
Empirical Software Engineering, 4, 263–287 (1999)

A Function Point-Like Measure for Object-Oriented Software

GIULIANO ANTONIOL
University of Sannio, Faculty of Engineering, Palazzo Bosco Lucarelli, Piazza Roma, I-82100 Benevento, Italy

CHRIS LOKAN
School of Computer Science, Australian Defence Force Academy, UNSW, Canberra ACT 2600, Australia

GIANLUIGI CALDIERA
PricewaterhouseCoopers, 12902 Federal Systems Park Dr, Fairfax, Virginia 22033, USA

ROBERTO FIUTEM
Sodalia Spa, Viale Zambra 1, Trento, Italy

Abstract. We present 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 organization 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" (Vicinanza, Mukhopadhyay and Prietula, 1991).

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 (DeMarco, 1982). Thus, it is necessary to estimate 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.

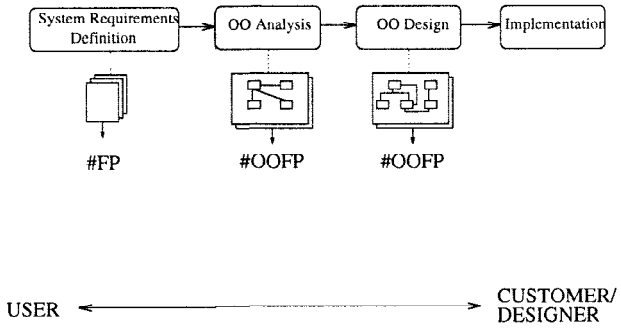

Figure 1. Perspectives and measures in the software development process.

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 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 modeling begins (i.e. from the OO analysis phase on), measurements can be obtained from the object models; OOFP 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 OOFP 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 counting of OOFPs, re-calculation is easy at any time.

3. Object Oriented Function Points

Practitioners have found function points (Albrecht and Gaffney, 1983; IFPUG, 1994) 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 (Rumbaugh, Blaha, Premerlani, Eddy and Lorensen, 1991). It would not be much different if the Booch notation (Booch, 1991) or Unified Modeling Language (Rational Software Corporation, 1997b) 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 (Rational Software Corporation, 1997b). 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 too generally map classes to files, and services or messages to transactions.

Whitmire (1993) considers each class as an internal file. Messages sent across the system boundary are treated as transactions. Schooneveldt, Hastings, Mocek and Fountain (1995) treat classes as files, and consider services delivered by objects to clients as transactions.This

method gives a similar count to traditional function points for one system. A draft proposal by the International Function Point Users Groups ("IFPUG") treats classes as files, and methods as transactions (IFPUG, 1995).

Fetcke, Abran and Nguyen (1997) define rules for mapping a *use case* model (Jacobson, Christerson, Jonsson and Övergaard, 1992) to concepts from the IFPUG Counting Practices Manual (IFPUG, 1994). 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 (1995) 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 by Minkiewicz (1997). 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 et al. (1997) define 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.

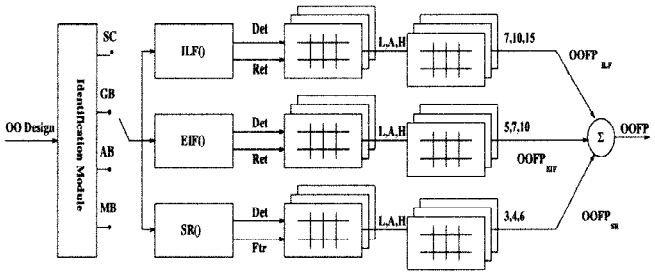

Figure 2. OOFP computation process.

OOFPs 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, 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 OOFP 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 (DETs, RETs 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. These numbers are the OOFP values for the individual logical files. The table used here can also be varied, and so is yet another parameter of the counting process.

4. If a logical file is a class which is annotated as "reused" (ie developed by reuse of another class), its OOFP value is multiplied by a scale factor (1.0 or less). The scale factor is another parameter of the counting process. This step is discussed in Section 4.4.

5. The OOFP values are summed to produce the total OOFP value.

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 OOFPs 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 this stage, the origin of a class does not matter. The scale factor used in step 4 of the counting process should be set to 1.0, so the class is counted with its full inherent value.

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.

From a designer's or implementer's point of view, reuse makes classes easier to develop. If the OOFP count is intended now to help predict the effort or duration needed to build the system, a scale factor of less than 1.0 should be used in step 4 of the counting process.

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/specializations 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.

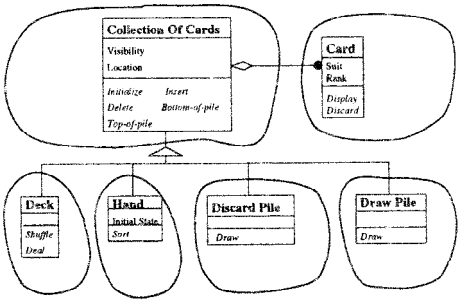

Figure 3. Single class ILFs.

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.

For example, Figures 3–6 show the different counting boundaries that result from these four strategies, on a sample object model.[1] 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

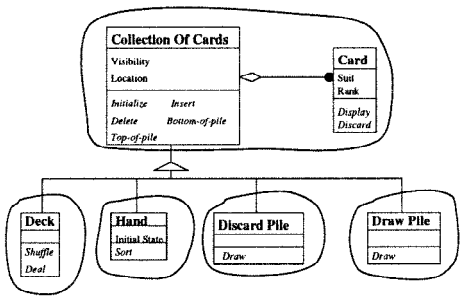

Figure 4. Aggregations ILFs.

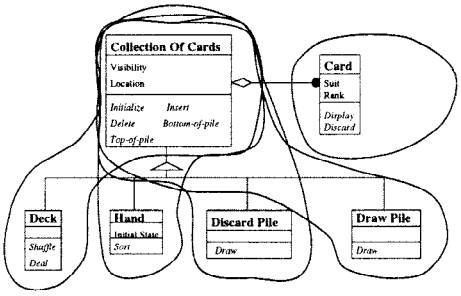

Figure 5. Generalization/specialization ILFs.

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 class 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.

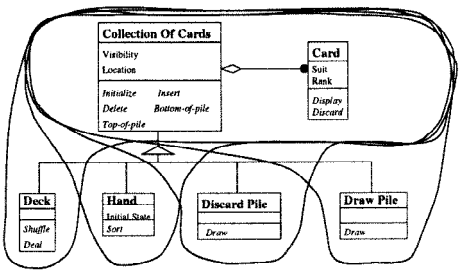

Figure 6. Mixed ILFs.

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" (IFPUG, 1994).

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 (IFPUG, 1994).

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 the EIF" (IFPUG, 1994).

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" (IFPUG, 1994)) 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 (IFPUG, 1994)). 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 up.[2]

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 (IFPUG, 1994) for external inputs and queries.

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. Allowing for Reuse

In an early count, where the main aim is to capture user-oriented functionality, the scale factor used in step 4 of the counting process should be set to 1.0. A user doesn't care where a class comes from, so the class should be counted with its full inherent value.

From a designer's or implementer's point of view, reuse makes classes easier to develop. In a later count, in which the OOFP count may be intended to help predict the effort or duration needed to build the system, a scale factor of less than 1.0 would be appropriate.

4.5. 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.

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 I shows the result of applying IFPUG 4.0 complexity tables with each variant. The value *Low* is rated as 7 OOFP, according to the IFPUG tables.

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.

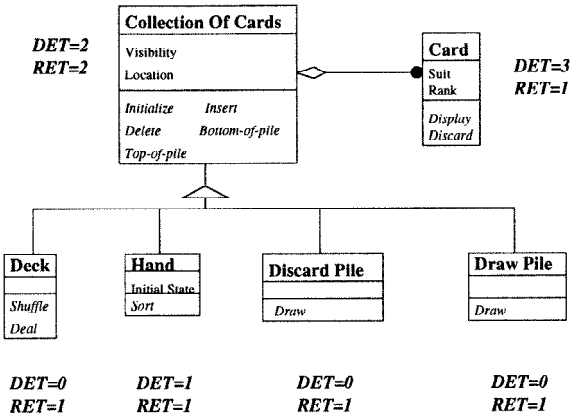

Figure 7. DET/RET computation for LFs on the example system.

5. Tools for Counting OOFPs

The process for computing OOFPs 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 OOFP 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 (Rational Software Corporation, 1997b), a superset of the OMT notation that is becoming the standard in object oriented design. Since UML is a visual description

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

	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 OOFP	42	35	35	28
SR OOFP	48	48	48	48
Total OOFP	90	83	83	76

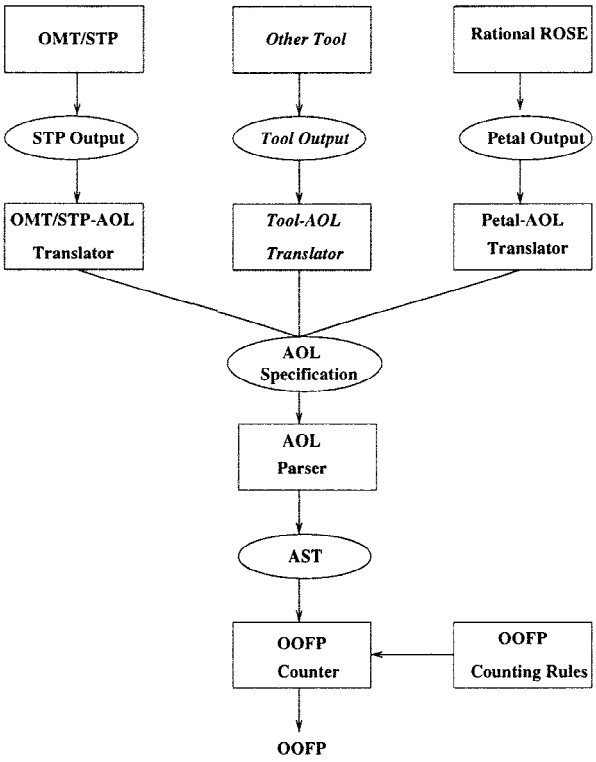

Figure 8. OOFP computation process.

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 automatically into an equivalent AOL specification. One translator has been implemented, to convert the output from OMT/STP (Interactive Development Environments, 1996) to an AOL specification. Other translators could be implemented for other CASE tools, such as Rational Rose (Rational Software Corporation, 1997a) 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 OOFP Counter implements the OOFP Counting Rules described in Section 4. The OOFP Counter is very different from other measurement and counting tools, because instead of assuming a specific counting strategy it allows one of several strategies to be chosen.

```
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.

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 OOFP counter have been implemented in both Refine (Reasoning Systems, 1990) 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 OOFP 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. These eight systems were chosen for study because all were developed by the same people, in the same environment, using the same language (C++). Design documents and the final source code were both available. Measurements of design characteristics were taken from the design documents, not "reverse engineered" from the source code.

Table II shows the numbers of various design elements in each system. Table III shows the size of each system, spreading from about 5,000 to 50,000 lines of code. Table III also shows the OOFP count for each system, using each of the four different strategies for identifying logical files.

Table II. Design characteristics (Att) = Attributes, Ope = Operations, Ass = Associations, Agg = Aggregation, Inh = Generalization, Cls = Classes).

System	Att	Ope	Ass	Agg	Inh	Cls
A	0	0	10	0	4	9
B	16	118	6	1	8	13
C	9	73	2	0	5	7
D	79	295	12	2	9	17
E	29	110	78	7	6	29
F	24	90	62	22	5	35
G	58	222	38	10	1	38
H	78	364	112	23	70	113

The four OOFP 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 III, differences between the methods become appreciable only for the projects with large LOC values.

The high correlation between the four OOFP 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.

6.1. Model Evaluation

A leave-one-out cross-validation procedure (Stone, 1974) was used to measure model performance. Each model was trained on $n-1$ points of the data set L (sample size is currently $n = 8$) and accuracy tested on the withheld datum. The step was repeated for each point in L and accuracy measures averaged over n. This method gives an unbiased estimate of future performance on new data, and enables quite different models to be compared directly.

Table III. System sizes and OOFPs. (S = Single Class, A = Aggregation, G = Generalization/Specialization, M = Mixed).

System	LOC	S	A	G	M
A	5807	63	63	35	35
B	10424	476	469	462	455
C	15267	284	284	270	270
D	19863	1071	1057	1057	1043
E	22980	562	513	548	499
F	31011	518	403	483	368
G	47057	1142	1100	1124	1072
H	54300	2093	1947	1872	1737

Model error was estimated as the cross-validation version of the *normalized mean squared error* (NMSE). This is the mean squared error, normalized over the variance of the sample:

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

Regression based on least square minimization assumes that the distribution of errors is Gaussian. Statistical tests for skewness and kurtosis do not cause the hypothesis of normality to be rejected for any of the five distributions (LOC and the four OOFP values). But given the small size of our data set, it is not clear that the distributions really are normal.

The least squares approach is sensitive to outliers (data points far removed from other data points) in the data because it minimizes squared deviations. An outlier can have almost as much influence on the regression results as all other points combined. Standard box-and-whisker plots do not identify any of our systems as outliers, although system H is right on the edge of being considered an outlier.

The impact of such influential points can be lessened by reducing the weight given to large residuals—for example, by minimizing the sum of absolute residuals rather than the sum of squared residuals. Thus another measure for errors, based on absolute values, was also considered to check inconsistencies due to possible influential or outlier points. This measure is *normalised mean absolute error* (NMAE):

$$NMAE = \frac{\sum_{k \in L} |y_k - \hat{y}_k|}{\sum_{k \in L} |y_k - med_y|}$$

where $\mu_y = mean(y)$ and $med_y = median(y)$ are the mean and median of the observed values in the sample L. Where available, the cross-validation estimates of the standard error σ of the residuals $y_k - \hat{y}_k$ and of the r-squared R^2 of the fit were also computed.

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.

6.2. Models Considered

Several regression techniques were considered to model the relationships of LOC with OOFP. Other predictors of LOC, based on direct indicators of OO size such as the number of classes or methods in the design, were considered for comparison.

First, linear models (`lms` in Table IV) 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 (`l1s` in Table IV). This method minimizes the sum of the absolute values of the residuals, to reduce the effect of large error values.

Robust regression techniques were also investigated, to handle non-obvious outliers. A family of M-estimators (see the Appendix) was considered (`rregs` and `rlms` in Table IV). 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. Examples

Table IV. Model performance for linear regressors (`lms` and `l1s`) and robust methods (`rregs` and `rlms`).

Method	NMSE	NMAE	$\hat{R}^2$	b_0	b_1
lm-S	0.39	0.66	0.73	7993	23.0
lm-S-1	0.54	0.81	0.90	0000	29.4
lm-A	0.43	0.66	0.69	8505	23.8
lm-G	0.38	0.60	0.73	7435	25.2
lm-M	0.46	0.68	0.68	8187	25.8
l1-S	0.55	0.81	–	9139	21.6
l1-A	0.63	0.86	–	8601	23.5
l1-G	0.39	0.69	–	8688	24.4
l1-M	0.46	0.73	–	8083	26.6
rreg-S	0.40	0.67	–	7875	23.0
rreg-A	0.43	0.66	–	8255	24.0
rreg-G	0.37	0.60	–	7332	25.5
rreg-M	0.44	0.66	–	7862	26.4
rlm-S	0.40	0.67	–	8002	23.0
rlm-S-1	0.63	0.86	–	0000	29.3
rlm-A	0.44	0.66	–	8518	23.8
rlm-G	0.38	0.60	–	7522	25.6
rlm-M	0.46	0.68	–	8162	26.3

of smoothers are Andrews, bisquare, fair, Hampel, Huber, and logistic (Venables and Ripley, 1994). Each corresponds to a different weight function.

Finally, multivariate linear models were developed that predict LOC directly from the numbers of OO design elements shown in Table II.

6.3. Results

Table IV shows each of the models, parameterized over LF selection methods and the type of regressor. The model coefficients b_0 and b_1 indicated were 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 smallest system 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.

The results summarized in Table IV are encouraging. For example, the `lm-G` model has an NMSE of 38%, meaning that the square error variance is less than half of the sample variance. From another point of view, models based on the OOFPs counted using the

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

Method	NMSE	Comments
rreg-default-G	0.37	–
rreg-andrews-G	0.37	–
rreg-bisquare-G	0.37	–
rreg-fair-G	0.48	converged after 50 steps)
rreg-hampel-G	0.38	–
rreg-huber-G	0.38	–
rreg-logistic-G	0.36	$c = 1.25$
rreg-logistic-G-0.8	**0.34**	$c = 0.80$
rreg-talworth-G	0.38	–
rreg-welsch-G	0.38	–

Generalization strategy achieve a cross validation average error of 47%, which is very good.

The best model in Table IV is `rreg-G`. Further investigation of `rreg-G` was done, with the results shown in Table V.

The best predictive accuracy ($NMSE = 0.337$) was achieved the the *rreg-logistic-G* model with tuning parameter $u = .8$. This corresponds 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 $LOC = 7435.1 + 25.2\ OOFP_G$.)

Table IV suggests that in this data set the Generalization strategy is consistently best. This is not proven statistically, though. A non-parametric bootstrap approach (Efron and Tibshirani, 1993) was used to assess the models. The null hypothesis that there are no differences between errors from the `lm-S` and `lm-G` models cannot be rejected; similar results were obtained for the other models. Thus it is not clear that any counting strategy should be preferred over any other.

For comparison, multivariate linear models were developed that predict LOC directly from the numbers of OO design elements (shown in Table II). Poorer results are obtained from such models. For example, a model based on classes and methods has $NMSE = 2.14$, $NMAE = 1.25$, $\hat{R}^2 = 0.79$. Models based on OOFP perform much better.

This pilot study was conducted in a specific project and environment, in a specific organization. The results are encouraging for size estimation in this context. The issue of external validity must be addressed by more extensive studies, targeting multiple organizations and different projects. We have taken the first step in empirical investigation; more needs to be done.

7. Discussion of Results

As can be seen in Table I, 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 (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 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 increases.[3] 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 modified at any time.

The pilot study suggests that for this organization there is no reason to prefer any of the four strategies for identifying LFs over any other. Other organizations may find differently. Although for this organization the best size predictions appear to be obtained with the Generalization strategy, its superiority is not proven statistically and may be an accident of the data.

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.

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 (Gamma, Helm, Johnson and Vlissides, 1995) on the structure within object models; this may lead to other strategies for identifying logical ILFs.

Appendix: M-Estimation

The M-estimation procedure used applies iteratively reweighted least squares to approximate a robust fit. Residuals from the current fit are passed through a weighting function to give weights for the next iteration. Several weighting functions are available in S-Plus: andrews, bisquare, cauchy, default, fair, hampel, huber, logistic, median, talworth, welsch. All of these have been compared in this study.

The calculations for some of the weight functions are described below. The description closely follows (Härdle, 1990) and (Venables and Ripley, 1994).

The vector u is the vector of residuals divided by the (Gaussian consistent) MAD of the residuals, while c is a "tuning" constant with default values as indicated for each method.

andrews The weighting function is

$$sin(u/c)/(u/c)$$

for $abs(u) \leq \Pi * c$ and 0 otherwise. The default c is 1.339.

bisquare The weight function is

$$(1 - (u/c)^2)^2$$

if $u < c$ and 0 otherwise. The default is $c = 4.685$.

cauchy The weight function is

$$1/(1+(u/c)^2)$$

with $c = 2.385$ the default.

fair $1/(1+abs(u/c))^2$

is the weight function and 1.4 is the default for c.

hampel has three tuning constants, a, b and c. The weight function is 1 if $abs(u) \leq a$; it is $a/abs(u)$ for $a < abs(u) \leq b$; it is $(a * ((c - abs(u))/(c - b)))/abs(u)$ for $b < abs(u) \leq c$; and it is 0 otherwise. The defaults for a, b and c are 2, 4 and 8.

huber The weight function is 1 for $abs(u) < c$ and $c/abs(u)$ otherwise. The default is $c = 1.345$.

logistic The weight function is

$$tanh(u/c)/(u/c)$$

with $c = 1.205$ the default.

talworth If $abs(u) \leq c$, the weight function is 1, otherwise it is 0. $c = 2.795$ is the default.

welsch The weight function is

$$exp(-2 * (abs(u/(2 * c))^2))$$

with a default c of 2.985.

Acknowledgments

We thank the 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. When this work was undertaken G. Caldiera and C. Lokan were with the Experimental Software Engineering Group at the University of Maryland, and G. Antoniol was with the ITC-IRST, Istituto per la Ricerca Scientifica e Technologica, I-38050 Povo (Trento), Italy.

Notes

1. The model is drawn from Rumbaugh et al. (1991).
2. The 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.
3. These 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 (1990).

References

Albrecht, A., and Gaffney, J. 1983. Software function, source lines of code and development effort prediction: a Software Science validation. *IEEE Transactions on Software Engineering* 9(6): 639–648.

Booch, G. 1991. *Object-Oriented Design with Applications*. Benjamin-Cummings.

Card, D., and Glass, R. 1990. *Measuring Software Design Quality*. Prentice-Hall.

DeMarco, T. 1982. *Controlling Software Projects*. Yourdon Press.

Efron, B., and Tibshirani, R. 1993. *An Introduction to the Bootstrap*. Chapman & Hall.

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

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

Härdle, W. 1990. *Applied Nonparametric Regression*, Vol. 19 of *Econometric Society Monographs*, Cambridge University Press.

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.

Interactive Development Environments. 1996. *Software Through Pictures Manuals*.

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

Mehler, H., and Minkiewicz, A. 1997. Estimating size for object-oriented software. *Proc. 8th European Software Control and Metrics Conference*, Berlin.

Minkiewicz, A. 1997. Measuring object-oriented software with predictive object points. *Proc. ASM'97—Applications in Software Measurement*, Atlanta.

Rational Software Corporation. 1997a. *Rational Rose/C++ Manuals*, Version 4.0.

Rational Software Corporation. 1997b. *Unified Modeling Language*, Version 1.1.

Reasoning Systems. 1990. *Refine User's Guide*.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. 1991. *Object-Oriented Modeling and Design*. Prentice-Hall.

Schooneveldt, M., Hastings, T., Mocek, J., and Fountain, R. 1995. Measuring the size of object-oriented systems. *Proc. 2nd Australian Conference on Software Metrics*, Australian Software Metrics Association, 83–93.

Sneed, H. 1995. Estimating the costs of object-oriented software. *Proc. of Software Cost Estimation Seminar*, Systems Engineering Ltd., Durham, UK.

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

Venables, W., and Ripley, B. 1994. *Modern Applied Statistics with S-Plus*. Second end. New York: Springer-Verlag.

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. *Software Engineering Productivity Handbook* (J. Keyes, ed.) McGraw-Hill, 229–244.

Giuliano Antoniol received his doctoral degree in Electronic Engineering from the University of Padua in 1982. In 1987 he joined the Artificial Intelligence Division of the Instituto per la Ricerca Scientifica e Tecnologica (IRST), Trento, Italy as Senior Researcher. From 1994 he leads the IRST Program Understanding and Reverse Engineering (PURE) Project team.

Results have been achieved in the architectural analysis of distributed software system and in C/C++ design/code analysis. His current research interests include software engineering, Object-Oriented development, architectural recovery, reverse engineering, and distributed systems.

He is currently affiliated with the University of Sannio, Faculty of Engineering, where he works in the area of software metrics, process modeling, software evolution and maintenance.

Chris Lokan is a Senior Lecturer in the School of Computer Science at the 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. He received the Defence Academy's Teaching Excellence Award in 1995. His current 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.

Dr. Gianluigi Caldiera is Principal Consultant with PricewaterhouseCoopers where he manages projects on software process improvement, distance learning, software quality engineering, and Year 2000 Renovation. He has more than 20 years of experience in the area of system and software engineering. He has consulted for

Government and private industry both in the United States and Europe designing and implementing systems and applications, developing methodologies, managing or supporting quality management, measurement, software process improvement, and certification preparation programs. Dr. Caldiera has been on the Faculty of the University of Maryland Institute for Advanced Computer Studies (UMIACS) where he has served for eight years. His research activities are in software engineering, with special focus on software reuse, software architectures, software quality assurance and management, software metrics, software process improvement, standards, and distance learning.

Dr. Caldiera is a guest speaker in many conferences and professional meetings and has published many papers in international journals and conference proceedings.

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. He was involved in artificial intelligence projects regarding autonomous navigation systems and automatic speech recognition. Then he was a member of the Maintenance of Software Systems (MASS) project at IRST, performing research activities on reverse engineering, reengineering and software maintenance.

He is now at Sodalia s.p.a, a telecom software industry in Trento, where he works in the Research and Technology department on software engineering methodologies and tools, in particular in the object-oriented domain.

His current research interests include software engineering, software process, object oriented methodologies and languages, software architecture, software metrics, distributed systems and graphical user interfaces.