A Study of Productivity and Efficiency for Object-Oriented Methods and Languages

Daniel Port
Univ. of Southern California
Center for Software Engineering
941 W. 37th Place #328
Los Angeles, CA 90089 USA
+1 213 740 7275
dport@sunset.usc.edu

Monica McArthur
Hughes Space and Communications
El Seguendo, CA
+1 310 416 2916
Monica.McArthur@hsc.com

ABSTRACT
A study was commissioned by Hughes Space and Communications software engineering group to determine the effectiveness of the recent introduction of object-oriented languages, technologies, and development methodologies. Of particular was any effects on development productivity. Fundamentally productivity metrics are difficult to apply across non-homogeneous projects and development teams. Furthermore, owing to many uncontrolled variables such as the lack of a solid control project and non-rigorously collected data, productivity measures alone are were sufficient to determine meaningful results as requested for the study. A new, robust means of comparing non-homogenous development efforts (that does not require a control project) called “efficiency” was introduced and used to augment the comparative analysis of the projects and address possible concerns with the use of productivity metrics alone. Efficiency measures the actual effort compared to an estimate of that effort with respect to an independent and well-defined baseline – for purposes of the study this was COCOMOII. Conclusive evidence was found to support the hypothesis that object-oriented languages coupled with object-oriented methods result in greater productivity and efficiency as compared to other efforts. Furthermore it is concluded that efficiency metrics along with the non-standard use of COCOMOII is a meaningful, useful and practical approach to compare development efforts.

Keywords
productivity, efficiency, COCOMO, development effectiveness, software metrics, empirical metrics

1. INTRODUCTION
The software engineering organization within Hughes Space and Communications (HSC) requested a study of the effect of the recent insertion of object-oriented technology into their software development. They wanted to compare the performance of projects using various levels of OO technology through productivity metrics, but this approach proved to produce less-than-satisfactory results. Another approach, using a newly defined metric of “efficiency”, was substituted; this metric produced more meaningful results and allowed for conclusions about the effectiveness of OO technologies. The approach to computing efficiency was inspired by Barry Boehm’s study on the impact of evolutionary software technology on development efforts. We present here both the new metric and the conclusions about OO technologies.

The paper is organized as follows: Section 1 discusses the nature of the study, the projects, and the organization. Section 2 provides some basic definitions, use and difficulties of productivity metrics followed by the same for efficiency in section 3. The data, methods of collection, and metrics are displayed in section 4 as well as an analysis of what these metrics imply in regards to the study objectives. Finally the overall conclusions are presented in section 5.

1.1 Study Objectives
The study was conducted based on the request of a software engineering organization that had recently begun to introduce object-oriented methodologies and languages and wanted to assess the impacts of the introduction. This organization had expected that insertion of object-oriented technology would have certain results, most importantly increased productivity as measured by source lines of code (SLOC)/hour generated per developer, reduced project costs, and overall improved software quality such as fewer defects and more rapid implementation of changes. Quantifiable models of software development productivity and efficiency are highly elusive and generally require great care and consideration regarding how the sample project (or projects) are set up, monitored, and analyzed. Meaningful definitions can be difficult to create, as is finding valid data to use in conjunction with them. There are underlying assumptions that must be justified within the models used. Assumptions are also implicit in the manner data is processed into information and used within the models. Within a fast paced commercial software development environment such luxuries are generally not practical to contend with. Owing to this and of the necessity of utilizing assumptions within modeling practice, as well as inherent constraints on the quality of data, there is no universally applicable model that faithfully represents inherent “truth” or significance. Thus our goal is not to find an absolute answer, rather it is to uncover evidence via the models, which may then be used to either support or refute any given hypotheses meaningful to HSC.
In this work we will step by step construct, justify, and then use models of productivity and efficiency. Ultimately, we will analyze the results and evaluate to what extent their expectations were met.

1.2 Background Organization
The organization that requested the study is the software engineering component of Hughes Space and Communications (HSC), a major aerospace company. HSC has engaged in the development and production of state-of-the-art space and communications systems for military, commercial, and scientific uses since the early 1960’s. HSC has a long history of developing software in-house to support these systems, but they have only recently begun to introduce object-oriented technology into their software development.

Background of Projects Studied
Four projects were studied, three of which used object-oriented technologies to varying degrees, and one which used standard development processes to serve as a “control”. Below is a table summarizing the projects studied.

<table>
<thead>
<tr>
<th>Label</th>
<th>Type</th>
<th>Development Approach</th>
<th>Implementation Technologies</th>
<th>Language(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Advanced prototype, semi-unprecedented</td>
<td>Mostly traditional</td>
<td>Mostly OO</td>
<td>Ada 95</td>
</tr>
<tr>
<td>B</td>
<td>Custom product, somewhat unprecedented</td>
<td>Mixed OO and traditional</td>
<td>Mostly OO</td>
<td>C++</td>
</tr>
<tr>
<td>C</td>
<td>New product, unprecedented</td>
<td>OO</td>
<td>Mostly OO, some non-OO</td>
<td>Java, C</td>
</tr>
<tr>
<td>D</td>
<td>Product line extension, not unprecedented</td>
<td>Traditional</td>
<td>Non-OO</td>
<td>Ada 83</td>
</tr>
</tbody>
</table>

Table 1. Summary of Projects Studied

The terms in the table above are defined as follows:

- Traditional development approach: follows a waterfall type development lifecycle, modeling with state charts and data flows, and up front requirements specifications.
- OO development approach: are generally iterative, use object/component/class interaction models, and start with use-cases or system responsibility lists.
- OO implementation: use of an object-oriented programming language, e.g., C++, Java, Ada 95.
- Non-OO implementation: use of a non-object-oriented programming language, e.g., C.
- Project precedence: see [Royce] and [Boehm].

2. PRODUCTIVITY
The organization requesting the study hoped to compare the projects through the metric of productivity. It was assumed that this metric would be useful in capturing the extent to which introducing OO technology had met all of the organization’s major goals for it.

2.1 Historical Background and Definitions
Programmer productivity is a measure of the rate a given programmer (or an average over the team) contributes to the overall system implementation effort. Typically, this is thought of as the number of source lines of code (SLOC) a programmer writes within a given time. That is,

\[
\text{Productivity} = \frac{\text{SLOC}}{\text{(team size)} \times \text{(effort)}}
\]

Productivity as a software metric has been widely used and studied for many years. There are many excellent expositions on the subject such as [Fenton] P. 407-416 and [Boehm].

2.2 Usefulness
Typically within a given project productivity is useful in evaluating the overall effort at a particular time with respect to a schedule in that it provides a means in which to estimate if a schedule can be satisfied or estimate how large a project will be. The customer for this study was particularly interested in this aspect.

2.3 Assumptions
Using productivity as a metric makes several critical assumptions that may fail when used to compare projects with different technology and development approaches.

Assumption 1: There is a meaningful definition of a single line of source code (SLOC) and there is a well-defined and consistent manner in which to enumerate them within a project.

Assumption 2: All the implementation is done in the same style or the SLOC measure can account for style differences.

Assumption 3: The same implementation language is used throughout.

To mitigate possible deviations from the above assumptions we made use of the following within the study:

1. SLOC was measured automatically via a tool (such CCCC) providing uniformity and consistency.
2. All SLOC are counted as “logical” lines as defined in Code Count, which is similar to the CCCC utility used for several projects (see http://sunset.usc.edu/CODECOUNT/index.html), thereby greatly reducing the effects of coding style differences.

3. All SLOC counts were converted to UFPs as defined in [Royce sec. 3.1.1] see also [Abran] to account for language differences. The conversion of language dependent SLOCs to UFPs is based on statistical averages across many projects and detailed analysis of the particular languages. It provides and averages the number of SLOCs per typical function point. The productivity equation will use UFPs rather than SLOC as indicated in [Fenton P.413]. The UFP/SLOC conversion rate used here is based on the following (taken from [Royce]):

<table>
<thead>
<tr>
<th>Language</th>
<th>SLOC/UFP</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>128</td>
</tr>
<tr>
<td>Ada83</td>
<td>71</td>
</tr>
<tr>
<td>C++</td>
<td>56</td>
</tr>
<tr>
<td>Ada95</td>
<td>55</td>
</tr>
<tr>
<td>Java</td>
<td>55</td>
</tr>
</tbody>
</table>

Table 2. SLOC/UFP Conversion Factors

2.4 Limitations

Although still in common use, productivity measures are understood have serious limitations. Regrettably it is still often abused in practice. For the purposes of this study, we discuss areas that are most relevant to the customers organization or issues that are not widely discussed in the literature. The first problem with using productivity as a metric for comparison is that there are a very large number of factors that greatly affect it ([Fenton P.408], [Jones]). Some of the most prevalent and widely studied are programmer experience, development process formality, task partition-ability, component pre-positioning, and communications overhead; less well-studied effects include different project types (for example, if the project is part of a product line where there is a large amount of code re-use, is the re-used code counted in the productivity measure?) Typically, under such diverse unknowns a “control” project is used to calibrate the measure. However, it is difficult to achieve a true “control” project, as there are far too many variables to control. This is true not only in the sense of having a control that meets rigorous statistical requirements, but even in the sense of having a control that allows for any meaningful baseline. In this particular study, one project (project D) was intended to be the control, but the fact that it differed from the other projects in development methodology, language used, personnel, and type of project made it very difficult to use as a control.

The second problem is simply a well-known fact from statistics; productivity represents an average, and therefore it is not terribly robust in the face of “noisy data” errors and is overly sensitive to outliers. Since the amount of data available in most project comparisons is small, noisy data is inevitable; the well-documented existence of “super-programmers” and “dead wood” (see the 80-20 rules in [Royce]) means that outliers are quite likely. Thus, for a purely statistical point of view, it would be better to use a statistic that is more robust (e.g., a median, although what to take the median of is not clear, or some other estimator).

However, the major flaw with productivity measures is that they are inherently not well-defined. It is clear that the application of the mitigation techniques described above do not necessarily produce interpretable results. That is, the same raw data, when processed into a productivity statistic using the mitigation techniques, may imply two different (and incompatible or contradictory) semantic results. For example it is well known that coding rates and language “denisty” (i.e. SLOC per UFP) are not inversely related, yet this is exactly what is assumed with a productivity measure. Say a Java based project and an Ada83 based project have the same number of UFPs developed in the same amount of time with the same number of people. This indicates on the one hand that Java project is “dense” in that there is a higher rate of UFP implementation at a lower SLOC coding rate than the Ada83 project. On the other hand, the Ada83 project must the inverse. We expect equal raw data values should produce equal productivity values that have exchangeable (i.e. trade-offs) results. This doesn’t even begin to address the complications due to code re-use, patterns, libraries, and physical programmer time and coding rate constraints. Furthermore it is difficult or impossible to “select out” how productivity is effected by non-coding development activities that are critical to project success such as analysis, design, testing, and so forth.

3. EFFICIENCY

What we would like to have is a global measure of the effectiveness of development for a particular project that is robust and comparable across many different project types, implementation approaches and technologies. The problem is analogous to the problem of measuring usability of an application as it too must deal with comparisons across diverse types (users rather that projects). The MUSiC project [Bevan] dealt directly with a number of quantitative performance measures of software usability from which we can draw upon. In particular they defined a measure of the effort required for a user to learn and operate system relative to that of an “expert” user. They called this metric “relative user efficiency”. This expert serves as an “independent baseline” to make meaningful comparisons against in the light of many complex individual factors that effect usability. As discussed previously, we face similar such complexities when attempting to compare productivity across non-homogeneous projects. In this regard, we submit the following concept of efficiency:
Definition: The efficiency of a development project is the amount of effort estimated for that project according to an independent baseline relative to the actual effort expended.

In the above definition the independent baseline serves as the “expert” to measurements against. If the actual effort is less than estimated then this implies a manner of “schedule compression” with respect to a baseline. This is generally regarded as positive within software development, as the same amount is accomplished in less time without loss of quality or coverage. This is intuitively what we mean by “efficiency” which has a number of obvious advantages and limitations. Assuming for the moment that we have a meaningful and adequate independent baseline, efficiency has many of the qualities we desire and avoids many of the undesirable aspects of productivity. The assumption of the existence of an independent baseline valid for each project removes the need for a control project. Efficiency is by definition relative (a ratio), not an average, and thus avoids the undesirable statistical properties of averages. And, finally, efficiency does not require any mitigation strategies, so that for a given independent baseline it is only possible to compute one efficiency value for a given effort, so efficiency is well-defined. A notable limitation of efficiency is that clearly (as with productivity) there are physical constraints as to the maximum efficiency that can be attained or even measured in a well-defined way.

Note that all of the features of efficiency rely on the single, very large assumption that there exists an independent baseline for all projects. Fortunately there is a very large and well studied field within software engineering for producing exactly such a baseline - project effort estimation (see [Royce] and [Boehm] for examples). We chose COCOMOII (see [Royce A2.3] for a through justification of the use of COCOMOII) as the particular project effort estimator to use for all projects. This choice allows us to take advantage of all of the software project development expertise and data stored in COCOMOII to normalize predicted efforts. The COCOMOII model accounts for a large variety of project types, sizes, languages, and methodologies thereby simplifying the computation of our project metrics in this study.

3.1 Approach and Justification: Reversing COCOMO
The details of our approach are as follows. COCOMOII has a number of parameters that effectively tailor it to a particular project. Since we already know the actual size and effort of each project under consideration, the approach is to “run COCOMOII backwards”, having COCOMOII estimate the effort needed to complete a project of the same size and type. The resulting estimate reflects a “typical” project using traditional (i.e. “waterfall”) development approaches. Here typical means that the estimate is relative to the hundreds of projects COCOMOII is calibrated against. The COCOMOII model takes into account many complex factors and effects such as differing project types, tool use, and communications overhead. Given a size estimate, COCOMOII will produce a reasonably accurate estimate of the effort (upwards of 85%) needed to complete the project; this statistic reassures us that COCOMOII is indeed a credible baseline.

The notion of efficiency can now be meaningfully defined as the magnitude of how much effort a typical project of that size and type is estimated to need (as estimated by COCOMOII) relative to the actual effort expended on such a project. This necessarily must target a specific part of a development effort such as “overall schedule” or “programming effort”, as we must consider exactly what it is we want an efficiency measure of. Thus, there are several measure of efficiency we are interested in, but the general measure is:

\[ Z \text{ Efficiency} = \frac{\text{estimated effort for } Z}{\text{actual effort expended on } Z} \]

Where \( Z \) can be Schedule, Programming, Plans and Requirements, Design, or Integration and Test. These correspond to the traditional waterfall phases and are more fully detailed in the COCOMOII model. (See Royce A2.3).

For example, Schedule is the total development time in person months, and Programming includes coding, detailed design, and unit testing.

4. RESULTS
4.1 The Data
The table 3 summarizes the type of data collected for each project. This table includes raw data as well as productivity and efficiency metrics.

<table>
<thead>
<tr>
<th>Collected Data</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total SLOC</td>
<td>total size in SLOC</td>
</tr>
<tr>
<td>Configured SLOC</td>
<td>current baseline SLOC</td>
</tr>
<tr>
<td>Critical defects</td>
<td>number of SCOs type 0</td>
</tr>
<tr>
<td>Normal defects</td>
<td>number of SCOs type 1</td>
</tr>
<tr>
<td>Improvements</td>
<td>number of SCOs type 2</td>
</tr>
<tr>
<td>New features</td>
<td>number of SCOs type 3</td>
</tr>
<tr>
<td>Number of Software Change Orders (SCOs)</td>
<td>cumulative SCOs</td>
</tr>
<tr>
<td>Open rework (breakage)</td>
<td>cumulative broken SLOC due to SCO type 1, 2</td>
</tr>
<tr>
<td>Closed rework (fixes)</td>
<td>cumulative fixed SLOC</td>
</tr>
<tr>
<td>Rework effort</td>
<td>cumulative effort expended fixing SCOs type 0, 1, 2</td>
</tr>
<tr>
<td>Usage time</td>
<td>hours that a given baseline has been operating under realistic usage scenarios</td>
</tr>
</tbody>
</table>

Table 3. Collected Data
### Table 4. Summary of Data

<table>
<thead>
<tr>
<th>Project</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Language</td>
<td>Ada95</td>
<td>C++</td>
<td>Java</td>
<td>C</td>
<td>Ada83</td>
</tr>
<tr>
<td>SLOC</td>
<td>42K</td>
<td>34.5K</td>
<td>11K</td>
<td>1.7K</td>
<td>28K</td>
</tr>
<tr>
<td>SLOCs reused</td>
<td>2K</td>
<td>0</td>
<td>0</td>
<td>333</td>
<td>25K</td>
</tr>
<tr>
<td>Effort (months)</td>
<td>19</td>
<td>20</td>
<td>16</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>Team size</td>
<td>11.0</td>
<td>10.0</td>
<td>6.0</td>
<td>7.5</td>
<td></td>
</tr>
<tr>
<td>Programming %</td>
<td>0.43</td>
<td>0.40</td>
<td>0.11</td>
<td>0.60</td>
<td></td>
</tr>
<tr>
<td>Programming effort (months)</td>
<td>8.17</td>
<td>8.00</td>
<td>1.78</td>
<td>5.40</td>
<td></td>
</tr>
<tr>
<td>Plans and requirements %</td>
<td>0.12</td>
<td>0.30</td>
<td>0.44</td>
<td>0.20</td>
<td></td>
</tr>
<tr>
<td>Plans and requirements effort (months)</td>
<td>2.28</td>
<td>6.00</td>
<td>7.11</td>
<td>1.80</td>
<td></td>
</tr>
<tr>
<td>Design %</td>
<td>0.12</td>
<td>0.20</td>
<td>0.22</td>
<td>0.20</td>
<td></td>
</tr>
<tr>
<td>Design effort (months)</td>
<td>2.19</td>
<td>4.00</td>
<td>3.56</td>
<td>1.80</td>
<td></td>
</tr>
<tr>
<td>Integration and test %</td>
<td>0.33</td>
<td>0.10</td>
<td>0.22</td>
<td>Not reported</td>
<td></td>
</tr>
<tr>
<td>Integration and test effort (months)</td>
<td>6.27</td>
<td>2.00</td>
<td>3.56</td>
<td>Not reported</td>
<td></td>
</tr>
<tr>
<td>SLOC/effort (lines/month)</td>
<td>4896</td>
<td>4313</td>
<td>6937</td>
<td>556</td>
<td></td>
</tr>
<tr>
<td>UFPs</td>
<td>727</td>
<td>616</td>
<td>200</td>
<td>10</td>
<td>38</td>
</tr>
<tr>
<td>UFP/effort (per month)</td>
<td>89.02</td>
<td>77.01</td>
<td>118.4</td>
<td>7.12</td>
<td></td>
</tr>
<tr>
<td>UFP/effort (per month/person)</td>
<td>8.09</td>
<td>7.7</td>
<td>19.73</td>
<td>0.95</td>
<td></td>
</tr>
<tr>
<td>COCOMOII effort (months)</td>
<td>136.7</td>
<td>350.4</td>
<td>105.4</td>
<td>49.8</td>
<td></td>
</tr>
<tr>
<td>COCOMOII effort/ team size (months/person)</td>
<td>12.43</td>
<td>35.04</td>
<td>17.57</td>
<td>6.64</td>
<td></td>
</tr>
<tr>
<td>COCOMOII team size</td>
<td>10.90</td>
<td>13.30</td>
<td>6.40</td>
<td>4.00</td>
<td></td>
</tr>
<tr>
<td>Schedule efficiency</td>
<td>0.65</td>
<td>1.75</td>
<td>1.10</td>
<td>0.74</td>
<td></td>
</tr>
<tr>
<td>Programming efficiency</td>
<td>0.88</td>
<td>2.54</td>
<td>6.34</td>
<td>0.72</td>
<td></td>
</tr>
<tr>
<td>Plans and requirements efficiency</td>
<td>0.38</td>
<td>0.41</td>
<td>0.17</td>
<td>2.58</td>
<td></td>
</tr>
<tr>
<td>Design efficiency</td>
<td>0.97</td>
<td>1.49</td>
<td>13.44</td>
<td>0.63</td>
<td></td>
</tr>
<tr>
<td>Integration and test efficiency</td>
<td>0.50</td>
<td>4.40</td>
<td>1.12</td>
<td>Not available</td>
<td></td>
</tr>
</tbody>
</table>

#### 4.2 Methods of Collection

Data was collected from documentation and code and metrics already in place at HSC. The types of data collected are listed in table 3 and processed as indicated in table 4.

Through extensive interviews with the developers and data analysis we were able to compile an accurate depiction of the projects (development type, project type, etc.) under consideration for use in our models. Each piece of critical information was collected by asking the same question in several different ways so as to cross-validate and assure data integrity. Every attempt was made to validate interview information with evidence gathered from project data. This information was used in the models, in particular in setting the COCOMOII model parameters as well as confirm (or cast doubt on) the conclusions drawn from the data. An example COCOMOII output is provided below:

![Screenshot of COCOMOII output for project C](image)

#### 4.3 Data, comparisons, and analysis

The analysis of the model results will be conducted by comparing and contrasting the data generated from the measures with respect to the project characteristics listed in table 1. We will often implicitly make references to these characteristics rather than repeatedly re-stating them.

##### 4.3.1 Productivity

From table 4, we find that in productivity (expressed as UFPs per month per person) Project C comes out way ahead with nearly 20 UFPs generated per month per person on average. At less than half this rate are Project B and Project A. Finally, at less than an eighth of the productivity of Project B and Project A is Project D. This measure does not fully support the stipulation that OO is overall more productive than non-OO, for several reasons. First, our measure of productivity only shows the rate of new UFPs generated. This has a negative impact on Project D since by its nature as a product line extension, it very sensibly re-uses a majority of code from previous products and this is not factored into the measure (although surely effects it). The problem lies mainly in that re-use of code slows down the overall programming effort in ways that are not easily measured as implementers must spend time integrating new code with old code. However, implementers cannot simply be given credit for the re-used code as it did not originate with them (if we did then suddenly implementers could be “hyper-productive” and greatly exceed the limitation on the rate a human can generate code).

Another consideration is that pure OO languages, such as Java, by design produce more dense functionality through use of extensive standard object libraries, which are tightly coupled to the language (e.g. Java AWT). This is a hidden
form of re-use for which implementers gets productivity credit for when considering UFPs as a project size metric. The OO languages C++ and Ada95 have less rich standard object libraries and are not oriented towards re-use to the same degree as Java. Hence, implementers tend to recreate many basic data structures, such as hash tables and trees. If they use COTS object library packages, such as Rogue Wave (as Project B did), implementers must spend time understanding and integrating them within their own structures and the productivity measure does not account for this. This explains the discrepancy in productivity between Project C and Projects B and A. Although such productivity gains from Java are impressive, it should be noted that there is a tradeoff for getting it. The tradeoffs are usually recognized in the need to satisfy project constraints such as customer mandates (e.g. Government projects must use Ada) and time/space performance (e.g. system must use less than 256K RAM, system must process commands within 400ms). For example, C++ executables are smaller and perform faster than their Java equivalents, as there is no runtime system overhead.

Finally, we see from Table 2 the Project A productivity is slightly better than that of Project B. Given that Project A is an advanced prototype, we would expect the programmer productivity to be quite high as there are less critically specified functionality requirements, and experimentation is encouraged. Ada95 and C++ object libraries are approximately equivalent in terms of richness (with Ada95 actually slightly ahead, particularly in terms of pre-set data structures and so forth – see appendix for a comparison). Furthermore, re-use is likely not a factor as Project A re-used approximately 2000 SLOC and Project B made moderate use of the Rogue Wave libraries, so where Project A lost SLOC due to re-use, Project B had their SLOC count impacted due to library use. Both projects had to contend with integration issues. This leaves development approach. From table 2, Project A used a most traditional development approach whereas Project B incorporated some OO-development techniques (in terms of system analysis and design). Hence Project B was likely more productive through the use of such techniques, and Project A was less productive to the point that the expected greater productivity for a prototype effort was diminished to the productivity level of a commercial product effort.

4.3.2 Efficiency
All measures here are magnitude of effort (in person months) factors that compare a given project to a COCOMOII estimate for a project with similar characteristics. The specific efforts, such as programming and design are compared to a traditional waterfall style (non-OO) development lifecycle. In general, an efficiency value of less than one indicates that the project in question took greater effort than estimated by COCOMOII. Values of greater than one indicate that the project took less effort than estimated by COCOMOII.

Programming Efficiency
Project C, as indicated in table 4, did more than six times better in overall programming effort than a traditional non-OO project of the same type. Project B did over two times better. Project A did about as well as was estimated by COCOMOII, which is to be expected given that it is a prototype project and that the size was determined by the schedule (i.e. 19 months) more than by completed functionality. It is interesting to note that the estimate is nearly within COCOMOII accuracy limits (85%), which of course inspires greater confidence in the model. It will be interesting to re-measure the programming efficiency when Project A concludes. Project D performed at a level that was slightly less than estimated. Given the 85% accuracy of COCOMOII, it is significant that it is off by nearly 1/3 of the estimated effort (although we have no evidence to explain why this might be so).

Schedule Efficiency
This measure is difficult to draw conclusions from as it measures how the project’s overall effort compares to an estimate. It provides little insight in the particulars of the effort itself. The winner here is Project B; they beat the estimate by more than 50%. With a hard deadline, which was likely aggressive from the start, this is understandable. Following respectably is Project C, doing almost exactly as estimated. This may in part be due to the fact that the project lead used COCOMOII to estimate the development effort! Project D came in third, under par by a statistically significant amount. In last place, not unexpectedly, is Project A. Schedule efficiency is not a great concern for a prototype effort.

Plans and Requirements Efficiency
Project D spent more than 2.5 times less effort on plans and requirements than estimated. This is a point of validation for our model in that we fully expect a project that expands upon a previous product (a product line) will begin with already well-established plans and requirements. Quite naturally, we would expect Project A to spend an extended amount of time in plans and requirements because a part of prototyping is to explore and define the concept of operations and system issues. Consequently, Project A was well over estimates. As Project B and Project C both employed OO development approaches, which are by design “front-end heavy” in that a greater amount of time is spent in analysis, lifecycle design (mostly iterative schemes, which require substantially more planning and control), and requirements negotiation and re-negotiation (to accommodate changes over the iterations), we expect these projects to run over estimates in this area.

Design Efficiency
From table 2, Project C beat estimates by a factor of 13. This is vastly greater than the next best, Project B, which
was about 1.5 times better than estimated. This overwhelming evidence supports the conclusion that OO development approaches coupled with “pure” OO languages enable an efficient, productive design effort. From our observations of Project C, their design effort followed easily from their analysis, which in turn led to a straightforward implementation. Very few critical design problems were encountered. However, Project B had difficulty in moving from the analysis to design and then from the design to implementation. Generally, it was felt that they were lacking the skills necessary to perform this part of the development process in a mature OO fashion. This is in part due to inadequate training and experience. As a result, Project B abandoned a significant portion (perhaps half) of the OO development approach, replacing it with more traditional techniques. This may explain some of the significant differences between Project B and Project C design efficiency. Notwithstanding, Project B beat estimates by a respectable 50%. Project D had difficulty integrating their new functions with the previous product architecture and thus were far above estimates. This may be due to the lack of an effective architecture description to work from (as compared to Project C that explicitly defined and made extensive use of a software architecture).

Integration and Test Efficiency

It is clear that this measure has some relation to OO issues, however it is affected by many factors outside the scope of this study. As Project D did not provide test effort data, it would be difficult to draw conclusions regarding comparisons of OO vs. non-OO from the results at present. We will simply note that the two projects that employed OO development approaches beat estimates. This result is in alignment with industry perceptions that OO development practices more readily lead to efficient integration and testing as the objects defined through the process more closely relate to the actual domain of the system rather than pure software structures. This gain appears to outweigh the complications of setting up test harnesses for objects which are generally less deterministic, and thus more complex to test.

4.3.3 Overall Results

A non-parametric assessment of the overall results of the study is shown in the following two tables. These tables include the productivity results as another statistic comparable with the efficiency statistic for a single part of the development effort. Using the information in table 4, each project was ranked relative to the others. A ranking of ‘1’ indicates “top” relative to the others, and a ranking of ‘4’ indicates “bottom”. The table below shows what the rankings are for each measure.

<table>
<thead>
<tr>
<th></th>
<th>Rank 1</th>
<th>Rank 2</th>
<th>Rank 3</th>
<th>Rank 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>UFP/effort/person</td>
<td>C</td>
<td>A</td>
<td>B</td>
<td>D</td>
</tr>
<tr>
<td>Schedule efficiency</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>A</td>
</tr>
<tr>
<td>Programming efficiency</td>
<td>C</td>
<td>B</td>
<td>A</td>
<td>D</td>
</tr>
<tr>
<td>Plans and requirements efficiency</td>
<td>D</td>
<td>B</td>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>Design efficiency</td>
<td>C</td>
<td>B</td>
<td>A</td>
<td>D</td>
</tr>
<tr>
<td>Integration and test efficiency</td>
<td>B</td>
<td>C</td>
<td>A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 5. Project Rankings (Non-parametric)

The table below shows for each project the number of times it placed in each rank. The last row shows the un-weighted average of all a project’s rankings where no emphasis is placed on how far above or below a particular ranking is relative to the other ranks. For example, C was approximately 13 times more efficient in the design phase than B, yet C will be weighed only 1 point less as B was the second most design effort efficient. The purpose of this is to highlight that the smaller the value, the more highly ranked that project is overall.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rank 1</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Rank 2</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Rank 3</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Rank 4</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Average rank</td>
<td>3.000</td>
<td>1.183</td>
<td>1.183</td>
<td>3.200</td>
</tr>
</tbody>
</table>

Table 6. Overall Project Ranking (Non-parametric)

The single conclusion one can draw, solely on the basis of Table 6, is that use of an OO development approach (Projects B and C) produced significantly better results in more metrics than use of a traditional development approach (Projects A and D). The presence of an OO implementation technology in Project A did not appear to significantly improve its results over the non-OO technology of Project D.

5. CONCLUSIONS

5.1 Conclusions about Study Approach

Surveying the conclusions drawn using productivity alone, efficiency, and productivity and efficiency together, we see that using productivity alone would have resulted in some
misleading results. According to productivity alone, Project A was a close second to Project C, with Project B a distant third. However, when the projects are compared using efficiency, Project B and Project C achieve near-parity, with Project A a distant third. An identical conclusion is reached using a non-parametric combination of productivity and efficiencies for various project phases. The latter conclusion matches much better with the perceptions of people interviewed at the organization. The high productivity of Project A is most likely due to its being a prototype, with the concomitant reduction in pressure to produce solid, robust code and increase in pressure to produce something that works.

Thus, based on this study, using productivity alone as a metric would be misleading. Using efficiency as a metric instead produces results that are not affected by differences in project parameters and that correspond better to subjective perceptions. Using efficiency and productivity together allows the robustness of efficiency to ameliorate the sensitivity of productivity, but since this approach, in this study at least, produced the same results as considering efficiency alone, it is not clear there is anything to be gained by using productivity in addition to efficiency.

There have been many related works on evaluating project performance in higher level languages such as Klepper and Bock who conclude that productivity is significantly higher with 4GLs than with 3GLs [Klepper].

5.2 Conclusions about Object-Oriented Technology
The use of efficiency metrics in addition to the productivity metric allows us to reach the following conclusions about object-oriented technology from the data in this study.

1. Use of an OO development approach, coupled at least in part with OO implementation technology, significantly improves overall project efficiency.

2. Use of an OO implementation technology without an OO development approach does not appear to improve overall project efficiency.

3. OO development approaches are less efficient than traditional approaches in the planning/requirements phase, but they more than make up for this with increased efficiency in the design, programming, and test phases.

Conclusion (1) is in alignment with industry perceptions as indicated in several related studies on use of OO languages and methodologies such as one undertaken by NASA. Conclusion (3) agrees with common “folklore” about OO development.

6. ACKNOWLEDGMENTS
The authors thank Dr. Barry Boehm for many stimulating and fruitful conversations on software development