Object-Oriented Productivity Metrics

John L. Connell, Sterling Software, Inc.
Nancy Eller, Sterling Software, Inc.

Software productivity metrics are useful for sizing and costing proposed software and for measuring development productivity. Estimating and measuring source lines of code has proven to be a bad idea because it encourages writing more lines of code and using lower level languages. Function Point Analysis, as espoused by Dreger [1], is an improved software metric system but it is not compatible with newer rapid prototyping and object-oriented approaches to software development. A process is presented here for counting object-oriented effort points, based on a preliminary object-oriented analysis. It is proposed that this approach is compatible with object-oriented analysis, design, programming, and rapid prototyping. Statistics gathered on actual projects are presented to validate the approach.

Problems With Existing Productivity Metrics

The software engineering field has been searching for decades for a way to definitively size software. The purpose of precise sizing of software is to provide a means for determining, exactly, the answers to the following questions:

- How much will it cost to develop a proposed new software application?
- How well will the software fit within the target computer's available storage and memory?
- How are individual programmers performing in terms of units of software produced per unit of project time elapsed?
- When will the work in progress be ready for use?

Good metrics would make new application cost/benefit analyses lead to correct decisions more frequently. There would be less processing of expensive change requests during system development to reconfigure baseline hardware architectures as true software size becomes apparent. Total Quality Management (TQM) programs could make use of better process improvement metrics. Project management, with good metrics, would be more effective.

The most prevalently used metrics are heuristics based on experience; experienced software developers are sometimes pretty good at guessing the answers to the four preceding questions. The second most widely used metric is the Source Line of Code (SLOC); developers guess how many SLOCs will be in the final product and then multiply the error of this estimate by errors in the estimate of how many lines of code can be produced in one person hour. Less widely used is Function Point Analysis [1] — a measure of software size that is language independent and has a more precise definition than the SLOC.

The argument can be made that estimates based on experience are not really metrics at all. A counter to this is that most existing software productivity metrics are also fairly unscientific, but are disguised to appear scientific. In fact, there is little conclusive evidence that use of any of the current metrics will produce any more accurate results than experience based estimates. Following are the specific disadvantages of using SLOCs or FPA metrics for your next software development project.

Source Lines of Code (SLOC) Metrics Discourage Productivity

Software developers are paid to develop software applications, not to write lines of code. Suppose that two developers are working on the same application in different organizations. One is using a third generation language (3GL), such as FORTRAN, and another using a fourth generation language (4GL), such as an SQL dialect. The 4GL developer will have fewer lines of code than the 3GL developer (by one order of magnitude) for the same amount of functionality. This is documented by many studies and is well presented in Dreger's Function Point Analysis [1].

This fact is complicated by several arguments. Some argue that 4GL code is not as efficient or robust as 3GL. Others say that, although the 4GL programmer writes fewer lines of code, s/he is done sooner, so the metrics work out satisfactorily. Finally, some advocate collecting SLOC metrics that are language specific, thus compensating for differences.

It is not necessary to argue over the goodness of 4GL versus 3GL. Do you believe that third generation languages were a significant productivity improvement over second generation languages such as Assembler and Autocoder? Wasn't Assembler a big improvement over raw machine language? Isn't there a noticeable trend here in terms of number of lines of code needed to produce a given amount of functionality? The history of programming environment development has always been to create new languages that increase programmer productivity by requiring *fewer lines of code* to produce a given amount of functionality. If you use a metrics system that rewards programmers based on how many lines of code they write per hour, you will be implementing a bad incentive (rewarding productivity degradation).

While it is true that 4GL programmers will finish coding a given amount of functionality sooner than their 3GL programmer counterparts, this does not mean that they will finish a similar development project significantly faster. The reason this is true is that they will not write requirements and design specifications or test plans any faster. Suppose that coding is 20% of total development effort. Then, suppose that a 4GL programmer can do the coding in 10% the time required for a 3GL programmer. This will result in a total 18% productivity improvement — not very dramatic. On the other hand, the 4GL programmer will probably write 10% of the number of lines of code needed by the 3GL programmer. Therefore, a SLOC metrics system will show that a 4GL programmer is very unproductive, compared to a 3GL programmer — unless the organization has been collecting 4GL metrics previously and never attempts to compare productivity between environments. The bottom line is that, even if you collect 4GL metrics, there are always vendors out there developing new higher productivity languages and development systems. Consider how badly SLOC systems fail in visual programming environments where icons, menu choices, and drawing tools take the place of text-based syntax.

Function Point Analysis (FPA): Improved, but Short of Ideal

FPA requires estimating how many Inputs, Outputs, Queries, Files, and Interfaces a proposed system will contain. An Input is basically a program that is mostly about capturing data. An Output is a program that mostly issues data. Queries are combinations since they require a fair amount of input and always produce output. Files are static data storage locations, including database tables. Interfaces are external entities: other applications, users, and devices.

In his book, Dreger presents a scheme for identifying and classifying each of these elements as to their complexity and assigning each element a number of Function Points depending on its complexity. He, of course, recommends collecting your own data to use in converting Function Points to hours, but does give some examples so that you can see that a good starting place, if you have not been collecting data, might be about 20 hours per Function Point in a typical 3GL environment. The beauty of this approach is that it is language independent; you will develop the same number of Function Points regardless of which language you choose.

FPA rewards productivity realized from using more advanced development tools by producing statistics that show more Function Points being created in fewer hours. It is true you must still estimate how many Inputs, Outputs, Queries, Files, and Interfaces a proposed system will contain, and then multiply the error of that estimate by the error in conversion to hours, but conventional structured analysis and design (SA/SD) methodologies will produce specifications from which these estimates can be rather precisely extracted. For this reason, FPA estimates based on fairly complete structured specifications, have proven to be much more accurate than the average SLOC based estimate.

Unfortunately, the very reason for the success of Function Point Analysis is also its major weakness. Since structured specifications must be fairly complete before a meaningful estimate can be generated, FPA does not work well when a rapid prototyping approach such as that recommended in Structured Rapid Prototyping [2] is used. The rapid prototyper needs a reliable estimate in order to plan for when prototype iteration must be complete, but does not want to completely pre-specify requirements before they have been discovered through prototyping.

Also, it is not clear that FPA is compatible with modern Object–Oriented development techniques. In the Object–Oriented paradigm, the concept of *Program* (Inputs, Outputs and Queries) is obsolete, as is the concept of *File*. These archaic concepts are replaced with the new term *Object*. An Object encapsulates both data (as Object attributes), and methods, or services (what the object does). The best of the new Object–Oriented Analysis methodologies [3] do not provide a means of developing specifications from which Function Points could be easily derived.

Introducing Object-Oriented Productivity Metrics (OOPM)

The following material presents something new — Object–Oriented Productivity Metrics (OOPM). This is an approach that, similar to FPA, is language independent, but is also very compatible with Object-Oriented Analysis and rapid prototyping. The developer using OOPM will be counting Object–Oriented Effort Points (OOEPs) instead of Function Points. An OOEP is intuitively straightforward, it is a unit of measure used to determine how long it takes to develop an Object Class.

In order to determine how long it will take to develop a new Object Class, you will need to specify how many attributes the Object will have, how many services or methods of various types it will contain, what external entities it will get data from, and to what external entities it will deliver data. Objects will be simple, average, or complex depending on how many attributes they have. A simple Object might be defined as one with fewer than seven attributes. An average Object would perhaps have seven to 14 attributes, and complex Objects would have greater than 14 attributes. This classification is similar to and based on Dreger's system [1] for classifying the complexity of files, except that files usually have significantly more fields than Objects have attributes [3].

Experience on actual projects shows that most of the time spent developing the data structure of an Object Class is spent in requirements analysis and design. Actual development of data structure instantiation scripts takes almost no time once the design details have been specified. This is why it is important to classify Objects as to their data complexity. Then give simple Objects 3 OOEPs, average Objects 5 OOEPs, and complex Objects 8 OOEPs — an assignment similar to FPA File classification.

Services, or methods, should be counted separately, as each one will contribute significantly to effort required to implement an Object Class. Here you could classify Services into four categories, consistent with both Dreger [1] and Coad/Yourdon [3]: Add/Modify/Delete Services, System Screen (Menus, Helps) Services, Output Services, and Computationally Intensive Services. Some of these categories would be further classified as simple, average, or complex, depending primarily on how much data is processed. Add/Modify/Delete Services could get 3,4, or 6 points, and Output Services 4, 5, or 7 points. System Screen Services, such as menus and help screens, do not normally process data, so they would always get the same number of points, say 4. One would classify services as computationally intensive so that they will always get a high number of points, regardless of amounts of data processed, say 8.

Finally, count the external entities the proposed application will have to interface with, similar to the Interface count in FPA. Classify external entities as simple, average, or complex depending on how many Object Classes the external will interface with. Give less than two Classes 7 points, two to five Classes 10 points, and more than five Classes 15 points. Figure 1 summarizes the system for classifying Object–Oriented Effort Points.

	Simple	Average	Complex
Object Class	< 7 Attributes	7 - 14 Attributes	> 14 Attributes
	3 OOEPs	5 OOEPs	8 OOEPs
Service:			
Add/Modify/ Delete	3 OOEPs	4 OOEPs	6 OOEPs
Output	4 OOEPs	5 OOEPs	7 OOEPs
Sys. Screen	4 OOEPs	N/A	N/A
Comp. Intense	N/A	N/A	8 OOEPs
External Entity	< 2 Classes	2 - 5 Classes	> 5 Classes
	7 OOEPs	10 OOEPs	15 OOEPs

Figure 1, Object-Oriented Effort Point Classification System

Using Object-Oriented Analysis to Count OOEPs

What the OOPM estimator will need to do at the start of a new software development project is to prepare a preliminary requirements specification, using graphic models depicting the Object Classes, Services, and External Entity Interfaces the new application will need. This can be very incomplete, for a rapid prototyping project, and an expansion factor applied to the initial estimate to obtain a total development effort estimate.

For rapid prototyping, OOPM will work better than an FPA approach based on SA/SD because there will be a direct correlation between expansion of the OOA specification and expansion of the prototype — more Object Classes and Services will be added with each prototype iteration. With SA/SD, expansion of a prototype resulted in more primitive processes to specify in the Structured Analysis, requiring re-partitioning and balancing of all higher levels in the Dataflow Diagram hierarchy. This is what is recommended in *Structured Rapid Prototyping* [2] — possible, but somewhat awkward.

Figure 2 shows an example of an Object–Oriented information model diagram for a Harbor Information System. This application will contain ten simple Object Classes for an OOEP count of 30. It will contain 6 simple Add/Modify/Delete Services for 18 OOEPs, 4 simple Output Services for 16 points and, let's say, one menu and one help screen, for 8 more points. This gives a total of 72 OOEPs.

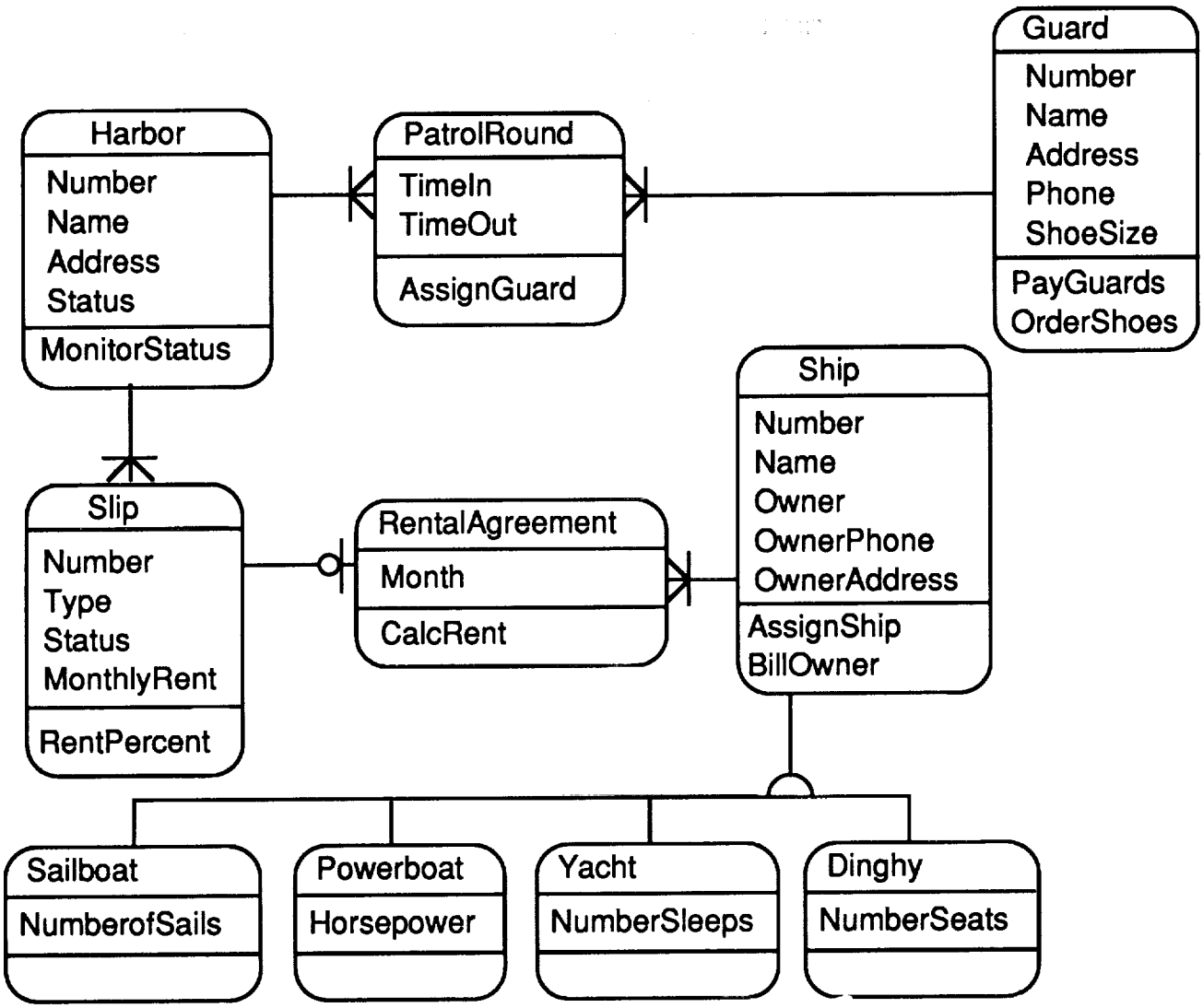

Figure 2, Harbor Information System Object-Oriented Information Model

But, what about External Entity Interfaces? In Structured Rapid Prototyping, [2] two dataflow diagrams, The Context Diagram and the Essential Functions Diagram, are recommended as part of the preliminary rapid requirements analysis to help work out the external data interfaces. How to model these interfaces seems to be missing from the Coad/Yourdon OOA approach. On the other hand, it does not seem appropriate any longer to advocate that an Object–Oriented developer begin by drawing Dataflow Diagrams. Instead, consider an Object Class Source/Sink Diagram as shown in Figure 3.

In Figure 3 there are 5 external entities. Harbor Manager and Employee are of average complexity and get 10 points each. Acme Placement Agency Database, Ship Owner, and Payless Shoes are simple and get 7 points each. The total External Interface OOEPs are 41. This brings the grand total for the Harbor Information System to 113 OOEPs.

Hours per OOEP

In Function Point Analysis [1], Dreger states that the norm for hours required to code one Function Point, in a third generation language, such as COBOL is about 20. Examples given of languages in this range are Pascal, JOVIAL, FORTRAN, COBOL, ALGOL and C (C is identified as the least

productive language in this list). Object–Oriented software development projects will not use one of these languages; they will hopefully use an Object–Oriented Programming Language (OOPL) such as Smalltalk or C++. Dreger says that such languages will require about one-fifth as many lines of code per Function Point as a third generation language. Presumably this is due to language extensibility and component reuse through inheritance. This puts an OOPL in almost the same category as a 4GL — about 4 hours per Function Point.

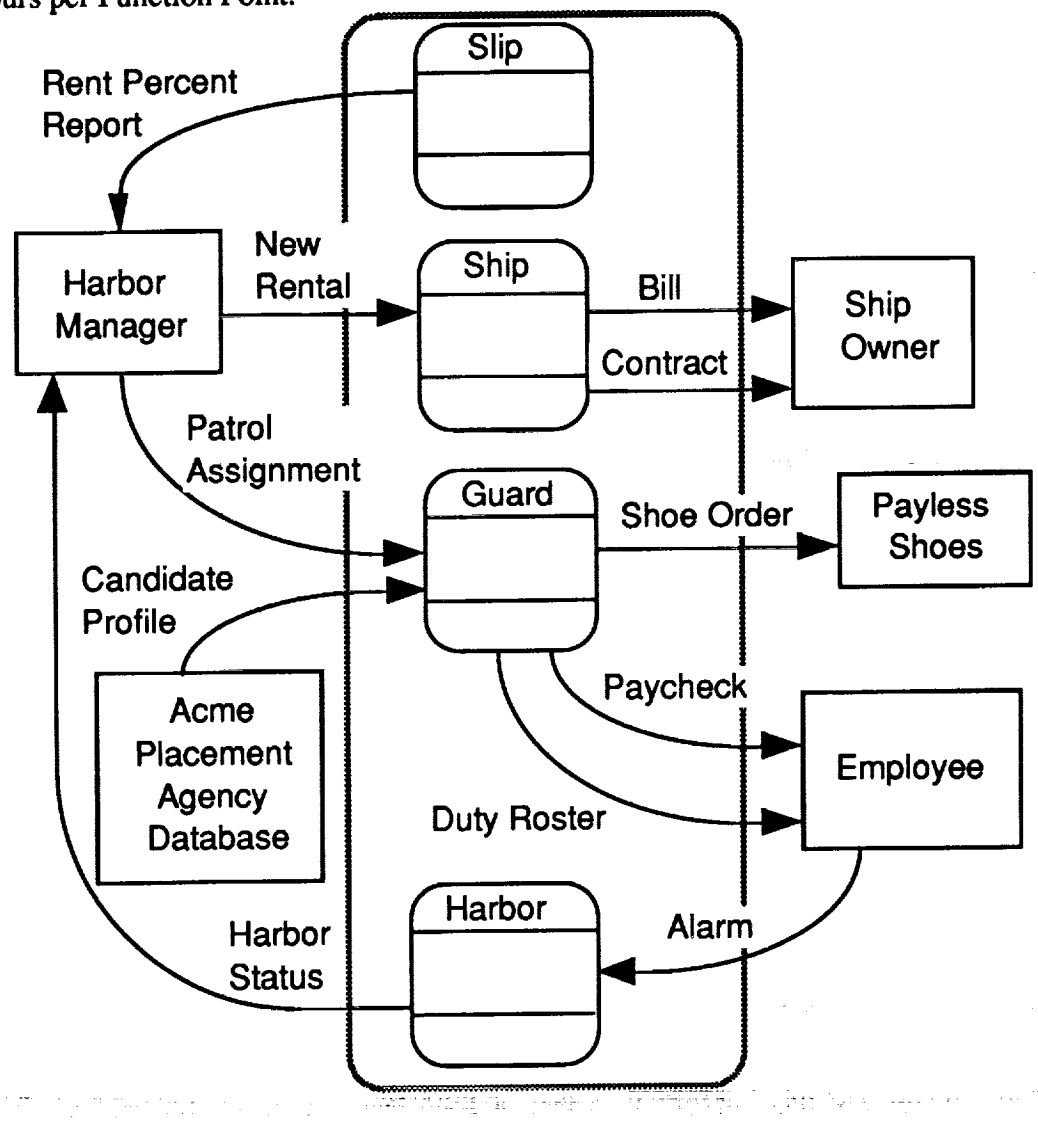

Figure 3, Harbor Information System Source/Sink Diagram

From this analysis, it can be determined that, using OOPM, a simple Add/Modify/Delete Service (data entry screen) should take an average of about 12 hours of effort (3 OOEPs times 4 hours per OOEP). The conversion of 4 hours per OOEP can be applied to all 113 OOEPs for The Harbor Information System, which then should take about 452 person hours to develop. This estimate should include OOA, OOD, programming with an OOPL, test and all documentation. Keep in mind that documentation effort is one of those aspects of development that will vary widely according to various organizations' documentation standards and that all such estimates will need to be adjusted to specific project conditions, such as developer skills and experience.

Benefits of Using Object-Oriented Productivity Metrics

The metrics presented here will be compatible with, and encourage the use of, very advanced development environments such as those used in Object–Oriented Rapid Prototyping [4]. Developers will be rewarded according to their proficiency in rapid development of new Objects (something of more perceived value to application users than lines of code or Function Points). Software reuse, an important modern productivity enhancing technique is fostered by Object–Oriented techniques; it will be rewarded by Object–Oriented Productivity Metrics. OOPM will generate metrics that are tightly coupled to Objects. When those Objects are reused, their OOEPs are encapsulated and will move with the Objects to the new environment where they become available for processor sizing and other purposes.

Whether or not rapid prototyping is used, OOPM provides a way to get reliable sizing and effort estimates very early in the project — before requirements have been finalized — based on preliminary Object–Oriented graphics models. When using a rapid prototyping approach, OOPM will provide an estimate of how long it will take to develop the initial prototype and an expansion ratio can be used to estimate how long it will take to develop the entire application. The same expansion ratio can also be applied to estimate how many Objects will exist in the final application.

Research Evidence Supporting Feasibility of OOPM

Dreger has been cited as a source of valuable software metric research [1] and Coad/Yourdon OOA [3] has been cited as a good approach to Object–Oriented requirements analysis. The basic philosophy of FPA is that software metrics should be based on things the user wants to pay for, not lines of code. The basic philosophy of OOA is that users are mostly interested in Objects within the domain of their field of interest. It seems logical, therefore to merge these two philosophies to create Object–Oriented Productivity Metrics.

Attempts at defining what a software Object is, to the universal satisfaction of all software engineers, have mostly met with failure. Undaunted, we will propose one more definition: an Object is a thing of interest to a software application user — defined, for the purposes of that application, by the attributes of the Object that are of interest to the user.

The reason this working definition of an Object is useful is that it allows for comparisons that may tie OOPM back to FPA for purposes of supporting feasibility. An Object has attributes; in terms of Information Modeling methodology, so does an Entity. In implementation, an entity often becomes a database table. Thus, Object attributes are very similar to database attributes which are, in turn, very similar to fields in files (except that database tables are usually normalized and have fewer attributes than files have fields). Therefore, FPA File counts can be converted to OOPM Object counts.

For FPA, SA/SD methodologies are used to provide precise counts of how many files will be created for a new application. For OOPM, you can use OOA models to accurately determine how many Objects of what level of complexity will be created. Determining Object complexity from preliminary OOA models may, in fact, be easier than determining File complexity from preliminary Structured Analysis. It is difficult, during the early stages of requirements analysis, to determine how many fields will be in a proposed file. It is not so difficult, in the early stages of OOA, to determine how many attributes a proposed Object will have.

One of the authors is the manager of several projects at NASA Ames Research Center that have been collecting metrics on application development in an environment using Sybase development tools. Sybase is a relational database management system with a bundled collection of development tools including a 4GL, reportwriters, and a forms generator. On these projects, entities are defined and modeled in a manner consistent with defining and modeling Objects in OOA (even though Sybase is not Object–Oriented).

Object-Oriented Productivity Metrics

The metrics kept are with respect to how long it takes to develop a Form and how long it takes to develop a Report. Only development with Sybase tools is tracked; very little development work on these projects uses third generation language programming. Four years of data have been averaged with the result that it can be stated that the average Form takes 24 hours to develop and the average Report 16 hours. There is a variance, depending on complexity, yielding a range of 4 to 40 (sometimes more) hours per Screen and 4 to 24 hours per Report. These metrics do not include final system documentation or full integration testing.

Some of the Screens are very complex. They include what has been described above as Add/Modify/Delete Services, System Screen Services, and Computationally Complex Services. The Reports are equivalent to the Output Services described above. Data has not been kept on the amount of requirements and design effort required to develop an Entity (Object).

The purpose of reporting these effort metrics here is that they fall well within an acceptable range of the suggested values for OOPM given above. Suppose that the average input Service is worth 4 OOEPs and the average output Service 5 OOEPs. Then suppose, using Sybase tools, an OOEP takes 4 hours to develop (this would be consistent with statistics provided by Dreger [1]. Then, developing an input Service would require an average of 16 hours and an output Service 20 hours, according to OOPM metrics. These estimates are acceptably close to the actuals of 24 hours and 16 hours, respectively, considering that Dreger says the average of 20 hours per COBOL Function Point has a range of 3 to 87 hours. To get really accurate metrics, you will have to collect your own statistics.

Summary of Findings

With the advent of rapid prototyping; Object–Oriented Analysis (OOA), Design (OOD), and programming (OOP); and, recently, Object–Oriented Rapid Prototyping (OORP) — it is time to abandon software productivity metrics based on either Source Lines of Code (SLOCs) or Function Point Analysis (FPA). Productivity metrics need to support and encourage the use of the best modern software engineering practice if the desired result is continuous process improvement.

A new approach, Object-Oriented Productivity Metrics (OOPM) has been defined here that is compatible with modern software development techniques such as Object–Oriented Rapid Prototyping. Arguments made in and supported by Coad/Yourdon's Object–Oriented Analysis [3] and Dreger's Function Point Analysis [1] provide proof of the soundness of the OOPM approach. Actual project experience, using metrics similar to those that would be used with OOPM, provides evidence indicating that OOPM based estimates would be highly accurate. Based on this research, the following recommendations can be made without hesitation:

- Use OOA, OOD, and OORP techniques for software development.
- Estimate effort using OOPM.
- Evaluate productivity using OOPM.
- Collect your own OOPM statistics and continually refine them over time.

References

1. Coad, P. & Yourdon, E. *Object-Oriented Analysis*, New York: Yourdon Press (Prentice-Hall), 1990, 1991.

2. Connell, J. & Shafer, L., *Structured Rapid Prototyping*, Englewood Cliffs, New Jersey: Yourdon Press (Prentice–Hall), 1989.

3. J. Brian Dreger, *Function Point Analysis*, Prentice Hall, Englewood Cliffs, NJ, 1989.
4. Connell, J., Gursky, D. & Shafer, L., "Object-Oriented Rapid Prototyping," Embedded Systems Programming, September, October, 1991.