Empirical Software Engineering, 8, 225–254, 2003.

Object-Oriented Function Points: An Empirical Validation

G. ANTONIOL antoniol@ieee.org
RCOST—Research Centre on Software Technology, University of Sannio, Department of Engineering, Palazzo ex Poste, Via Traiano, I-82100 Benevento, Italy

R. FIUTEM fiutem@sodalia.it
Research and Technology Department Sodalia SpA, via V. Zambra, 1, 38100 Trento, Italy

C. LOKAN cjl@cs.adfa.edu.au
School of Computer Science, Australian Defence Force Academy, UNSW, Canberra ACT 2600, Australia

Editor: Sandro Morasca

Abstract. We present an empirical validation of object-oriented size estimation models. In previous work we proposed object oriented function points (OOFP), an adaptation of the function points approach to object-oriented systems. In a small pilot study, we used the OOFP method to estimate lines of code (LOC). In this paper we extend the empirical validation of OOFP substantially, using a larger data set and comparing OOFP with alternative predictors of LOC. The aim of the paper is to gain an understanding of which factors contribute to accurate size prediction for OO software, and to position OOFP within that knowledge.

A cross validation approach was adopted to build and evaluate linear models where the independent variable was either a traditional OO entity (classes, methods, association, inheritance, or a combination of them) or an OOFP-related measure.

Using the full OOFP process, the best size predictor achieved a normalized mean squared error of 38%. By removing function point weighting tables from the OOFP process, and carefully analyzing collected data points and developer practices, we identified several factors that influence size estimation. Our empirical evidence demonstrates that by controlling these factors size estimates could be substantially improved, decreasing the normalized mean squared error to 15%—in relative terms, a 56% reduction.

Keywords: Size prediction, OO size estimation, software metrics.

1. Introduction

Cost and effort estimation is an important aspect of managing software development projects. Most methods for estimating effort require an estimate of the size of the software. Accurate estimation of size is vital.

Traditionally, size meant the length of the delivered program, in lines of code (LOC). There is a large body of experience now in estimating effort from LOC, using models such as COCOMO (Boehm, 1981) or knowledge of team/company dependent productivity factors.

Unfortunately, estimating LOC has proved to be just as difficult as estimating effort, especially early in development when the estimates are of most use.

An alternative is to identify other aspects of the size of a system, preferably that can be measured or estimated early in development, from which it might be possible to estimate effort. These might be structural (e.g. number of packages, subsystems, files, classes, subroutines) or functional (e.g. use cases; transactions and files, as in traditional function points (IFPUG, 1994)).

The ideal is to identify size measures that can be used to estimate effort directly. But it takes time, and extensive data, to develop and validate direct estimation approaches. In the mean time, an alternative is to estimate LOC as an intermediate step (i.e., use other measures as a basis for estimating length in LOC), and then use existing LOC-based models to estimate effort. This provides immediate access to the body of LOC-based experience, which is still very widely used.

Estimation is not a one-time activity at project initiation. Estimates should be refined continually throughout a project (DeMarco, 1982). Thus, it is necessary to estimate size repeatedly throughout development.

More detail becomes known about the design and eventual implementation of the system as the project proceeds. Progressive size estimates should take advantage of this increased knowledge. Thus we need to identify things that can be measured at different stages of development, using all of the detail available at the time, that can be used to predict the final project length in LOC. This might mean measuring different things at different stages. Or it might mean measuring the same things throughout the project, if they can capture increasing knowledge about the system. Either way, it must be possible to obtain the measurements easily and objectively.

Most research on estimating size has dealt with traditional applications and traditional software development practices. Few methods have been proposed for object oriented (OO) software development (see Card et al., 2001, for a recent survey).

A particular advantage of the OO paradigm is that since object models are used throughout the whole software development process, and not only at the requirement specification phase, repeated measures can be taken at different times during the development process (as shown in Figure 1) using the same method. This allows one to track and verify size predictions along the entire development cycle.

We have proposed object oriented function points (OOFPs) as a size measure for OO software (Caldiera et al., 1997; Caldiera et al., 1998; Antoniol et al., 1999). This is an adaptation of the traditional function points approach, for OO software.

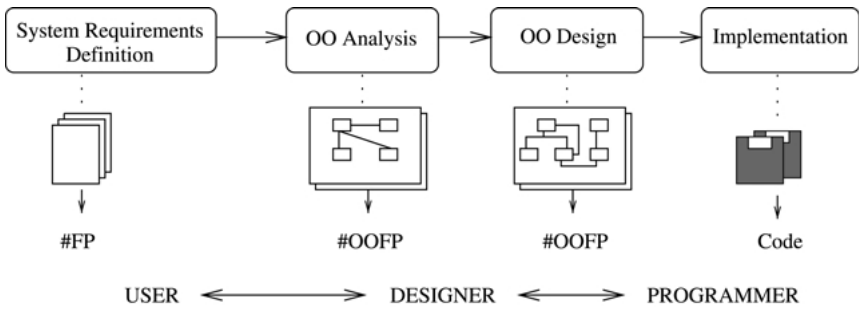

Figure 1. Measures in the software development process.

A crucial goal in defining OOFPs was that the method can be automated. This makes it easy to recalculate OOFP throughout a project, supporting re-estimation. It requires that the method be objective, with no subjective judgments and no ambiguities.

For our initial investigations, we use OOFPs to estimate LOC. Direct estimation of effort requires data that is not available to us in this study. Meanwhile, we look at estimating LOC as a step towards estimating effort and cost. Ultimately, we hope to investigate direct relationships between OOFPs (and other measures of OO software) and project effort and cost.

Earlier evaluations of OOFPs showed encouraging results for size estimation, but the data set was small. Here we report further empirical validation of OOFPs, using a larger data set.

Two specific research questions were investigated:

- How accurate are size (LOC) estimates based on OOFPs, in a larger data set than previously investigated?

- How does this compare with size estimates based on more traditional aspects of object models?

Our aim was to gain an understanding of which factors contribute to accurate size prediction for OO software, and to position OOFPs within that knowledge. In our development environment, we noted the use of COTS (Component off the Shelf), libraries, reused code, middleware and automatic code generators. We categorized these into three factors that influence size estimation: mis-counted code (automatically generated code, and test code), over-specified design, and under-specified design. By controlling these factors, we were able to greatly improve estimation accuracy.

We report one organization's experience and the identified influence factors affecting software size estimation. Other organizations may learn from it, even though they would not necessarily be able to use the specific models identified here directly in their own environments.

The paper is organized as follows. Section 2 summarizes the OOFP approach and compares it with related work. Section 3 describes the data set, and the experimental method used to analyze it. Section 4 presents the experimental results. This section includes a discussion of some factors that affect the accuracy of the size estimates; controlling these factors reduced the average prediction errors by about 60%. Section 5 reports some lessons we learned from experimentation. Finally, in Section 6 conclusions and outline for future work are presented.

2. Object Oriented Function Points

2.1. Background

One of the best-known methods for size estimation is function points. Since they were proposed in 1979 (Albrecht, 1979), function points (FPs) have become a well

known and widely used software metric. The most widely used definition of FP analysis is presented in the International Function Point Users Group (IFPUG)'s *Counting Practices Manual* (IFPUG, 1994).

Function points attempt to capture the user's view of software functionality. The key features of FPs are that they are available early (at the specification phase), and they are a measure of the problem independent from any particular implementation.

Despite some concerns (Verner et al., 1989; Kitchenham and Känsälä, 1993; Kitchenham et al., 1995; Jeffery and Stathis, 1996), practitioners have found FPs to be useful in the data processing domain, for which they were invented.

Several variants of FPs have been proposed to extend their use to other domains (see Hastings, 1995 for a survey). These variants follow from the need to exploit the understanding gained with function points in their traditional domain. Some variants relate specifically to the OO paradigm (see Section 2.5).

2.2. FPs must be Adapted for OO Software

If OO is regarded as a development approach, one might argue that function points should apply as well to OO software as to any other software. Function points are meant to be counted from a specification document, and to be independent of the implementation.

One problem with this can be the forms of documentation available. Although FP are not restricted to projects using traditional structured design techniques, the items counted in FP are often identified from the documents produced with those techniques (e.g. data flow diagrams, hierarchical process models, or database structures). Those documents are not provided in OO methodologies. Some OO function point proposals (see Section 2.5) seek to derive the required counting information from use cases. The focus is still on traditional function points, but the counting information is found from non-traditional sources.

In our view, in OO systems the goal of measuring the functionality requested by the customer cannot be fulfilled with the traditional FP counting rules. Although the traditional FP approach can be used as an analogy, it cannot be used directly with OO software. Function points must be adapted for use with OO software. There are several reasons for this. First, different abstract models are used. The FP counting method is based on transaction and file types that are visible to the end user. Software components that are not visible from a user's viewpoint are not considered. The OO approach, by contrast, models software systems as collections of cooperating objects.

Second, in the function point model the "user" is an end user—a person. In OO, the concept is broader. "Actors" in UML include users, but can also be external applications or devices. A one-to-one mapping between OO actors and FP users is unlikely.

Most importantly, FP files and transactions do not map directly to OO concepts. Use cases might be seen as corresponding to transactions, except that there is no one-to-one relationship. One use case may need to be counted as one or as many

transactions, depending on the task it performs (Fetcke et al., 1997). Logical files do correspond to the OO concept of an object—more precisely, to the type of object that in an analysis model is labeled an entity object (to be distinguished from interface and control objects). But this distinction is not visible at domain analysis when the model of domain object identifies the data concepts which are relevant for the application domain. Besides, it should be noted that in the IFPUG *Counting Practices Manual* (IFPUG, 1994), the weights of transactions and files are based on detailed rules requiring the determination of data element types (DET), record element types (RET) and file types referenced (FTR). But, in an OO software engineering context, this information is also not visible. To find them, analysts would have to deepen their investigations (following ways that do not comply with OO software engineering rules) for the sake of measurement only.

Finally, inheritance (a mechanism whereby one object acquires characteristics from one or more other objects) is a genuine OO concept that has no direct representation in FP analysis (even though in IFPUG, 1995 the complete hierarchy is taken as a single file, with RETs for each subclass).

2.3. Adapting Function Points to OO

In order to adapt function points to OO, we need to map function point concepts to object oriented concepts, and must decide how to handle OO-specific concepts such as inheritance and polymorphism.

An important goal for us is that the measurement method should be easy to apply repeatedly throughout a project. The method should be able to be automated. It must therefore not involve subjective judgments that would require human intervention. This consideration drives some of the choices of what to include in the method.

We do not aim to predict size from code abstractions. We aim to predict size from analysis and design abstractions that are captured earlier in a project—in particular, from object models.

In our approach, we only deal with the static, structural information that can be found in object models (essentially class diagrams). Dynamic information (such as, for example, that contained in UML interaction or activity diagrams) is not considered in the current formulation.

The object model is the most important model for our purposes. The object model is developed in the early phases of the software development cycle, and so can be measured earliest. It is the one that represents what is actually to be built and what mostly may contribute to the size of the application. In a sense, the other models help in completing the object model.

In traditional FPs, the items counted are logical files and the transactions that operate on those files. The central analogy we used to map FP to OO software relates logical files to objects, and transactions to object operations (i.e., methods).

A class in an object model encapsulates a collection of data items, so it is the natural candidate to map logical files into the OO paradigm.

The FP approach differentiates between internal logical files (ILFs) and external input files (EIFs). This division clearly identifies the application boundary. In the OO counterpart, objects within the application boundary correspond to ILFs while objects belonging to external systems (or components if a single component of an application is being modeled) are considered as EIFs.

FP transactions are classified as external inputs, external outputs or external inquiries, according to the direction of information flow and the particular operations performed. This categorization makes sense for the DP domain, but is not easily applicable in general. Moreover, in an object model the information needed to distinguish the direction of information flow in an operation is rarely present. To provide it would require considerable human analysis and intervention, contrary to our goal of a method that is objective and can be automated. Hence, we do not try to distinguish the above three categories. We simply treat them as generic service requests (SRs), issued by objects to other objects to delegate operations.

2.4. OOFP Measurement Process

This section presents a brief summary of the OOFP measurement process. For full details and explanations of the method, see Antoniol et al. (1999) and Caldiera et al. (1998).

OOFPs are assumed to be a function of objects comprised in a given object model. The model might be produced at the analysis or the design stage, or extracted from source code.

Counting OOFPs is a four-step process (depicted in Figure 2).

Identify logical files. The object model is analyzed to identify the units that are to be counted as logical files.

There is not necessarily a 1-to-1 mapping from classes to logical files. Aggregation and inheritance relationships between classes can suggest that a group of classes should be counted a single logical file. Different choices of how to deal with

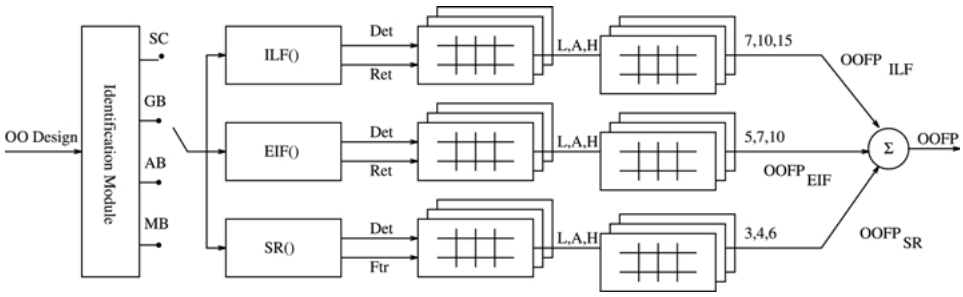

Figure 2. OOFP computation process.

aggregations and inheritance relationships lead to different ways to identify logical files. These choices correspond to four OOFP counting strategies:

- *Single class (SC) boundary*. Count each separate class as a logical file, regardless of its aggregation and inheritance relationships.

- *Aggregations boundary (AB)*. Count an entire aggregation structure as a single logical file, recursively joining lower level aggregations.

- *Generalization/specialization boundary (GB)*. Given an inheritance hierarchy, consider as a different logical file the collection of classes comprised in the entire path from the root superclass to each leaf subclass.

- *Mixed boundary (MB)*. Combination of option 2 and 3. First, aggregation structures are combined into a single structure; then the inheritance is done.

Determine complexity. The complexity of each logical file and service request is determined.

For each logical file, the numbers of DETs and RETs are counted.

DETs are computed by counting simple data elements. Simple attributes are counted as DETs because they represent a "unique user recognizable, non-recursive field of the ILF or EIF" (IFPUG, 1994). Single-valued associations are considered as DETs, since IFPUG suggests counting a DET for each piece of data that exists because the user requires a relationship with another ILF or EIF to be maintained (IFPUG, 1994). Aggregations are a special case of associations, so single valued aggregations are also counted as DETs (there may be multiplicity specified on aggregations).

RETs are computed by counting complex data elements. Each logical file counts one RET in its own right, because it represents a "user recognizable group of logically related data" (IFPUG, 1994). A complex attribute (an object, or a reference to an object) within a logical file counts as an additional RET, since it can be seen as "a user recognizable subgroup of data elements within an ILF or EIF" (IFPUG, 1994). A multiple-valued association or aggregation also counts as an additional RET, because an entire group of references to objects is maintained in one attribute.

To classify the complexity of a logical file, we adopt the approach used in the traditional Function Points method. After the DETs and RETs of a logical file are counted, a table is consulted to classify the file as having "low", "average", or "high" complexity. As a starting point, we adopt the IFPUG tables directly. For example, IFPUG classifies an ILF with 4 RETs and 55 DETs as having "high" complexity (IFPUG, 1994). We use the same tables and would make the same classification.

A similar approach is used to classify each method (service request) in each class.[1] Simple and complex elements, mainly extracted from the method's signature, respectively represent SR DETs and FTRs. A table is used to classify the service request as having "low", "average", or "high" complexity, depending on the number of DETs and FTRs.

Allocate OOFP values. The complexity scores are translated into OOFP values, using tables derived from the IFPUG *Counting Practices Manual*. For example, an ILF with "high" complexity receives 15 OOFPs; an SR with "low" complexity receives 3 OOFPs.

Sum OOFP values. The individual OOFP values are summed to produce the final OOFP result.

The computation of OOFPs has been automated through a series of tools,[2] which allow us to count OOFPs both from CASE tools and code.

2.5. Related Work

Other authors have proposed methods for adapting function points to object oriented software.

Some retain a focus on traditional function points as the output from a count. The issue is how to relate OO concepts to FP elements:

- Whitmire (1993) considers each class as an internal file. Messages sent across the system boundary are treated as transactions.

- Schooneveldt (1995) treats classes as files, and considers services delivered by objects to clients as transactions.

- A draft proposal (IFPUG, 1995) treats classes as files, and methods as transactions.

- Fetcke et al. (1997) defines rules for mapping a use case model (Jacobson et al., 1992) to concepts from the IFPUG *Counting Practices Manual* (IFPUG, 1994).

Others develop new measures, tailored to OO software but analogous to FPs in some aspects of their construction.

- Sneed (1996) proposed object points as a measure of size for OO software. Object points are derived from the class structures, the messages, and the processes or use cases, weighted by complexity adjustment factors.

- Predictive object points (POPs) (Mehler and Minkiewicz, 1997; Minkiewicz, 1997) are based on counts of classes and weighted methods per class, with adjustments for the average depth of the inheritance tree and the average number of children per class. Methods are weighted by considering their type (constructor, destructor, modifier, selector, iterator) and complexity (low, average, high), giving a number of POPs in a way analogous to traditional FPs.

- Graham (1996) proposed task points as a size measure that can be computed at the requirements analysis stage. A task point is an atomic task that the system will carry out in support of user requirements. Task points are equivalent to the leaf nodes in the task object model—a hierarchical model of the business tasks to be supported by the system.

Our OOFP proposal shares characteristics with each group. Structurally, OOFPs match the first group in mapping classes to files, and services or messages to transactions. As an initial formulation, OOFPs also adopt classification and weighting tables from the traditional FP method. But the philosophy behind OOFPs is more like the second group. Rather than finding a way to count traditional function points, the aim is to develop a measure based on OO concepts that is useful for estimating project attributes such as size, drawing on FPs for inspiration.

The various methods may be compared on several criteria, including structure, how much information is considered, and when the information is available; objectivity and the potential for automation; and value to a user.

We noted above that OOFPs are similar in structure to most other approaches, in mapping classes to files, and services or messages to transactions. Sneed and Fetcke each exploit different information, by considering use cases rather than just the object model; Sneed considers several other things as well. POPs also consider extra information, principally when counting methods since the OOFP approach does not distinguish between method types; the disadvantage is that information about method type is seldom identifiable at the design stage.

OOFPs have the advantage that counting them requires no human intervention. Tools have been constructed to automate the process. This is inherently difficult, if not impossible, in the other approaches.

In the end, the value of each method depends on how useful the measures turn out to be for project management. This is difficult to judge from the literature, because the proposals have been validated in different ways. For example, Fetcke concentrated on demonstrating that his method was unambiguous enough to be applied in practice; Schooneveldt showed in a case study that his method gave a result similar to a traditional FP count; Graham developed a tool to estimate effort from task points, but its accuracy is not reported in the literature.

The main purpose of this paper is to report some empirical evaluations of OOFP, to aid comparisons between this and other methods for measuring OO software.

3. Experimental Method

3.1. Description of the Data Set

Twenty-nine subsystems, from four complete projects in the area of telecommunications, were measured. They were all developed in Sodalia's industrial environment.

Sodalia is a medium-sized enterprise (about 250 programmers) located in Trento, Italy. Initially a joint venture between Bell Atlantic and Telecom Italia, Sodalia is an

IT Telecom company today, the business unit of Telecom Italia Group for Information Technology, sector leader in Italy. Sodalia is a fast growing company in the area of telecommunications operations support systems.

Although the software engineering environment, tools, middleware, and general corporate culture can be considered uniform across Sodalia projects, it is hard to control all factors—especially the human factors. The 29 subsystems were thus selected to be representative of the corporate application domain, the corporate skill and the project teams. C++ was the target language, and the CORBA environment was the standard platform for distributed computing.

Our analysis was performed at the subsystem level, rather than the system level, mainly because that is the level at which the design is generally documented and at which developers work.

The design class diagram and the final code were available for each subsystem. The class diagrams were produced during detailed design, although not necessarily at the end of detailed design. Classes almost always have a constructor (with void argument) and a destructor. Nested inner classes were not represented. Many methods have their parameters specified in full detail, but some do not. A few classes are completely unspecified (no attributes or methods). Thus the design class diagrams represent a mixture of high level and detailed design, perhaps closer to high level design.

OOFP related measures, other OO measures, and the final size of each system in LOC[3] were measured for each subsystem.[4] Subsystem size is spread from a few hundred to about 50,000 lines of code, for a total of about 350 KLOCs. The mean system size measured in LOC is 9983 (standard deviation 11,578 LOC); design documents contain a fairly spread number of classes with a maximum of 113 and a minumum of one (mean value 17, standard deviation 20). Indeed, design documents have quite different details levels; for example, the mean number of specified methods is 98, however, the method standard deviation is 95, i.e., there are designs not specifying methods. Detailed information on the 29 subsystems can be found in the appendix tables.

3.2. Model Generation

Several regression models were developed, to relate LOC to software metrics computed at the design stage. Basic diagnostics (Rawlings et al., 1998) (e.g., F-test, t-test, statistical significance, distribution of residuals, quantile-quantile plots), were used to assess all models. In every case, very strong evidence was obtained that the slope of the model was non-zero. At standard significance level (i.e., 5% and 10%), intercepts were never significantly different from zero (intercept achieved significance level between 0.16 and 0.87 and model p-value between $1e-5$ and $1e-10$), so all of the models described in this paper are forced to pass through the origin.

Models with a small number of independent variables were preferred, for ease of use and interpretation, and because the number of data points is still not large.

In a previous paper (Antoniol et al., 1999) we compared robust regression techniques with the more traditional least squares line fitting. Our experience is that robust models do not gain much in explaining the data better. More is gained by removing or alleviating influential factors. Thus, only simple regression models were investigated here.

The metrics[5] that were investigated for predicting size were OOFP; OOFP related quantities (raw counts of RETs and DETs); and OO measures such as the total numbers of classes, methods, associations, and inheritances in an object model.

The four different ILF identification strategies summarized above give rise to four different OOFP measures for each data point. We found previously (Antoniol et al., 1999) that the GB strategy consistently gave slightly better predictors of LOC than the other strategies. We found the same again here in every case, so in the rest of this paper only the OOFP_{GB} value is reported.

3.3. Model Assessment

A leave-one-out cross-validation procedure (Stone, 1974) was used to measure model performance: each given model was trained on $n-1$ points of the data set L (sample size $n = 29$) and accuracy tested on the withheld datum. The step was repeated for each point in L and accuracy measures averaged over n. This methodology gives an unbiased estimate of future performance on novel data and it is thus indicated in the design of predictive models. Moreover, different choices of model class, or parameter tuning, data preprocessing choices become directly comparable.

Different measures of the error were considered: the normalized mean squared error (NMSE), the mean magnitude of relative error (MMRE) and the prediction level(s) (PL).[6] The cross-validation estimates of the standard error of the residuals $y_k - \hat{y}_k$ (σ_{err} in the following) and of average R^2 ($\overline{R}^2$ in the following) were also computed. NMSE, MMRE, and $\overline{R}^2$ are reported as percentage values in the following sections.

3.4. Sequence of Models

Models were investigated in four stages.

At each stage, several different models were developed, to predict LOC from different combinations of independent variables. First OOFP was used as the predictor; then various OO quantities; then the basic OOFP quantities (RETs and DETs).

To begin with, the baseline data[7] was analyzed. Results are presented in Section 4.1.

As outlier values were observed, code and design inspections, plus interviews with the programmers, were used to verify and validate the data. From these we recognized that several common practices of designers and developers may have a

strong influence on the collected software metrics (details can be found in Section 5). A series of refinements of the data was used to account for the effect of these factors. Three stages of refinements were used. In summary, we adopted a twofold approach: develop simple models, and whenever possible treat outliers by refining the data. Results are presented in Section 4.2.

4. Experimental Results

4.1. Baseline Models

Recall that we map FP to OO by relating files and transactions to classes and operations (methods). The first models that we investigated relate classes and methods directly to LOC.[8] If they predict LOC well, FP could be used directly for OO software.

Table 1 shows that models based directly on classes and methods are less accurate (i.e., NMSE and MMRE are worse) than models based on $OOFP_{GB}$. This is not surprising: predictions based simply on class and/or method counts are over-simplified, in that they do not take account of associations and inheritance, both of which are recognized in OOFP.

Given the current state of the art in empirical software engineering, the traditional interpretation of MMRE and PL is that a model is considered to be "good" if MMRE is 25% or less and PL (for $L = 25$) is at least 75% of the time (Conte et al., 1986). A model is "acceptable" if MMRE is 30% or less and PL (for $L = 30$) is at least 70% of the time (Conte and Campbell, 1989). But these values are rarely achieved; indeed, some consider an average error of 100% "good" and an average error of 30% "outstanding" (Vicinanza et al., 1991).

Even so, despite high values of $\overline{R}^2$ which indicate a strong association between each predictor and LOC, the results in Table 1 are not sufficiently accurate to be useful in practice. An average relative error of almost 100% leaves a lot of room for improvement.

The first improvement that we investigated was to add associations and inheritances as additional predictors, along with classes and methods. This is the classes, methods, inheritances and associations predictor (also referred to below as the "OO entities" predictor for short) in Table 2; it predicts LOC from four

Table 1. Performance of models based on classes, methods, OOFP.

Explanatory variables	NMSE	NMRE	$\overline{R}^2$	σ_{err}
Classes	52	99	81	8353
Methods	58	98	73	8768
Classes, methods	44	98	84	7667
$OOFP_{GB}$	38	88	82	7103

Table 2. Performance of models based on OO entities and OOFP elements.

Explanatory variables	NMSE	MMRE	$\overline{R}^2$	σ_{err}
Classes, methods, inheritances and associations	40	107	92	7278
OOFP elements	20	64	91	5152
DETs	20	54	91	5152

independent variables: the numbers of classes, methods, associations, and generalizations.

This model performs better than the simpler model that just uses classes and methods. Although MMRE is higher, NMSE, $\overline{R}^2$ and σ_{err} all improve. It is clear that associations and generalizations must be considered to obtain accurate size estimates.

For our next alternative, we looked at the way OOFPs are calculated. After the DETs and RETs are counted, OOFP values are found using tables that were taken from the IFPUG *Counting Practices Manual*. Those tables were not developed with regard to OO technologies. If we remove the table weighting steps, we might be able to predict LOC directly from the OOFP primary measures. In other words, instead of using DETs and RETs to determine complexity, complexity to determine OOFPs, and OOFPs to predict LOC, we might be able to predict LOC directly from DETs and RETs. This is the OOFP elements predictor in Table 2; LOC is predicted from three independent variables: ILF DETs and RETs, and SR DETs (SR FTRs are not considered, as they are almost always zero). The ILF RETs coefficient has a low significance level (p-value 0.28) and it may be dropped. Removing the ILF RETs term slightly improved the average model predictive capability and positively affects prediction levels.

From Tables 1 and 2 it is clear that LOC is predicted better from raw OOFP elements than from OOFP_{GB}. NMSE, MMRE, and σ_{err} are all much better. Thus, in their present form, the FP weighting tables do not seem to contribute usefully to accurate predictions of LOC for OO software; they even appear counter-productive. (This does not mean that they have no place; they might be valuable if recalibrated, and they may be useful for predicting other things than size.)

Figure 3 shows PL^9 for different levels of L, for each of the predictors discussed so far. There appears to be an order from OO entities to DETs. OOFP elements and DETs appear best for any level. OO entities attain the lower predictive capability. The best model so far is based on ILF DETs and SR DETs, with the GB strategy being used to identify ILFs.

4.2. Data Refinement Strategies

The measurements analyzed in Section 4.1 were obtained from the final design model and final system code. These include some extra classes that were not in the original design model, and some extra code that the programmers did not write. Removing

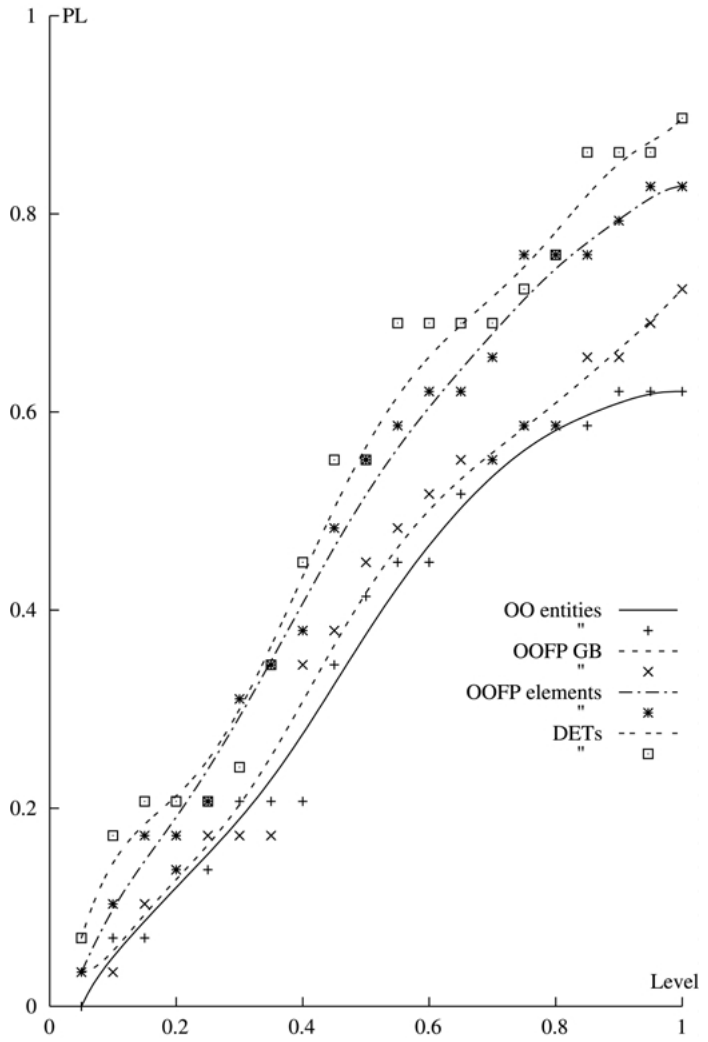

Figure 3. PL vs L for different predictors.

these extra classes and code gets us back to the design model from which an estimate would be made, and to the code written by the programmers. We need to do this, since our aim is to be able to estimate from a design model the size of the code that a programmer will write.

In this section we describe the steps taken to understand which subsystems have a particular influence on our results, and to allow for factors that introduce a distance between a design and its implementation.

The following steps were performed: visual inspection of the data; influential statistics study; traceability analysis and code/design inspection/reading; interviews

with programmers. These steps were iterated over the subsystems, until designers and programmers considered the design to code traceability mapping to be satisfactory and representative of the company's actual practice.

Visual inspection was performed on the data; scatter plots for several (x, y) choices (e.g. OOFP_{GB} vs LOC) of the data superimposed with the linear model studied to identify potential influential points. Common to the scatter plots were points apart from the line representing the linear model: dropping X1 and X6 from the data set consistently improved our models. To verify that X1, X6 were points on the fringes of the cloud of sample points in X-spaces, influential statistics were considered. Cook's distance, DFFITS, DFBETAS and COVRATIO (Rawlings et al., 1998) were computed, confirming the above conjecture concerning the existence of influential points.

The first eight subsystems were examined first: A traceability mapping between design and code (Antoniol et al., 2000; Fiutem and Antoniol, 1998) was constructed. The result was considered worth further investigation in that a high distance between design and code was discovered (141% of added classes, 43% of deleted classes and 57% of common classes).

By comparing code with design, and talking to the programmers, we identified some important factors that are likely to introduce a distance between a design and its implementation:

- *COTS*. The use of components off the shelf (COTS) may introduce classes in source code which are not modeled in design.

- *Library code*. Library classes are not necessarily reported in design. They may be reported to highlight subclassing, but the use of a library class is not regularly documented in design.

- *Middleware*. Like COTS, middleware software layers, e.g. CORBA, introduce new classes in code as a result of the process of stub generation. Moreover, include files containing CORBA classes, used by components which need remote object communication, introduce new classes in source code.

- *Automatically generated components*. Components of user interfaces are often generated automatically through graphical user interface builders, usually making heavy use of libraries. Since most of the code is automatically produced, only a rough design is necessary.

- *Application environment*. Quite often, extra classes were added to support interaction between applications, interfaced databases and third-party or legacy systems. In other words, the application context is modeled.

In our data set, automatically generated components, including test drivers and stubs, created extra code in eight subsystems. Extra classes were added to the original design (to clarify the application environment, the reuse of corporate components or

Table 3. Test, automatically generated code size and related classes.

Subsystem	Test LOC	Generated LOC	Extra classes
X1	4457	11,670	6
X6	1742	9135	11
X5	—	8153	—
X7	3678	—	—
X2	2905	—	—
X4	1584	—	8
X3	713	—	6
X8	258	—	0

the adoption of middleware and COTS) in four subsystems. Table 3 summarizes the effect on those subsystems.

COTS and automatically generated code cause another phenomenon: two designs (X3, Z3) contained no methods. We further observed that there were designs with either no ILF DETs, ILF RETs or SR DETs. Programmers confirmed that these underspecified subsystems were interfaces to a commercial database, to libraries handling and dispatching errors or to COTS tracing and logging system events. For example, classes were derived from the database connector with no need to override methods. All these subsystems rely on COTS and automatically generated code with a minimal designer and programmer intervention.

It is interesting that the influential factors we identified are all connected to rapid development approaches, such as the presence of reused or automatically generated code, and to the style used in design which may tend to over-specify or under-specify a software system. We believe that these factors are likely to become even more important for the software industry due to the need of mass software production.

To take account of these factors, a sequence of three new data sets was derived from the baseline data shown in Tables 6 and 7:

- *Revised LOC*. Automatically generated code, stubs and test code were removed from each component's source code, recomputing the corresponding LOC value.

- *Revised components*. Design classes introduced to clarify the application context (e.g. lib_ftp_a a standard ftp library) were then removed.

- *Revised Design*. Subsystems X3, Z1, Z2, Z3, Z4, Z5, Z9 corresponding to underspecified designs were removed from the final data set, under the hypothesis that they were special purpose components, not representative of the company's normal practices.

The final data set is presented in Tables 8 and 9 (in Appendix A). Data is presented for all 29 projects, but the removed subsystems are separated in the tables.

Table 4 reports the performance of the main models of interest ($OOFP_{GB}$, OO

Table 4. Comparison of model performance for the sequence of revised data sets.

	$OOFP_{GB}$			OO entities		
	NMSE	MMRE	σ_{err}	NMSE	MMRE	σ_{err}
Baseline system	38	99	7103	40	107	7278
Rev. LOC	34	67	4898	23	64	4035
Rev. components	29	52	5168	18	51	4117
Rev. design	35	54	5319	32	45	5068
	OOFP elements			DETs		
	NMSE	MMRE	σ_{err}	NMSE	MMRE	σ_{err}
Baseline system	20	64	5152	20	54	5152
Rev. LOC	15	43	3214	16	38	3270
Rev. components	18	43	4043	16	40	3673
Rev. design	15	27	3459	15	25	3530

entities (classes, methods, associations and inheritances), OOFP elements (ILF DET, ILF RET and SR DET), and DETs (ILF DET, SR DET) on each data set.

MMRE for each model tends to improve with each revision of the data. The greatest improvement is seen with the change from the baseline data set to the Revised LOC data set. Removing automatically generated code has the biggest impact; removing extra classes and atypical subsystems helps further, but the gain is not as great. NMSE, $\overline{R}^2$, and σ_{err} also improve a lot with the change to the Revised LOC data set, but further revisions to the data set do not help those measures.

The models based on OOFP elements and DETs are consistently better than the other two models. The model based on DETs appears best over all.

On the final data set, the model based on DETs achieves an NMSE of 15% (thus, the cross validation variance of the mean square error is significantly smaller than the sample variance). From a different point of view, the model has an MMRE of 25% and PL (for $L = 25\%$) of 73% which in industry terms may be considered satisfactory (Vicinanza et al., 1991).

4.3. Comparing the Models

Table 4 suggests that (in this data set) models based on OOFP elements or DETS are consistently better than models based on $OOFP_{GB}$ or OO entities. To verify this hypothesis, a non-parametric bootstrap approach (Efron and Tibshirani, 1993) was used to evaluate the statistical significance of differences between the models (using the revised design data set). The achieved significance level (ASL) of the test represents the probability of the observed difference between errors from two models, when the null hypothesis (that there are no differences between the errors from two given models) is true.[10]

Table 5. Significance levels for comparisons between models.

Base model	Alternative model	ASL
$OOFP_{GB}$	OO entities	0.020
$OOFP_{GB}$	OOFP elements	0.021
$OOFP_{GB}$	DETs	0.019
OO entities	OOFP elements	0.425
OO entities	DETs	0.19
DETs	OOFP elements	0.25

Table 5 presents the ASL levels for comparisons between the models (using 1500 bootstrap replications).

- The null hypothesis, that there are no differences between the errors from $OOFP_{GB}$ and OO entities, can be rejected with high statistical significance. The model based on OO entities is clearly better than the model based on $OOFP_{GB}$.

- Similarly, the models based on OOFP elements or DETs are clearly better than the model based on $OOFP_{GB}$.

- There is no statistically significant difference between the errors from the models based on OOFP elements or DETs, and OO entities.

Further tests (the non-parametric bootstrap BC_a and ABC methods (Efron and Tibshirani, 1993), and a paired T-test (Khoshgoftaar et al., 1996) on the cross-validation residual square errors) confirmed the results of Table 5.

Although the model based on DETs (or OOFP elements) appears to perform slightly better than the model based on OO entities, on the available data there is no proven reason to prefer one of these models over the other. However, the sample size is small so it pays to be cautious when drawing conclusions.

5. Lessons Learned

5.1. Size of Data Sample

In previous size estimation research with a smaller data set (Antoniol et al., 1999), the best NMSE obtained with the $OOFP_{GB}$ predictor was 34% using a robust logistic regression, and 39% using a simple regression. With a larger data set, we find now the same NMSE using a simple regression.

This improvement is not surprising, since it is well known that there is no better data than more data.

Moreover, a larger data set has enabled us to identify influential factors, and to obtain a better understanding of how to estimate size accurately.

5.2. *Influential Factors*

The aim to reduce software development cycles leads to the adoption of various strategies and development methodologies.

Some of these can lead to differences between the documented design and the final code, to the point that the design is no longer an abstraction of the code.

This distance between design and code can greatly influence the accuracy of size estimation methods. Factors that cause the distance must be allowed for to increase estimation accuracy.

We noted the use of COTS, libraries, reused code, middleware and automatic code generators. We categorized these into three factors that influence size estimation: mis-counted code (automatically generated code, and test code), over-specified design, and under-specified design. By controlling these factors, we were able to greatly improve estimation accuracy.

Removing the effect of influential factors has a larger impact than using different regression techniques. Previously we reduced NMSE from 0.39 to 0.34 by using robust regression techniques (Antoniol et al., 1999). In relative terms this is a reduction of only about 13%. Compare this with the 25% reduction in NMSE (from 0.20 to 0.15—see Table 4) and 54% reduction in MMRE (from 0.54 to 0.25) by removing the influential factors.

We discovered that these factors were important through *a posteriori* analysis of the data. Now that we know they are important, and knowing their effect, we can allow for them in future. Indeed, at design stage, those factors are very likely to be known and thus quantifiable; as we experienced, designers add meta-classes to represent COTS, middleware, and reused code.

More broadly, there is a clear link between the accuracy of size estimates and how closely a design model is an abstraction of what is actually present in the code. The closer the fit, the better the estimate. In effect, we can estimate an "inherent" size from a design model, and then adjust it to allow for "corrupting" factors.

The influential factors (e.g. COTS, reuse, code generators) are generally related to implementation strategies to reduce effort. Instead of adjusting the size estimate to allow for the influential factors, it may be best to treat those factors as productivity drivers in models for estimating effort.

5.3. *Size Estimation using OOFP*

Empirical evidence here shows that the stage of OOFP calculation that uses FP weighting tables decreases the value of OOFP as a size predictor.

We discovered that a model using raw counts of OOFP elements (DET_{ILF}, RET_{ILF}, and DET_{SR}) or the DETs (DET_{ILF} and DET_{SR}) as independent variables outperforms models using OOFP_{GB} (which in turn outperforms the other three OOFP counting strategies).

The weighting tables that we adopted were based on those from the IFPUG *Counting Practices Manual* (IFPUG, 1994), in which quite large numbers of RETs

and DETs are needed to reach average or high complexity (for example, to obtain an average complexity weight an ILF needs a DET value between 20 and 50 and a RET value between 2 and 5).

It is possible that the weighting tables might be valuable if they were recalibrated—for example, needing fewer DETs and RETs to be judged as having "average" or "high" complexity. (Research is in progress on just this point.) A large gain in estimation accuracy would be needed to make them worthwhile, however.

5.4. Size Estimation using Traditional OO Entities

We have seen that traditional aspects of object models such as classes, methods, inheritances and association may also be exploited to develop accurate size predictors.

Tables 1 and 2 clearly show that inheritance and associations must be counted to obtain accurate size estimates.

Our data support the hypothesis that reuse by inheritance influences size estimation: the coefficients of the inheritance counts were always negative.

It makes sense that incorporating associations also improved the models. The presence of an association means that a class may delegate to other classes; code must be written to implement the association, preparing parameters and invoking methods.

We found no benefit if the number of attributes or the number of aggregations were added as a further independent variable to the OO entities model.

5.5. Research Questions

As noted in Section 1, our aim was to understand which factors contribute to accurate size estimation for OO software, and to position OOFP within that knowledge. More specifically, we sought to determine the accuracy of size estimates using OOFP, on a larger data set than previously studied; and to compare it with estimates based on more traditional aspects of OO software.

5.5.1. Prediction Accuracy

The best model we found, which uses DETs as a predictor, performs well. NMSE is 0.15, meaning that the variance of the mean square error is much smaller than the sample variance. From a different point of view, the model has a predictive cross validation average error of 25%.

In a world where an average error of 100% can be considered "good" and an average error of 30% "outstanding" (Vicinanza et al., 1991), this is very good. It is worth noting, that the component sizes (LOCs, number of classes, methods, inheritances) are spread fairly evenly across a broad range; our dataset comprises a high variety of components: user interfaces, database connector, component to handle the phone call routing, etc. Other researchers (Abran and Robillard, 1993) have obtained comparable results, but on a much more homogeneous set of projects. Moreover, when compared with traditional function points, our approach is not tied to the subjectivity of adjusting factors and manual counting.

For this organization, models based on traditional OO entities perform almost as well in terms of NMSE and σ_{err}. Differences between models based on OOFP elements and OO entities or DETs are not statistically significant. Other organizations may find differently.

In practical terms, errors of plus/minus 25% are still large enough to cause problems. Practitioners would prefer estimates to be accurate to within 5% or perhaps 10%. That is difficult to achieve, in a world of volatile requirements and rapidly changing technology. It requires detailed local experience and understanding (stable and well understood requirements, process, and environment), and knowledge of which factors can cause how much variation. (Alternatively, it requires the freedom to tailor the project, adjusting what is delivered, and how, in order to fit the project to the estimate.) Nonetheless, improving the accuracy of our estimates remains a target for us.

5.5.2. Which Factors Predict Size?

In models based directly on counts of OO entities, the factors we found to contribute to size estimation were classes, methods, associations, and inheritance. These entities at least must be considered for accurate size estimation. Including attributes did not help.

Extra information is considered in the OOFP process. To count OOFP elements one must take into account classes, attributes, methods, method arguments, and associations. Inheritance and aggregations may be considered when identifying the logical files—classes or groups of classes—to count.

It is unclear whether the extra information captured within OOFP aids size estimation. The apparent superiority of the OOFP elements model is not proven statistically, and may be an accident of the data.

5.5.3. What Role for OOFP?

If there is no reason to prefer a model based on OOFP over a model that uses more traditional OO entities, it might seem that the OOFP approach is not needed at all. Further research is still needed to answer this question.

It certainly appears that the full OOFP measurement process is not needed. Size estimates based on raw counts of OOFP elements are significantly more accurate, in this organization.

For this organization it does appear that the best size estimates are obtained from the DETs model, but the superiority of this model is not proven statistically. This may change with further data, and other organizations may find differently.

6. Summary and Future Work

We have presented a family of models for estimating the size of object oriented software once design artifacts are available. In previous work (Antoniol et al., 1999), we have shown how FP concepts could be applied to OO software.

Models were developed based on the FP analogy as well as on traditional aspects of object models. Validation of the models was performed in an industrial environment.

The empirical validation results have been reported. The results show promise for size estimation. This is important, since an estimate of size is needed for many effort estimation models.

On the data available to us so far, the best performance appears to be obtained with a model based on ILF DET and SR DET, although its superiority is not statistically significant once compared with a model based on OO entities (classes, methods, inheritances and associations). Further experimentation is needed, with data from more systems, to statistically evaluate the difference, if any, among the proposed models.

Perhaps the most valuable result of our study is not the set of models, but a methodology able to consistently improve model performance. Indeed, during our validation, many factors were discovered that may affect the performance of the size prediction model.

The influential factors we identified are all connected to rapid development approaches, such as the presence of reused or automatically generated code, and to the style used in design which may tend to over-specify or under-specify a software system. We believe that these factors are likely to become even more important for the software industry due to the need of mass software production.

We have shown that by removing influential factors we can obtain accurate results that are useful in an industrial environment.

Future work will take several directions. More data will be collected to further validate our approach, the refinement process and the prediction models. Furthermore, we need to consider the impact of using design patterns (Gamma et al., 1995) on the structure within object models and the influence of software reuse.

Appendix A: Tables

Table A1. Baseline measurements of analyzed subsystems. From left to right: the name of the subsystem (system name and subsystem number), the total LOCs, DETs, RETs, service request DETs, OOFP value from each counting strategy.

Subsystem	LOC	OOFP elements			OOFP counting strategy			
		DET_{ILF}	RET_{ILF}	DET_{SR}	SC	MB	GB	AB
X1	40,522	55	38	322	932	858	925	865
X2	9074	7	13	114	445	424	431	438
X3	5539	10	9	0	63	35	35	63
X4	20,640	85	29	102	533	470	519	484
X5	26,702	84	35	108	515	365	480	400
X6	47,579	135	113	266	1884	1528	1663	1738
X7	18,808	14	17	370	1004	976	990	990
X8	13,904	2	7	85	268	254	254	268
Y1	10,481	10	32	283	998	900	900	984
Y2	7972	42	24	177	579	523	523	579
Y3	6942	5	23	165	720	664	671	685
Y4	6735	8	14	118	453	411	425	439
Z1	2282	0	1	35	56	56	56	56
Z2	787	2	3	0	45	45	45	45
Z3	10,577	9	12	0	84	41	73	52
Z4	4304	18	18	0	245	231	231	245
Z5	437	5	6	0	102	81	88	95
Z6	17,904	26	12	431	627	620	620	627
Z7	8182	10	7	198	328	321	321	328
Z8	8401	2	9	88	198	184	184	198
Z9	2458	0	2	0	83	83	83	83
Z10	1923	7	9	29	111	83	104	90
Z11	1699	13	14	11	167	107	160	114
Z12	6948	18	17	122	532	451	518	465
W1	1043	2	3	36	93	86	86	93
W2	1965	8	9	23	171	157	157	171
W3	2615	8	22	68	466	389	410	438
W4	1806	6	7	25	154	140	140	154
W5	1287	5	10	26	151	123	144	130

Table A2. Baseline measurements of analyzed subsystems. From left to right: the name of the subsystem (system name and subsystem number), the total LOCs, numbers of classes, methods, associations, aggregations, inheritances.

Subsystem	LOC	Design characteristics				
		Cls	Meth.	Assoc.	Aggreg.	Inh.
X1	40,522	38	222	38	10	1
X2	9074	13	118	6	1	8
X3	5539	9	0	10	0	4
X4	20,640	29	110	78	7	6
X5	26,702	35	90	62	22	5
X6	47,579	113	364	112	23	70
X7	18,808	17	295	12	2	9
X8	13,904	7	73	2	0	5

Table A2. Continued.

Subsystem	LOC	Design characteristics				
		Cls	Meth.	Assoc.	Aggreg.	Inh.
Y1	10,481	29	265	8	2	21
Y2	7972	24	137	42	0	16
Y3	6942	18	198	0	5	12
Y4	6735	12	123	6	2	6
Z1	2282	1	16	0	0	0
Z2	787	3	8	2	0	0
Z3	10,577	12	0	0	9	3
Z4	4304	17	42	17	0	2
Z5	437	6	20	4	1	3
Z6	17,904	12	181	26	0	9
Z7	8182	7	93	10	0	5
Z8	8401	9	45	2	0	6
Z9	2458	2	23	0	0	0
Z10	1923	9	16	4	3	1
Z11	1699	14	23	4	9	1
Z12	6948	16	140	8	10	3
W1	1043	3	24	2	0	1
W2	1965	9	36	8	0	2
W3	2615	19	111	4	4	13
W4	1806	7	35	6	0	2
W5	1287	7	34	2	3	3

Table A3. Final measurements of analyzed subsystems. From left to right: the name of the subsystem (system name and subsystem number), the total LOCs, DETs, RETs, service request DETs, OOFP value from each counting strategy.

Subsystem	LOC	OOFP elements			OOFP counting strategy			
		DET_{ILF}	RET_{ILF}	DET_{SR}	SC	MB	GB	AB
X1	24,395	42	32	322	890	862	883	869
X2	6169	7	13	114	445	424	431	438
X4	19,056	71	21	90	429	380	422	387
X5	18,549	84	35	108	515	365	480	400
X6	36,702	124	102	266	1804	1476	1611	1665
X7	15,130	14	17	370	1004	976	990	990
X8	13,646	2	7	85	268	254	254	268
Y1	10,481	10	31	279	970	879	879	956
Y2	7972	42	23	177	569	520	520	569
Y3	6942	5	23	165	720	664	671	685
Y4	6735	6	11	118	432	397	411	418
Z6	17,904	26	12	431	627	620	620	627
Z7	8182	10	7	198	328	321	321	328
Z8	8401	2	7	88	184	177	177	184
Z10	1923	4	3	29	69	62	69	62
Z11	1699	9	7	11	118	86	118	86
Z12	6948	15	14	122	511	444	504	451
W1	1043	2	3	36	93	86	86	93
W2	1965	6	5	23	143	129	129	143
W3	2615	3	12	42	315	266	266	308
W4	1806	5	5	25	140	126	126	140
W5	1287	5	8	26	137	116	137	116

Table A3. Continued.

Subsystem	LOC	OOFP elements			OOFP counting strategy			
		DET_{ILF}	RET_{ILF}	DET_{SR}	SC	MB	GB	AB
X3	4826	5	3	0	21	21	21	21
Z1	2282	0	1	35	56	56	56	56
Z2	787	1	2	0	38	38	38	38
Z3	10,577	9	12	0	84	41	73	52
Z4	4304	16	12	0	203	203	203	203
Z5	437	5	5	0	83	69	76	76
Z9	2458	0	2	0	83	83	83	83

Table A4. Final measurements of analyzed subsystems. From left to right: the name of the subsystem (system name and subsystem number), the total LOCs, numbers of classes, methods, associations, aggregations, inheritances.

Subsystem	LOC	Design characteristics				
		Cls	Meth.	Assoc.	Aggreg.	Inh.
X1	24,395	32	222	26	9	1
X2	6169	13	118	6	1	8
X4	19,056	21	94	65	6	6
X5	18,549	35	90	62	22	5
X6	36,702	102	363	101	23	65
X7	15,130	17	295	12	2	9
X8	13,646	7	73	2	0	5
Y1	10,481	28	258	8	2	20
Y2	7972	23	136	42	0	14
Y3	6942	18	198	0	5	12
Y4	6735	9	123	4	2	5
Z6	17,904	12	181	26	0	9
Z7	8182	7	93	10	0	5
Z8	8401	7	45	2	0	5
Z10	1923	3	16	3	1	0
Z11	1699	7	23	4	5	0
Z12	6948	13	140	6	9	2
W1	1043	3	24	2	0	1
W2	1965	5	36	6	0	2
W3	2615	12	77	2	1	9
W4	1806	5	35	5	0	2
W5	1287	5	34	2	3	0
X3	4826	3	0	5	0	0
Z1	2282	1	16	0	0	0
Z2	787	2	8	1	0	0
Z3	10,577	12	0	0	9	3
Z4	4304	11	42	15	0	0
Z5	437	5	16	4	1	2
Z9	2458	2	23	0	0	0

Table A5. Model equations NMSE, MMRE and PL(MMRE).

Data set	Model	NMSE	MMRE	PL
Baseline	$\text{LOC} = 511.86 \times \text{Cls}$	0.52	0.99	0.72
	$\text{LOC} = 95.14 \times \text{Meth}$	0.58	0.82	0.66
	$\text{LOC} = 357.49 \times \text{Cls} + 35.79 \times \text{Meth}$	0.44	0.98	0.69
	$\text{LOC} = 25.687 \times \text{OOFP}_{GB}$	0.38	0.88	0.66
	$\text{LOC} = 208.278 \times \text{DET}_{ILF} + 92.952 \times \text{RET}_{ILF} + 35.629 \times \text{DET}_{SR}$	0.20	0.64	0.62
	$\text{LOC} = 261.202 \times \text{DET}_{ILF} + 38.990 \times \text{DET}_{SR}$	0.20	0.54	0.69
	$\text{LOC} = 483.68 \times \text{Cls} + 44.27 \times \text{Meth} + 123.41 \times \text{Assoc} - 566.95 \times \text{Inh}$	0.40	1.07	0.66
Revised LOC	$\text{LOC} = 20.484 \times \text{OOFP}_{GB}$	0.34	0.67	0.69
	$\text{LOC} = 145.373 \times \text{DET}_{ILF} + 80.672 \times \text{RET}_{ILF} + 31.626 \times \text{DET}_{SR}$	0.15	0.43	0.59
	$\text{LOC} = 191.307 \times \text{DET}_{ILF} + 34.543 \times \text{DET}_{SR}$	0.16	0.38	0.55
	$\text{LOC} = 209.909 \times \text{Cls} + 42.998 \times \text{Meth} + 135.800 \times \text{Assoc} - 263.764 \times \text{Inh}$	0.23	0.64	0.62
Revised components	$\text{LOC} = 25.590 \times \text{OOFP}_{GB}$	0.29	0.52	0.69
	$\text{LOC} = 162.610 \times \text{DET}_{ILF} + 148.266 \times \text{RET}_{ILF} + 37.573 \times \text{DET}_{SR}$	0.18	0.43	0.62
	$\text{LOC} = 242.891 \times \text{DET}_{ILF} + 42.967 \times \text{DET}_{SR}$	0.16	0.40	0.55
	$\text{LOC} = 265.31 \times \text{Cls} + 57.72 \times \text{Meth} + 174.29 \times \text{Assoc} - 345.98 \times \text{Inh}$	0.18	0.51	0.66
Revised design	$\text{LOC} = 21.060 \times \text{OOFP}_{GB}$	0.35	0.54	0.59
	$\text{LOC} = 161.562 \times \text{DET}_{ILF} + 75.368 \times \text{RET}_{ILF} + 32.935 \times \text{DET}_{SR}$	0.15	0.27	0.68
	$\text{LOC} = 201.979 \times \text{DET}_{ILF} + 35.728 \times \text{DET}_{SR}$	0.15	0.25	0.73
	$\text{LOC} = 184.50 \times \text{Cls} + 47.78 \times \text{Meth} + 176.43 \times \text{Assoc} - 283.87 \times \text{Inh}$	0.32	0.45	0.64

Appendix B: Definitions

Measures for Evaluating Models

NMSE

Let y_k be a data point belonging to a set of observations of the dependent variable Y, and let $\hat{y}_k$ be its estimate. NMSE is the mean squared error normalized over the variance of the sample:

$$NMSE = \frac{\sum_{k \in L} (y_k - \hat{y}_k)^2}{\sum_{k \in L} (y_k - \mu_y)^2} \tag{1}$$

where $\mu_y = mean(Y)$ is the mean of the observed values in the sample L.

MMRE
The magnitude of relative error (MRE):

$$MRE_k = \frac{|y_k - \hat{y}_k|}{y_k} \tag{2}$$

when averaged over the n observations, gives raise to the MMRE:

$$MMRE = \frac{\sum_{k \in L} MRE_k}{n} \tag{3}$$

PL
PL, for a given percentage l, is defined as the percentage of the points such that $MRE_k \leq l$.

ASL
Let *Base Model* and *Alternative Model* be two models. A handy comparison statistic is:

$$\hat{\theta} = \frac{1}{n}[RSE(Base\ Model) - RSE(Alternative\ Model)] \tag{4}$$

where n is the number of observations, and *RSE* is the residual square error, the total squared difference between the predictions and the observations for the given model. The bootstrap replications allow one to estimate the $\hat{\theta}$ standard deviation. Having observed $\hat{\theta}$, the ASL of the test is defined to be the probability of observing at least that large a value when the null hypothesis (that there are no differences between the errors from two given models) is true. For a detailed explanation of the computation and the bootstrap theory see Efron and Tibshirani (1993).

Elementary Measures of OO Software

Size—LOC
LOCs were measured as the number of non-blank, non comment lines including pre-processor directives.

OOFP
OOFP represents the single final number that results from applying the full OOFP process. It involves identifying and counting the OOFP elements, classifying the complexity of each class and method in the object model, and allocating a number of OOFP's accordingly, and summing the individual OOFP values. Full details are given in Caldiera et al. (1998) and Antoniol et al. (1999).

OOFP elements
OOFP elements are the raw numbers of logical file DETS, logical file RETs, and method DETs in an object model. These are defined in Section 2.3.

OO entities
OO entities are the raw total numbers of classes (Cls), methods (Meth), associations (Assoc), aggregations (Aggreg), and inheritances (Inh) in an object model.

Notes

1. Abstract methods are not counted, and concrete methods are only counted once, in the class in which they are declared, even if they are inherited by several subclasses.
2. OOFP counting tools have been developed for the StP/OMT tool and for the C + + language.
3. LOCs were measured as the number of non-blank, non-comment lines including pre-processor directives.
4. Tables A1 and A2 (in Appendix A) show the measurements for each subsystem and its design.
5. See Appendix B for definitions.
6. See Appendix B for definitions.
7. Reported in Tables A1 and A2.
8. Details of these and all subsequent models, showing all coefficients, are summarized in Table A5 in Appendix A.
9. See Appendix B for definitions.
10. See Appendix B for details.

References

Abran, A., and Robillard, P. N. 1993. Reliability of function point productivity model for enhancement projects (a Field Study). In *Proceedings of IEEE International Conference on Software Maintenance*. Montreal, Quebec, Canada, pp. 134–142.

Albrecht, A. J. 1979. Measuring application development productivity. In *Proceedings of IBM Applications Development Symposium*, pp. 83–92.

Antoniol, G., Caprile, B., Potrich, A., and Tonella, P. 2000. Design-code traceability for object oriented systems. *The Annals of Software Engineering* 9: 35–58.

Antoniol, G., Lokan, C., Caldiera, G., and Fiutem, R. 1999. A function point-like measure for object oriented software. *Empirical Software Engineering* 4(3): 263–287.

Boehm, B. W. 1981. *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice-Hall.

Caldiera, G., Antoniol, G., Fiutem, R., and Lokan, C. 1998. Definition and experimental evaluation of function points for object-oriented systems. In *Proceedings of 5th International Symposium on Software Metrics*. pp. 167–178, IEEE.

Caldiera, G., Lokan, C., Antoniol, G., Fiutem, R., Curtis, S., La Commare, G., and Mambella, E. 1997. Estimating size and effort for object oriented systems. In *Proceedings of 4th Australian Conference on Software Metrics*.

Card, D., El Emam, K., and Scalzo, B. 2001. Measurement of object-oriented software development projects. Technical report, Software Productivity Consortium, Herndon VA.

Conte, S. D., and Campbell, R. L. 1989. A methodology for early software size estimation. Technical Report SERC-TR-33-P, Purdue University.

Conte, S. D., Dunsmore, H. E., and Shen, V. Y. 1986. *Software Engineering Metrics and Models*. Benjamin-Cummings.

DeMarco, T. 1982. *Controlling Software Projects*. Englewood Cliffs, NJ: Prentice Hall, Yourdon Press Computing Series.

Efron, B., and Tibshirani, R. J. 1993. *An Introduction to the Bootstrap*, Vol. 57 of *Monographs on Statistic and Applied Probability*. London: Chapman & Hall.

Fetcke, T., Abran, A., and Nguyen, T.-H. 1997. Mapping the OO-Jacobson approach to function point analysis. In *Proceedings of IFPUG 1997 Spring Conference*, pp. 134–142.

Fiutem, R., and Antoniol, G. 1998. Identifying design-code inconsistencies in object-oriented software: a case study. In *Proceedings of IEEE International Conference on Software Maintenance*. Bethesda MD, pp. 94–102.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley.

Graham, I. 1996. Making progress in metrics. *Object Magazine* 6(8): 68–73.

Hastings, T. 1995. Adapting function points to contemporary software systems: a review of proposals. In *Proceedings of 2nd Australian Conference on Software Metrics*. Australian Software Metrics Association.

IFPUG. 1994. *Function Point Counting Practices Manual, Release 4.0*. International Function Point Users Group, Westerville, Ohio.

IFPUG. 1995. *Function Point Counting Practices: Case Study 3—Object-Oriented Analysis, Object-Oriented Design Draft*. International Function Point Users Group, Westerville, Ohio.

Jacobson, I., Christerson, M., Jonsson, P., and Övergaard, G. 1992. *Object Oriented Software Engineering: A Use Case Driven Approach*. Reading, MA: Addison-Wesley.

Jeffery, D., and Stathis, J. 1996. Function point sizing: structure, validity and applicability. *Empirical Software Engineering* 1(1): 11–30.

Khoshgoftaar, T., Allan, B. E., Kalaichelval, K. S., and Goel, N. 1996. The impact of software evolution and reuse on software quality. *Empirical Software Engineering* 1(1): 31–44.

Kitchenham, B., and Känsälä, K. 1993. Inter-item correlations among function points. In *Proc. 15th International Conference on Software Engineering*. IEEE pp. 477–480.

Kitchenham, B., Pfleeger, S., and Fenton, N. 1995. Towards a framework for software measurement validation. *IEEE Transactions on Software Engineering* 21(12): 929–944.

Mehler, H., and Minkiewicz, A. 1997. Estimating size for object-oriented software. In *Proceedings of ASM'97 Applications in Software Measurement, Berlin*. Berlin.

Minkiewicz, A. 1997. Measuring object-oriented software with predictive object points. In *Proceedings of 8th European Software Control and Metrics Conference*. Atlanta.

Rawlings, J., Pandula, S. G., and Dickey, D. A. 1998. *Applied Regression Analysis a Research Tool*, Springer Texts in Statistics, second edition. New York: Springer-Verlag.

Schooneveldt, M. 1995. Measuring the size of object oriented systems. In *Proceedings of 2nd Australian Conference on Software Metrics*. Australian Software Metrics Association.

Sneed, H. 1996. Estimating the development costs of object-oriented software. In *Proceedings of 7th European Software Control and Metrics Conference*. Wilmslow, UK.

Stone, M. 1974. Cross-validatory choice and assesment of statistical predictio ns (with discussion). *Journal of the Royal Statistical Society B* 36: 111–147.

Verner, J., Tate, G., Jackson, B., and Hayward, R. 1989. Technology dependence in function point analysis: a case study and critical review. In *Proceedings of 11th International Conference on Software Engineering*, pp. 375–382.

Vicinanza, S., Mukhopadhyay, T., and Prietula, M. 1991. Software-effort estimation: an exploratory study of expert performance. *Information Systems Research* 2(4): 243–262.

Whitmire, S. 1993. Applying function points to object-oriented software models. In *Software Engineering Productivity Handbook*. New York, NY: McGraw-Hill, pp. 229–244.

Giuliano Antoniol received his doctoral degree in electronic engineering from the University of Padua in 1982. He worked at Irst for ten years were he led the the Irst Program Understanding and Reverse Engineering (PURE) Project team. He has published more than 60 papers in journals and international conferences and served as a member of the Program Committee of international conferences and workshops such as the International Conference on Software Maintenance, the International Workshop on Program Comprehension, the International Symposium on Software Metrics.

He is presently a member of the Editorial Board of the *Software Testing Verification & Reliability Journal*, the *Information and Software Technology Journal*, *Empirical Software Engineering* an international journal and the *Journal of Software Quality*. He is currently Associate Professor at the University of Sannio, Faculty of Engineering, where he works in the area of software metrics, process modeling, software evolution and maintenance.

Roberto Fiutem received the Laurea degree in electronic engineering from the Politecnico of Milano, Italy in 1988. Since 1989 he has been a researcher at the Istituto per la Ricerca Scientifica e Tecnologica (IRST), Trento, Italy, working in artificial intelligence and software engineering research projects. In 1998 he joined Sodalia S.p.A., a telecom software industry in Trento, where he worked on software engineering methodologies and tools, in particular in the object-oriented domain. He is now at IT Telecom S.p.A., the Information Technology company of Telecom Italia, where he is working mainly in EU funded research and development projects on telecommunications, in particular on IP quality of service and SLA management.

Chris Lokan is Head of the School of Computer Science at the University of New South Wales, Australian Defence Force Academy, where he has worked since 1987. He earned his doctorate in computer science from the Australian National University in 1985. His teaching focuses on software engineering. His research interests include software metrics, software size measurement and estimation, benchmarking, and object-oriented systems. He is a member of the ACM, IEEE Computer Society, Australian Software Metrics Association, and International Software Benchmarking Standards Group.