Adapting Function Points to Object Oriented Information Systems*

G. Antoniol[1], F. Calzolari[1], L. Cristoforetti[1], R. Fiutem[1] and G. Caldiera[2]

[1] I.T.C.-I.R.S.T., Via alla Cascata, I-38050 Povo (Trento), Italy
tel. +39 461 314-444
e-mail: antoniol,calzolar,cristofo,fiutem@irst.itc.it
[2] University of Maryland, Dept. of Computer Science,
College Park, Maryland 20742, USA
tel. +1 301 405-2707
e-mail: gcaldiera@cs.umd.edu

Abstract. The object oriented paradigm has become widely used to develop large information systems. This paper presents a method for estimating the size and effort of developing object oriented software. The approach is analogous to function points, and it is based on counting rules that pick up the elements in a static object model and combine them in order to produce a composite measure. Rules are proposed for counting "Object Oriented Function Points" from an object model, and several questions are identified for empirical research.
A key aspect of this method is its flexibility. An organization can experiment with different counting policies, to find the most accurate predictors of size, effort, etc. in its environment.
"Object Oriented Function Points" counting has been implemented in a Java tool, and results on size estimation obtained from a pilot project with an industrial partner are encouraging.

Keywords: Object oriented design metrics, function points, size estimation.

1 Introduction

Cost and effort estimation is an important aspect of the management of software development projects and it could be a critical point for complex information systems. Experience shows how difficult is to provide an accurate estimation: in literature [18] an average error of 100% is considered to be "good" and an average error of 32% to be "outstanding". Most research on estimating size and effort has dealt with traditional applications and traditional software development practices, while few works have been experimented for object oriented (OO) software development.

* This research was funded by SODALIA Spa, Trento, Italy under Contract n. 346 between SODALIA and Istituto Trentino di Cultura, Trento, Italy.

This paper presents a method for estimating the size and development effort of object oriented software, supported by a tool, implemented in Java. The proposed approach, that we call "Object Oriented Function Points" (OOFP), is based on an adaptation for object oriented paradigm of the classical Function Point (FP) methodology [2].
As shown in Figure 1, we will measure Object Oriented Function Points, and correlate them with actual system size and development effort to identify estimation models tailored for a specific environment. One of the advantages of this approach is that different estimation models can be developed for different stages of a software project, as soon as the software artifact becomes more detailed while the project goes on.
The *OOFP_Counter*, the Java tool that implements the proposed approach, provides a way to finely tune the counting rules by setting several parameters related to which counting policy is better suited for a given software project.

This paper is organized as follows: Section 2 explains how we map main concepts of function points to object oriented software. The rules for counting Object Oriented Function Points are then described in Section 3, with emphasis on different counting policies that can be adopted. Section 4 presents the *OOFP_Counter*, the tool developed to automatize the counting process. This tool has been used to produce results for an industrial pilot project, focused on size estimation, reported in Section 5. Finally, conclusions are drawn.

2 Object Oriented Function Points

Since they have been proposed in 1979 [1], function points (FP) have become a well known and widely used software metric. Despite some concerns [10, 11, 12, 17], practitioners have found FPs to be useful in the data processing domain, for which they were invented.
Function points are available at the specification phase since they are based on the user's view of software functionality. FPs are generally considered to be independent from the technology used to implement the solution. The key features of function points are that they are available early, and they are a measure of the problem independent from any particular implementation. The International Function Point Users Group (IFPUG) publishes guidelines to standardize their definition [6].

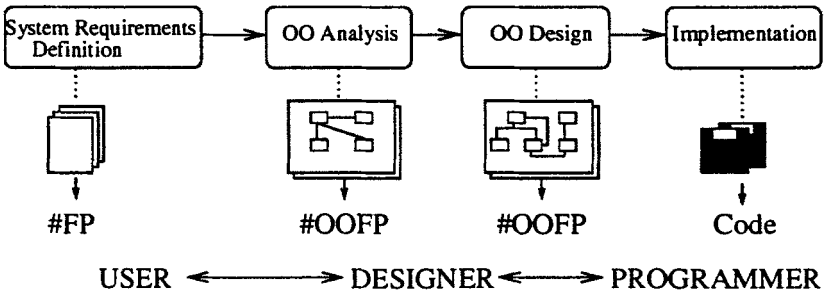

Fig. 1. Measures in the software development process.

Several variants have been proposed to extend FPs use to other domains (see [5] for a survey). Since OO paradigm had become widely adopted to design large information systems, different attempts have been proposed to adapt function points concepts to object oriented software, in order to exploit the understanding gained with function points in their traditional domain.

In the object oriented approach, an object model uses classes and their inter-relationships to represent the structure of a system. While the development proceeds the object model evolves: in addition to the problem-related classes, the model includes design- and implementation-oriented classes with new inheritance relationships. These changes do not concern the user, but reflects the developer's view of the system. A measure derived from the object model should be now a better predictor of development size and effort.

The OOFP approach enables a smooth transition from the user's view to the developer's view, and the same methodology can be used to measure the object model at each stage, as shown in Figure 1.

2.1 Mapping function points to object oriented software

Object model, dynamic model, and functional model may be used to represent information about object oriented software [14]. The object model is usually the first to be developed, and it is the only one that describes the system using specifically object-oriented concepts. We focus our attention to object model to map traditional FP concepts to OOFP, translating logical files and transactions to classes and methods. A Logical File (LF) in the function point approach is a collection of related user identifiable data. Since a class encapsulates a collection of data items,

it seems to be the natural candidate for mapping logical files into the OO paradigm. Objects that are instances of a class in the OO world correspond to records of a logical file in data processing applications.

In the FP method the application boundary identifies *Internal* Logical Files (ILFs) (logical files maintained by the application) and *External* Interface Files (EIFs) (referenced by the application but maintained by other applications). In the OO counterpart, we could consider *external* classes encapsulating non-system components, such as other applications, external services, and library functions. Classes within the application boundary correspond to ILFs. Classes outside the application boundary correspond to EIFs. In the OO paradigm operations are performed by methods (which are usually at a more fine-grained level than transactions). Since object models rarely contain the information needed to tell whether a method performs an input or an output or is dealing with an enquiry, we simply treat them as generic Service Requests (SRs), issued by objects to other objects to delegate some operations.

Issues such as inheritance and polymorphism affect the structure of the object model, and how the model should be counted. This problem will be addressed in Section 3.1.

2.2 Related work

Several authors have proposed methods for adapting function points to object oriented software. In [15] classes are treated as files, and services delivered by objects to clients as transactions, while in [19] each class is considered as an internal file, and messages sent across the system boundary are treated as transactions. Sneed [16] 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.

A draft proposal by IFPUG [7] treats classes as files, and methods as transactions. Fetcke [3] defines rules for mapping a "use case" model [9] to concepts from the IFPUG Counting Practices manual, but no attempt has been made to relate the results to other metrics, such as traditional function points, lines of code, or effort.

The key aspect of our approach is its flexibility. For example, Fetcke [3] defines that aggregation and inheritance should be handled in a particular way. We define several options (one of which is Fetcke's approach) and leave it to the user to experiment which parameter settings produce the most accurate predictors of size, effort, etc. in its environment. Thus we have a method which can be tailored to different organizations or

environments. Moreover, the measurement is not affected by subjective ratings of complexity factors, like those introduced in classical function point analysis.
Finally, the *OOFP_Counter* will automatically count OOFPs, for a given setting of parameters.

3 Measurement Process

OOFPs are assumed to be a function of objects comprised in a given object model D (D can be that produced at design stage or extracted from the source code) and they can be calculated as:

$$OOFP = OOFP_{ILF} + OOFP_{EIF} + OOFP_{SR}$$

where:

$$OOFP_{ILF} = \sum_{o \in A} W_{ILF}(DET_o, RET_o)$$

$$OOFP_{EIF} = \sum_{o \notin A} W_{ELF}(DET_o, RET_o)$$

$$OOFP_{SR} = \sum_{o \in A} W_{SR}(DET_o, FTR_o)$$

A denotes the set of objects belonging to the application considered and o is a generic object in D. Dets, Rets and Ftrs are elementary measures to be calculated on LFs and SRs and used to determine their complexity

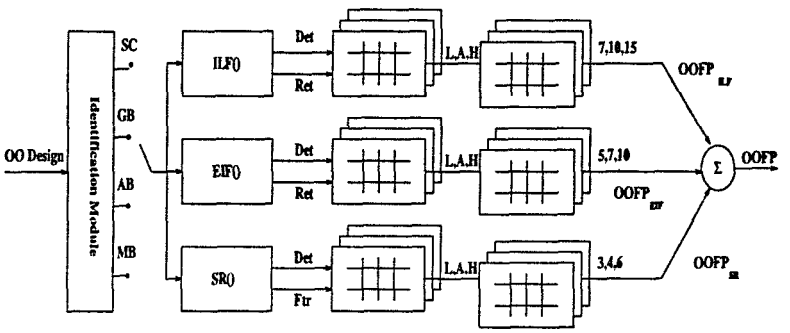

Fig. 2. OOFP computation process.

through the complexity matrixes W. Such meaasures are further detailed in Sections 3.2 and 3.3.
Counting OOFPs is a four steps process:

1. The object model is analyzed to identify the units that are to be counted as logical files.
2. The complexity of each logical file and service request is determined. Structural items are mapped to complexity levels of low, average, or high.
3. The complexity scores are translated into values.
4. The values are summed to produce the final OOFP result.

Figure 2 outlines the counting process. The counting rules used in these steps are described in Sections 3.1 to 3.3, while Section 4.1 explores the effect of counting classes in different ways.

3.1 Identifying logical files

Classes are generally mapped into logical files. However, relationships between classes (aggregations and generalization/specializations in particular) can sometimes require to count a group of classes as a single logical file. Different choices of how to deal with aggregations and generalization/specialization relationships lead to different ways to identify logical files. In what follows we are going to present the four different choices we identified: a simple example taken from [4] will support explanation.

1. **Single Class**: count each separate class as a logical file, regardless of its aggregation and inheritance relationships (Figure 3).
2. **Aggregations**: count an entire aggregation structure as a single logical file, recursively joining lower level aggregations (Figure 4).
3. **Generalization/Specialization**: 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, i.e. inheritance hierarchies are merged down to the leaves of the hierarchy (Figure 5).
4. **Mixed**: combination of option 2 and 3 (Figure 6).

Merging superclasses into subclasses makes intuitive sense. It seems right to count leaf classes, with their full inherited structure, since this is how they are instantiated.
Dividing a user-identifiable class into an aggregation of sub-classes is an implementation choice. Thus from the point of view of the function point

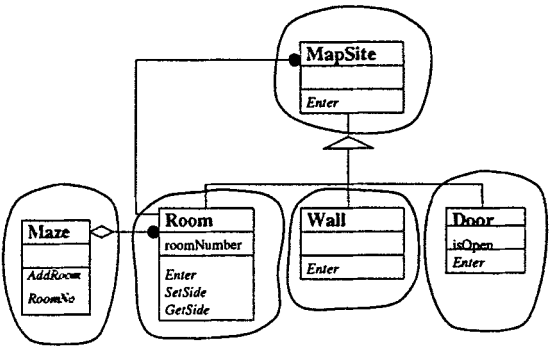

Fig. 3. Single class ILFs.

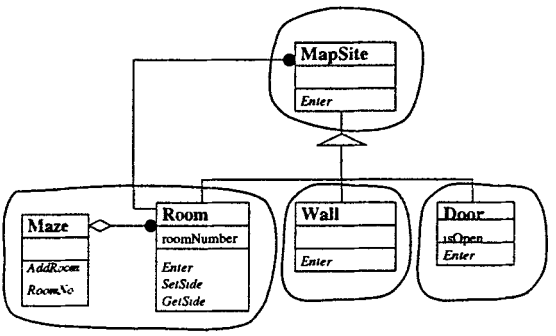

Fig. 4. Aggregations ILFs.

measurement philosophy, the OOFP value should not be affected. From this perspective, the aggregation structure should be merged into a single class and counted as a single logical file.
Merging aggregations or not seems to depend on whether the user's or designer's perspective is chosen. However, a hybrid solution can be adopted as well, flagging on the design which aggregations must be considered as a unique entity and thus must be merged.

3.2 Complexity of Logical Files

For each logical file it is necessary to compute the number of DETs (Data Element Types) and RETs (Record Element Types). Counting rules depend on whether it is a *simple* logical file, corresponding to a single class, or a *composite* logical file, corresponding to a set of classes.

For simple logical files:

- One RET is counted for the logical file as a whole, because it represents a "user recognizable group of logically related data" [6].

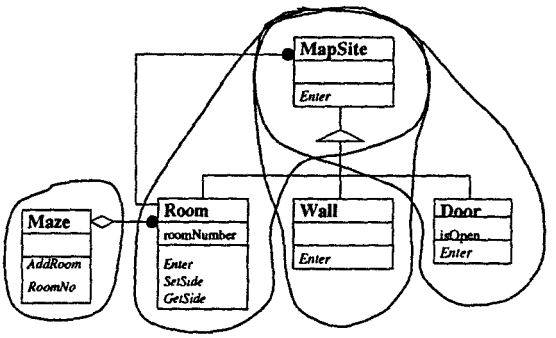

Fig. 5. Generalization/Specialization ILFs.

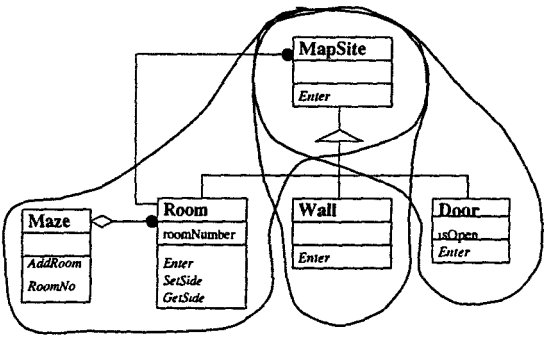

Fig. 6. Mixed ILFs.

- Simple attributes, such as integers and strings, are considered as DETs, as they are a "unique user recognizable, non-recursive field of the ILF or EIF" [6].
- Complex attributes are counted as RETs. A complex attribute is one whose type is a class (i.e. "a user recognizable subgroup of data elements within an ILF or EIF" [6]) or a reference to another class.
- A single-valued association is considered as a DET (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[6]).
- A multiple-valued association is considered as a RET, because an entire group of references to objects is maintained in one attribute.
- Aggregations are treated simply as associations.

For composite logical files:

- Using the rules for simple logical files, except for the handling of aggregations, DETs and RETs are counted separately for each class within the composite.
- In a composite logical file aggregations represent a subgroup. One RET, assigned to the container class, is counted for each aggregation, whatever its cardinality. One more RET is also counted for the logical file as a whole.
- The individual DETs and RETs are summed to give an overall total for the composite logical file.

When the DETs and RETs of a logical file have been counted, tables (derived from those given in the IFPUG Counting Practices Manual Release 4.0 [6] for ILFs and EIFs) are used to classify it as having low, average, or high complexity.

3.3 Complexity of Service Requests

Each method in each class is considered: abstract methods are not counted, while concrete methods are only counted once (in the class in which they are declared), even if they are inherited by several subclasses.
If a method is to be counted, the data types referenced in it are classified as *simple items* (analogous to DETs in traditional function points) for simple data items referenced as arguments of the method, and *complex items* (analogous to File Types Referenced (FTRs) in traditional function points) for complex arguments [2].
Again tables are used to classify the method as having low, average, or high complexity. Notice that sometimes the signature of the method provides the only information on DETs and FTRs. In such a case, the method is assumed to have average complexity.

3.4 An Example

The counting procedure for each individual class gives the DETs and RETs shown in Figure 7, while Table 1 shows ILF and SR contribution to OOFP counting. Since service requests (methods) are only counted once, it does not matter how the classes are aggregated into logical files. Because the signatures are unknown for the methods in the example, each method is assumed to have average complexity.

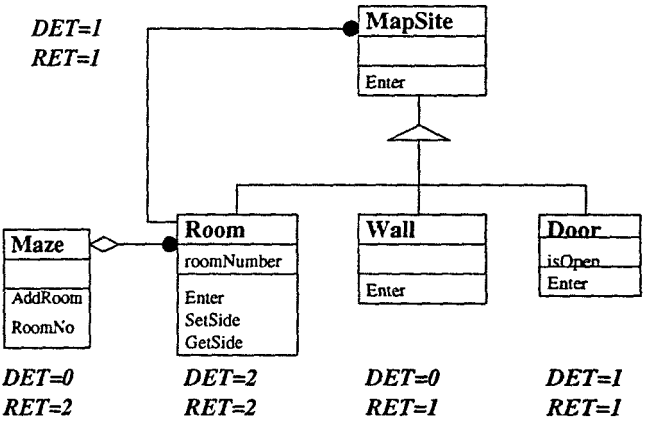

Fig. 7. DET/RET computation for LFs on the example system.

Values in third and fifth columns show the results of applying IFPUG 4.0 complexity tables with each variant. The value 7 is rated as *Low* and it is weighted 4. For more details about how counting rules have been applied the interested reader could refer to [2].

	ILF	ILF OOFP	SR	SR OOFP	*Total OOFP*
Single Class	5	35	7	28	63
Aggregation	4	28	7	28	56
Generalization/Specialization	4	28	7	28	56
Mixed	3	21	7	28	49

Table 1. ILF and SR complexity contribution.

The highest OOFP count comes when each class is counted as a single ILF. All the other variants have the effect of reducing the OOFP value, as they reduce the number of ILFs. Although there is an increase in DETs/RETs in the merged ILFs, it is not enough to raise the ILF complexity to higher values.

For this example, and for the pilot project that will be presented in Section 5, the complexity of each ILF and SR are always determined to be low. The tables used to determine complexity are based on those from the IFPUG Counting Practices Manual [6], 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). On the data

available to us so far, we suspect that recalibration of the OOFP tables for logical files might improve the accuracy of OOFP as a predictor of size, but further experimentation is needed on this topic.

4 The OOFP_Counter Tool

We have developed the *OOFP_Counter* tool, presented in Figure 8, to automate the OOFP counting process. This tool has been implemented using Java.
The *OOFP_Counter* inputs Abstract Object Language (AOL) specification of the object oriented model. AOL is a general-purpose design description language capable of expressing concepts of OO design. It has been adopted in order to keep the tool independent of the specific CASE tool used. AOL is based on the Unified Modeling Language [13], which represents *de facto* a standard in object oriented design.
The OOFP_Counter tool parses AOL specification and produces an abstract syntax tree representing the object model. The parser also resolves references to identifiers, and performs some simple consistency checking (e.g. names referenced in associations have been defined).
To improve portability, the AOL parser and the OOFP counter, the two parts of the OOFP_Counter tool have been implemented in Java.
For the project presented in Section 5, OMT/STP [8] has been used as CASE tool; an automatic translator to convert from OMT/STP output to AOL specifications has been implemented.

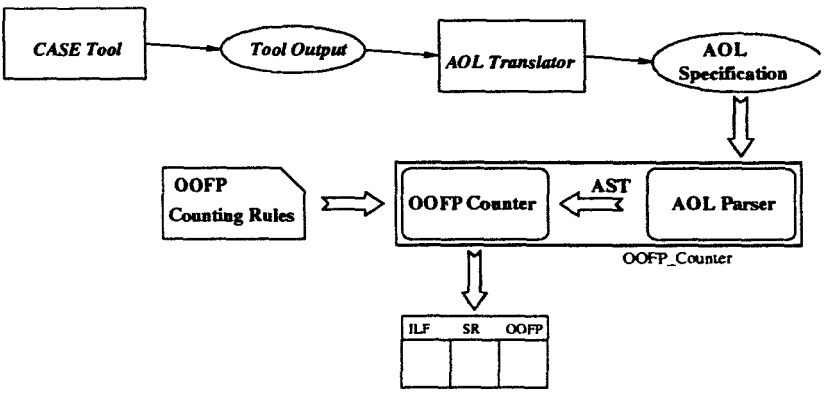

Fig. 8. The OOFP_Counter tool.

4.1 Parameters Setting

The OOFP_Counter works on the abstract syntax tree and implements the OOFP Counting Rules described in section 3. It is possible to set several parameters, that may influence the counting policy:

- ILF counting strategy (see Section 3.1)
- External classes inclusion
- Private methods counting;
- Private attributes counting;
- Values of DET, RET, and FTR thresholds between low, average, and high complexity.

Parameter setting might be guided by some philosophy. For example, from a traditional function point perspective one would wish to count only user-visible abstractions, ignoring all implementation aspects. This might mean selecting the Mixed strategy for grouping classes into logical files, counting only those methods which are publicly visible and related to classes at the system boundary, and giving full weight to classes whether they are reused or not.
From a designer's point of view, one might want to take account of all implementation details, in an attempt to get an accurate estimate of development effort. This might mean counting each class as a separate logical file, including all methods and attributes, and reducing the weight given to reused classes.
Different parameter settings could be tried on a purely experimental basis in order to identify that company specific profile that gives the best overall performance for estimating size or effort.

5 An Industrial Case Study

The described methodology has been applied in an industrial environment. Our first study is of the relationship between the OOFP measure of a system and its final size in lines of code (LOC), measured as the number of non-blank lines, including comments. Size estimation is important, since it is needed for most effort estimation models, thus we can make use of existing models that relate size to effort.
Eight completed (sub-)systems were measured, for which both an OO design model and the final code were available. All were developed in the same environment, using the C++ language. Table 2 shows the size of each system, spreading from about 5,000 to 50,000 lines of code.

Table 2 also shows the OOFP count for each system, using each of the four different strategies for identifying logical files.

System	LOC	Single Class (SC)	Aggregation (AB)	Generalization (GB)	Mixed (MB)
A	5089	63	63	35	35
B	6121	476	469	462	455
C	15031	284	284	270	270
D	16182	1071	1057	1057	1043
E	21335	562	513	548	499
F	31011	518	403	483	368
G	42044	1142	1100	1124	1072
H	52505	2093	1947	1872	1737

Table 2. System sizes and OOFPs.

The four OOFP series are strongly correlated each other, with all correlations within the .992 – .998 range (Pearson), the lowest corresponding to SC *vs* MB. As shown in Table 2, differences between the methods become appreciable only for the projects with large LOC values.
Several regression techniques were considered to model the LOC-OOFP association. Given the reduced size of the database, a leave-one-out cross-validation procedure was used to achieve unbiased estimates of predictive accuracy for the different models. Model error was expressed in terms of *normalized mean squared error* (NMSE): each model was trained on $n-1$ points of the data base L (sample size is currently $n = 8$) and tested on the withheld datum; NMSE is obtained over L normalizing over the sample variance of the observed values ($\mu_y = mean(y)$).
The small size of the database and a limited knowledge of LOC measures validity required the use of simple models capable to handle non obvious outliers in the response variable LOC. In this study, the basic least squares linear fit was compared with resistant techniques. Regression estimates based on least square minimization are in fact sensitive to outliers in the response variable when the error distribution is not Gaussian. Robust regression techniques may improve the least-squares fit and handle model inadequacies due to unusual observations.
First linear models (`lms`) based on the minimization of the sum of squares of the residuals were developed for each ILF selection method. Least absolute deviation, based on L_1 error was also applied (`l1s`) . The regressor is build minimizing the sum of the absolute values of the residuals to resist the effect of large error values.

Method	NMSE	NMAE	$\hat{R}^2$	b_0	b_1
lm-SC	0.391	0.661	0.730	7992.5	23.0
lm-SC-1	0.539	0.811	0.901	0000.0	29.4
lm-AB	0.434	0.656	0.691	8504.7	23.8
lm-GB	0.380	0.601	0.728	7435.1	25.2
lm-MB	0.464	0.681	0.680	8187.4	25.8
l1-SC	0.547	0.812	–	9139.1	21.58
l1-AB	0.629	0.855	–	8601.1	23.48
l1-GB	0.389	0.693	–	8688.4	24.36
l1-MB	0.457	0.734	–	8083.0	26.61
rreg-SC	0.399	0.672	–	7875.2	23.0
rreg-AB	0.431	0.661	–	8255.3	24.0
rreg-GB	0.368	0.599	–	7331.7	25.5
rreg-MB	0.443	0.664	–	7861.9	26.4
rlm-SC	0.402	0.670	–	8001.9	23.0
rlm-SC-1	0.633	0.860	–	0000.0	29.3
rlm-AB	0.440	0.660	–	8517.5	23.8
rlm-GB	0.377	0.600	–	7521.5	25.6
rlm-MB	0.456	0.676	–	8161.6	26.3

Table 3. Model performance for linear regressors (`lms` and `l1s`) and robustified methods (`rregs` and `rlms`). The normalized mean squared error (NMSE) and the normalized mean absolute error (NMAE) are estimated by cross-validation.

A family of M-estimators was therefore considered (`rregs` and `rlms`). The basic idea of M-smoothers is to control the influence of outliers by the use of a non-quadratic local loss function which gives less weight to "extreme" observations. Non-linear modelling was also attempted, expecting instability and lack of convergence due to the sample size.

Estimated model accuracy for each model $\hat{y} = b_0 + b_1 x$ of each experimented family is collected in Table 3, parametrized over ILF selection methods and type of regressor. The model coefficients b_0 and b_1 are indicated as computed from the full data set. Estimated R-squared measure is also included for the linear models for comparison with other results separately obtained on these data.

A point of concern is the inclusion of an intercept term b_0 in model: it is reasonable to suppose the existence of support code unreferred to

Method	NMSE	Comments
rreg-default-GB	0.368	–
rreg-andrews-GB	0.367	–
rreg-bisquare-GB	0.367	–
rreg-fair-GB	0.480	converged after 50 steps)
rreg-hampel-GB	0.381	–
rreg-huber-GB	0.378	–
rreg-logistic-GB	0.357	$c = 1.25$
rreg-logistic-GB-0.8	**0.337**	$c = 0.80$
rreg-talworth-GB	0.380	–
rreg-welsch-GB	0.380	–

Table 4. Model performances for different weighting functions of the M-estimator *rreg*. Results are given for the GB selection method only.

the functionalities being counted, and prediction is improved whith the term. However, the intercept term is not significant in a non-predictive fit of the data. More important, the fact that the intercept term is always larger than the first LOC value might indicate poor fit for small OOFP values. It would be interesting to apply a Bayesian procedure to select the intercept from given priors.

The estimates for different weighting functions of the M-estimator are listed in Table 4.

The best predictive accuracy (NMSE= 0.337) was achieved by the *rreg-logistic-GB* model with tuning parameter $u = .8$, corresponding to the linear predictor $LOC = 7183.4 + 25.6\ GB$.

As shown in Figure 9, the *rreg-logistic-GB* model is very close to the basic linear model *lm-GB*, whose equation is $LOC = 7435.1 + 25.2\ GB$.

As the GB method is consistently better for all models and for both the predictive error measures NMSE and NMAE, these results indicate that the choice of ILF selection method may influence prediction. Lowess, supersmoother and predictive splines have been also tested and showed instability of convergence due to the small sample size.

Although more experimental work is needed, obtained results are encouraging for size estimation.

6 Conclusions

This paper shows how the concepts of function points can be applied to object oriented software.
We presented a methodology for estimating the size and effort of object oriented software. The method is based on an adaptation of traditional function points to object oriented paradigm. Mapping from FP concepts to OO concepts have been defined, and the OOFPs counting process

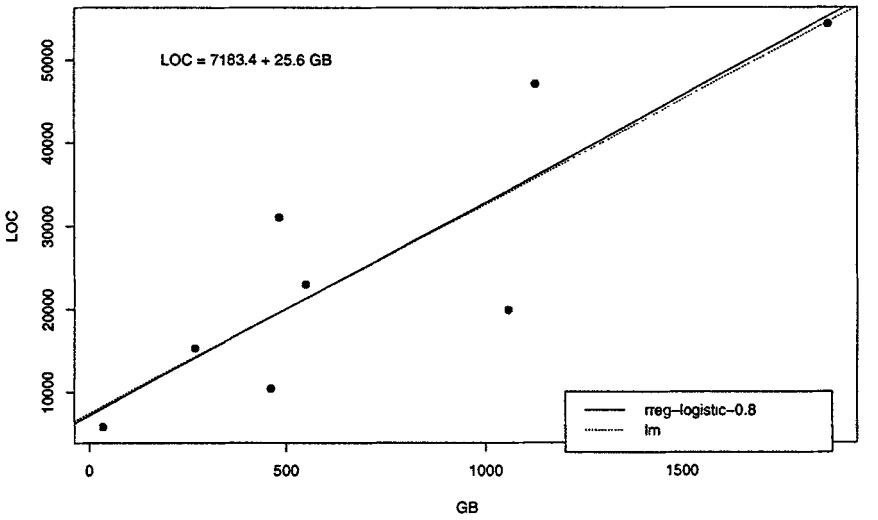

Fig. 9. The rreg-logistic-GB model (c=0.8) compared with the linear model lm-GB.

has been described. The OOFP_Counter tools has been developed to automate the counting process. Results obtained from a pilot study in an industrial environment have been reported.
The results for size estimation are encouraging, and they can be used with many effort estimation models.
Future work will investigate the effect of recalibrating the complexity tables and analyzing the statistical correlation between the collected measeres (DETs, RETs, FTRs) and program size. Other relationships, beyond just OOFP and code size, will be studied; those between OOFP and traditional FP, and OOFP versus effort, are of particular interest.

7 Acknowledgement

The authors are indebted with Cesare Furlanello who performed most of the statistical analysis in the pilot study.

References

1. A. J. Albrecht. Measuring application development productivity. In *Proc. IBM Applications Development Symposium*, pages 83–92. IBM, Oct. 1979.
2. G. Caldiera, C. Lokan, G. Antoniol, R. Fiutem, S. Curtis, G. L. Commare, and E. Mambella. Estimating Size and Effort for Object Oriented Systems. In *Proc. 4th Australian Conference on Software Metrics*, 1997.
3. T. Fetcke, A. Abran, and T.-H. Nguyen. Mapping the OO-Jacobson approach to function point analysis. In *Proc. IFPUG 1997 Spring Conference*, pages 134–142. IFPUG, Apr. 1997.
4. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software.* Addison-Wesley, 1995.
5. T. Hastings. Adapting function points to contemporary software systems: A review of proposals. In *Proc. 2nd Australian Conference on Software Metrics.* Australian Software Metrics Association, 1995.
6. IFPUG. *Function Point Counting Practices Manual, Release 4.0.* International Function Point Users Group, Westerville, Ohio, 1994.
7. IFPUG. *Function Point Counting Practices: Case Study 3 - Object-Oriented Analysis, Object-Oriented Design (Draft).* International Function Point Users Group, Westerville, Ohio, 1995.
8. Interactive Development Environments. *Software Through Pictures manuals*, 1996.
9. I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard. *Object Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley, 1992.
10. D. Jeffery and J. Stathis. Function point sizing: Structure, validity and applicability. *Empirical Software Engineering*, 1(1):11–30, 1996.
11. B. Kitchenham and K. Känsälä. Inter-item correlations among function points. In *Proc. 15th International Conference on Software Engineering*, pages 477–480. IEEE, May 1993.
12. B. Kitchenham, S. Pfleeger, and N. Fenton. Towards a framework for software measurement validation. *IEEE Transactions on Software Engineering*, 21(12):929–944, Dec. 1995.

13. Rational Software Corporation. *Unified Modeling Language, Version 1.0*, 1997.
14. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object Oriented Modelling and Design.* Prentice-Hall, 1991.
15. M. Schooneveldt. Measuring the size of object oriented systems. In *Proc. 2nd Australian Conference on Software Metrics.* Australian Software Metrics Association, 1995.
16. H. Sneed. Estimating the Costs of Object-Oriented Software. In *Proceedings of Software Cost Estimation Seminar*, 1995.
17. J. Verner, G. Tate, B. Jackson, and R. Hayward. Technology dependence in Function Point Analysis: a case study and critical review. In *Proc. 11th International Conference on Software Engineering*, pages 375–382. IEEE, 1989.
18. S. Vicinanza, T. Mukhopadhyay, and M. Prietula. Software-effort estimation: an exploratory study of expert performance. *Information Systems Research*, 2(4):243–262, Dec. 1991.
19. S. Whitmire. Applying function points to object-oriented software models. In *Software Engineering Productivity Handbook*, pages 229–244. McGraw-Hill, 1993.