SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper., **29**(10), 833–847 (1999)

Productivity Analysis of Object-Oriented Software Developed in a Commercial Environment

THOMAS E. POTOK[1]*, MLADEN VOUK[2] AND ANDY RINDOS[3]
[1]*Oak Ridge National Laboratory, Post Office Box 2008, Building 6010, Oak Ridge, TN 37830, USA (email: potokte@ornl.gov)*
[2]*College of Engineering, Department of Computer Science, North Carolina State University, Box 8206, Raleigh, NC 27695, USA*
[3]*International Business Machines Corporation, Dept. CE6A, Bldg. 664, P.O. Box 12195, 3039 Cornwallis Rd, Research Triangle Park, NC 27709-2195, USA*

SUMMARY

The introduction of object-oriented technology does not appear to hinder overall productivity on new large commercial projects, but nor does it seem to improve it in the first two product generations. In practice, the governing influence may be the business workflow, and not the methodology. Copyright © 1999 John Wiley & Sons, Ltd.

KEY WORDS: object-oriented; software development productivity

INTRODUCTION

As software development cycles shorten and software markets become more competitive, improved software development productivity continues to be a major concern in the software industry. Many believe that object-oriented technology provides a breakthrough solution to this problem, but there is little quantitative evidence to support this belief. Furthermore, most studies related to object-oriented productivity do not consider it in conjunction with the business processes and culture under which the software is developed.

In this paper we present a comparison of empirical software productivity data for a number of commercial software products developed in the same organization and business model, using both 'classical' procedural methods and object-oriented methods. Our results indicate that, although the introduction of object-oriented technology does not appear to hinder overall productivity on new large commercial projects, it does not seem to improve it in a systematic way, at least not in the first two product generations. Furthermore, examination of the data indicates that the governing influence may not be the methodology, but the business model imposed through schedule deadlines.

*Correspondence to: Thomas E. Potok, Oak Ridge National Laboratory, Post Office Box 2008, Building 6010, Oak Ridge, TN 37830, USA.

CCC 0038–0644/99/100833–15$17.50

Received 27 August 1998
Revised 29 March 1999
Accepted 29 March 1999

Evidence

There is surprisingly little *quantitative* evidence that the productivity of object-oriented software development is indeed consistently better than that of 'classical' procedural software development in a commercial environment. Published evidence appears to derive primarily from productivity studies made in non-commercial environments under non-commercial business models, and the scalability of the results to commercial environments is not clear.

For example, Lewis *et al.* [1] performed an experiment with undergraduate software engineering students to study the effect of reuse. Based on their productivity metrics, they concluded that the object-oriented paradigm can improve productivity when reuse is present by about 50 per cent (about 1.5 times). However, they did not find any statistically significant evidence that the object-oriented paradigm has a higher productivity rate than procedural methods when reuse is not a factor. Melo *et al.* [2] also conducted an experiment with graduate students that yielded seven projects ranging in size from 5000–25,000 lines of code. The projects were developed using the Waterfall process model, object-oriented design, C++, and varying levels of reuse. Their results support the conclusion that reuse rates can increase programmer productivity by as much as two to three times. When reuse levels become cost-effective is still an open question. Optimistic economic models [3] of reuse indicate that break-even reuse levels may be as low as 10–20 per cent, while pessimistic models [4] contend that cost-effective levels of reuse may be much higher as well as difficult to achieve. Fichman *et al.* [5] and Lee *et al.* [6] report that there are significant barriers to the adoption of reuse in an organization.

There is also evidence that other factors may confound the picture. For example, different development methodologies may impact on software development productivity in ways other than through reuse. Boehm-Davis *et al.* [7] report on a comparison of Jackson's program design, object-oriented design, and functional decomposition. They found that Jackson's method and object-oriented methodologies produce more complete solutions; require less time to design and code a problem; and produce less complex designs than functional decomposition. However, a quantitative comparison of productivities associated with different methodologies was not given. Similarly, Zweben *et al.* [8], again in an experiment with graduate and undergraduate students, show that language-based layering and encapsulation (an object-oriented trait) may reduce software development effort. There are many other studies concerned with the value of the object-oriented approach, but most are not quantitative in nature.

Recent work by Hansen [9] and Fichman *et al.* [10] correctly asserts that software development must first be viewed as a business. The referenced productivity studies focus on the productivity effects of object-oriented technology isolated from the effects of a typical business workflow. Noticeably absent are convincing *quantitative* studies that focus on productivity related to new object-oriented software developed by professional programmers under commercial business models. It is not difficult to understand that, although many industrial organizations claim to practice object-oriented software development, many practicing software engineers and managers are quite cautious on the subject of object-oriented productivity. In fact, it would appear that many organizations simply do not systematically measure software reuse (and the associated productivity), and therefore may not have more than anecdotal evidence for or against object-oriented productivity gains [11].

Productivity

Productivity can be measured in many ways. A traditional approach is to use project size or amount of functionality, e.g. in Lines Of Code (LOC) or function-points, and divide that by the time or effort spent in developing the code. In an object-oriented environment, LOC may not be an ideal metric for software size or functionality, but Tegarden *et al.* [12] report that traditional metrics, such as LOC, may be still appropriate for measuring the complexity of object-oriented systems.

In fact, in a commercial situation there are many other factors besides size that impact on software costs and productivity. This is particularly true if issues such as marketing, staff training, applied research, long-term maintenance, and customer support are taken into account. None of these extra factors are reflected in traditional LOC metrics, but do require expenditure of effort. Unfortunately, in practice, LOC is often the only available metric. Therefore, we define productivity in terms of effective developed/changed LOC, but with an understanding that the effort (or time) expended may include many non-coding activities that are necessary in viable commercial products (see equations (1) and (2).

We define the average productivity of the software development team by the following relationship:

$$\textit{Team Productivity} = \frac{\textit{Project Size}}{\textit{Team Effort}} \tag{1}$$

where team productivity is the measure of the team output for a given unit of time or effort, e.g. thousands-of-lines-of-code (KLOC) per month. The project size is the number of KLOCs required to develop the project, and the team effort is the number of person-months required to develop the project. To derive the average productivity of an individual programmer, we divided the team productivity by the average number of programmers on the software development team. Finally, to find the effort expended by individual programmers, we merely rearranged the resulting equation, which gives:

$$\textit{Programmer Effort} = \frac{\textit{Project Size}}{\textit{Programmer Productivity}} \tag{2}$$

These equations imply a linear relationship between the effort and size of a project, which assumes that the profile of the work performed by each product team is approximately the same, for example, each team spends about 5 per cent on training activities and about 3 per cent on quality initiatives. There is no data available to support this assumption; however, business guidelines tend to dictate the amount of time that team members should spend on various activities. Clearly, this varies from organization to organization and person to person, but the assumption that work profiles from team to team are consistent is an assumption that we are comfortable with. Although this type of linear relationship may be suitable for comparing team productivity on multiple products, practice shows that the size and productivity of a software team often varies over the duration of a project; and that average programmer productivity is a non-linear function of a number of factors. Boehm [15] defines the effort required to develop a software project in terms of size

$$\textit{Effort} = \alpha(\textit{Size})^{\beta} \tag{3}$$

Other researchers used similar models [13–15] The parameters α and β are constants that are typically determined using regression on the loglinear version of this model, i.e.

$$\ln(\textit{Effort}) = \ln(\alpha) + \beta \ln(\textit{Size}) \tag{4}$$

The typical experience with this type of model is that $\beta > 1$, i.e. larger projects have lower productivity than smaller projects, however, some researchers have reported β values less than one [15].

Data†

The empirical data discussed in this paper was collected at the IBM Software Solutions Laboratory in Research Triangle Park, North Carolina. This laboratory employs about 650 people, with approximately 90 per cent being directly involved in software development. The laboratory was ISO 9000 Certified in 1994, and has consistently received high marks in internal assessments against the Malcolm Baldrige Criteria. The lab was formed in 1984, and produces a wide range of software products, ranging from mainframe end-user interface design tools to workstation visual builders.

We examined 19 commercially available software products developed at this laboratory. The measurements collected are defined by a corporate metrics council. This data is recorded by members of each product development team, and used by lab management in managing and controlling projects. Of the 19 products, 11 were developed using object-oriented methods and eight using traditional procedural methods. All object-oriented projects are either first or second generation, while all procedural projects are second or higher generations. Four of the object-oriented products were inter-platform software ports, where the original software was developed using object-oriented methods, then ported to work with another operating system. Five projects were developed for mainframe use, and 14 for workstation use. The product development activity ranges in size from about 1 thousand (KLOC) to about 1 million lines of new or modified code. There is a very wide range of team productivity for the products, with the highest productivity rate being nearly 50 times more productive than the lowest productivity rate. Project development duration is recorded in calendar months from the time when the project was officially funded to the first customer ship date. The effort is reported in person-years, and includes the effort of the programmers, testers, writers, planners, managers, and vendors. In this number is also included a person-year equivalent for purchased software. For example, if software was purchased for $300,000, and the average programmer cost is $150,000 per year, then this purchase would be equated to two person years of effort. Software reuse data is listed as a collected metric, however, none of the investigated products explicitly reported reusing any code. This is most likely due to a mixture of process omissions and widely differing application areas and platforms of the examined products.

BUSINESS MODEL

The general business model that drives software development at this laboratory recognizes two major software product sub-categories: versions and releases. A new version is typically quite large, and contains a significant product enhancement, or change in functionality. A version is ordinarily followed by one or more maintenance releases that are usually much smaller than a version, and contain fixes to defects, and minor enhancements. The calendar-time duration for development of both versions and releases is strongly driven by market forces. Versions tend to take longer than releases, but are within an 18–24 month window

†Data used by permission. The scales appearing on the axes of all graphs, and any product and date-related information, has been altered to provide discretion.

common in the industry today. Release development will normally not be shorter than 9–12 months. There are a number of reasons for this, some of which are distribution costs, arrival rate of release-type fixes and changes, and possibly the issue of user-perceived quality (e.g. scheduling of a release very soon after a version can give the impression of quality problems). While all new development must be completed with a limited number of personnel, existing projects will have an established team. Typically, an effort is made to maintain or even increase the size of the team because it may not be cost-effective to dismantle the team between versions. Therefore, it is not unusual to have a large version developed with tight resource and time constraints, yet have a smaller follow-up maintenance release developed over a more relaxed schedule using the same team.

The development of both versions and releases is subject to frequent high-level reviews of their schedule status against key development dates (or milestones) established at the beginning of the product cycle. The progress towards these dates is reviewed regularly and in detail, and schedule slips in any major milestones are strongly discouraged. Detailed project schedules are required at the beginning of the product development cycle, and they trigger business processes including funding, planning, marketing, supporting, and certification of the quality of a product.

The software development teams are formed according to the skill and experience requirements of the project. The overall experience level from team to team is typically about the same. The teams that develop software using object-oriented technology are well skilled in the discipline with numerous internal and external courses available. Further, the lab employs several highly respected object-oriented experts.

At the time this data was collected there was a program in place to encourage software reuse by moderately rewarding contributions to software reuse libraries, and highly rewarding authors of reused components. This program has since been halted due to lack of participation. Why this program did not succeed is an open question with several possible explanations, including: (1) the well documented difficulties of reusing software; (2) the heterogeneous nature of the products developed at the lab; or (3) possibly the lack of focus during staff reductions.

EMPIRICAL PRODUCTIVITY

In Figure 1 we plot for each of the 19 products the logarithm of the product size versus the logarithm of the product effort. We further categorize the products into three groups: those developed using object-oriented methods, dark circles; those developed using procedural methods, light circles; and those object-oriented projects that were ported from one platform to another, dark squares. Based on the model described in equation (4), we see an apparently linear relationship between the logarithm of product size and effort, i.e. as a product gets larger, more effort is required. If there were a distinct productivity difference between two groups, this would appear on the graph in Figure 1 as two distinct groups of points, with the higher productivity products being closer to the x-axis. From this plot, it appears that the ported products, as expected, have significantly higher productivity than the non-ported projects, and that there are no obvious differences between the procedural and object-oriented products.

Porting software can be viewed as software development where most of the design and significant parts of the code are reused, and although it should not be confused with reuse, it offers a hint of the reduction in effort that can be obtained through reuse. While ports are generally less costly than new software development, and in itself the observed difference

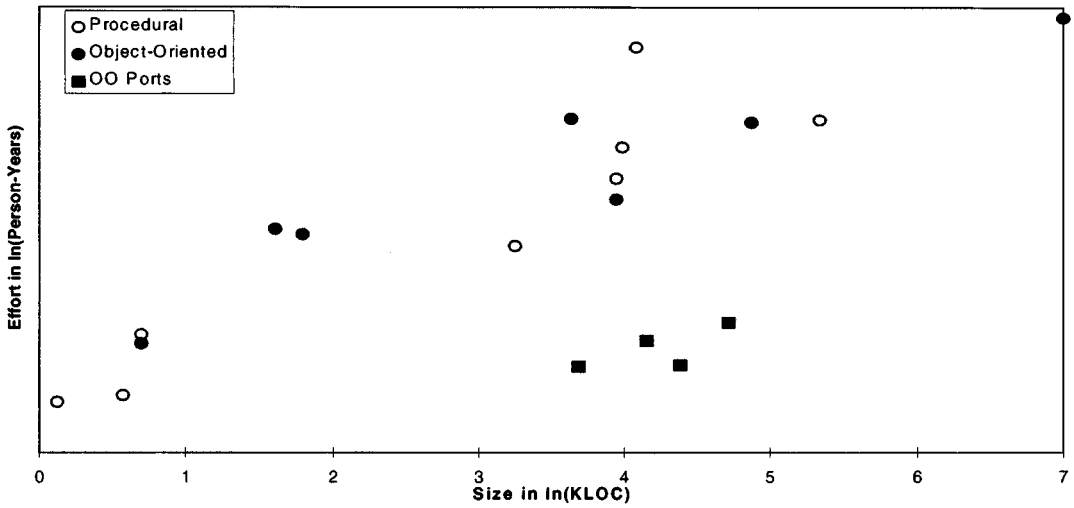

Figure 1. A plot of effort vs. size for procedural, object-oriented and object-oriented port projects

between the ported and other categories is not unusual, it is important to remember that the object-oriented approach *inherently* provides a mechanism for reuse not only when software is ported, but also for enhancements and in development of future releases [16]. Hence, the productivity gains observed during the porting of object-oriented software may be a good reflection of the possible gains the technology offers if reuse is the key productivity driver.

The second observation, that there appears to be no productivity distinction between products developed using object-oriented and procedural methodologies, runs counter to the limited study on object-oriented productivity that has been performed to date, and to the commonly held view that object-oriented methods improve productivity. There are many possible explanations as to why this result has been observed, for example: (1) the observation is not supported by statistical analysis: (2) LOC is an inadequate metric for object-oriented products; (3) the object-oriented methodology was not properly used; (4) more generations of object-oriented projects are needed to see a productivity gain; or (5) there is an underlying factor that is skewing the result for object-oriented products.

ANALYSIS

To examine these issues further, we extend the loglinear model described in equation (4), so that procedural and object-oriented product data can be compared statistically. This extension adds a methodology factor to the model so that a significant difference between projects of different methodologies can be tested. Equation (4) is extended to include a methodology factor, and applied to only the non-ported data, as shown in equation (5):

$$\ln(PM) = \alpha_1 + (\beta_1)\ln(KLOC) + (\beta_2)Method + (\beta_3)\ln(KLOC)Method + \varepsilon \qquad (5)$$

where α_1, β_1, β_2, β_3 are constants, $\ln(PM)$ is the logarithm of effort recorded in Person-Months, $\ln(KLOC)$ is the logarithm of the effective project size in KLOC, *Method* is a class variable that indicates the development methodology, either object-oriented or procedural, and ε is the regression error term. This relationship provides a single regression model for the full data, and a means of testing whether the methodology is a significant factor in this data.

This analysis was performed using the proposed regression model, with the results provided in the Appendix. This analysis confirms the observation made in Figure 1, that for this data *there is no statistically significant evidence that the productivity of object-oriented software development is different than that of procedural software development*. As expected, a similar evaluation of the data on ported software versus other software shows a significant difference between the two. It is encouraging that the introduction of object-oriented technology does not appear to carry excessive productivity penalties, a fear that some managers may have. It is less welcome, but probably not so surprising, that there is no obvious productivity gain in the first and second generations of object-oriented projects.

This analysis reveals two interesting points: first, there exists a very strong economy of scale that can be seen in the parameters reported in Table II of the appendix, which yield the model[‡], $Effort = 70.1(KLOC)^{.43}$. An economy of scale for software development is unusual, however, cases have been reported [15,17,18]. The second interesting point is the high R^2 value recorded for the regression model. A low R^2 value would indicate that LOC is a poor representative of project size, i.e. two projects may have similar LOC values, but require vastly different amounts of effort. Another way to state this is to argue that the LOC metric is skewed for object-oriented projects, namely that more functionality can be created with an object-oriented methodology and language than a comparable amount of code in a procedural environment. A high R^2 provides some support for the use of the LOC metric as a reasonable predictor of project size/complexity for procedural development as well as object-oriented development.

DISCUSSION

As stated above, there are several possible explanations as to why no productivity gain was seen in this data. It appears that two can be ruled out: (1) the observations reported about Figure 1 are statistically valid; and (2) the LOC metric seems to be reasonable for this data. Another possible explanation is that a procedural methodology (perhaps mistakenly called object-oriented) was used in all of these projects, but with some using an object-oriented language. After reviewing the processes used by several of the object-oriented projects, and assessing the skill level of several of the key object-oriented developers, we believe there is little doubt that an object-oriented methodology was being followed.

Another possible explanation is that only two generations of object-oriented projects are analyzed, and that the productivity enhancing effects of reuse may not be seen until later generations. Although this may be the case, there appears to be no trend towards an increase in the productivity from the first to the second-generation object-oriented projects (see Figure 2). If reuse was evident, one would hope to see some indications that the second-generation products (the hollow circles) are taking advantage of reuse, but unfortunately there is no evidence of this in this data sample. Additionally, two of the object-oriented productivity studies above seem to show that the object-oriented methodology improves productivity by producing a simpler design, not merely through reuse [7,8]. Improvements due to improved designs would seem to be independent of product generation, yet do not appear to be present in this data either.

Moreover, it may be possible that weak teams were somehow assigned to the object-oriented projects, while the stronger teams worked on the procedural projects. Based on interviews and observations, it appears that the development teams had roughly the

[‡]Note that $e^{4.25} = 70.1$, where 4.25 is the regression coefficient $\ln(\alpha)$ from equation (4).

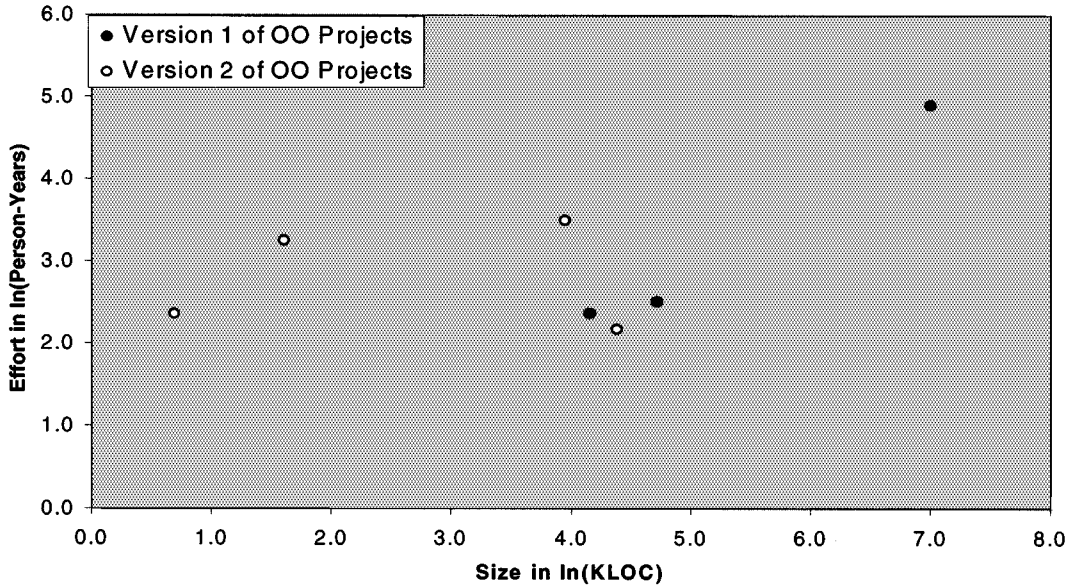

Figure 2. A plot of the development effort versus the size of the projects for first and second generation object-oriented projects

same experience and skill level throughout the organization. Furthermore, there was some speculation among the lab's senior technical people that perhaps the object-oriented teams are stronger than the procedural teams, since new technologies often attract higher performing people. In any case, it appears unlikely that the make up of the teams caused a reduction of the productivity rates for the object-oriented projects.

UNDERLYING FACTORS

There appears to be no obvious explanation for the lack of a productivity gain from the object-oriented methodology in this data. However, exploring the unusual economy of scale observed in the statistical analysis may provide some further clues. We explore the economy of scale issue by analyzing the project staffing and productivity characteristics. This analysis shows some puzzling relationships. For example, Figure 3 shows the logarithm of average *programmer* productivity (LOC/Person-Month) versus the logarithm of project size (changed or modified KLOC).

This plot shows clearly what was seen in the regression model developed above, that counter-intuitively programmer productivity increases as the project size increases. We see an equally puzzling plot (Figure 4), that shows the logarithm of *team* productivity versus the logarithm of project size. It can be seen that team productivity (expressed as LOC/month) appears to significantly increase with project size. Typically, one would expect to see a wide variance in this value, since changes in programming team size should strongly influence the amount of code the team produces. For example, one would expect a 20-person team will have a much higher team productivity rate for a given project than will a 5-person team on the same project. However, this plot shows a remarkably strong relationship between team productivity and size.

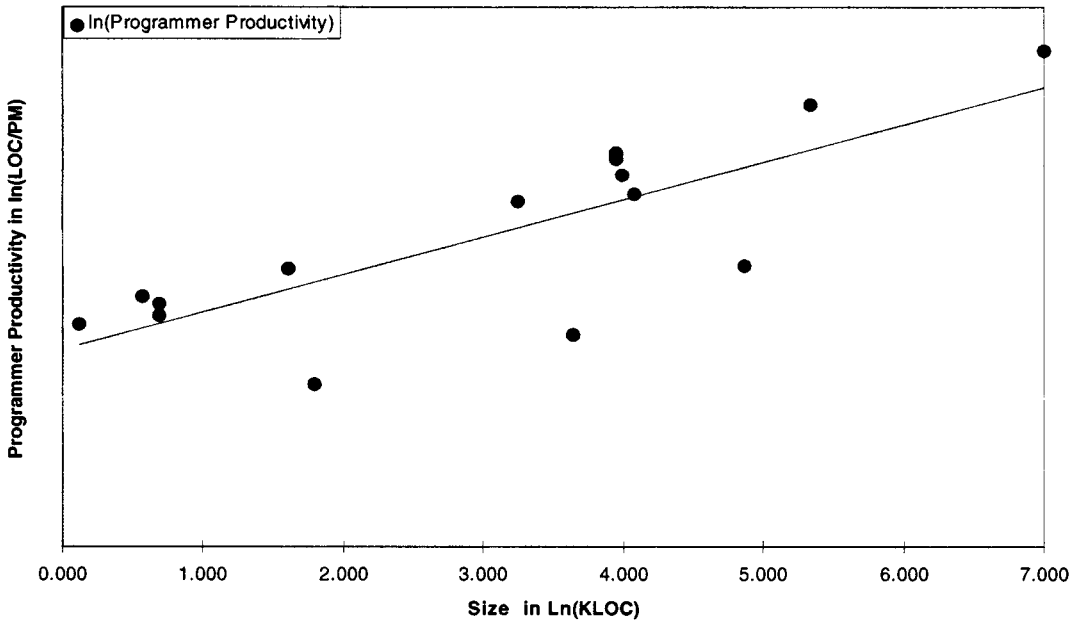

Figure 3. A plot of the average programmer productivity (LOC/Person-Month) as the project size (KLOC) increases

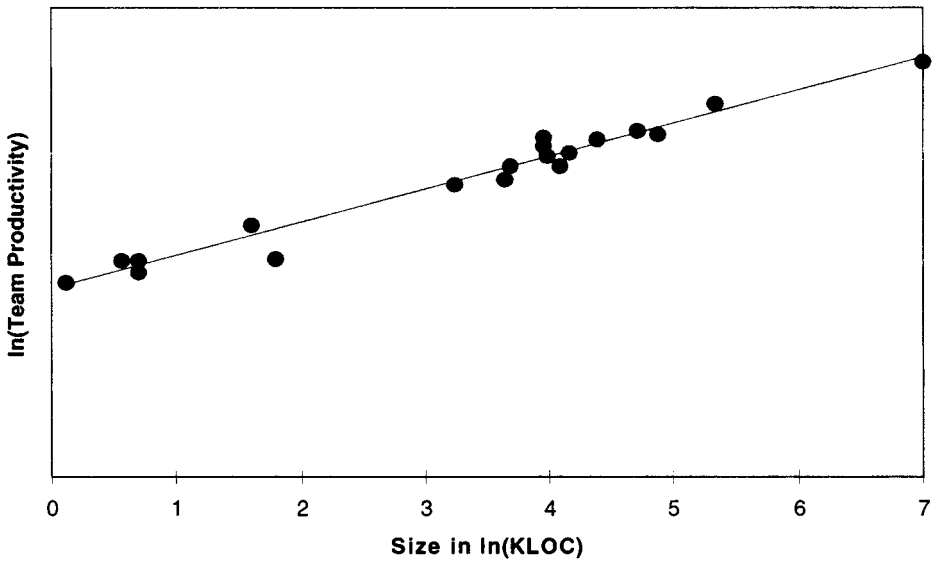

Figure 4. Team productivity vs. project size

Based on the reported productivity factors, again there is a variety of possible explanations for the economy of scale seen in this data. One explanation may be that there is a large but constant overhead associated with all projects, or that smaller projects are, for some reason, more complex. Unfortunately, there is no evidence to support either of these

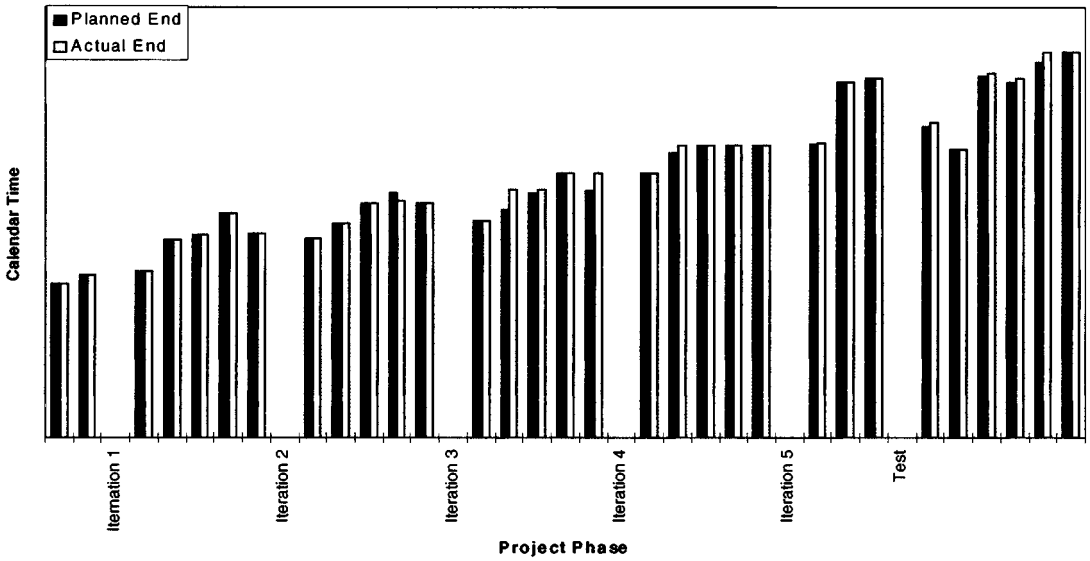

Figure 5. Second generation object-oriented project. The actual (light bars) and planned (dark bars) project duration vs. milestone number

assertions. However, based on interviews with the process owners and a review of the project documentation, it appears that the business model may have dictated the establishment of larger teams for certain types of smaller projects. For example, small intermediate product releases were required to preserve continuity of skills and expertise between large versions of the product, that in turn require large amounts of effort. While this may provide a partial explanation for the first six small projects, it does not really explain the productivity growth observed for larger projects.

The question is whether this growth is the result of schedule over-estimation for smaller projects, or something else. Given the fact that some of the larger projects exhibited productivity larger than the 'nominal' values that might be expected for the product based on, for example, the classical COCOMO model, it appears that some schedule compression may have taken place. This suggests that the gains in the productivity of larger products may be a result of schedule pressure.

This prompted an examination of project schedule histories of several projects in detail. Figure 5 illustrates the typical effects and trends that were observed. The plot shows a comparison of the planned and actual milestone completion durations for a second-generation object-oriented project. The dark bars represent the planned task duration, while the light bars represent the actual task durations. The vertical axis is the calendar time unit for task completion, and the horizontal axis represents the project milestones. For emphasis, the graph shows all tasks starting times superimposed, i.e. as if they were starting at the same time, which is obviously not the case in reality. It appears that the actual completion of tasks follows quite closely the planned completion for those same tasks. Similarly, compliant tracking patterns were observed for both procedural and object-oriented projects.

One possible explanation for Figure 5 is that the product planning process is very accurate. However, given a very wide variation in the average productivity over the examined projects,

it is unlikely that the productivity rate of the associated programming teams was so well known in advance. Another explanation could be that schedules were met because software functionality was changed or testing time was reduced to meet them. Examination of the project records show that no major functions were added or deleted in these projects, and that time was not saved by shortening testing cycles. For example, the project in Figure 5 shows delays in Iteration 3 that occurred in coding milestones, and that the schedules were brought back in line during this phase, not during testing. The project also has delays in the testing phase; however, the testing phases were entered and exited on schedule, indicating that the testing effort may have been shifted, but not shortened.

Yet another possible explanation is that the actual schedule data has been modified to match the planned expectations. In other words, regardless of when a task actually completed, a date close to the planned milestone was recorded. Unfortunately, this issue of potentially altered data must be seriously addressed. We are confident that the data used in this graph is accurate and reliable. First, the initial schedule used to drive the software development process is generated early in the software development lifecycle and used by the project management as a means of tracking progress on the project and triggering various business activities. The software development team typically reports progress to management on a weekly basis, with detailed project reviews held by the laboratory director each month. There are a multitude of groups that rely on the output of the team in order to do their jobs, i.e. technical writers, testers, support personnel, marketing groups, distribution groups, language translation groups, business partners to name a few. For a team to supply incorrect task completion dates, they must provide false information to their management chain running the very real risk that the dates will be questioned if the output of the task has not been received by a dependent group. Presenting false information in this manner can result in dismissal from the company.

Secondly, project histories from four projects were examined in total. These histories, of which, only one was shown, do report large delays (up to 20 weeks) in meeting some non-critical milestones, however, the final project completion milestone were consistently met in all four projects. The reporting process makes it very difficult to present false information, and there is evidence of large task completion delays. For these reasons, we believe that this graph provides initial evidence that dynamic schedule enforcement and compression took place, and may have been a factor in achieving the milestone compliance.

The detailed project data along with an understanding of the controlling business model seem to indicate that the business deadlines may strongly influence the overall productivity of a software development project. The characteristics we observed are: (1) a common programming team is formed regardless of project size; (2) there is an established maximum and minimum duration for the development cycle; (3) strong pressure exists to meet key project deadlines; and (4) there are two classes of development, large versions and small releases. In this environment, it appears that large versions are developed with a relatively small programming team on a relatively short cycle, with strong pressure to finish on schedule. This most likely results in an increase in apparent productivity. Smaller releases have relatively large programming teams, with a relatively lax schedule, and no incentive to finish early. For a software release, therefore, it seems that the conditions allow for lowering of the apparent productivity. This leads to the conjecture that the business model may drive smaller releases to have low productivity, and larger versions to have higher productivity [18]. Furthermore, it may be possible that the business model exerts influence over the productivity of object-oriented software development, and this influence may offset potential gains in the methodology. A simulation study of this phenomenon confirms that this may indeed be the governing influence [19].

SUMMARY

Our results indicate that in a commercial environment, there may not be a consistent statistically significant difference in the productivity of object-oriented and procedural software development, at least not for the first couple of generations of an object-oriented product. The reason may be low reuse level, but it could also be the underlying business model. Investigation of 19 commercial products has shown an unusual economy of scale for both object-oriented and procedural software that is difficult to explain with traditional productivity drivers. However, a review of the underlying business workflows has suggested that business deadlines may strongly influence the overall productivity. In an environment where a typical delivery cycle for product versions or release is on the order of 12–24 months it may be more economical to preserve development team skills and expertise by keeping them together whether they operate under the new release or maintenance schedules. This may produce aggressive schedules for new releases, and lax schedules for maintenance releases. Our data appears to indicate that business workflows can play a key role in realizing the potential productivity benefits from a new technology such as object-orientation. For example, funding, staffing, and scheduling an object-oriented project in the same way that is done for a procedural project appears to dictate the productivity of the team, regardless of the potential benefit of a given methodology. The adoption of an object-oriented methodology may necessitate changes beyond merely the new technology. The estimation of project effort, the scheduling of project tasks, and the tracking of task completions should all be examined based on the characteristics of a new technology. Otherwise, investment in technology that has the potential to increase productivity may be lost unless the underlying business work flows are adjusted to take advantage of the improved software development capabilities.

APPENDIX: REGRESSION MODEL STATISTICS

The regression statistics for equation (5) are summarized in Table I. Originally there were 19 data points of which data from the four ported projects was set aside, leaving 15 data points. The model fits the data with an R^2 value of 0.88.

Note that the T-statistics in Table I only test for each individual parameter, not a combination of parameters. By reducing the model of both the methodology factor, and the interaction between methodology and size, i.e. yielding the equation $\ln(PM) = \ln(\alpha_1) + \beta_1 \ln(KLOC) + \varepsilon$, (equation (4)), a model irrespective of methodology is produced. By comparing the full and reduced models, the joint hypotheses that methodology is not a significant factor, H_0: $\beta_2 = \beta_3 = 0$, can be tested. An F-test is used to evaluate the differences between the two models. The reduced model has an R^2 value of 0.85, and the regression statistics summarized in Table II.

Table I. The regression statistics for the model with a factor for methodology

	Coefficients	Standard error	T Stat	P-value
α_1	4.37	0.23	18.64	1.14E-09
β_1	0.49	0.07	6.90	2.6E-05
β_2	0.56	0.36	1.57	0.14
β_3	−0.13	0.10	−1.34	0.21

Table II. The regression statistics for the model without a factor for methodology

	Coefficients	Standard error	T Stat	P-value
$\ln(\alpha_1)$	4.25	0.18	23.64	4.56E-12
β_1	0.43	0.05	8.56	1.06E-06

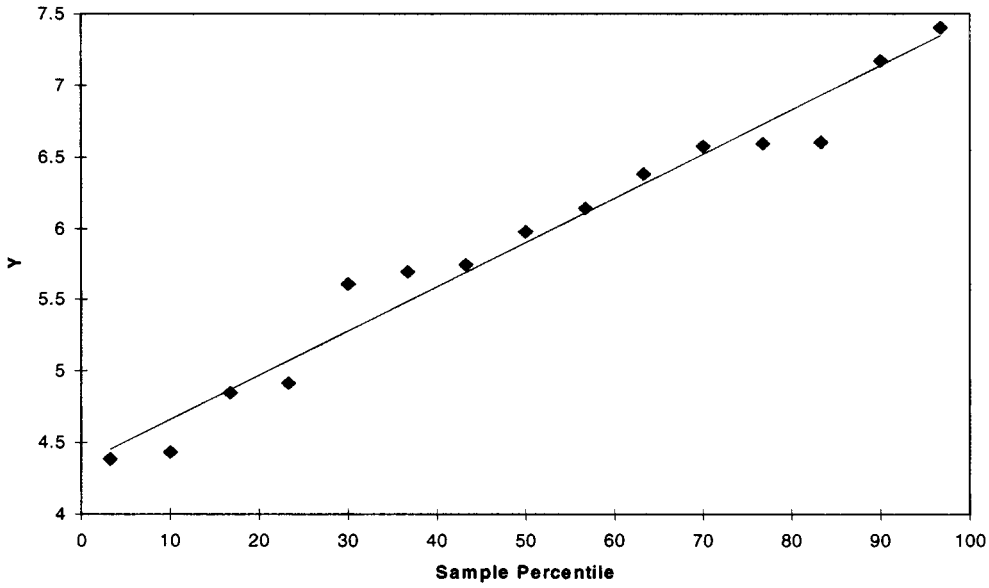

Figure 6. Normal probability plot of the residuals for the full model

Comparing the two models yields an F statistic of 1.23 for F(2, 11)[§]. The tabulated value of F(2, 11) at $\alpha = 0.05$ is 3.98; therefore, the null hypothesis is not rejected at that level, indicating that there is no statistical support for a methodology factor in modeling this data. To ensure that the underlying assumptions of the regression model are valid, the residuals of the full model are analyzed. Figure 6 shows the normal probability plot of the residuals, in order to detect nonnormality. The residual values in the plot appear to be linear, however, the trend-line through the data does not cross the origin. Given the relatively small sample size of the data, the variance of the data can easily cause this result [20].

Satisfied that the errors in the full regression model appear to be normal and independently distributed, further analysis of the residuals and standardized residuals is done to identify potentially influential points (see Table III). For the most part, the standardized residuals are relatively close to zero, with the exception of observation 12, which is roughly twice as large as the next largest residual value.

[§]The F statistic is calculated from the equation $F(\gamma_H, \gamma_{full}) = [(SSE_{reduced} - SSE_{full})/\gamma_H]/MSE_{full}$, where SSE is the sum of squares of error, and MSE is the mean squared error. γ_H d.f. for H_0, and γ_{full} for error under the full model. For this data, $SSE_{reduced} = 1.852$, $SSE_{full} = 1.513$, and $MSE_{full} = 0.138$ at $\gamma_H = 2$ and $\gamma_{full} = 11$ degrees of freedom. This produces an F-statistic of 1.228 at F(2, 11).

Table III. A list of the residuals, and standard residuals for the full model

Observation	Predicted Y	Residuals	Standard residuals
1	4.425392	−0.03838	−0.10349
2	4.643842	−0.21303	−0.57444
3	4.703311	0.215209	0.580328
4	5.175575	−0.32981	−0.88937
5	5.50018	0.242823	0.654792
6	5.564769	0.13484	0.363607
7	5.946235	−0.33896	−0.91404
8	6.218671	0.387742	1.045577
9	6.329106	−0.35073	−0.94576
10	6.289296	−0.15114	−0.40755
11	6.307668	0.077189	0.208147
12	6.353243	0.820256	2.211883
13	6.654392	−0.07734	−0.20854
14	6.965289	−0.37115	−1.00083
15	7.408761	−0.00753	−0.0203

The project represented in observation 12 was a procedural project that appeared to take an unusually large amount of effort to complete. After further review of the data, the values reported are accurate, and there is no reason to remove this point from the analysis.

ACKNOWLEDGEMENTS

This research was funded by IBM. The document preparation was supported by Lockheed Martin Energy Research.

REFERENCES

1. A. Lewis, S. M. Henry and D. G. Kafura, ‘An empirical study of the object-oriented paradigm and software reuse’, *Proceedings of the Conference on Object-oriented Programming Systems, Languages and Applications*, 1991, pp. 184–196.
2. W. L. Melo, L. C. Briand and V. R. Basili, ‘Measuring the impact of reuse on quality and productivity in object-oriented systems’, *Technical Report*, University of Maryland, Department of Computer Science, CS-TR-3395, 1995.
3. B. Henderson-Sellers, ‘The economics of reusing library classes’, *Journal of Object Oriented Programming*, **6**, 43–50 (1993).
4. D. Schimsky, ‘Software reuse: some realities’, *Vitro Technical Journal*, **10**, 47–57 (1992).
5. R. G. Fichman and C. F. Kemerer, ‘Object technology and software rese: lessons from longitudinal case studies of early adopters’, *IEEE Computer*, **30**, 47–59 (1997).
6. N. Y. Lee and C. R. Litecky, ‘An empirical study of software reuse with special attention to Ada’, *IEEE Trans. Software Engineering*, **23**, 537–549 (1997).
7. D. A. Boehm-Davis and L. S. Ross, ‘Program design methodologies and the software development process’, *International Journal of Man Machine Studies*, **36**, 1–19 (1992).
8. S. H. Zweben, S. H. Edwards, B. W. Weide and J. E. Hollingsworth, ‘The effects of layering and encapsulation on software development cost and quality’, *IEEE Trans. Software Engineering*, **21**, 200–208 (1995).
9. G. A. Hansen, ‘Simulating software development processes’, *IEEE Computer*, **29**, 73–77 (1996).

10. R. G. Fichman and S. A. Moses, 'An incremental process for software implementation', *Sloan Management Review*, **40**, (1999).
11. W. B. Frakes and C. J. Fox, 'Sixteen questions about software reuse', *Commun. ACM*, **38**, 75–87 (1995).
12. D. P. Tegarden, S. D. Sheetz and D. E. Monarchi, 'Effectiveness of traditional software metrics for object-oriented systems', *Proceeding of the 25 International Conference on System Sciences*, 1992, pp. 359–368.
13. C. E. Walston and C. P. Felix, 'A method of programming management and estimation', *IBM Systems Journal*, **16**, 54–73 (1977).
14. J. W. Bailey and V. R. Basili, 'A meta-model for software development resource expenditures', *Proceedings of the Fifth Internations Conference on Software Engineering*, 1981, pp. 107–116.
15. B. W. Boehm, *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs, NJ, 1981.
16. M. A. Vouk, 'On the cost of mixed language programming', *ACM SIGPLAN Notices*, **19**, 54–60 (1984).
17. S. E. Elmaghraby, E. I. Baxter and M. A. Vouk, 'An approach to the modeling and analysis of software production process', *International Trans. Operational Research*, **2**, 117–135 (1995).
18. T. E. Potok and M. A. Vouk, 'Development productivity for commercial software using object-oriented methods', *Proceedings of CASCON'95*, 1995.
19. T. E. Potok and M. A. Vouk, 'The effects of the business model on object-oriented software development productivity', *IBM Systems Journal*, **36**, 140–161 (1997).
20. G. E. Box, W. G. Hunter and J. S. Hunter, *Statistics for Experimenters*, Wiley, New York, NY, 1978.

