

Definition and Experimental Evaluation of Function Points for Object-Oriented Systems

G. Caldiera

PricewaterhouseCoopers
12902 Federal Systems Park Dr
Fairfax, Virginia 22033, USA
(703) 633-4128
gcaldiera@cs.umd.edu

G. Antonioli,

R. Fiutem

ITC-IRST
Via alla Cascata
I-38050 Povo (Trento), Italy
+39 461 314444
antonioli,fiutem@irst.itc.it

C. Lokan

Dept of Computer Science
Australian Defence Force
Academy
Canberra ACT 2600, Australia
+61 2 6268 8060
c-lokan@adfa.edu.au

Abstract

This paper presents a method for estimating the size, and consequently effort and duration, of object oriented software development projects. Different estimates may be made in different phases of the development process, according to the available information. We define an adaptation of traditional function points, called “Object Oriented Function Points”, to enable the measurement of object oriented analysis and design specifications. Tools have been constructed to automate the counting method. The novel aspect of our method is its flexibility. An organisation can experiment with different counting policies, to find the most accurate predictors of size, effort, etc. in its environment. The method and preliminary results of its application in an industrial environment are presented and discussed.

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

1. Introduction

Cost and effort estimation is an important aspect of the management of software development projects. Experience shows that accurate estimation is difficult: an average error of 100% may be considered “good” and an average error of 32% “outstanding” [18].

Most methods for estimating effort require an estimate of the size of the software. Once a size estimate is available, models can be used that relate size to effort.

Cost estimation is not a one-time activity at project initiation. Estimates should be refined continually throughout a project [3]. Thus, it is necessary to esti-

mate size repeatedly throughout development.

Accurate estimation of size is vital. Unfortunately it has proved to be very difficult, especially early in development when the estimates are of most use.

Most research on estimating size has dealt with traditional applications and traditional software development practices. Few methods have been proposed for object oriented software development.

This paper presents a method for estimating the size of object oriented software development projects. It is based on an adaptation of the classical Function Point method to object oriented software.

In the following sections, we present our approach to measurement. We explain how we map the concepts of function points to object oriented software. We describe the process of counting *Object Oriented Function Points* (OOFPs), and give an example. Results from a pilot study are presented and discussed.

2. Measurement Perspective

Figure 1 shows the main phases of an object oriented (OO) development process, and measurements that can be obtained at different points in development. The purpose of these measures is to give the project manager something from which to estimate the size, effort, and duration of a project. These estimates can be revised as new artifacts become available during development.

As we move through the phases of the process, the measurement perspective changes from that of the user to the designer.

At the end of the requirement specification phase, the classical Function Point (FP) counting method is applied on the requirements document. The function point method takes the perspective of the *end user*. What is actually measured are the functions of the system that are visible to the end user. This measure is

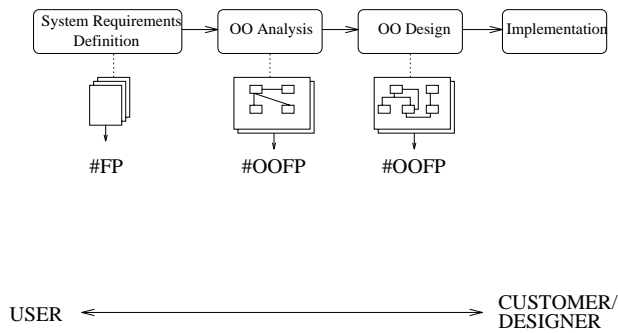


Figure 1: Perspectives and measures in the software development process.

generally considered to be independent of any particular implementation.

Some aspects of a system (e.g. a graphical user interface) are not included in the classical function point count. Nevertheless, they contribute to the final size of the system. If the objective of measuring functionality is to estimate the size of an implementation of a system, and from that the effort and duration of a software project, these aspects should be taken into account. This changes the perspective from that of the user to that of the *customer* i.e. the organization that acquires the system, accepts it and pays the development costs; and of the system designer, who has to produce an implementation of the given specifications.

Once object oriented modelling begins (i.e. from the OO analysis phase on), measurements can be obtained from the object models; OOFPP are used in place of FP. As development proceeds, this gives a natural shift of perspective.

In the OO analysis phase, most of the elements in the object models are still related to the application domain, so the perspective is still that of the user.

At the OO design and later phases, the object models reflect implementation choices. This includes aspects of the system that are not specified in the requirements documents. The count of OOFPP on these models will thus include such functionalities. The measurement perspective is now that of the designer.

Different size estimation models can be developed for different stages of development. More detailed information is available, as the system evolves from an abstract specification to a concrete implementation. It should be possible to refine a size estimate repeatedly, as long as the estimation process is not too difficult. Since we have developed tools to automate the count-

ing of OOFPPs, re-calculation is easy at any time.

3. Object Oriented Function Points

Practitioners have found function points to be very useful within the data processing domain, for which they were invented. We aim to exploit the experience that has been obtained with function points in traditional software development, in the OO paradigm. In adapting 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.

Our presentation uses notations from the OMT method [15]. It would not be much different if the Booch notation [1] or Unified Modeling Language [12] was used instead, since the main models and views in the different methodologies carry similar information.

The OMT method uses three different, orthogonal views to describe a software system:

- Object Model: a static representation of the classes and objects in a system, and their relationships.
- Functional Model: data flow diagrams provide a functional decomposition of the activities of the system.
- Dynamic Model: state machines represent the dynamic and control aspects of a system.

Although all three models provide important information about an object-oriented system, the object model is the most important for our purposes. It is usually the first to be developed, and so can be measured earliest. It is the one that represents what is actually to be built. In a sense the other models help in completing the object model: the functional model helps in identifying and designing some of the methods; the control model helps in identifying attributes that are needed to maintain state information, and events that must be implemented as classes or methods.

There is, however, an ongoing discussion in the practitioners community on the content and role of those models. The functional model seems to have fallen into disuse and is not required any more by some methodologies [12]. The dynamic model is often replaced with a list of use cases and scenarios. The object model is the only one that is present in all methodologies and describes the system using specifically object-oriented concepts. For these reasons, we decided to restrict our attention to object models.

In traditional developments, the central concepts used in counting function points are *logical files* and *transactions* that operate on those files. In OO systems, the core concept is no longer related to files or data bases; instead the central concept is the “object”.

The central analogy used to map function points to OO software relates logical files and transactions to classes and their methods. A logical file in the function point approach is a collection of related user-identifiable data; a class in an object model encapsulates a collection of data items. A class is 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, logical files (LF) are divided into *internal* logical files (ILFs) and *external* interface files (EIFs). Internal files are those logical files that are maintained by the application; external files are those referenced by the application but maintained by other applications. This division clearly identifies the application boundary. In the OO counterpart, the application boundary is an imaginary line in an object model, which divides the classes belonging to the application from the classes outside the application. External classes encapsulate non-system components, such as other applications, external services, and reused library classes (both directly instantiated and subclassed and parameterized classes). Classes within the application boundary correspond to ILFs. Classes outside the application boundary correspond to EIFs.

Transactions in the FP method are classified as *inputs*, *outputs* and *inquiries*. This categorization is not easily applicable outside the data processing domain.

In the OO paradigm the locus of operation are class 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, an output or is dealing with an enquiry, we do not attempt to distinguish the three categories. We simply treat them as generic Service Requests (SRs), issued by objects to other objects to delegate to them some operations.

In short, we map logical files to classes, and transactions to methods.

Issues such as inheritance and polymorphism affect the structure of the object model, and how the model should be counted. They are addressed in Section 4.

3.1. Related Work

Other authors have proposed methods for adapting function points to object oriented software. They also generally map classes to files, and services or messages to transactions.

Whitmire[19] considers each class as an internal file. Messages sent across the system boundary are treated as transactions. Schooneveldt[16] treats classes as files, and considers services delivered by objects to clients as transactions. This method gives a similar count to traditional function points for one system. A draft proposal[7] by the International Function Point Users Group (IFPUG) treats classes as files, and methods as transactions.

Fetcke[4] defines rules for mapping a *use case* model[9] to concepts from the IFPUG Counting Practices Manual[6]. Three case studies have confirmed that the rules can be applied consistently. No attempt has been made to relate the results to other metrics, such as traditional function points, lines of code, or effort.

Sneed [17] 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.

The closest analogue to our method is *Predictive Object Points* (POPs), proposed recently by Minkiewicz [10, 11]. POPs 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. POPs have been incorporated into a commercial tool for project estimation.

Our work differs from Minkiewicz in several ways. In two respects we consider more information: we count the data in a class as well as the methods; and we consider aggregation and inheritance in detail, instead of as averages. We consider less information when counting methods, since we do not distinguish between method types. Information about method type is seldom available at the design stage. Automatic tools would need to gather that information from the designer, which might be a tedious task for very large systems. For that reason we do not attempt to base our method complexity weighting on method type; instead we try to exploit information about a method’s signature, which is most likely to be present in a design,

at least at the detailed design stage.

The key thing which is new about our method is its flexibility, with much scope for experimentation. For example, Fetcke[4] defines that aggregation and inheritance should be handled in a particular way. As discussed below in Section 4.1, we define several options (one of which is Fetcke’s approach) and leave it to the user to choose. We have written programs to count OOFPs automatically, with several parameters to govern counting decisions. An organization can experiment to determine which parameter settings produce the most accurate predictors of size 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.

4. Measurement Process

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

OOFPs can be calculated as:

$$OOF P = OOF P_{ILF} + OOF P_{EIF} + OOF P_{SR}$$

where:

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

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

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

A denotes the set of objects belonging to the application, and o is a generic object in D . DETs, RETs and FTRs are elementary measures, calculated on LFs and SRs and used to determine their complexity through the complexity matrices W . Details are given in Sections 4.1–4.3.

Figure 2 shows the phases of the OOF P computation process:

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

four ways in which this might be done; which to use is a parameter of the counting process. This step is described in Section 4.1.

2. The complexity of each logical file and service request is determined. W tables are used to map counts of structural items (DET, RET and FTRs) to complexity levels of low, average, or high. These tables can be varied, and represent another parameter of the counting process. This step is described in Sections 4.2 and 4.3.
3. The complexity values are translated to numbers, using another table. This table too can be varied, and so is yet another parameter of the counting process.
4. The numbers are summed to produce the final OOF P value.

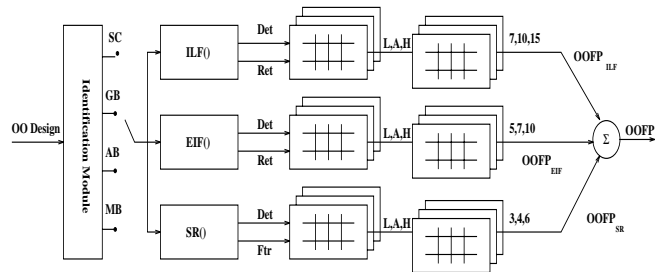


Figure 2: OOF P computation process.

4.1. Identifying logical files

Conceptually, classes are mapped to logical files. It may not always be appropriate to count each class simply as a single logical file, however. Relationships between classes (aggregations and generalization/specializations in particular) can sometimes make it appropriate to count a group of classes together as a logical file.

Aggregation and inheritance relationships pertain mostly to implementation aspects (internal organization, reuse). There tend to be few of them in an analysis object model. There may be many of them in a design or implementation model, as whole-part assemblies and inheritance hierarchies are identified.

How these relationships affect the boundaries around logical files depends on the perspective chosen, and the artifact on which the OOF Ps are computed.

At the analysis phase, the user’s perspective is the important one. It is too early to take the designer’s point of view. At this stage, most of the classes in an object

model represent entities in the application and user domain. There are few aggregation and inheritance relationships to complicate things. Counting each single class as a logical file is usually appropriate.

At the design phase, the object models contain much more information related to the implementation. From a designer’s perspective, considering each single class as a logical file will again be the correct choice.

Counting a design object model from the user’s perspective is more complicated. To count what can actually be perceived by the user of the system, the original abstractions present in the requirements and analysis models have to be recovered. Implementation details should not affect the count. There might no longer be a strict mapping of single classes to logical files; collections of classes may sometimes need to be counted together as a single logical file.

There may be many different ways to identify logical files. We consider four, which are defined by different choices of how to deal with aggregations and generalization/specialization relationships:

1. **Single Class:** count each separate class as a logical file, regardless of its aggregation and inheritance relationships;
2. **Aggregations:** count an entire aggregation structure as a single logical file, recursively joining lower level aggregations.
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.
4. **Mixed:** combination of options 2 and 3.

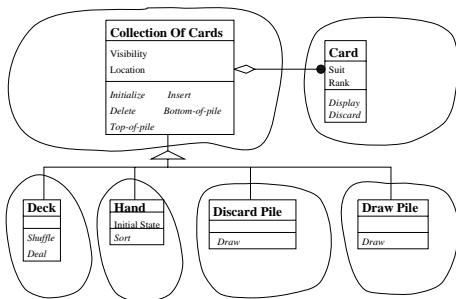


Figure 3: Single class ILFs.

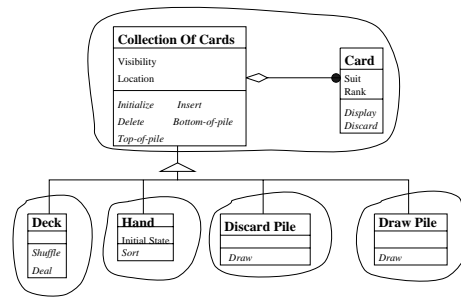


Figure 4: Aggregations ILFs.

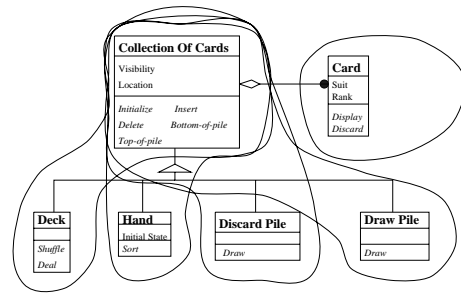


Figure 5: Generalization/Specialization ILFs.

For example, Figures 3-6 show the different counting boundaries that result from these four strategies, on a sample object model¹. Aggregation merging decreases the number of classes in the object model from 6 to 5 as `CollectionOfCards` is merged with `Card`; the resulting logical file contains all the data members and methods of the two classes. Generalization/specialization merging projects the superclass `CollectionOfCards` onto its subclasses, again reducing the number of logical files from 6 to 5. Finally, combining Aggregation and Generalization/Specialization merging first aggregates `CollectionOfCards` with `Card` and then projects the result onto the subclasses of `CollectionOfCards`, resulting in 4 logical files.

Conceptually, it makes sense to merge superclasses into subclasses for OOFP counting. It seems right to count the leaf classes, with their full inherited structure, since this is how they are instantiated. (The non-leaf classes of a hierarchy usually are not instantiated — they are created for subsequent re-use by deriving subclasses, and for exploiting polymorphism.) Also, two classes linked by a generalization/specialization relationship are intuitively less complex than two separate classes, because the subclass represents a refinement of the superclass.

Associations may present a problem. If non-leaf classes

¹The model is drawn from Rumbaugh[15].

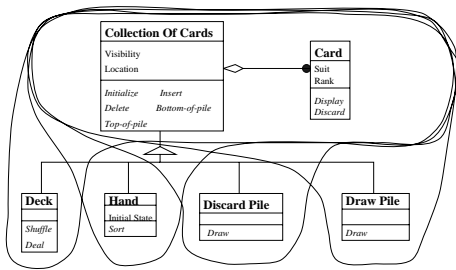


Figure 6: Mixed ILFs.

of an inheritance hierarchy participate in associations, replicating the superclass association into each subclass would increase artificially the number of associations. In fact, the original superclass association contributes to complexity only in the superclass, and in code it will only be implemented once.

Merging aggregations into a single entity for OOFP counting seems less intuitive. The objects that form aggregations are separate objects, that exist independently of each other and have their own methods and data members. At run-time, different objects will be instantiated for each class in the aggregation.

However, it can be argued that dividing a user-identifiable class into an aggregation of sub-classes is an implementation choice. From the point of view of the end user, and of the function point 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.

Whether or not it is right to merge aggregations seems to depend on whether the user's or designer's perspective is chosen. A hybrid solution can be adopted, in which the treatment of aggregations is considered as a parameter of the OOFP counting process. Three options can be identified, with the choice left to the measurer:

1. merge aggregations;
2. do not merge aggregations;
3. flag on the design which aggregations should be considered as a unique entity and so must be merged.

4.2. Logical Files

What is considered an ILF/EIF varies, according to the particular ILF/EIF identification strategy used. Merging aggregations or generalizations can generate ILFs

or EIFs that correspond to sets of classes in the design. We call these *composite* ILFs/EIFs, to distinguish them from those consisting of a single class, called *simple*.

For each ILF/EIF it is necessary to compute the number of DETs (Data Element Types) and RETs (Record Element Types). The rules for DET/RET computation are slightly different for simple or composite ILFs/EIFs.

In both cases, one RET is associated to each ILF/EIF, because it represents a “user recognizable group of logically related data” [6].

When the DETs and RETs of an ILF or EIF have been counted, tables are used to classify the ILF/EIF as having low, average, or high complexity. We base these tables on those given in the IFPUG Counting Practices Manual Release 4.0[6].

4.2.1 Simple ILFs/EIFs Simple attributes, such as integers and strings, are considered as DETs, since they are a “unique user recognizable, non-recursive field of the ILF or EIF” [6].

A complex attribute in the OO paradigm is an attribute whose type is a class (this models the analogy of a complex attribute with a RET, i.e. “a user recognizable subgroup of data elements within an ILF or EIF” [6]) or a reference to another class.

Associations need to be counted as well, since they contribute to the functionality/complexity of an object. An association is usually implemented as a data member referencing the associated objects; this reference is used in methods to invoke the associated object's services.

Associations are counted as DETs or RETs according to their cardinality. 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 a special case of associations. For simple ILFs/EIFs, they are treated as normal associations.

4.2.2 Composite ILFs/EIFs DETs and RETs are counted for each class within the composite, and summed to give the overall total for the composite ILF/EIF.

DETs and RETs are counted using the same rules as for simple ILFs/EIFs, except for aggregations. Aggregations are dealt with in a special way because in a composite ILF/EIF they represent a subgroup. One RET is counted for each aggregation, whatever its cardinality. The RET is assigned to the container class.

In practice, the values of DET and RET for any ILF/EIF are computed by counting DETs and RETs for each component class on its own (this is trivial for a simple ILF/EIF), and just adding them up².

4.3. Service Requests

Each service request (method) in each class in the system is examined. Abstract methods are not counted. Concrete methods are only counted once (in the class in which they are declared), even if they are inherited by several subclasses, because they are only coded once.

If a method is to be counted, the data types referenced in it are classified:

- *simple items* (analogous to DETs in traditional function points) are simple data items referenced as arguments of the method, and simple global variables referenced in the method;
- *complex items* (analogous to File Types Referenced — FTRs — in traditional function points) are complex arguments, objects or complex global variables referenced by the method.

Several approaches are possible to distinguish complex items from simple ones. For example, compiler built-in types might be considered as simple and all other types as complex. This choice might not be appropriate, since all user-defined types would be counted as complex, even if they were scalar types or aliases of built-in types. Another approach is to regard a complex item as one whose type is a class or a reference to another class. This approach is used here.

When the DETs and FTRs of a method have been counted, tables are used to classify the method as having low, average, or high complexity. We base these tables on those given in the IFPUG Counting Practices Manual Release 4.0[6] for external inputs and queries.

²The counting rules defined make DET-RET additive. The only exception is the aggregation relation, which is handled differently in simple and composite ILFs. However, in practice, the contribution of aggregation in composite ILFs corresponds to considering one RET for each class involved in the aggregation structure, which becomes equivalent to summing the RETs of each component class separately.

Most of the time, the signature of the method provides the only information on DETs and FTRs. Sometimes, especially early on, even that is not known. In such a case, the method is assumed to have average complexity.

4.4. An example

Figures 3-6 show four different ways that classes in an object model might be merged, according to which of the four different LF identification strategies is used. Here we show the OOFPs that are computed for each variant.

Service Requests:

Service requests (methods) can be counted immediately. Since they are only counted once anyway, 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. They each receive the four OOFPs that are scored for an average service request.

As there are 12 concrete methods in the model, service requests contribute $12 \times 4 = 48$ OOFPs.

Logical files:

The counting procedure for each individual class gives the DETs and RETs shown in Figure 7.

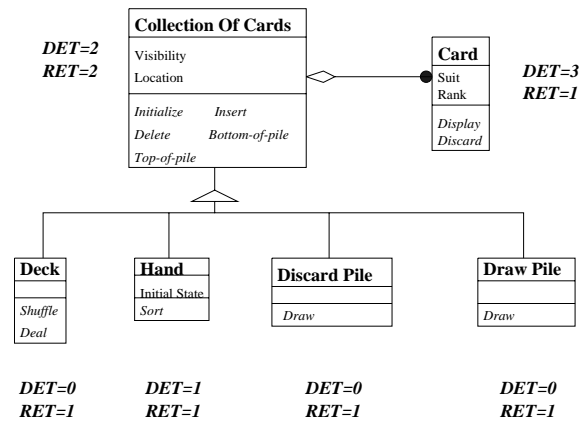


Figure 7: DET/RET computation for LFs on the example system.

The class `Card` has three DETs (two due to the two data items and one due to the many-to-one association with `CollectionOfCards`) and one RET (since the class itself is a collection of related data items).

CollectionOfCards has two DETs due to its two data items, one RET due to the one-to-many aggregation with Card, and one RET for its own structure. Each other class has one RET and as many DETs as it has data items.

Depending on which ILF identification strategy is used, there are four different ILF variants. Each variant merges classes together in different ways, resulting in different total DET and RET counts. Table 1 shows the result of applying IFPUG 4.0 complexity tables with each variant. The value *Low* is rated as 7 OOFF, according to the IFPUG tables.

	S	A	G	M
Collection of Cards	Low	Low	-	-
Card	Low	-	Low	-
Deck	Low	Low	Low	Low
Hand	Low	Low	Low	Low
Discard Pile	Low	Low	Low	Low
Draw Pile	Low	Low	Low	Low
ILF OOFF	42	35	35	28
SR OOFF	48	48	48	48
Total OOFF	90	83	83	76

Table 1: ILF and SR complexity contribution (S = Single Class, A = Aggregation, G = Generalization/Specialization, M = Mixed).

The highest OOFF count comes when each class is counted as a single ILF. All the other variants have the effect of reducing the OOFF 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.

5. Tools for Counting OOFFs

The process for computing OOFFs has been automated, as shown in Figure 8. Object models produced with CASE tools are translated to an intermediate representation. The intermediate representation is parsed, producing an Abstract Syntax Tree (AST), to which the OOFF counting process is applied.

In order to be independent of the specific CASE tool used, an intermediate language, called Abstract Object Language (AOL), has been devised. The language is a general-purpose design description language, capable of expressing all concepts available at the design stage of object oriented software development. This language is based on the Unified Modeling Language [12], a superset of the OMT notation that is becoming the standard in object oriented design. Since

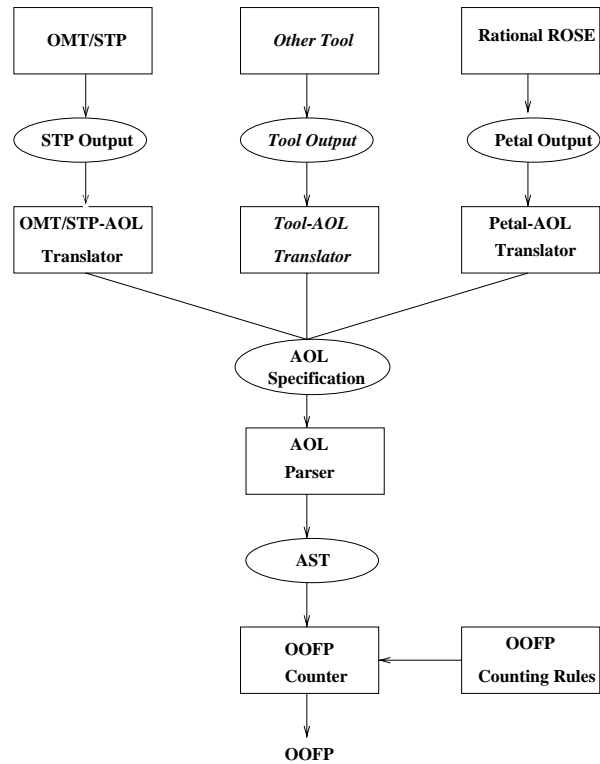


Figure 8: OOFF Computation Process

UML is a visual description language with some limited textual specifications, we had to design from scratch many parts of the language, while remaining adherent to UML where textual specifications were available. Figure 9 shows an excerpt from the AOL description of the object model depicted in Figure 7.

The output of the specific CASE tool used is translated into an equivalent AOL specification, by an automatic translator. One such translator has been implemented, to convert the output from OMT/STP[8] to an AOL specification. Other translators could be implemented for other CASE tools, such as Rational Rose [13] which fully supports UML and represents its output using a language called Petal.

The AOL specification is then parsed by an AOL parser, producing an AST 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).

The OOFF Counter implements the OOFF Counting Rules described in Section 4. The OOFF Counter is very different from other measurement and counting tools, because instead of assuming a specific counting

```

class Deck
  operations
    PUBLIC Shuffle() : void,
    PUBLIC Deal() : void;
class Hand
  attributes
    PRIVATE InitialState : void
  operations
    PUBLIC Sort() : void;
...
aggregation
  container class CollectionOfCards mult one
  parts      class Card mult many;

generalization CollectionOfCards
  subclasses Deck, Hand, DiscardPile, DrawPile

```

Figure 9: Excerpt of the AOL specification for the example object model.

strategy it allows one of several strategies to be chosen. This makes it suitable for experimentation. The tool is very flexible, being parameterizable with respect to the rules used in the counting process.

The AOL parser and the OOFPP counter have been implemented in both Refine[14] and Lex/Yacc.

6. Pilot Study

The described methodology has been applied in an industrial environment producing software for telecommunications. Our first study is of the relationship between the OOFPP measure of a system and its final size in lines of code (LOC), measured as the number of non-blank lines, including comments.

Eight sub-systems of a completed application were measured, for each of which 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 OOFPP count for each system, using each of the four different strategies for identifying logical files.

The four OOFPP series are strongly correlated with each other. The lowest Pearson correlation, between the Single Class (S) and Mixed (M) strategies, is .992. Other correlations range up to .998. As shown in Table 2, differences between the methods become appre-

System	LOC	S	A	G	M
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 OOFPPs. (S = Single Class, A = Aggregation, G = Generalization/Specialization, M = Mixed).

ciable only for the projects with large LOC values.

The high correlation between the four OOFPP series suggests that they are essentially linear transformations of each other. In that case, changing the strategy for identifying logical files might not make much difference to the accuracy of size estimation models. When different models are compared, however, some differences do emerge.

Several regression techniques were considered to model the LOC-OOFPP association.

First, linear models (1ms) based on minimizing the sum of squares of the residuals were developed for each LF selection method. Least absolute deviation, based on L_1 error, was also applied (11s). This method minimizes the sum of the absolute values of the residuals, to reduce the effect of large error values.

Regression based on least square minimization assumes that the distribution of errors is Gaussian. Regression estimates are sensitive to outliers in the response variable if this assumption is not satisfied. Given the small size of our data set, it is not clear that this assumption holds. Hence robust regression techniques were also investigated. Robust techniques may improve the least-squares fit and handle model inadequacies due to unusual observations.

A family of M-estimators was therefore considered (rregs and r1ms). 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, although instability and lack of convergence were expected due to the sample size.

Model error was expressed in terms of *normalized mean squared error* (NMSE) and *normalized mean absolute*

error (NMAE).

Given the small size of the data set, a leave-one-out cross-validation procedure was used to achieve unbiased estimates of predictive accuracy for the different models. Each model was trained on $n - 1$ points of the data base L (the sample size is currently $n = 8$) and tested on the withheld datum; NMSE and NMAE are obtained over L normalizing over the sample variance of the observed values ($\mu_y = \text{mean}(y)$).

Even with cross-validation, care is needed in interpreting the results. The small sample means that any observations must be regarded as indicative rather than conclusive.

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

Table 3: Model performance for linear regressors (lms and l1s) and robust methods (rregs and rlmS).

Table 3 shows each of the models that was obtained, parametrized over LF selection methods and the type of regressor. The model coefficients b_0 and b_1 are indicated as computed from the full data set. The estimated model errors (NMSE and NMAE) are shown for each model. The estimated R-squared measure is also included for the linear models.

A point of concern is whether an intercept term b_0 should be included in the model. It is reasonable to suppose the existence of support code not directly related to the functionalities being counted, and prediction is improved with the term. However, the intercept

term is not significant in a non-predictive fit of the data. More importantly, the fact that the intercept term is always larger than our two smallest systems might indicate a poor fit for small OOFp values. It would be interesting to apply a Bayesian procedure to select the intercept from given priors.

Table 3 shows that in this data set the Generalization strategy is consistently better, for all models and for both the predictive error measures NMSE and NMAE.

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

Table 4: Model performances for different weighting functions of the M-estimator *rreg*, for the Generalization selection method.

The estimates for different weighting functions of the M-estimator are listed in Table 4. Results are given for the Generalization selection method only. Lowess, supersmoother and predictive splines were also tested, but showed instability of convergence due to the small sample size.

The best predictive accuracy (NMSE=0.337) was achieved by the *rreg-logistic-G* model with tuning parameter $u = .8$, corresponding to the linear predictor $LOC = 7183.4 + 25.6 OOFp_G$. (This model is very close to the basic linear model *lm-G*, whose equation is $LOC = 7435.1 + 25.2 OOFp_G$.)

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

7. Discussion of Results

As can be seen in Table 1, the complexity of each LF is always determined to be low, even when several classes are merged together. The same is true for service requests. 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 LF needs a DET value between 20 and 50 and a

RET value between 2 and 5). This is due to the data processing origins of the function points method, and doesn't seem to apply as well to all kinds of systems. Therefore the complexity tables should be recalibrated, to provide more discrimination.

The implicit assumption in the use of these tables is that the complexity of a class, and hence the size of its implementation and the effort required to implement it, increases as the number and complexity of its attributes increases. Similarly, the complexity of a method (and hence its size and development effort) is assumed implicitly to increase as the number and complexity of its parameters increases³. Whether these assumptions are true needs to be determined experimentally.

The assumption seems reasonable for classes as a whole, but perhaps not for methods. What works for transactions in traditional function points may not work for methods in an object model, because transactions tend to be much more coarse grained than methods.

At the analysis and design stages, we often have no more information about a method than its signature. If it turns out that this is unrelated to complexity and size, we have nothing to go by in counting OOFPs. One possibility would be to permit the designer to annotate a method with a complexity rating. This would introduce a subjective element to the process, however, which we want to avoid. Another approach would be simply to count the methods, making no attempt to classify them by complexity. A promising approach would take advantage of the information available in use cases and scenarios to derive a complexity rating for methods.

On the data available to us so far, it seems that recalibration of the OOFP tables for logical files might improve the accuracy of OOFP as a predictor of size; recalibration of the table for methods might not. Further experimentation is needed on this topic, with data from more systems. In order to support such experimentation, the tool used to count OOFPs is designed to consider the table entries as parameters that can be defined at any time

The pilot study suggests that for this organization, the best predictions of size are obtained using the Generalization strategy for identifying LFs. Other organizations may find differently.

³These assumptions are fairly common. They underlie the philosophy of the classical function point method. They also feature in the design metrics work of Card and Glass[2].

Modifying the complexity tables might make a difference in determining the best strategy for selecting LFs.

Once a counting scheme has been chosen, it is important that it be applied consistently. Consistent counting is straightforward for us, since tools are used to automate the process.

8. Conclusions

We have presented a method for estimating the size of object oriented software. The method is based on an adaptation of function points, to apply them to object models. The proposed method takes full advantage of the information contained in the object model and eliminates the ambiguities of the traditional function points method. It can be parameterized in order to take into account more closely the characteristics of a specific design environment or of a particular problem.

We have defined the mapping from FP concepts to OO concepts, and described the counting process. Tools have been developed that automate the process. Preliminary results from a pilot study in an industrial environment have been reported. The results from the pilot study show promise for size estimation. This is important, since an estimate of size is needed for many effort estimation models.

In summary, we have shown that we can apply the concepts of function points to object oriented software and that the results are accurate and useful in an industrial environment. Integrating function points with common object oriented modeling tools, and automating the counting process, can help to extend the use of function points in the software industry.

Future work will take several directions. One is to investigate the effect of recalibrating the complexity tables. Other relationships, beyond just OOFPs and code size, will be studied; those between OOFPs and traditional FPs, and OOFPs versus effort, are of particular interest. Another avenue is to consider the impact of using design patterns [5] on the structure within object models; this may lead to other strategies for identifying logical ILFs.

Acknowledgements

We thank the anonymous referees for their constructive comments. This research was funded by SODALIA Spa, Trento, Italy under Contract n. 346 between SODALIA and Istituto Trentino di Cultura, Trento, Italy. G. Caldiera and C. Lokan were with the Experimental Software Engineering Group at the University of Maryland when this work was undertaken.

REFERENCES

- [1] G. Booch. *Object-Oriented Design with Applications*. Benjamin/Cummings Publishing Company, 1991.
- [2] D.N. Card and R.L. Glass. *Measuring Software Design Quality*. Prentice-Hall, 1990.
- [3] T. DeMarco. *Controlling Software Projects*. Yourdon Press, 1982.
- [4] 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, April 1997.
- [5] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley, 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] H. Mehler and A. Minkiewicz. Estimating Size for Object-Oriented Software. *8th European Software Control and Metrics Conference*. Berlin, May 1997.
- [11] A. Minkiewicz. Measuring Object-Oriented Software With Predictive Object Points. *ASM'97 — Applications in Software Measurement*. Atlanta, October 1997.
- [12] Rational Software Corporation. *Unified Modeling Language, Version 1.0*, 1997.
- [13] Rational Software Corporation. *Rational Rose/C++ Manuals, Version 4.0*, 1997.
- [14] Reasoning Systems. *Refine User's Guide*, 1990.
- [15] J. Rumbaugh et al. *Object-Oriented Modelling and Design*. Prentice-Hall, 1991.
- [16] M. Schooneveldt et al. Measuring the size of object oriented systems. In *Proc. 2nd Australian Conference on Software Metrics*. Australian Software Metrics Association, 1995.
- [17] H. Sneed. Estimating the Costs of Object-Oriented Software. *Proceedings of Software Cost Estimation Seminar*, Systems Engineering Ltd., Durham, UK, March 1995.
- [18] S.S. Vicinanza, T. Mukhopadhyay, and M.J. Pritula. Software-effort estimation: an exploratory study of expert performance. *Information Systems Research*, 2(4):243–262, December 1991.
- [19] S. A. Whitmire. Applying Function Points to Object Oriented Software. In Keyes J. (ed.), *Software Engineering Productivity Handbook*, Chapter 13. McGraw-Hill, 1993.