PROTOTYPING VS. SPECIFYING: A MULTI-PROJECT EXPERIMENT

Barry W. Boehm*, Terence E. Gray, and Thomas Seewaldt**

University of California, Los Angeles
Computer Science Department

*also with TRW Defense Systems Group **present affiliation: Universitaet Kaiserslautern

ABSTRACT

In this experiment, seven software teams developed versions of the same small-size (2000-4000 source instruction) application software product. Four teams used the Specifying approach. Three teams used the Prototyping approach.

The main results of the experiment were:

> Prototyping yielded products with roughly equivalent performance, but with about 40% less code and 45% less effort.
>
> The prototyped products rated somewhat lower on functionality and robustness, but higher on ease of use and ease of learning.
>
> Specifying produced more coherent designs and software that was easier to integrate.

The paper presents the experimental data supporting these and a number of additional conclusions.

1. INTRODUCTION

1.1 Motivation

Should the current specification-driven approach to software development be dropped in favor of an approach based on prototyping? There have been several recent proposals of this nature, and a great deal of discussion of the relative merits of the two alternative approaches.

Prototyping offers a number of attractive advantages, such as the early resolution of high-risk issues, and the flexibility to adapt to changing environmental characteristics or perceptions of users' needs. To date, however, there is not much information on whether the prototyping approach retains all of the advantages of the specification-driven approach, such as visibility and control of the software development process, and the ability to manage integration of many small programs into a large product. Nor is there much information on how the nature of a software product developed via prototyping compares with the nature of a product developed via the specification-driven approach.

In order to illuminate these and related issues, we undertook the experiment described in this paper. We had seven teams develop the same software product (a user-interactive software cost estimation model, comprising roughly 3,000 Pascal source instructions); four teams used a specification-driven approach, and three used a prototyping approach. The resulting data on the teams' experiences and products provide at least a start toward understanding the relative strengths and weaknesses of the prototyping and specifying approaches, and toward understanding how they may best fit into a next-generation software development methodology.

1.2 Background

At the beginning of every software project, the project manager is faced with a critical choice of approach. The primary choices are:

> **Building and Fixing.** Proceed to build the full system with minimal or no specifications. Rework the resulting product as necessary until it satisfies its users.
>
> **Specifying.** Develop a requirements specification for the product. Develop a design specification to implement the requirements. Develop the code to implement the design. Again, rework the resulting product as necessary.
>
> **Prototyping.** Build prototype versions of parts of the product. Exercise the prototype parts to determine how best to implement the operational product. Proceed to build the operational product, and again rework it as necessary.

There are a number of variations on these three basic approaches, of course, but their essential distinctions are identified reasonably well in the above three options.

Approach 1, Building and Fixing, has been shown to work poorly on most projects of any reasonable size. This is largely because of the highly increased cost of fixing a software product once it is completed and operational [1]. The Speci-

0270-5257/84/0000/0473$01.00©1984 IEEE

fying approach evolved to avoid the problems encountered in Building and Fixing [2,3,4] and led to the familiar "waterfall" model of software development most frequently seen today.

The Specifying approach has been highly successful in many application areas [5,6]. However, it encounters difficulties in application areas in which it is hard to specify requirements in advance. This happens most frequently in human-machine interface systems, in which the requirements analyst often has to deal with user responses of the form, "I'm really not sure what I want, but I'll know it when I see it."

In such situations, the Prototyping approach appears attractive. A number of papers have proposed refinements of the waterfall model of the software life-cycle to incorporate prototyping options [7,8,9]. Some authors have gone so far as to suggest that prototyping options make all current life-cycle models completely obsolete and even harmful [10].

1.3 Open Questions

At this stage, however, not enough is known about the relative merits of specifying and prototyping to summarily reject either approach in favor of the other. The results of recent workshops such as the ACM-IEEE-NBS Workshop on Rapid Prototyping [11] indicate that a number of significant open questions still exist, such as:

> What characterizes application areas in which prototyping is likely to be more successful than specifying?
>
> What effect does prototyping have on a software project's effort distribution, schedule distribution, and productivity; and on the product's size, quality, maintainability, etc.?
>
> How does prototyping change the mix of skills needed on a software project?
>
> Is there need to adopt a mixed strategy using both specifying and prototyping? If so, when and how?

Clearly, such questions need to be further illuminated via analysis or experimentation before we can formulate definitive recommendations on the critical issue of specifying vs. prototyping.

1.4 The Experiment

The experiment described here was designed to investigate such questions. It involved having seven teams develop their own versions of the same product, four teams using the specifying approach and three teams using the prototyping approach.

The product to be developed was an interactive version of the COCOMO model for software cost estimation [12]. The tables and equations in the model were the same for all projects, and provided an overall definition of the product's requirements. However, each team was to determine and create its own user interface to the model. A previous experiment [13] indicated that the interactive COCOMO model would be a suitable product for such an experiment. It is a reasonably sized job for an experiment, and the user interface constitutes the dominant portion of the product.

The experiment took place in early 1982 as part of a one-quarter first year graduate course in software engineering at UCLA. The four Specifying teams produced a requirements specification, a design specification, and an end product consisting of operational code, a user's manual, and a maintenance manual. The three Prototyping teams produced the same end products, but were required to produce and exercise a prototype by the midpoint of the course, rather than to develop specifications. All of the projects were instrumented to collect data relevant to the open questions above.

1.5 Outline of Paper

Section 2 of this paper describes the experimental project in more detail. Section 3 presents the experimental results. Section 4 presents the resulting conclusions.

2. THE EXPERIMENTAL PROJECT

This Section discusses the key aspects of the experiment: the product developed; the project schedule and work environment; the organization into teams; the experimental data collection procedures; and some of the experimental limitations caused by the course schedule and teaching objectives.

2.1 The Product

Each team was to develop an interactive version of the COCOMO model for estimating the costs of a software product. The model accepts descriptions of the components of the future product in terms of their size, and their ratings with respect to 16 cost-driver attributes (e.g., hardware constraints, data base size, personnel experience, use of tools and modern programming practices). It uses these to calculate the amount of time and effort (and resulting dollar cost) required to develop each component and the overall system, and provides a breakdown of the effort into the major development phases and activities.

The model algorithms and tables were provided in [12]; each team was to develop its own file system and user interface The user interface for such a product is considerably more extensive than the cost model algorithms. The user interface software must support the selective creation, addition, modification, query, and deletion of the cost-driver parameters describing each component of the software product whose costs are to be estimated. It must support the specification, generation, formatting, and dispatching of desired outputs: overall cost, effort, and schedule estimates, and their breakdown by component, by phase, and by activity. It must detect and provide messages for erroneous inputs, and provide some level of on-line help. There are also a wide variety of further options which may be included, and many alternative ways to accommodate inputs (menus, commands, tables, forms), display outputs, and support user control. Thus, there are a good many issues to be addressed via prototying or specifying which have a significant influence on the nature of the project and its resulting product.

2.2 Project Schedule

The major milestones for each type of team were:

Week	*Specifying Teams*	*Prototyping Teams*
3	Requirements	
5		Prototype Demo
6	Design Spec Draft User Manual	
10	Acceptance Test User Manual Maintenance Manual	Acceptance Test User Manual Maintenance Manual
11	Project Critique Maintenance Vote	Project Critique Maintenance Vote

The requirements and design specifications were subjected to a thorough review by the instructors. This resulted in a set of Problem Reports returned to the project teams and discussed in a set of Requirements Reviews and Design Reviews. The prototypes were exercised by the instructors, who provided similar feedback on errors, suggested modifications, missing capabilities, etc.

The acceptance test consisted of the instructors' exercising each program to determine whether it performed all of the required capabilities, whether it handled error conditions with useful responses, and whether it exhibited a high degree of user-friendliness. Subsequently, the authors of this paper exercised each product in more detail, and each author rated it on a scale of 0 to 10 with respect to four particular criteria:

> **Functionality:** the relative utility of the various computational, user interface, output, and file management functions provided by the product.
>
> **Robustness:** the degree to which the user was protected from aborts, crashes, loss of working files, etc.
>
> **Ease of Use** (or, lack of frustration): the degree of user convenience in performing desired functions, and the avoidance of overconstrained or unexpected program behavior.
>
> **Ease of Learning:** the ease with which new users could master the product's workings and get it to do what they wished. This rating covered not only program prompts, help messages, and error messages, but also the user manual and associated job aids or "crib sheets" provided by the teams.

The project critiques were ten-page essays written by each student, addressing the question, "If we were to do the project over again, how could we do it better." These critiques were analyzed for the degree of consensus among the participants of the most important factors influencing the project results.

The maintenance ballots asked each student to rate each of the other teams' products in the order in which they would prefer to have the product as their product to maintain. The average rating for each product was then calculated as an index of its maintainability. 20% of each person's course grade was based on his product's maintainability rating.

2.3 Development Environment

The products were developed in UCB Pascal, using a UCLA VAX 11/780 running the Unix (TM: Bell Labs) operating system. The Unix environment provided excellent support for both documentation and code development functions. Some difficulties were the overload on the VAX at the end of the quarter combined with poor documentation of the separate compilation facilities in UCB Pascal, which made product integration and test much more complex and time-consuming than expected.

2.4 Team Organization and Staffing

At the beginning of the course, the students were given a description of the project, and a form to indicate their level of experience with Pascal, with Unix, and with programming; their grade point average; and their preference for which approach to use on the project. The instructors then selected teams based on the students' preferences and on experimental balance.

The 11 students expressing a preference for specifying were divided into four teams (S1-S4). The 7 students expressing a preference for prototyping were divided into three teams (P1-P3). The resulting team characteristics are given in Table 1 below.

Table 1. Experimental Team Characteristics

	Specifying					Prototyping			
Team	S1	S2	S3	S4	Avg.	P1	P2	P3	Avg.
No. of people	3	3	2	3	2.75	2	3	2	2.33
Avg. programming experience (mo)	25	47	42	30	36	54	46	60	53
Avg. Pascal exp. (mo)	1	17	7	3	7	30	16	9	18
Avg. Unix exp. (mo)	0	1	12	5	4.5	3	4	0	2.3
Avg. grade point average	3.1	3.6	3.6	3.3	3.37	3.4	3.0	3.3	3.27

Each team was given the freedom to organize in whatever way the members found most appropriate. Most teams used a highly democratic consensus-based organization,with all members performing some design, some programming, some documentation, and some integration and test. Some teams had a single individual develop certain documents, such as the user's manual.

2.5 Experimental Limitations

The teaching objectives of the course introduced several experimental limitations which somewhat reduced the sharpness and representativeness of the results.

Technical Leveling. A pure experiment would have isolated the teams to minimize any cross-fertilization of ideas or technical leveling between projects. Here, our teaching objectives caused us to hold every requirements review, design review, and prototype exercise in front of the entire class. Thus, prototypers got some added insights from the specifiers' reviews, and vice versa. However, our impression is that the students did not significantly change their approaches as a result of this information.

Nonrepresentative reviewing. In order to provide thorough feedback on specifications, and to show the value of early verification and validation, the instructors performed more thorough reviews of specifications than are performed on the typical project. The prototype exercises were also somewhat nonrepresentative in being one-shot exercises by expert users rather than sustained usage by non-expert users.

Choice of Approach. The Specifying teams were staffed entirely with students who had expressed a preference for the Specifying approach, and similarly for the Prototyping teams. This is largely nonrepresentative of actual projects -- although some students' critiques indicated that they would prefer taking the opposite approach if they were to do a similar project again.

Data collection procedures. The instructors explained to the students that their grade had nothing to do with the timesheet data they turned in, so there was no reason to falsify data. However, students occasionally exhibit procrastination and lapses in discipline. Thus, it was not too surprising that some of the timesheets were turned in late, with the attendant possibility that some of the data provided was created "from memory".

As stated above, these factors tend to reduce the sharpness and representativeness of the results. However, the net impression from the project critiques is that none of these factors played a critical role in the outcome of the experiment. Thus, the experimental results described below appear to transcend these acknowledged limitations. As a further point of perspective, it is worth noting that many conclusions reached in the software engineering field are still based on sample sizes of one project. Thus, a sample of seven reasonably comparable, moderately representative projects is not too bad.

3. EXPERIMENTAL RESULTS

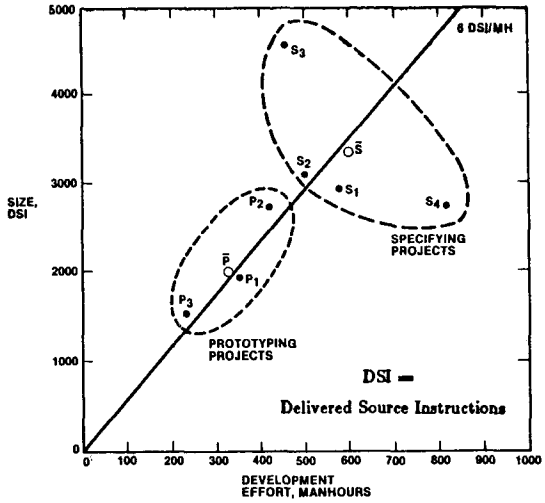
3.1 Prototyping vs. Specifying

Product Size and Development Effort

The comparisons of the relative sizes of the products and the relative effort required to develop them produced a striking result: the prototyping teams' products were 40% smaller, on the average, and required 45% less effort to develop.

The products of the prototyping groups had an average size of 2064 delivered source instructions (DSI), while the products of the specifying groups had an average of 3391 DSI. The average development effort of the prototyping groups was 325 man hours (MH), and for the specifying groups, 584 MH. Figure 1 shows the relative results for each project and the averages by type of group.

Figure 1. Prototyping vs. Specifying:
Size and Effort Comparisons



Both differences might be partly due to the smaller average team size of the prototyping groups (2.33 persons vs. 2.75), but comments in the project critiques indicate that the group type most significantly influenced these results. Specifically, the specifying people indicated that it was very easy to overpromise in their specifications. For example, when confronted with a review comment such as, "Some users would like to enter data by rows as well as columns," the specifiers would tend to say, "Sure, that's just another sentence in the spec." When confronted with this sort of comment in their prototype review, prototypers had a better feel for the programming implications, and tended to say,"We'll put that in if we have time."

The range of product sizes was from 1514 DSI to 4606 DSI (Table 5 provides data on each project). The second largest product was 3391 DSI, so we consider the 4606 DSI product somewhat anomalous. Even so, the 3:1 range in product sizes in remarkable, considering that each team was developing essentially the same product. This range, and the comparable 3.4:1 range in project effort, tend to corroborate the ranges in [12, Chapter 21] on the relative accuracy of early software sizing and costing efforts.

Overall Productivity

One of our hypotheses was that the prototyping projects would have higher "productivity" in terms of Delivered Source Instructions per Man-Hour (DSI/MH), primarily because the prototyping teams did not have to expend the ex-

Table 5. Summary of Project Results

	Specifying Teams					Prototyping Teams				ANOVA Significance	
	S1	S2	S3	S4	AVG.	P1	P2	P3	AVG.	Spec/Proto	Team Size
TEAM SIZE	3	3	2	3	2.75	2	3	2	2.33		
PROGRAM SIZE (Deliv. Source Inst.)											
File	64	622	246	714	411	45	356	204	201		
User Interface	1462	1910	2830	1064	1817	1123	1505	815	1148		
Compute	143	62	648	195	262	84	178	70	111		
Output	931	264	267	405	467	349	513	293	385		
Other	385	306	615	331	434	351	174	132	219		
TOTAL	2985	3164	4606	2809	3391	1952	2726	1514	2064	0.0674•	0.7891
Omitting S3 data					2986				2064	0.0668•	0.0039••
DOCUMENTATION (Pages)											
Rqts. Spec.	19	13	11	11	14	-	-	-	-		
Design Spec.	38	123	59	83	76	-	-	-	-		
User Manual	32	45	73	35	46	38	37	30	35		
TOTAL	165	231	181	179	189	58	49	56	54	0.0001••	0.3743
MANHOURS											
thru Rqts.	83	44	44	70	60	-	-	-	-		
thru Design	225	160	144	242	193	-	-	-	-		
thru Prototype	-	-	-	-	102	129	84	105	-		
Reading	29	33	40	72	43	21	24	43	29		
Planning	44	7	30	41	30	22	15	15	17		
Designing	81	46	59	82	67	16	13	12	13		
Programming	276	162	135	289	216	109	147	39	98		
Documenting	48	82	53	92	69	54	50	33	46		
Testing	45	49	55	29	44	27	75	46	49		
Reviewing	2	3	6	20	8	19	3	5	9		
Fixing	27	42	19	48	34	33	39	27	33		
Meeting	30	49	24	80	46	19	44	11	25		
Misc.	7	26	40	37	28	3	13	2	6		
TOTAL	589	498	459	789	584	323	422	232	325	0.0471••	0.0817•
PERFORMANCE											
Functionality	6.33	7	5	6	6.08	5.33	5	4	4.78	0.0799•	0.0799•
Robustness	4.67	5.5	6	4.33	5.13	4.33	4.33	3	3.89	0.0875•	0.7535
Ease of Use	2.33	4	2.67	4	3.25	6	5.33	2.67	4.67	0.2144	0.9133
Ease of Learning	3.67	3.5	4	3.67	3.71	5.67	5.33	3.67	4.89	0.0771•	0.6022
MAINT. SCORE (low is good)	45.5	57	59.5	59	55	21	27.5	45.5	31	0.0217••	0.6968
PRODUCTIVITY (DSI/MH)											
Overall	5.1	6.4	10	3.6	5.8	6	6.5	6.5	6.3	0.9727	0.1737
Coding	10.8	19.5	34.1	9.7	15.7	25	18.5	38.8	21		

tra effort to develop requirements and design specifications. This hypothesis was not borne out by the experimental results: both the prototyping and specifying groups averaged roughly 6 DSI/MH, where the number of man-hours includes effort expended for all phases of the project, not just coding. The prototyping groups had an average productivity of 6.3 DSI/MH; the specifying groups, 5.8 DSI/MH.

The range in overall productivity for prototyping groups was from 6 to 6.5 DSI/MH; for specifying groups it was from 3.6 to 10, but with only one group (the same one that produced the largest product) exceeding 6.4 DSI/MH. As shown in Fig. 1, the development effort is generally proportional to the size of the developed product.

Note that the large variation in product size does not invalidate the DSI/MH productivity measure as an indicator of efficiency in producing code. Rather, it illustrates the need for further research into why ostensibly similar products can differ so dramatically in size, and therefore, development cost.

Coding Productivity

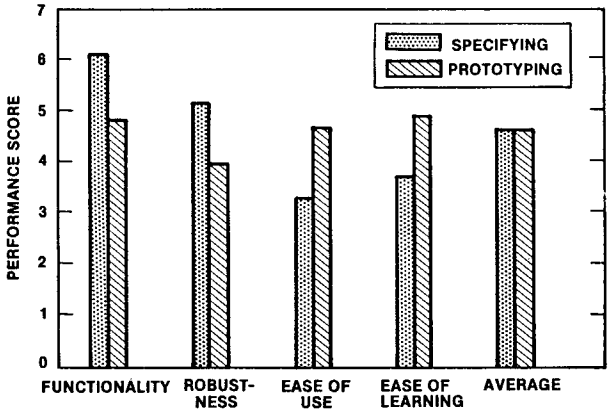
An unexpectedly large variation in coding productivity (DSI/programming MH) was observed. The low was 9.7 and the high was 38.8 DSI/MH. While the average for specifying teams (15.7) was 25% lower than that of prototyping teams (21), much of this difference is attributable to the pronounced team-size effect discussed in section 3.2.

Product Performance

Since the products of the prototyping groups were smaller and developed with less effort, one might think that they would rate correspondingly lower on performance. However, their overall performance was the same as the performance of the specifying groups (Fig. 2). The prototyped products were rated lower in overall functionality and in their tolerance of erroneous input, but correspondingly higher in their ease of learning and ease of use (i.e., the frustration caused by overconstrained or unexpected program behavior was less for the products of the prototyping groups). There was not a uniform dominance of prototyping or specifying projects on any of the performance ratings. Practically all the ratings were in the range of 3 through 7 (a "5" rating corresponded to an "acceptable" product), but there was considerable variation in ratings within the range in each group.

Figure 2. Specifying vs. Prototyping:

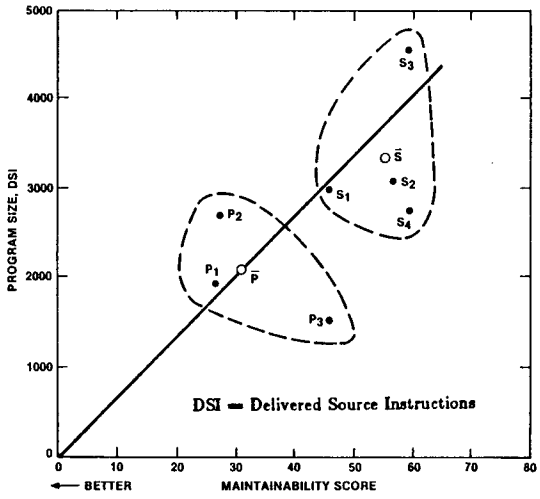
Performance Comparisons



Maintainability Ratings

At the end of the quarter, students indicated which products they would prefer to maintain and ranked the products accordingly. The maintainability of the prototype group products was rated remarkably higher than the maintainability of the specifying group products. Therefore, the student's subjective evaluations did not confirm the hypothesis that the specifying approach leads to lower maintenance costs. Somewhat paradoxically, however, the products of the prototype groups were judged worse as a basis for planning add-ons.

Figure 3. Maintainability Rating vs. Size



In exploring which factors influenced the maintainability rating, we compared the rating to the size of the products (Fig. 3). At first glance, it appears as if the students preferred to maintain smaller products. However, when asked in the follow-up questionnaire what the main criteria for their maintenance rating were, the students ranked the size of the products only third, and size was mentioned only as one factor besides design, programming style and documentation, and performance (see Table 2).

Table 2.

Ranking of Maintenance Rating Criteria

Factor	# Students Citing Factor
design	8
programming style	5
size of the product	4
documentation	3
product performance at acceptance test	2

Effort Distribution

During the development process, the groups had to meet different deadlines. The specifying groups had to hand in a requirements specification in the third week and a requirements and design specification in the sixth week. The prototype groups had to present their prototype for a prototype exercise in the fifth week. For both groups, the acceptance test took place in the tenth week after project start. Figures 4 and 5 show the average effort distribution by phase and activity for both group types. While the "deadline effect" observed in [13], with major peaks of the total effort before the deadlines, is fairly distinct in the effort distribution of the specifying groups, the effort distribution by phase for the prototyping groups is much smoother. Especially at the end of the quarter, the programming, testing, and fixing effort of the prototyping groups did not peak, as it did for the specifying groups. Instead, the effort-peak for the prototyping groups at the end was mostly due to documentation effort.

Figure 4. Effort Distribution by Phase and Activity: Specifying Groups

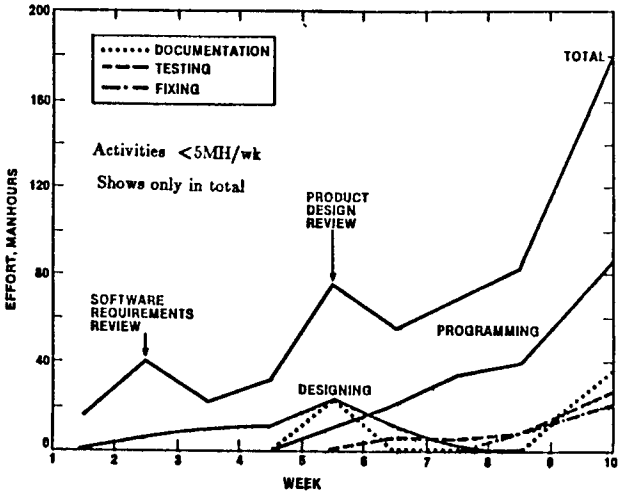


Figure 5. Effort Distribution by Phase and Activity: Prototyping Groups

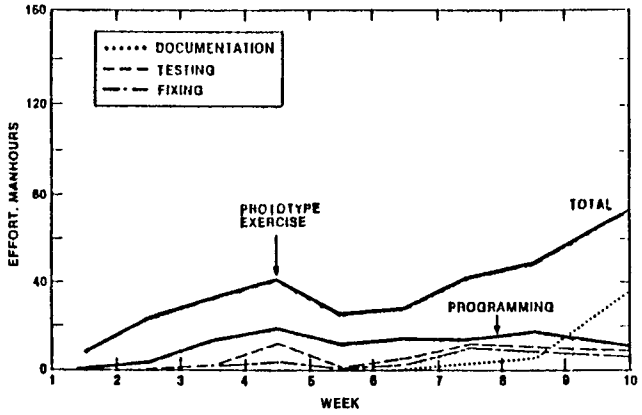
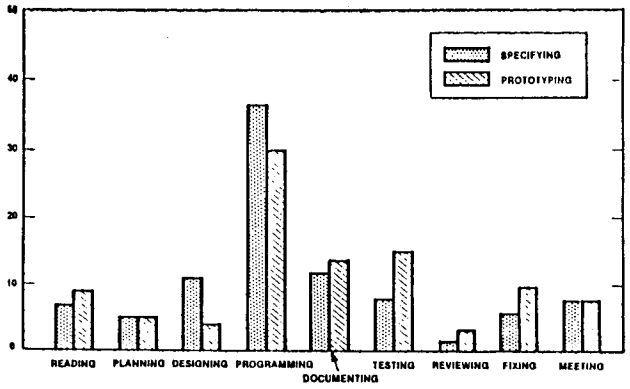


Fig. 6 shows the effort distribution by activity in percent of the total effort for both group types. Proportionately, the prototype groups spent less time for designing and programming, more for testing, reviewing, and fixing. The higher effort needed to integrate the prototype products was confirmed by comments in the project critiques.

Figure 6. Distribution of project effort by activity



Going through the whole design process before coding seemed to simplify the integration by forcing developers "to think before coding". On the other hand, building a prototype had the advantage of "always having something that works".

Documentation Productivity

As mentioned already, no difference between the two groups in the overall productivity (DSI/MH) could be observed. However, there appeared to be a significant difference in the documentation productivity. The specifying groups produced, on the average, 2.8 pages per documentation MH, the prototype groups only 1.2 (Table 3). An explanation for this effect might be that the members of the specifying groups were more motivated to write documents, for they chose their group type knowing that a lot of documents would have to be produced. In addition, the specifying groups had 3 deadlines where documents were to be presented, the prototype groups only 1. Therefore, the "deadline effect" might have influenced the documentation productivity of the specifying groups more than the documentation productivity of the prototyping groups. Another explanation may be that a good deal of documentation was produced in the process of design; if the design man-hours are added to the documentation man-hours, the difference in pages/MH is reduced to 1.2 vs 0.9.

Table 3. Comparative Documentation Productivity

	specifying groups	prototyping groups
pages of documentation	161	54
MH for documentation	69	46
MH for design	67	13
productivity (page/MH)	2.8	1.2
pages/(design + doc. MH)	1.2	0.9

Retrospective Comments

In the follow-up questionnaire, the students were asked how their project outcome would have differed, if they would have belonged to a different group type. The answers of the students of the specifying groups were not uniform. Some would have expected a better product, some a worse product, if they had been in a prototyping group. On the other hand, the students of the prototyping groups mostly indicated that specifying would have increased the performance of their product and would have resulted in a faster development process. It is interesting that the data of the experiment generally did not confirm this expectation.

Analysis of Variance

An analysis of variance (ANOVA) was performed to determine the relative levels of statistical significance of the results above. The results are shown in the "Spec/Proto" column of Table 5.

In general, a difference between treatment groups is considered significant if its significance score is less than 0.05 (indicated by two asterisks in Table 5). A score between 0.05 and 0.10 is considered reasonably significant (indicated by one asterisk in Table 5).

Table 5 thus summarizes the following conclusions about the significance of the differences between the specifying and prototyping groups:

- The differences in documentation size, total manhours, and maintenance score are significant;
- The differences in program size, functionality, robustness, and ease of learning are reasonably significant;
- The differences in productivity were not significant.

3.2 Smaller vs. Larger Teams

A second analysis of the data was conducted, investigating the influence of the group size on the product and the development process, independent of the type of team.

Although the smaller teams of both types needed 41% less effort to develop their product (338 vs. 575 MH), the average size of their products was only 8% smaller than the average size of the products of the larger teams (2690 vs. 2921 DSI). Therefore, the productivity, as a measure of delivered source instructions per man hour, was higher for the smaller teams than for the larger teams (7.5 vs. 5.4 DSI/MH).

However, the almost equal average product size and the higher productivity might perhaps be due to one exceptional project: One of the two-person teams (S3) developed a product which was significantly larger than the products of all the other groups (4606 DSI vs. 3164 DSI for the second largest product). The productivity of this group was also by far the highest (10 DSI/MH), due to one extremely prolific programmer. Removing this anomalous case brings the average productivity of smaller teams from 7.5 DSI/MH down to 6.25 DSI/MH and reveals a significant relationship between team size and product size.

The performance of the products of the smaller teams (again, independent of team type) was rated somewhat lower, mainly due to a lower functionality score. It seems that the smaller teams were not that much concerned about providing "fancy" functions, but more about getting their work done. (This may also have been because two of the three smaller teams were prototyping teams.) Due probably also to the lower manpower available, their products were less debugged than the products of the larger teams. On all the other factors -- ease of use, ease of learning, and tolerance of erroneous input, the team size had only little correlation.

Comparing the effort the teams spent for different activities, for both prototyping and specifying teams, the smaller teams needed proportionally less time for programming than the larger teams (27% of total effort vs. 38%). As expected, they needed also less time for meetings (5% vs. 9%). Comments in the project critiques and follow-up questionnaires confirm that the larger teams had more communication problems and communication overhead than the smaller teams.

Asked how the product outcome would have changed if they would have been a group of the other group size, people of the larger groups indicated that reducing their team size by one person would have reduced the performance of their products. On the other hand, people of the small teams did not expect an increase in team size to lead to an increase in product performance.

The analysis of variance results comparing the 2-person and 3-person teams are shown in the "Team Size" column of Table 5. Most of the team size differences are not statistically significant; total manhours and product functionality rating achieve a reasonably-significant level. In the other evaluation categories team size does not appear to be a primary driver of the experimental results. However, as noted above, removing the anomalous data of team S3 yields a statistically significant ($s = 0.0039$) difference in product size across the groups with different team size. Thus, if we drop project S3, team size appears to influence product size more than development style.

3.3 Characteristics of the Development Process

As shown already in Fig. 6, the dominant activity in the development process of both group types was programming. The specifying groups spent 37% of their total effort for programming, the prototyping groups 30%. No other activity took more than 20%. This result conflicts with one of the results of the earlier experiment reported in [13], in which the percentages of effort for programming (12-17%) and documentation (27%-32%) were reversed. The main reasons

were most likely the requirements for more documents (project plans, test plans) and for updating documents, and the larger team sizes in the earlier experiment; and the improved documentation aids provided by Unix in the later experiment.

In the current experiment, due to the tight schedule (only 10 weeks were available for the project), the development documents (requirements specification and design specification) were only written once. After the software requirements review and product design review, they were not corrected to incorporate modifications due to problem reports. Also, they were not updated when changes were made during the development process. Since the team size was relatively small and, therefore, the communication within the groups was good, the lack of up-to-date documents was not critical. However, if up-to-date documents had been required for other reasons, the proportional effort spent for documentation would have increased and, probably, at least reached the programming effort.

In order to investigate whether the COCOMO model can be applied to this kind of class project, the data of the different products were entered in one of the products and a prediction was calculated. The results are shown in Table 4. Even allowing for the 30-40% difference due to non-project activities explained in [12], a significant discrepancy remains. Several factors might have influenced this discrepancy. First, as mentioned above, no final version of the requirements and design specification was written and the documents were not updated. In addition, the fact that 7 groups were working on the same kind of project simultaneously and the grade in the course was mostly based on the project outcome might have produced a much more competitive situation than is found in a normal program development environment. Also, the fixed schedule with no possibility for prolongation, imposed by the duration of the quarter, might have contributed to this effect.

Table 4. COCOMO Estimates vs. Project Actuals

Team	Man-Months actual	Man-Months predicted	Actual / Predicted
specifying	3.8	12	0.32
prototyping	2.1	6.6	0.32

Note that the COCOMO prediction offset was independent of the development style. For both specifying and prototyping groups, the model predicted the same percentage of the real effort. The mismatch between model estimates and project actuals is not overly surprising; in general, algorithmic cost models have a difficult time with small projects.

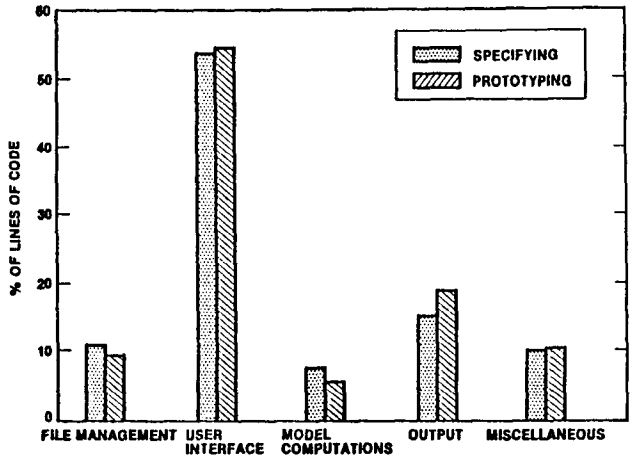
3.4 Characteristics of Products

Although all groups were given the same task, the product architectures differed significantly. They frequently seemed to be a reflection of the developer personalities (elaborate vs. simple displays, terse vs. verbose messages, free-form vs. directed sequential inputs, etc.).

Most of the groups (6 of 7) developed a menu driven system. Screen oriented interaction was preferred to line-oriented interaction. The man-machine interface and the flexibility of the dialogue was very product- and architecture-dependent. In addition, competition stimulated man-machine interface frills.

The distribution of source code by function for products of both group types was almost the same (Figure 7). Although the main purpose of the product was to calculate a cost, effort, and schedule estimation, the portion of code devoted to this purpose was very small (5% - 8%). The user interface turned out to be the most important part of the system. It took over 50% of the code. These results are consistent with the earlier results in [13].

Figure 7. Distribution of source code by function



The portion of code devoted to the file management system differs heavily between the products. It comprises anywhere between 5% and 40% of the code. Its size depends mostly on the provided capabilities and on how much file management functions of the operating system were used vs. implementing a new file management system for the product. (The original directions to the terms said simply "develop a single-user file system for input data.")

In the number of delivered source instructions per person there was only little variation. With one exception (2303 for team S3), they were all in the range of 757 - 1055 DSI/person. The same is true for the overall performance of the products. With one exception (13.3 points), all products were rated between 17 and 21.3 points.

3.5 Some Other Observations

The organization of the team was left to the individual team members. Since no group leader was explicitly determined, every group followed more or less a democratic approach. Yet, the preferred organization was highly people-dependent. Four students mentioned in their critiques that the democratic approach worked well, while three students would have preferred to have a team leader.

The prototyping strategies were quite consistent over the prototyping groups. After building the prototype, no group started new to build the final product. Rather, 67 - 95% of the prototype code was used in the final product. The size of the prototypes was between 40 - 60% the size of the final product. An overall summary of project results is given in Table 5.

4. CONCLUSIONS

The results of this experiment provide some useful quantitative and qualitative information on the relative effects of the specifying and prototyping approaches on the development of a small applications software product. However, as indicated at several points earlier, initial experiments of this nature are not likely to provide definitive conclusions applicable to all project situations. The experimental results can be sensitive to exceptional individuals' performance or to experimental boundary conditions, or the results may depend on the size and nature of the software application. Therefore, the conclusions below should be considered suggestive rather than definitive. The conclusions are thus presented in project-specific rather than general terms.

4.1 Conclusions on Specifying vs. Prototyping

Prototyping tended to produce a smaller product, with roughly equivalent performance, using less effort. The prototyped products averaged about 40% smaller than the specified products, and required about 45% less effort. In performance, they rated somewhat lower on functionality and robustness, but somewhat higher on ease of learning and ease of use. Statistically, these differences were at least reasonably significant.

The main reason for this effect appeared to be that prototyping fostered a higher threshold for incorporating marginally useful features into a software product. The process of prototyping gave software developers a more realistic feel for the amount of effort required to add features to a project, and the lack of a definitive specification meant that prototypers were less locked into a set of promises to deliver capabilities than were the specifiers. In the somewhat rueful words of one of the specifiers, remarking on his team's efforts to fulfill the promises in their ambitious specification, "Words are cheap."

Prototyping did not tend to produce higher "productivity" if "productivity" is measured in delivered source instructions per man-hour. However, if "productivity" is measured in equivalent user satisfaction per man-hour, prototyping did tend to be superior. This conclusion reinforces the desire for a better productivity metric than the number of source instructions developed.

Again, this conclusion does not necessarily apply to every project situation. The value-of-information decision guidelines for software projects in [12, Chapter 20] identify a number of situations in which the information value of a prototype will not be worth the investment in it. Even for projects similar to the one in this experiment, the Specifying approach may be able to produce similarly concise products if the specification reviews are strongly focussed on elimination of marginally useful product features.

Prototyping did tend to provide a number of benefits frequently ascribed to it. These included:

- Products with better human-machine interfaces;
- Always having something that works (at least for "build-upon" if not for throwaway prototypes);
- A reduced deadline effect at the end of the project.

Prototyping led to better maintainability ratings, but the effect was unclear. At the same time, participants' critiques indicated that specifications led to more coherent designs and that prototyping made it harder to plan additions.

Prototyping tended to create several negative effects. These included:

- Proportionally less effort planning and designing, and proportionally more testing and fixing;
- More difficult integration due to lack of interface specifications;
- A less coherent design.

These effects become particularly critical on larger products. This suggests that, especially for larger products, prototyping should be followed by a reasonable level of specification of the product and its internal interfaces.

4.2 Conclusions on Team-Size Effects

Smaller teams produced smaller products with less effort and a higher "productivity" in DSI/Man-hour. This conclusion is only statistically significant if the anomalous data for team S3 is removed from the analysis. The two-person teams spent smaller percentages of their effort in programming and meeting. Their products were rated somewhat lower on functionality, but about the same on ease of use, ease of learning, and robustness.

Some of this team-size effect may have accounted for some of the differences between the Specifying and Prototyping results, since the average size of the Prototyping teams was somewhat smaller (2.33 persons vs. 2.75). However, the corroborative evidence from the project critiques indicates that the projects' results were strongly influenced by whether they used a specifying or a prototyping approach.

4.3 Conclusions on Other Software Engineering Effects

The "deadline effect" observed on a previous project [13] was corroborated. Also, this experiment corroborated the previous observation that most of a product's code is devoted to largely application-independent "housekeeping" functions.

The previous conclusion in [13] that "documentation is the dominant activity during software development" was not corroborated by this experiment. In fact, programming was the dominant activity during this experiment, due most likely to differences in project groundrules and team sizes. This result emphasizes the need for follow-up experiments to confirm conclusions reached during software engineering experiments.

The most effective software project organization in strongly dependent on the nature of the people on the project. Some people's critiques emphasized the need for a strong leader, as in the Chief Programmer Team approach. A larger number of people felt that a more democratic team approach was more effective.

The COCOMO model strongly overestimated the amount of effort required to develop the experimental products. The overestimates were typically by a factor of about 2.5, much larger than could be explained by not counting the typical 30-40% of the workday devoted to non-project activities. Most likely, the extra productivity was a result of exceptional motivation of the people involved, both from the competitive-team aspects and from a Hawthorne effect.

Nothing succeeds like motivation. This was the major cause of both the high team productivity and the very high level of maintainability of their products. The software field in general needs a maintainability motivator similar in power to that of telling students, "20% of your course grade will depend on how much others want to maintain your product."

4.4 Future Research

It is clear that the large number of variables in the present experiment made it impossible to draw unambiguous conclusions. We believe it is equally clear that experiments such as this can make a significant contribution, particularly as others repeat them and thereby increase the sample size.

Subsequent experiments of this type should attempt to reduce the number of variables by:

- Making all of the teams the same size (prohibit prime numbers of students per class!)
- More precisely defining the user-interface requirements, so that everyone implements close to the same functionality.

Several different directions for investigation were also suggested by this work:

- Further examination of the effect of team size on programmer productivity and product size. Use the same development approach and environment, and a precisely defined product definition.
- Further examination of prototyping vs. specifying approaches applied to different phases of a development project: definition of functional requirements, design decisions, and implementation decisions. In other words, how does the development approach affect the *nature* of the product, as distinct from the cost of development?
- Examination of the effect of implementation language choice on programmer productivity and product size. A series of class projects should be an excellent way to investigate this question. Of particular interest: comparison of interpreted and compiled languages.
- Examination of factors that influence the style of user-interface chosen by a particular designer, e.g. available development tools, assumptions about user environment, and personality traits of the designer.
- Investigation of the effect of turn-around time on programmer productivity. Is the effect linear or non-linear? To what extent does it depend upon the programmer's expectations and previous experience?
- Examination of the "user-manual first" approach on product size, quality, and effort expended.
- Investigation of how accurately developers can predict final product size from requirements definitions. Also, how much does the existence of design specs improve accuracy of product size predictions?
- Further investigation of the effect of prototyping vs. specifying on maintenance and enhancement costs.
- Further examination of the wide variations observed in coding productivity, with respect to both team size and development approach.

4.5 Summary

The results of this experiment indicate that both prototyping and specifying have valuable advantages that complement each other. For most large projects, and many small ones, a mix of prototyping and specifying will be preferable to the exclusive use of either by itself. In particular, the results indicate that:

1. *The current specification-oriented model should not be completely scrapped, particularly on large projects.* The prototypers' experience indicated that interface and design specifications were still particularly valuable in supporting integration and change implementation.

2. *The current model needs to be reoriented to accommodate prototyping,* and such related techniques as incremental development. This involves establishing such new life-cycle milestones as a User Design Review (UDR) to achieve user validation of a prototyped user interface.

3. *Contracting for software acquisition needs to reflect the reoriented model.* This involves the use of competitive front-end prototyping and "fly-offs," and the organization of the development into a series of stabilized increments of functional capability.

4. *The bottom-line driver on selection of the specific mix of prototyping and specifying should be risk management.* Prototyping is not necessary on familiar projects where there is little risk of getting the wrong user interface, requirements, or design. Elaborate specifications are not necessary on smaller projects with good user-developer rapport, where there is little risk of botching the integration process or having an altercation over contract deliverables. Risk management considerations also drive most of the other key management decisions over the software life-cycle (how much to invest in analysis, simulation, new technology, testing, quality assurance, configuration management, etc.), leading to a final implication:

   Software projects should develop, maintain, and follow a Risk Management Plan, which identifies potential high-risk issues, establishes plans for resolving them, and highlights risk-item resolution in project status reviews.

As a final note, it is worth re-emphasizing the conclusion that the prototyping approach resulted in products that were easier to learn and use. For a field which is searching for ways to make its products more humane, this experiment indicated that the prototyping approach clearly has a great deal to offer.

REFERENCES

[1] B.W. Boehm, "Software Engineering," *IEEE Transactions on Computers,* pp. 1226-1241, Dec. 1976.

[2] H. D. Benington, "Production of Large Computer Programs," in *Proceedings, Symposium on Advanced Programming Methods for Digital Computers,* ONR, Washington, D. C., June 1956, pp.15-27

[3] W. A. Hosier, "Pitfalls and Safeguards in Real-Time Systems with Emphasis on Programming", *IRE Transactions on Engineering Management,* pp. 99-115, June 1961.

[4] W. W. Royce, "Managing the Development of Large Software Systems: Concepts and Techniques", *Proceedings, WESCON,* August 1970.

[5] R. D. Williams, "Managing the Development of Large-Scale Reliable Software", *Proceedings, 1975 International Conference on Reliable Software,* IEEE/ACM, April 1975, pp. 3-8.

[6] R. L. Glass, *Modern Programming Practices: A Report From Industry,* Prentice-Hall, 1982.

[7] M. M. Lehman, "Programs, Life Cycles, and Laws of Program Evolution", *IEEE Spectrum,* Sept. 1980.

[8] P. Kerola and P. Freeman, "A Comparison of Lifecycle Models," *Proceedings, Fifth International Conference on Software Engineering,* IEEE/ACM, March 1981, pp. 90-99.

[9] B. W. Boehm, "Software Design and Structuring", in E. Horowitz (ed), *Practical Strategies for Developing Large Software Systems,* Addison-Wesley, 1975.

[10] D. D. McCracken and M. A. Jackson, "Life Cycle Concept Considered Harmful", ACM SIGSOFT *Software Engineering Notes,* pp. 29-32, April 1982.

[11] M. Zelkowitz and M. Bramstad, *Proceedings, ACM SIGSOFT Rapid Prototyping Symposium,* Columbia, MD, April 1982.

[12] Boehm, B. W., *Software Engineering Economics,* Prentice-Hall, 1981.

[13] B. W. Boehm, "An Experiment in Small-Scale Application Software Engineering," *IEEE Transactions on Software Engineering,* pp. 482-493, September 1981.