# Mapping the OO-Jacobson Approach to Function Point Analysis

Thomas Fetcke, Alain Abran and Tho-Hau Nguyen

Université du Québec à Montréal*

## Abstract

The Function Point software measure does not require the use of a particular development technique. However, the high-level concepts of object-oriented development methods cannot be mapped directly to the concepts of Function Point Analysis. In order to apply this software measure early in the development process, the object-oriented concepts corresponding to transactional and data function types have to be determined.

Object-oriented methods differ, especially in their early development phases. The *Object-Oriented Software Engineering* method of Jacobson et al. is based on so-called *use cases*. The viewpoint of this method is similar to Function Point Analysis in the sense that it concentrates on the application's functionality from the user's perspective.

The OO-Jacobson approach identifies the functionality of an application with the requirements *use case model*. Data types are described with a *domain* or *analysis object model* on the requirements level. Our work proposes rules to map these models to the Function Point counting procedures. With the proposed rules, it is possible to count software developed with the OO-Jacobson method. Experimental counts have been conducted for three industry projects.

## 1. Introduction

Function Point Analysis was introduced by Albrecht (1979) as a measure of the functional size of information systems. Since then, the use of Function Points has grown worldwide and the counting procedures have been modified and improved several times since their initial publication. Function Point Analysis (FPA) is now maintained by the International Function Point Users Group (IFPUG). The current version of the counting rules is recorded in the Counting Practices Manual (IFPUG 1994).

Function Point Analysis as a measurement technique is intended to be independent of the technology used for implementation. It is formulated as a counting procedure of several steps in the Counting Practices Manual.

Though independent of implementation, the rules are based on implicit assumptions on how software applications are modeled. The items counted in Function Point Analysis are identified from the documentation, following the counting rules. These items include *transaction* and *file types*. They are typically identified from the documents of traditional, structured design techniques, e. g. data flow diagrams, hierarchical process models or database structures.

### 1.1. Function Point Analysis with object-oriented design methods

Object-oriented design methods, by contrast, model software systems as collections of cooperating objects. The models created with OO methods are different, especially in the early phases. They typically do not provide the documentation mentioned above.

However, the goal of measuring the functionality that the user requests and receives is still valid for applications developed with object-oriented technology.

* Université du Québec à Montréal, Laboratoire de recherche en gestion des logiciels, Case postale 8888, succursale Centre-Ville, Montreal (Quebec) Canada H3C 3P8, Telephone: +1 (514) 987-3000 (6667), Telefax: +1 (514) 987-8477, E-Mail: c3724@er.uqam.ca, fetcke@cs.tu-berlin.de, WWW: http://www.cs.tu-berlin.de/~fetcke.

6th Workshop on Software Metrics of the German Gesellschaft für Informatik, held together with the Germany-Quebec Workshop on Software Metrics, 1996 September 19–20 in Regensburg, Germany.

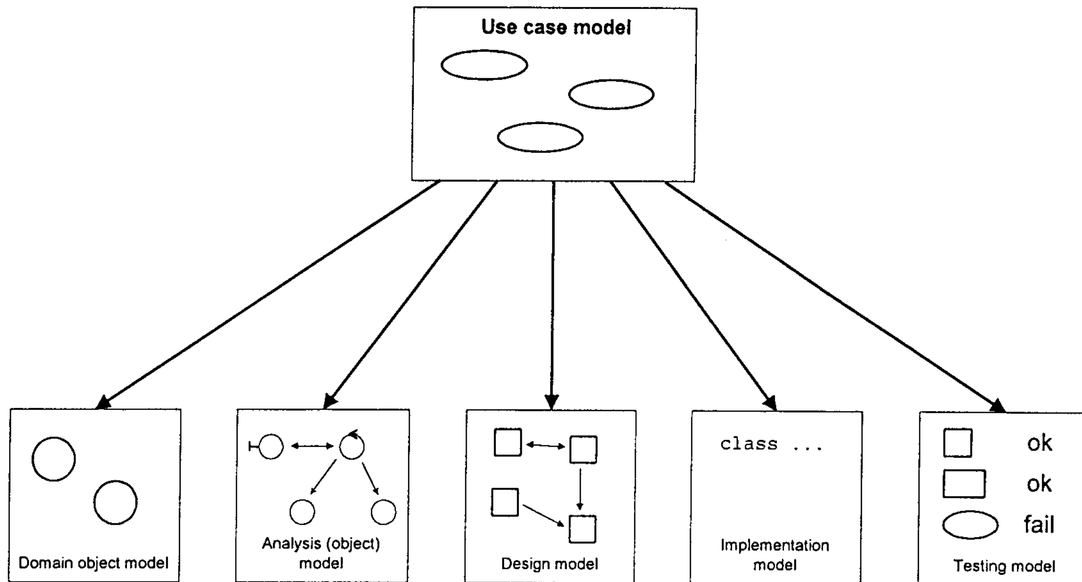


Figure 1: The use case model is the basis on which all other models of the OOSE approach are developed.

In the early phases of the software cycle, distinct object-oriented methods differ in the models developed. It is therefore necessary to discuss individual counting approaches for each method (see Jones 1995). In this project, we focus on the approach of Jacobson, Christerson et al. (1992). This method is called Object-Oriented Software Engineering (OOSE).

The OOSE method defines a process to transform formalized requirements into a sequence of models. The steps include the requirements, analysis, design, implementation and testing models (see Fig. 1).

## 1.2. Related work

Little work has been published on Function Point Analysis in the context of object-oriented software engineering techniques. These approaches are generally based on a model that displays objects together with their methods. Methods are then treated as transactions and objects as files. These approaches do not, however, properly reflect differences in OO approaches, and, in particular, are not applicable to early OOSE documents. It is also questionable whether each individual method is to be counted as a transaction[1].

Whitmire (1992) considers each class as an internal logical file and treats messages sent outside the system boundary as transactions.

The ASMA (1994) paper takes a similar approach. Services delivered by objects to the client are considered as transactions. The complexity of services is weighted based on accessed attributes and communications. Objects are treated as files, their attributes determining their complexity.

IFPUG (1995) is working on a case study which illustrates the use of the counting practices for object-oriented analysis and design. This case study, which is currently in draft form, uses object models in which the methods of classes are identical with the services recorded in the requirements. Under this assumption, the methods can be directly counted as transactions.

Karner (1993) proposes a new measure called Use Case Points for projects developed with the OOSE method. The structure of this measure is similar to Function Points, but it does not conform with the concepts of Function Points.

[1] The Function Point concept requires a transaction to be the smallest unit of activity that is meaningful to the user. The elementary process has to be self-contained and to leave the business of the application in a consistent state.

### 1.3. Function Point Analysis with OOSE

In order to apply the counting rules for Function Points to the software applications developed with OOSE, the Function Point and OOSE concepts and terminologies have to be set into relation to one other. The challenge of this research project is to identify and clarify this relationship and then to transform it into a mapping of respective concepts. The mapping must then be transformed into a set of rules and procedures. This set of rules and procedures will facilitate the counting of Function Points by practitioners in the field, helping them to apply the procedures of the Counting Practices Manual.

Two quality factors will have an impact on the ease of counting Function Points based on the results of this research work.

The first quality factor will be the quality of the mapping of the OOSE models to Function Point concepts, i. e. how easy it is to use the rules and procedures developed in this research project in order to identify and measure, from the OOSE documents, the components that contribute to functional size.

The second factor will, of course, be the degree of conformity between the project documentation and OOSE standards, as well as its quality and completeness. The measurement process is indeed very dependent on the quality and completeness of the project documentation, i. e. completeness in terms of the parts that are required for the count: if, for the project to be measured, important parts of the method are not used, it may be necessary to augment the formally documented items with information recorded differently, and determine their additional contribution to the Function Point count according to the counting rules.

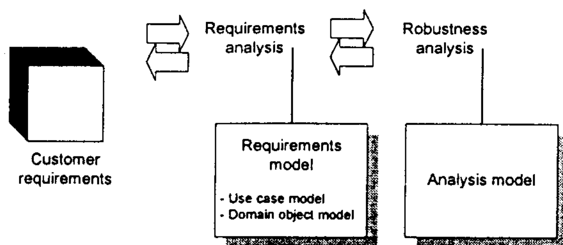


Figure 2: Analysis phases of OOSE life cycle.

## 2. Brief introduction to OOSE

The OOSE method is divided into three major consecutive processes: *analysis*, *construction* and *testing*. The analysis phase is further divided into two steps, called *requirements analysis* and *robustness analysis* (see Fig. 2). The first step derives the requirements model from the informal customer requirements. This model is expressed in terms of a *use case model*, and may be augmented by a *domain object model*. The second step, robustness analysis, then structures the use case model into the *analysis model*. The succeeding processes further transform these models, as indicated in Figure 1.

The focus of our work is the models developed in the analysis phase. As Jacobson et al. state, the requirements model can be regarded as formulating the functional requirements specification based on the needs of the users. Our goal is to count Function Points early in the life cycle, measuring the functionality requested by the user from these models.

In the following paragraphs, we give a short overview of the three models.

### 2.1. Use case model

The central model of OOSE is the use case model (cp. Fig. 1), and therefore Jacobson et al. call their approach "use case driven". This model has two types of entities, *actors* and *use cases*.

Actors represent what interacts or exchanges information with the system. They are outside the system being described. When an actor uses the system, he performs a behaviorally related sequence of actions in a dialogue with the system. Such a special sequence of actions is called a use case. With the *uses* and *extension* relationships, use cases can be structured into further detail.

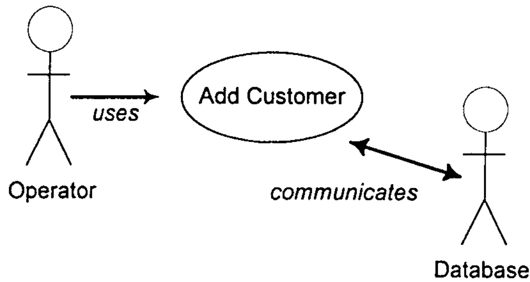


Figure 3: Use case model with the use case *Add Customer* and two associated actors.

Each use case is a specific way of using the system. The set of all use case descriptions specifies the complete functionality of the system. Figure 3 gives a small example of a use case model.

Interface descriptions of the use cases can support the use case model.

### 2.2. Domain object model

As an augmentation of the use case model, the *domain object* model consists of the objects found in the problem domain. Structuring these objects with the *inheritance* and *aggregation* relationships is an option. Some examples of domain objects are shown in Figure 4.

This model is meant to support the development of the requirements model. The OOSE method does not explicitly require a domain object model.

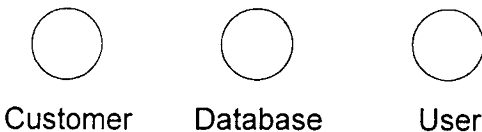


Figure 4: Domain object model.

### 2.3. Analysis (object) model

The analysis model is based on typed objects. The three object types are *entity*, *control* and *interface*. The purpose of the typing is to support the creation of a structure that is adaptable to changes. Thus, for example, changes to the interface requirements can be limited to interface objects.

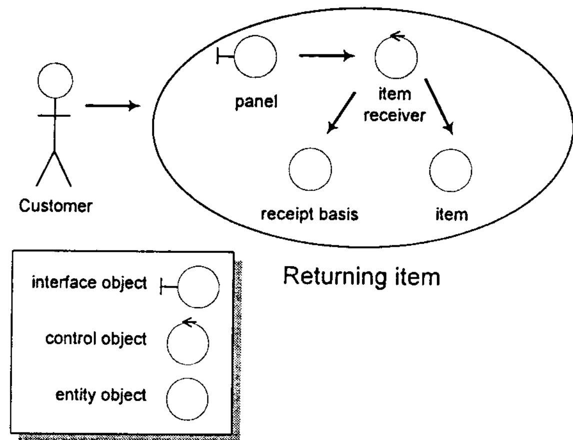


Figure 5: Analysis model for a use case.

Entity objects model information that exists in the system for a longer time, typically surviving a use case. Domain objects often become entity objects, but this is not necessarily the case.

Interface objects model behavior and information related to the presentation of the system to the outside world.

Control objects model functionality that is not naturally tied to other object types. A control object could, for example, operate on several entity objects, perform a computation and return the result to an interface object that would present it to the user.

The analysis model is derived from the use case model. The functionality of each use case is partitioned and allocated to the typed objects.

The use case example *Returning item* in Figure 5 is structured into four objects that will perform this service. The *customer* interacts with the interface *panel* when returning an item. The data on the items is stored in the entity objects *receipt base* and *item*. The *item receiver* object controls the process.

## 3. Function Point concepts

### 3.1. Function Point model

A high-level view of the Function Point Analysis model is given in Figure 6. The Function Point model specifies which component types of the software application will be measured and from which viewpoint this will be done. What is to be counted, and measured,

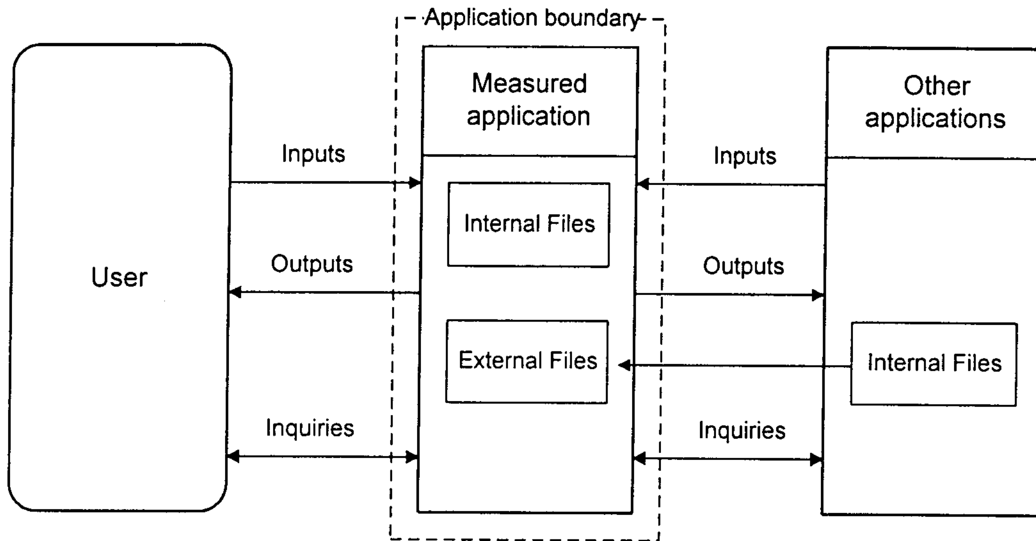


Figure 6: High-level view of Function Point Analysis with users and links to other applications. The dotted line marks the application boundary.

are the internal files and external files of the application, together with the inputs, outputs and inquiries from and to the user. Software components or deliverables which are not visible from a user viewpoint are not considered part of the Function Point measurement model.

However, within the Function Point model, the *user* concept is not equivalent to, nor restricted to, a human being as the user of the software, and other types of users are therefore admissible within its measurement model, such as mechanical devices or other software applications. Figure 6 also illustrates that, within the Function Point model, inputs, outputs and inquiries coming from, and going to, other software applications qualify as admissible items to be counted and measured.

### 3.2. Function Point measurement procedure

The Function Point measurement procedure for a software application consists of four major steps of abstraction of user-visible components of the software. The first abstraction step is identification of the application's boundary. The second major step is identification, within the previously identified boundary, of the files and transactions that have to be counted. The third step classifies the files and transactions identified in the second step into classes of file and transactional types respectively. In the last major step, the items to be counted are assigned weights based on their number of subcomponents.

The next section describes the proposed mapping of OOSE models to Function Points along these four major steps. The mapping has been formulated as rules to support their practical application.

## 4. Mapping of concepts

### 4.1. Step 1: Boundary concepts

The viewpoint of the user is essential in Function Point Analysis to determine which parts of the application contribute to the delivered functionality. The concept of the *counting boundary* is the high-level abstraction of an application which determines the artifact under measurement. Before any measurement

can take place, the object of the measurement process has to be specified.

*The Function Point counting boundary indicates the border between the project or application being measured and the external applications or user domain.*[2]

In Figure 6, the counting boundary is indicated by a dotted line. The counting boundary is always dependent on the purpose and the viewpoint of the count.

The view of the OOSE use case model corresponds to the boundary concept of Function Points, as the actors are outside the application and the use cases define the application's functionality[3].

#### 4.1.1. Actors, users and external applications

Since the OOSE concept of actors is broader than the concept of users and external applications in FPA, there cannot be a one-to-one mapping of actors to users or external applications. However, each user of the application has to appear as an actor. Similarly, every other application which communicates with the application under consideration must appear as an actor too.

In this sense, the set of actors gives us the *complete* view of the users and external applications outside the counting boundary. But the set may contain actors that are not considered as users in the Function Point view, as OOSE makes it possible to view the "functionality of the underlying system as an actor." Therefore we have to select those actors that fall into the Function Point categories of users and *external* applications.

The following rules have therefore been formulated to ensure a consistent and coherent mapping between the OOSE model and the Function Point measurement procedures.

#### 4.1.2. Proposed rules

1) Accept each human actor as a *user* of the system.

2) Accept each non-human actor which is a separate system not designed to provide functionality solely to the system under consideration as an *external application*.

3) Reject each non-human actor which is part of the underlying system, e. g. a relational database system or a printing device.

The documentation required for this step is the *use case model* displaying actors and use cases on a relatively high level.

### 4.2. Step 2: Identification of items within the boundary

#### 4.2.1. Step 2a: Transactional functions

The Function Point rules have two concepts of items that have to be counted. The first of these concepts is a user-visible elementary process which leaves the system in a consistent state, called a transaction. However, which user-visible deliverables have to be counted as transactions is determined by detailed counting rules. There can be a one-to-one, one-to-many or many-to-one relation between deliverables visible to the user and transactions, e. g. a single input screen can correspond to one *external input*, while a complex screen can contain masks for input and produce output from it. Furthermore, an input may have so many fields that it is split up into several screens.

Determining what is a transactions is therefore a process that requires analysis according to the Function Point counting rules.

Use cases are the OOSE concept corresponding to transactions. However, there is no one-to-one relation between them. As stated above, the view defined in the counting rules may imply that one use case has to be counted as one or as many transactions, depending on the tasks it performs[4]. Nevertheless, the set of use cases is the set of candidates for transactional functions.

[2] IFPUG (1994) p. 4-2.

[3] Jacobson et al. call this boundary the "system delimitation".

[4] This notion is not related to the number of actors that can execute one use case (cardinality of relationships), but to the abstract concept of different transactions performed.

#### Use cases and transactions

The level of detail in the use case model may vary. On the one hand, different flows of interaction may be grouped in one use case. On the other hand, use cases can be broken down into further detail, using the *uses* and *extension* relationships. Generally, the use case model does not provide enough information to make decisions, whether and how to count a specific use case according to the Function Point rules. For this purpose, the use cases have to be described in further detail. But, if use cases are hierarchically ordered using the uses relationship, it is possible to choose those use cases that directly communicate with users or external applications, i. e. the actors that have been identified as users or external applications respectively.

These use cases are candidates for transactions. Determining how many transactions of which types one use case corresponds to has to be made with more detailed information from use case descriptions.

#### Proposed rules

**4)** Select every use case that has a direct relation to an actor accepted by rule 1 or 2. This use case will be a candidate for one or several transactions.

**5)** Select every use case that extends a use case selected by rule 4 as a candidate. The extension may include interaction with a user or external application.

**6)** No other use cases will be selected.

The documentation required for this step is the use case model displaying actors and use cases on a relatively high level, the same as for rules 1-3.

### 4.2.2. Step 2b: Files

The second concept of items that have to be counted is the file, and the corresponding concept in OOSE is the domain object.

#### Domain objects and files

An optional part of the requirements analysis in the Jacobson approach is a model of domain objects. The domain objects are candidates for files.

In the analysis model, the objects are typed into three groups, namely entity, interface and control objects. Among these, the entity objects correspond to the Function Point notion of files, while interface objects relate to a (technical) presentation of data to the actor and control objects model the internal processes.

If the analysis model of typed objects is provided, the set of objects that have to be analyzed is limited to the entity objects and is thus typically smaller. In this case, rules 7a and 8a can be applied.

If, however, only the (untyped) domain object model exists, the set of candidates for files is the entire set of domain objects (rule 7b). These have to be analyzed according to the FP counting rules (see section 4.3 below).

#### Proposed rules

*(a) for typed objects*

**7a)** Select every object of entity type as a candidate for a file type.

**8a)** No other objects will be selected.

The documentation required for this step is the typed analysis (object) model.

*(b) for untyped objects*

**7b)** Select every domain object as a candidate for a file type.

**8b)** No other objects will be selected.

The documentation required for this step is the domain object model.

*Additional candidates for files*

Some data that are by Function Point convention considered as internal/external files may be not represented in an object model, although that functionality is required by the user. Error messages or help texts, for example,

may be a requirement and need a representation according to Function Point rules. These data are not normally modeled as objects, however.

9) If use cases make implicit use of logical files that are not represented in the object model, these files have to be included in the set of files.

The documentation required for this step are the use case descriptions and the object model used under (a) or (b).

### 4.3. Step 3: Determination of types of the items

#### 4.3.1. Step 3a: Transactional function types

Determining the types of transactions is based on a set of detailed rules in FPA. This process involves interpretation of the rules. The basis for the decisions made is the project documentation.

The rules are recorded in the IFPUG Counting Practices Manual. The relevant sections are

- "External Input Counting Rules",
- "External Output Counting Rules", and
- "External Inquiry Counting Rules".

#### 4.3.2. Step 3b: File types

Determining file types is also based on a set of detailed rules in FPA, and this process also involves interpretation of the rules. The basis for the decisions made is the project documentation.

The rules are recorded in the IFPUG Counting Practices Manual. The relevant section is "ILF/EIF Counting Rules".

### 4.4. Step 4: Weighting factors

The weights of transactions and files are based on detailed rules in the Counting Practices Manual. The rules require the determination of data elements, record types and files that are referenced. This information has to be extracted from detailed documentation of the use cases (for transactions) and of the domain objects (for files).

However, if this level of detailed documentation is not (yet) available, the weights can be estimated, based on expert judgment or experience. This makes it possible to obtain an estimate of the Function Point count in an early development phase.

## 5. Experimental results

The rules proposed in section 4 have been used to count three industry projects that were developed with the OOSE approach. The documentation provided included use case models and domain object models together with textual descriptions of these models.

The calculated sizes of the projects in Function Points were:

| | |
|---|---|
| Project 1 | 265 |
| Project 2 | 181 |
| Project 3 | 215 |

The rules proposed have been useful and have made the Function Point counts for these projects feasible.

## 6. Summary

In this work we have demonstrated the applicability of Function Points as a measure of functional software size to the object-oriented Jacobson approach, OOSE. This supports the thesis that Function Point Analysis measures independent of the technology used for implementation.

The high-level OOSE models had to be mapped to Function Point concepts. This mapping was divided into four levels of abstraction. The mapping has been expressed in a small set of rules, thus also supporting the actual counting procedure.

These rules were successfully applied to three industrial software projects.

Future work in the field has to deal with the application of Function Point Analysis to other object-oriented design techniques. This would make the measure available for these new techniques, and would make it possible to compare the counts of projects that were developed with different techniques. The resulting mappings of concepts could be incorporated in a future release of the IFPUG case study.

## Acknowledgements

We thank Ericsson for the support and funding of this work. This research was carried out at the Laboratoire de recherche en gestion des logiciels at the Université du Québec à Montréal. This laboratory is made possible through a partnership with Bell Canada. Additional funding is provided by the Natural Sciences and Engineering Research Council of Canada. The opinions expressed in this article are solely those of the authors.

# Literature

Albrecht, A. J. (1979). *Measuring Application Development Productivity.* IBM Applications Development Symposium, Monterey, CA.

ASMA (1994). *Sizing in Object-Oriented Environments.* Victoria, Australia, Australian Software Metrics Association.

Goh, F. (1995). *Function Points methodology for object oriented software model*, Ericsson Australia Pty Ltd.

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

IFPUG (1995). Function Point Counting Practices: Case Study 3 - Object-Oriented Analysis, Object-Oriented Design (Draft), International Function Point Users Group.

Jacobson, I., M. Christerson, et al. (1992). Object-Oriented Software Engineering. A Use Case Driven Approach, Addison-Wesley.

Jones, J. (1995). *FP Issues for O-O and K-B Systems.* IFPUG 1995 Spring Conference, Masville.

Karner, G. (1993). *Resource Estimation for Objectory Projects*, Objectory Systems.

Whitmire, S. A. (1992). Applying function points to object-oriented software models. in Software engineering productivity handbook. J. Keyes, McGraw-Hill, pp. 229–244.

## About the authors

**Thomas Fetcke** received his diploma in computer science from the Technical University of Berlin in 1995. From September to December of 1994, he was with the Gesellschaft für Mathematik und Datenverarbeitung (GMD), where he studied object-oriented software metrics. Currently, he is pursuing his Ph. D. on the Function Point software measure in the context of object-oriented software. In April of 1996, he joined the Software Engineering Management Research Laboratory of the Université du Québec à Montréal. He is also a Certified Function Point Specialist.

**Alain Abran** is currently professor at the Université du Québec à Montréal. He is the research director of the Software Engineering Management Research Laboratory and teaches graduate courses in Software Engineering. He has been in a university environment since 1993.

He has over 20 years of industry experience in information systems development and software engineering. The maintenance measurement he developed and implemented at Montreal Trust (Montreal, Canada) has received one of the 1993 *Best of the Best* awards from the Quality Assurance Institute (Orlando, Florida, USA).

Dr. Abran received his M.B.A. and Master of Engineering degrees from University of Ottawa, and holds a Ph. D in software engineering form École Polytechnique de Montréal. His research interests include software productivity and estimation models, software metrics, function points measurement models and econometrics models of software reuse. He has given presentations in various countries including Canada, USA, France, Germany, Italy and Australia.

**Tho-Hau Nguyen** graduated in Computer Science, and Management from École Polytechnique de Montreal, McGill university and Université du Québec à Montréal. Since 1979 he has worked in computer science in private and educational sectors. In 1983, he joined the Université du Québec à Montréal as a regular faculty member of the department of Computer Science. His research areas include object-oriented database design, and metrics.