Measuring the Effects of Object-Oriented Frameworks on the Software Process [1]

Simon Moser, Oscar Nierstrasz

University of Berne[2]

Abstract. A field study of over thirty projects using Object Technology has shown that the availability (or absence) of reusable frameworks has substantial productivity impacts. This can make it more difficult to reliably estimate the size and cost of such projects early in the software process. The newly proposed System Meter method tackles this problem by distinguishing functionality to be implemented from functionality supported by reusable components. It therefore yields more uniform and predictable productivity measures. Moreover, it can also be applied already after a preliminary analysis phase, in contrast to the more traditional Function Points approach.

1 Introduction

"How much will it cost?" "How long will it take?" These questions are notoriously difficult to answer accurately and early in the software process. Software metrics tackle this problem by assuming a statistical correlation between the "size" of a software project and the amount of effort that is typically required (in a given context) to realize it. A metric of size that will be useful for cost estimation must be based on the inherent complexity of the system to be developed (rather than on, say, lines of code, since no lines of code exist yet). Such metrics exist, and have been applied with varying degrees of success in the past, but the nature of software development has been changing, and some of the assumptions behind the established cost estimation techniques are slowly being invalidated. How can these techniques keep up?

The trend towards "open systems development" means that software systems are no longer completely knowable (if they ever were!): not only are systems being increasing built from existing components and frameworks, but systems need to be built more flexibly to cope with changing requirements [8]. This affects software estimation in two important ways. First, since modern software systems are constructed differently, this affects the way that system complexity is modelled and measured. Second, the boundary between what is to be implemented and what may be reused plays a more important role. Our goal in software size estimation is to obtain as accurate a picture as possible of the final "size" or complexity of the

system to be built as early as possible in the software process so we can translate this to a reasonable cost estimate based on known productivity rates.

On the other hand, when using Object Technology (such as object-oriented frameworks, or other collections of reusable components), it is common to report "increased productivity." Improvements in productivity are "good" (of course!), but they cannot be used in software estimation because they can only be measured after the fact. We cannot use our known productivity rates to reliably estimate project costs and also expect an increase in productivity.

The fallacy lies in the way we normally think about productivity, as "total functionality delivered per person days of effort spent." Individual team members do not magically become more productive by using Object Technology; rather they are able to be just as productive as before but at a higher level of abstraction! We can formalize this notion by measuring the *incremental functionality developed* rather than the total functionality delivered, by simply factoring out the reusable components insofar as they do not affect the incremental system complexity. Correspondingly, we can now measure *incremental productivity* in terms of incremental functionality developed per person days of effort. With this approach, we find that (incremental) productivity is more predictably uniform, but Object Technology helps us to reduce the complexity of the system to be developed.

The System Meter is a new software sizing approach based on an object-oriented model of system descriptions [5]. This model is intended to better reflect the structure and complexity of software systems developed using object-oriented methods, in contrast to the well-known Function Point method (cf. [1], [12]), which does not explicitly take object concepts into account. The System Meter approach additionally distinguishes between components to be developed and those to be reused, thus reflecting the idea of incremental functionality.

We present some preliminary results of a field study of thirty-six projects developed using Object Technology. Both the traditional Function Point metric and the new System Meter method were applied to all projects, and the results were compared. The overall analysis found the System Meter to be of slightly inferior predictive power than the Function Point approach, but it could be applied significantly earlier in the software process [6].

More interesting though are detailed results derived from a subset of the studied projects consisting of four sets of projects that developed and reused object-oriented frameworks. With the System Meter, each of these projects reflected (predictable) size reduction with consistent productivity, whereas the Function Point method only reflected (unpredictable) increased productivity.

2 Software Sizing and Object Technology

The use of Object Technology (OT) affects both the qualitative (or structural) aspects of the software process and the quantitative aspects. Whereas a great deal of work has been published on this first aspect — i.e., how the software process itself changes — there has been comparatively little analysis of how the use of OT affects the size of software delivered and the total effort expended. Software sizing has many important applications, but one of the most critical applications is in the estimation of the overall cost of a project during the earliest phases of the software process. The classical approach to cost estimation is to assume a correlation between the size of the system to be delivered and the productivity of the project team.

Productivity, of course, must be measured in advance and recorded in an empirical database. Software size, on the other hand, must be measured using techniques that allow the results of requirements specification or analysis to be used as predictors of the final system size and effort.

The first question to answer is "Should we measure (1) code size or (2) the amount of functionality to be delivered?" Research and practice (cf. [1], [2], [11] and [12]) in the area of software sizing strongly tend towards measuring functionality. The measure of choice for our research therefore was the Function Point (FP) metric originally defined by Albrecht [1] in the late seventies.

The FP metric, however, has already shown some deficiencies [3][4] and exhibits yet others when analysed in the context of OT. First of all, it is based on a rather old-fashioned underlying concept (or *metamodel*, cf. [5]) in which all systems are seen to consist of two parts: database structures, and functions that access those structures using four basic database operations, create, read, update and delete (commonly known as "CRUD"). Both parts are measured using historically founded fixed points (FP-factors) per data element and per access according to a complexity rating (easy, medium, complex) associated with the element (e.g. a create access of a complex data element yields 6 FP). In case your system does anything more than CRUD, you have the possibility of incorporating either a limited number of system wide "influence factors" or a set of heuristic mappings into the CRUD model. Both solutions lead to an undesirable bias inherent in the definition of the FPs (as shown by Kemerer [3]). If the majority of the requirements fit the basic metamodel, though, one can live with that "dark spot." The influence factors, however, are (like the FP-factors mentioned above) tied to unnecessarily historical "magic numbers" that have been shown not to contribute to the correlation to effort spent (cf. [4]) but only turn out to be a source of confusion and potential misuse (cf. [12]).

When we consider the effects of OT, two drawbacks become apparent: First, FPs were never designed to fit a metamodel completely different from the "database/function" (or CRUD) metamodel. Object-oriented models, on the other hand, are more generic with respect to the following three dimensions: (1) the kinds of entities or classes used, (2) the kinds of functions or methods used, and (3) the definition of which entities support which functions, thus allowing structural and functional requirements to be freely linked. In such situations the "influence factor" pseudo-solution is no longer sufficient because it only supports a limited number of kinds of requirements that may not be related to the other parts of the FP metamodel.

Second, FPs do not capture the effects of reuse, i.e. the availability of frameworks or libraries that address parts of the requirements. Rather than measuring the total functionality of the system to be delivered, we should be measuring only the incremental functionality to be implemented beyond that which is provided by the framework components. Similarly, our productivity ratings, to be useful for software cost estimation, should be based on incremental functionality implemented rather than on that which is delivered.

A final drawback of the FP metric is that it is based on a very low level of abstraction and thus depends on the results of detailed analysis. This means that it can only be applied relatively late in the software process with respect to the more rapid and tighter cycles common to modern development approaches. This is not so serious when we are only interested in ana-

lysing productivity after the fact, but it is a critical obstacle when we need the measures for software project estimation.

Summing up, we have the following requirements for measures of size:

1. Historical "magic numbers" should be avoided

2. Availability of reusable components and frameworks should be taken into account

3. Arbitrary kinds of functional requirements should be captured

4. Possible links between functional and structural requirements should be expressible

5. The metric should be applicable early in the software process

Much of the work on software metrics for OT focusses on descriptive quality metrics. Two recent proposals, however, address software sizing: Harry Sneed's Object Point (or rather Data Point) metric is a simplification of the FP metric which drops the functional part [11]. This is useful for data-oriented applications, because it simplifies the modelling and measurement task, thus supporting criterion 5. It does not address, however, the other four requirements. The Task Point metric is a development of the Swiss Bank Corporation of London (contact: Mark Lewis, Bezant Ltd., Wallingford, U.K.). It can be seen as a variant of the FP metric that has been terminologically adapted to business process descriptions, but still essentially is equivalent to FPs.

The weak support for these five requirements in both traditional and newly proposed approaches led us to search for a new kind of metric of software size. Such a new metric could be used not only to analyse the effects of OT on the software process of completed projects, but would actually be useful for controlling the software process itself. By "controlling" we mean that both the expectations and the results of the process can be measured in a sound and precise way. We seek to honour Tom DeMarco's statement "You can't control what you can't measure." [2]. In the next section we briefly present the System Meter, which is an attempt to address these goals. For details, refer to [5] and [6].

3 The System Meter — A New Approach for Measuring System Size

We all agree that it is not possible to measure things that are not defined precisely. The "things" in our context are object-oriented systems or models. In order to measure them, we need a so-called metamodel that tells us how they are constructed. The System Meter is based on a metamodel for "system descriptions" encompassing all manner of software items from requirements specifications to source code. This metamodel proposes that a system description is at its most detailed level of abstraction made up of "description objects" that may take on several (polymorphic) forms, such as "classes", "features", "methods", "formal parameters", "messages" and "actual parameters". This metamodel is referred to as the *generic system metamodel*.

Every object[3] has two parts: (1) its name or "external part," which enables the object to be referenced, and (2) the set of its implementation objects or its "internal part," which constitutes its behaviour. The implementation objects are firstly the messages that must be sent when the description object is created, modified or deleted, and secondly, the objects passed as parameters in those messages. We say that these implementation objects are "writers" of the description object in the sense that they contribute to its definition. A detailed and commented metamodel presented in [5] defines in a semi-formal way how the various objects listed above are mapped into this generic view of object implementation.

The aspect of reuse is captured by the fact that any given system description can be separated into "language objects", "library objects" and "project objects". The language objects are those that are given and fixed for the software phase at hand: for the analysis phase they might be analysis method concepts; for the implementation phase they might be programming language features. Library objects are reusable objects defined in terms of the language objects. Finally, project objects are those that describe the incremental functionality under development. This third class of object is the most important in determining the System Meter size for a system.

The System Meter is defined by two parts, corresponding to the two parts of an object. First, the external size or complexity depends entirely on the complexity of the name:

(1)   **externalSize** (Object o)   =   if (isNotAnonymous (o))
                                         then   numberOfTokens (name (o))
                                         else   1

In this formula, the function "numberOfTokens" returns the number of new tokens in the name, relative to the already known tokens during a sequential system scan. For example, the object name "currentRegistrationPolicy" might consist of three tokens, if each of the tokens "current", "Registration" and "Policy" are new. Anonymous objects are those referenced through a container or a pointer object, and are considered to have an external size of 1. The language objects are all assigned a fixed external size of 1.

Second, the inner complexity of a given object is determined by its set of implementation objects (denoted as "isWriterOf(o)"):

(2)   **internalSize** (Object o) = $\sum_{x \in \text{isWriterOf (o)}}$ externalSize (x)

The total size of an object then is just the sum of these two sizes:

(3)   **Size** (Object o) = externalSize (o) + internalSize (o)

When measuring whole systems, reuse should be taken into account. We can now exploit the distinction between "language", "library" and "project" objects in the definition of system size:

(4)   **Size** (System s) = $\sum_{o \in \text{isLibraryObject}}$ externalSize (o) + $\sum_{o \in \text{isProjectObject}}$ Size (o)

As we can see, the language objects are not counted at all because they do not vary when building a system. The external interfaces of library objects are counted because these interfaces are often not entirely stable. Total sizes, i.e. externals and internals, are counted only for

---

3. Since we are modelling object-oriented systems, it is natural that the descriptive metamodel should also be expressed in terms of object concepts. Note, however, that when we use the word "object" here, we mean a "description object," not a run-time object of the software system.
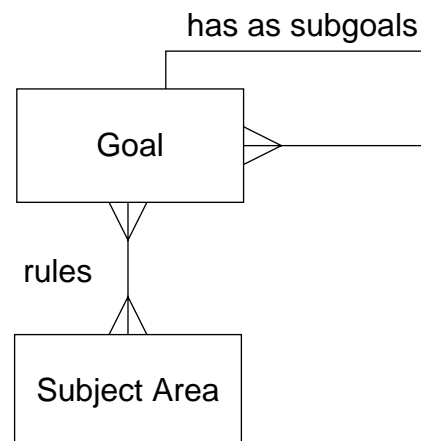
has as subgoals

Goal

rules

Subject Area

Figure 1 Metamodel of a Preliminary System Description

the project-specific objects, and so we only measure the incremental functionality developed rather than the total functionality delivered. With this definition, we can model virtually any degree of reuse in a straightforward way. The System Meter therefore fulfils requirements 1 and 2.

In order to fulfil criteria 3 and 4 we would want to — but according to criterion 5 we cannot afford to — wait until a complete implementation model is available consisting of classes, methods, parameters, etc. for all parts of the system. Instead, we must to be able to measure requirement models which are accessible earlier in the software process. For this reason each of the phases of the software process will have its own metamodel and its own corresponding "language" for the concepts at the appropriate level of abstraction. A preliminary system analysis, for instance, results in a description consisting mainly of a goal hierarchy, application subject areas, and links between those two parts (thus fulfilling criteria 3 and 4, see figure 1). A specific preliminary systems description would consist of a graph of objects and relationships that are instances of the metamodel.

A mapping between the various metamodels is both necessary and possible, so we can see, for example, how goals and subjects in the preliminary analysis are reflected as (description) classes, features and methods in the generic metamodel for example. Although a detailed description of the metamodel mapping is beyond the scope of this article (see [5] for details), we should mention one side effect of the metamodel mapping that is important for our purposes. Any metric defined for the generic metamodel (which is assumed to be the target of the mapping) is automatically also defined for the mapped-in metamodels. As a consequence, the System Meter fulfils criterion 5, because (1) it is defined for the generic metamodel and (2) the metamodel of the preliminary analysis is mapped into the generic metamodel (as explained in [5]).

Another, more detailed system description metamodel is that of an object-oriented domain analysis model (as proposed e.g by [10]). Research is currently ongoing in applying the System Meter at this level, which we expect will lead to more precise estimation and productivity analysis of object-oriented projects.

4 A Field Study

As explained in detail in [6], an empirical database of 36 projects was established in a field study conducted from mid-93 until 11-95. The correlation biases of (1) the System Meter method and (2) the Function Point method, were calculated and analysed with respect to overall software process effort.

The set of investigated projects varied in many aspects:

1. The variations regarding the *organizational and personal aspect* may probably best be captured by analysing the contractor companies involved. The set mainly consists of projects undertaken at 3 medium-sized information services companies (17, 9 and 4 projects). Additionally, there are 3 university projects and 3 projects from a large company from the chemical industry.

2. The *total project efforts* observed ranged from 1.5 person-months to more than 100 person-months. Whereas the *average team sizes* ranged from as few as 0.8 persons to slightly over 5 persons and *maximum team sizes* from 1.2 to 10.5. The month of project completion ranges from December 1987 until November 1995.

3. From the *technological point of view* the projects may be categorized into an overwhelming 25 projects using Smalltalk, 7 using 4GLs and 4 using C++.

4. From the *methodological point of view* there were 19 projects using the BIO [7] method (a synthesis of modern structured techniques with OMT [10] tailored for client/server database applications), 4 projects using modern structured techniques, and the rest of the projects not really following any method.

5. When regarding *project phases*, there are 6 main phases defined by the template process model of BIO: (1) a preliminary study, (2) essential (domain) analysis, (3) detailed requirements specification, (4) construction (identification and/or construction of design patterns), (5) implementation and test, and (6) introduction. Only a few projects went completely through all of the phases. Additionally, in order to support prototyping, it is allowed — even recommended — at a certain phase to elaborate preliminary versions of results of future phases. In phase 1, for example, BIO recommends that goals and critical parts of the domain subject areas be verified in a small exploratory prototype (phase 5). In phase 3, then, it is crucial to already plunge into phases 4 and 5 (with less rigid documentation and testing however), in order to arrive at a first version of an evolutionary prototype and to give feedback to the final system specification. In order to keep the efforts of the differently tailored processes comparable, they were normalized to the efforts of phases 2-6 of BIO.

6. The *application domains* varied from work flow administration, land registry, statistics, taxation, registration of chemical formulae to decision support and management information systems. Virtually every application (30 of 36) was built using a client/server architecture with a GUI-client and a database server. Four projects explicitly had to deliver reusable frameworks (cf. detailed comments in section 5).

Quadratic regression within the two empirical databases yielded the following approximation functions:

System Meter method: $\text{effort}(x) = 0.151 \cdot x + 0.0000182 \cdot x^2$

Function Point method: $\quad$ effort $(x) = 0.638 \cdot x + 0.0000436 \cdot x^2$

In these formulas x stands for the system size measured in units appropriate for each of the methods. The regressed coefficients have the dimensions of "person-days/units-of-size" and "person-days/square-units-of-size" respectively. The units of the returned values of the effort functions therefore are person-days.

More interesting, though, than the regressed coefficients are the bias indices that were calculated assuming a Gaussian distribution of efforts for any given value of size and with respect to a 95%-confidence level. This means that when applying such a function to calculate an estimation value **e**, the effective outcome of will be in the range of **e**±bias with a probability of 95%.

For the new System Meter method the bias is 33%, whereas for the conventional Function Point method it is 20%. At first sight, it seems that the new method is inferior. But considering the fifth criterion for sizing techniques, there is one major difference: the System Meter method can be used after phase 1 (the preliminary study) whereas the Function Point method cannot be applied until a more refined system model (the analysis model) is available. The effort spent until application of the System Meter typically is 5% of the total software process effort, a percentage also observed in the field study. In contrast, the completion of phase 2 required by the Function Point means that 18% of the total effort is already spent. The new method can therefore be viewed as being three times faster and more cost effective. This aspect of estimation speed is of growing importance due to the reduced development cycles in modern software producing units.

5 The Effects of Building and Reusing Frameworks

The survey of over thirty projects using OT (or other advanced techniques) also included four independent sets of projects that explicitly had to develop and apply frameworks. Although this is too small a number of projects to allow any statistically sound conclusions to be drawn, we were interested in seeing whether quantitative metrics could tell us anything about the impact of reusable components on the productivity of projects that both develop and reuse such components. We will briefly describe each of these four sets of projects, and then discuss what preliminary conclusions can be drawn from the productivity measurements taken using the System Meter.

The first project A1 had as its primary goal the development of a first usable version of a system for document registration and management of standard administrative procedures (such as legal initiatives, trials, applications, elections, etc.). Since this project was the first of a new generation of client/server applications, it also had a secondary goal to develop a framework dealing mainly with (1) the access of data stored in relational databases (e.g. ORACLE, Sybase, etc.) from an object-oriented client system, (2) the handling of complex interaction sequences involving several windows and (3) the construction of a programmable model for administrative procedures. Project A1 was immediately followed by a successor project A2 that produced a second release of the system using the framework and including some major enhancements, like incremental and keyword-based searches in the registry and ongoing procedures.

The second project B1 had the primary objective to produce a management information system (read only data access) for presenting information from several existing production systems (financial accounting, organisations and addresses, task management, documentation registry, etc.). At the same time it also developed a framework dealing with data access (but using stored procedures), data imports and exports, and presentation of data in flat and hierarchical browsers. It was followed by a project B2, using the framework, that dealt with acquiring and aggregating data for statistics of scholars (150'000 records).

Project C1 — in contrast — was explicitly and uniquely defined to produce a framework. This framework, too, had to accomplish (1) the task of accessing stored relational data from an object-oriented client, (2) an extensive tailoring of the MVC-framework for the search, display and modification of persistent data, and (3) the implementation of a state-transition interpreter that supports the state-based specification of the context-sensitive behaviour of a GUI application (dimming of impossible options and buttons). It was both paralleled and followed by several projects C2, C3, C4, C5, C6, C7 and C8 using this framework to develop a taxation system.

Finally, project D — which is still ongoing — has the task to produce a framework for the administration of personal and organisational information together with its addresses (postal, electronic, etc.). This is a kind of system component that is commonly required in administrative systems, and it is intended that this component can be easily "plugged" into existing and future systems. Project data from the "plug in" processes, however, are not yet available.

Data Analysis

Productivity was measured for all of the projects using both the Function Points and System Meter approaches. For project A1 we determined a productivity of 1.2 FP/Person Day (PD). In the follow-up project A2 the productivity rose to 1.5 FP/PD mainly due to the framework reuse, because all other influencing factors (project team, tools, methodology, domain area, infrastructure) remained constant. Expressed in percentages, this is an increase of 25%.

When looking at the System Meter numbers we observe a productivity of 5.2 SM/PD for A1 and 5.8 SM/PD for A2. This means only a 12% rise. Since the System Meter only measures incremental functionality, this means that only incremental productivity is observed. Since the System Meter values decrease when reusable parts are incorporated into a system, the incremental productivity will tend to be more stable. The reasons for the remaining size of 12% may be twofold: (1) there is a productivity bias outside the scope of models and metrics which ranges between 5% to 10%, and (2) substantial non-framework based functionality was still developed in A2.

This effect of the System Meter method may seem undesirable, because we want to encourage reuse. On the other hand this effect can also be viewed at as a positive stabilizing property of the new method which improves the quality of the estimates. When productivity rates are more constant, derived estimates will be more accurate (as demonstrated in [9] and [6]). In fact, this is a more consistent and reproducible way of understanding the effects of frameworks on productivity: project teams do not magically become more productive through the use of frameworks (as the FP method might have us conclude), but rather the size of the system to be designed and implemented is dramatically reduced. If we can accurately meas-

ure the size of the reduced system, then we can rely on our previous productivity rates to help us better estimate the actual cost of the project.

In case an organization would like to encourage reuse by setting productivity goals, the System Meter could alternatively be applied without its reuse modelling component, thus rating all system description parts as "project" parts. Values obtained with this System Meter variant are denoted as "flat" System Meters. Those flat SMs were prototypically measured in project A2. (In project A1 the flat SMs equalled the SMs because there were no reusable framework components available at the start of the project). A2's flat productivity rate was calculated as 7.2 SM/PD which means a 38% increase. These "magical" improvements in productivity, however, can only be observed after a project has been completed, and so are uninteresting from the perspective of predictive estimation.

Another independent case of framework construction and reuse was observed in the project pair B1 and B2. Function Point analysis yielded productivities of 1.5 and 2.0 respectively (a 33% increase) whereas the System Meter method yielded 5.3 and 5.7 rates (an 8% increase). Actually, more components could be reused without modification in project B2, which explains both the higher FP increase and the more constant behaviour of the SM values. Another difference between B2 and A2 was the tighter schedule of A2. This required the formation of a bigger team, which undoubtedly contributed to the lower increase in productivity of A2 with respect to B2 (in accordance with observations described by Putnam [9]). This "team size" effect, also known as the "Mythical Man Month" syndrome, of course, has nothing to do with frameworks or reuse, but is a general rule of management of complex development processes.

In the setting of projects C1 — C8, we observed the following series of productivity rates:

Project	*FP productivity*	*increase vs. C1*	*SM productivity*	*increase vs. C1*
C1	1.1	n.a.	5.7	n.a.
C2	1.6	45%	5.3	-7%
C3	1.6	45%	4.9	-14%
C4	1.5	36%	6.1	7%
C5	1.7	55%	5.3	-7%
C6	1.5	36%	4.9	-14%
C7	1.6	45%	5.6	-2%
C8	1.6	45%	6.2	9%

Table 1   Productivity rates and percentage deltas for projects C1-C8 (C1 = framework project)

The observations we made concerning the A and B projects are even more pronounced here. When measured using the System Meter method, the projects that reused the framework (C2-C8) exhibited even lower incremental productivity than the framework construction project C1.

In contrast to projects A1 and B1, where the framework construction was only a secondary goal after the development of a first version of an application software system, project C1 was entirely dedicated to establishing a framework. This explains why C1 rated notably low in the FP analysis, and consequently, why the follow-up projects C2-C8 exhibited unusually high

percentage FP productivity increases. The System Meter method, on the contrary, yielded rather uniformly distributed incremental productivity rates. Due to the heavy framework reuse, C2-C8 rated rather low. Notable exceptions are projects C4 and C8 which developed substantial non framework supported subsystems (C4 dealt with the development of a spreadsheet-like taxation calculation model; C8 encompassed a few non standard reports).

The quantitative analysis of Project D, finally, could not cover productivity increase rates, because no follow-up project exists yet for D's framework. We therefore concentrated our analysis on the estimating power of the two methods: D yielded 295 FPs and 1250 SMs. When applying the currently regressed coefficients out of the 36 project empirical database, we would have estimated 193 PDs (bias: ±20%) using the Function Points and we actually did estimate 229 PDs (bias: ±33%) using the System Meters. The effective effort spent is 220 PDs, which is substantially closer to the SM estimate than to the FP estimate. This, of course, may not be interpreted as empirical evidence of the superiority of the new method over the established FP method. On the other hand, the new method is not rejected by the empirical results, but rather, we are encouraged to continue our investigations.

6 Conclusions

The System Meter is a new approach to quantitative software metrics based on a rich meta-model for system description objects rather than on the better-known, but more simplistic Function Points approach. The System Meter has been developed to address the effects of Object Technology on the way that software projects are structured and composed. A Field Study of over thirty projects comparing the two kinds of metrics demonstrated that the System Meter was able to deliver estimates of comparable quality, but was able to do so earlier in the software process. The principle idea behind the System Meter is to distinguish between the "internal" and "external" sizes of objects, and thus to only measure complexity that has a real impact on the size of the system to be implemented.

More interesting, however, was the way in which the impact of framework reuse was reflected in the estimates. Whereas the Function Points approach only reported a "magical" increase in productivity, the System Meter reported more uniform productivity but for reduced project sizes. To capture this concept of reduced project size we introduced the notion of "incremental functionality." The key insight is that framework and component reuse does not make project teams inherently more productive; it just reduces the amount of work that has to be done. If this reduced work can be measured accurately and early in the software process, then we have a way of estimating the "productivity gains" we can expect from the use of Object Technology. The sample space for the framework-based projects was too small for the results reported here to be statistically significant (four framework development projects and nine subsequent framework reuse projects), but the results for the field study as a whole give us some confidence in the approach in general.

Acknowledgements

divisions, namely Pierre André Briod, Alfred Graber, Robert Siegenthaler and Åke Wallin, who were project leaders and who co-authored or reviewed the BIO manual.

References

[1]   A.J. Albrecht, "Measuring Application Development Productivity," in Proc. IBM Applications Develop. Symp. Monterey, CA, Oct. 14-17, 1979, GUIDE Int. and SHARE, Inc., IBM Corp., p. 83ff.

[2]   T. DeMarco, *Controlling SW Projects*, Prentice-Hall, Englewood Cliffs, N.J., 1982.

[3]   C.F. Kemerer, "Reliability of Function Points Measurement," CACM Vol. 36, No. 2, 1993, pp. 85-97.

[4]   B. Kitchenham, K. Kænsælæ, "Inter Item Correlations Among Function Points," IEEE publication Nr. 0-8186-3740 — 4/93, pp. 11-14.

[5]   S. Moser, "Metamodels for Object-Oriented Systems," in *Software-Concepts and Tools*, Springer Intl., 1995, pp. 63-80

[6]   S. Moser, "Estimating the Modern Software Process," IAM, U. Berne, Switzerland, 1995, submitted for publication.

[7]   S. Moser, R. Siegenthaler, "BI-CASE/OBJECT (BIO) V2.1 — An Object Oriented Software Development Method," Bedag Informatik, Berne, Switzerland, 1995.

[8]   O. Nierstrasz, D. Tsichritzis (ed.), *Object-Oriented Software Composition*, Prentice-Hall International, 1995.

[9]   L.H. Putnam, *Software Cost Estimating and Life Cycle Control*, Computer Society Press, IEEE, Los Alamitos CA, 1980.

[10]   J. Rumbaugh et al, *Object-Oriented Modelling and Design*, Prentice-Hall, Englewood Cliffs, N.J., 1991.

[11]   H. Sneed, "Calculating Software Costs using Data Points," SES, Ottobrunn/Munich, Germany, 1994.

[12]   C.R. Symons, *Software Sizing and Estimating Mk II FPA*, John Wiley & Sons, N.Y., 1993.