Mapping the OO-Jacobson Approach into Function Point Analysis

Thomas Fetcke, Alain Abran and Tho-Hau Nguyen

Université du Québec à Montréal
Software Engineering Management Research Laboratory
Case postale 8888, succursale Centre-Ville
Montréal (Québec) Canada H3C 3P8
Phone: +1 (514) 987-3000 (8900)
Fax: +1 (514) 987-8477
E-mail: fetcke@cs.tu-berlin.de, abran.alain@uqam.ca
WWW: http://saturne.info.uqam.ca/Labo_Recherche/lrgl.html

Abstract

Function Point Analysis measures user requested functionality independent of the technology used for implementation. Software applications are represented in an abstract model that contains the items that contribute to the functional size. When Function Point Analysis is applied to object-oriented software, the concepts of the development method have to be mapped into that abstract model.

This article proposes a mapping of the use case driven Object-Oriented Software Engineering method by Jacobson et al. into the abstract Function Point model. The mapping has been formulated as a small set of concise rules that support the actual measurement process. Our work demonstrates the applicability of Function Point Analysis as a measure of functional software size to the OO-Jacobson approach. This supports the thesis that Function Point Analysis measures independent of the technology used for implementation and that it can be used in the object-oriented paradigm.

1. Introduction

Function Point Analysis (FPA) was introduced by Albrecht [1] 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 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 [5].

Function Point Analysis as a measurement technique is intended to be independent of the technology used for implementation. It is formulated as a counting method of several steps in the Counting Practices Manual. This counting method is implicitly based on a high-level model of software applications. Any software requirements documentation has to be mapped into this model. The actual measurement, the mapping into numbers, takes then place in the context of the abstract model. Nevertheless, both abstraction and assignment of numbers are defined as a cohesive process in the Counting Practices.

Though independent of implementation, the counting rules are thus based on implicit assumptions on the abstract model of software applications. The items in the abstract model that are then counted include *transaction* and *file types*. These items 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 traditional documentation in data flow diagrams or database structures.

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

In the early phases of the software cycle, distinct object-oriented methods differ in the types and structures

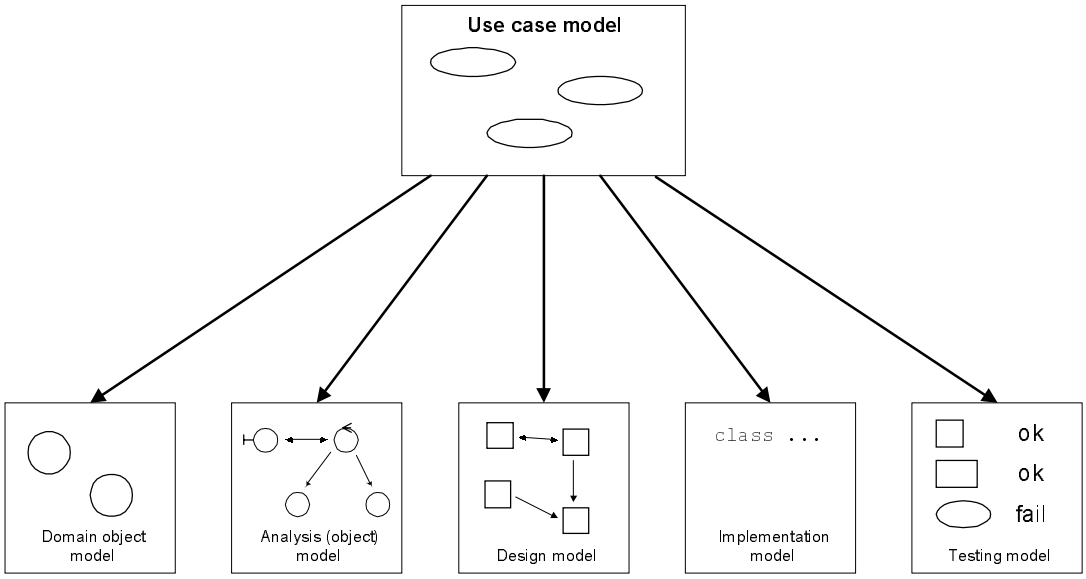

Figure 1: The use case model is the basis on which all other models of the OOSE approach are developed.

of the models developed. It is therefore necessary to discuss individual counting approaches for each method (see [8]).

In this project, we focused on the approach of Jacobson et al. [7]. The authors call their method 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. The use case model is the basis on which all other models are developed. Together with the domain object model it forms the requirements model (see Fig. 1).

The objectives of our project were:

1. The application of Function Point Analysis following the IFPUG standard.
2. To measure for software developed with the OOSE method.
3. To count early in the life cycle, in the requirements analysis phase.

1.2. Related work

Little work has been published on Function Point Analysis in the context of object-oriented software engineering techniques. None of the approaches published applies the IFPUG standard for Jacobson's OOSE method.

On the one hand, the majority of the approaches do not use the standard FPA defined by IFPUG, but defines variants or new measures. In consequence, the functional size measured with these approaches is not comparable with that of standard FPA counts.

On the other hand, only one publication was based specifically on the OOSE method. Given the early models developed with the Jacobson approach, none of the other approaches can be applied, as those measures cannot handle use cases.

Whitmire bases his approach on a class diagram including messages sent between classes [10]. He considers each class as an internal logical file and treats messages sent outside the system boundary as transactions. He rejects the existence of external interface files, because *externally maintained data are never accessed directly*. The standard FPA, however, understands under an external interface file a group of data that is read but not maintained by the measured application.

We also disagree with the view that every message should be a transaction. The FPA concept of transactions is based on smallest activities meaningful to the users. A single message is not necessarily meaningful to the user.

The ASMA paper [2] takes an approach similar to that of Whitmire. 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.

The paper refers to the IFPUG standard, but it replaces the items that count and their internal structures with objects and services. Again, it is questionable if there exists a simple one-to-one relationship between

original FPA concepts and objects and services. The ASMA article also discusses internal size measurement that is not considered in FPA.

IFPUG is working on a case study [6] 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.

While the case study has the objective to comply with the FPA standard, it is based on an object model. It cannot be used with the OO-Jacobson approach, because the assumption that the required services are represented in object methods does not hold for early life cycle phases.

Karner proposes a new measure called Use Case Points for projects developed with the OOSE method [9]. The structure of this measure is similar to Function Points, but it is not based on the items that represent functional size in FPA.

Its results cannot be compared directly to size measured in Function Points.

Gupta and Gupta define Object Points as a new measure which is different from Function Points [4]. The overall structure is similar to FPA. But instead of user functionality, objects are the items that are counted. Objects are weighted based on their "complexity", derived from so called effective attributes, instance and message connections.

Again, the results obtained with this measure proposed by Gupta et al. cannot be compared to size measured in Function Points.

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 in relation to each 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 factors will have an impact on the ease of counting Function Points based on the results of this research work.

The first factor will be the ease of use 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.

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.

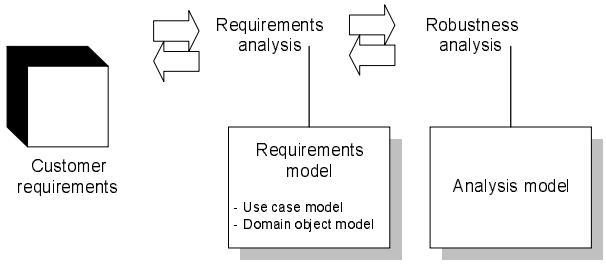

Figure 2: Analysis phase of the OOSE life cycle.

At the focus of our work are 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 (cf. 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 subjects that interact or exchange information with the system. They are outside the system being described. When an actor uses the system, he follows a course of behaviorally related actions in dialog with the system. Each such special sequence of actions is called a *use case* and defines 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.

Use cases can be refined with the *uses* and *extension* relationships. Common parts of use cases can be extracted and modeled as *abstract* use cases, that are then used by other use cases, for example *Print* in Fig. 3.

With the extension relationship a use case may be inserted into the flow of action of an existing use case. The extension adds functionality independently of the existing, in itself complete use case. Exceptional situations can be handled in this way.

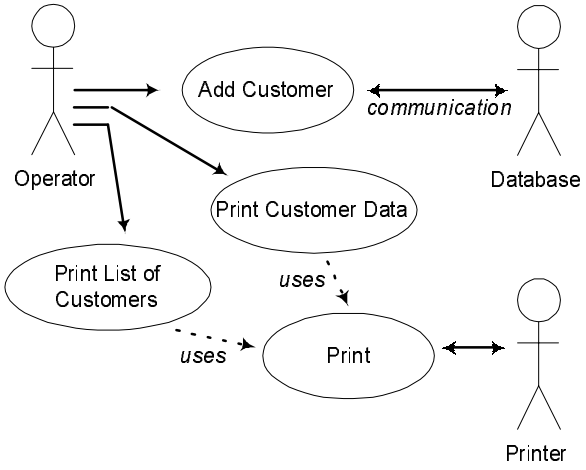

Figure 3: Use case model with four use cases and three associated actors. *Print* is an abstract use case.

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. These objects can be structured with the *inheritance* and *aggregation* relationships. Some examples of domain objects are shown in Figure 4.

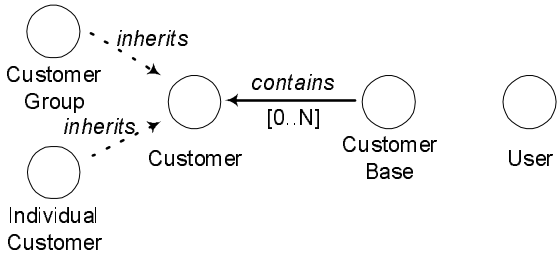

Figure 4: Domain object model with inheritance and aggregation.

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

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. Entity objects can be structured with inheritance and aggregation relationships as described above for domain objects.

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

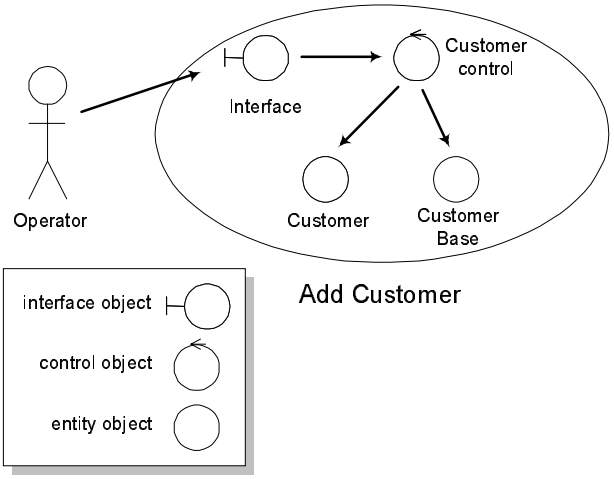

Figure 5: Analysis model for the *Add Customer* use case.

Control objects model functionality that is not naturally tied to the 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 *Add Customer* in Figure 5 is structured into four objects that will perform this service. The *Operator* interacts with the *Interface* when adding customer data. This data is stored in the entity objects *Customer* and *Customer Base*. The *Customer control* object controls the process.

3. Function Point concepts

3.1. Function Point model

A high-level view of the FPA 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, 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 sub-components.

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.

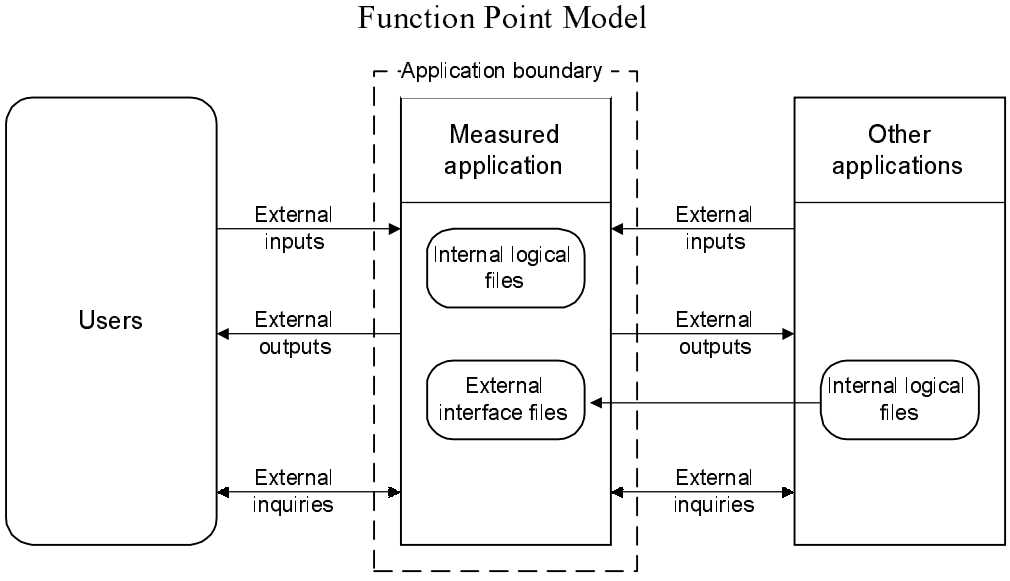

Figure 6: High-level view of the abstract Function Point model with users and links to other applications. The dotted line marks the application boundary.

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.*[1]

In Figure 7, this counting boundary is indicated. The 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[2]. The similarities of these concepts, however, are not exact equalities. We therefore have to discuss a mapping of the use case model into FPA.

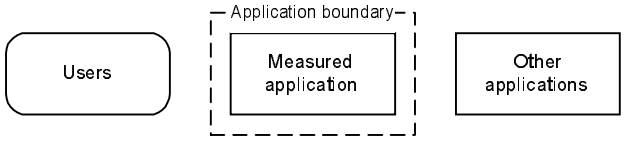

Figure 7: Step 1 – Identification of the counting boundary.

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.

Proposed mapping 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.

The result is a representation of the application boundary as a set of users and applications external to the one under consideration.

4.2. Step 2: Identification of items within the boundary

In Function Point Analysis, two sorts of items determine functional size: *transactional functions* and *files* (see Fig. 8). To identify these different sorts of items, we divide this step into two sub-steps.

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

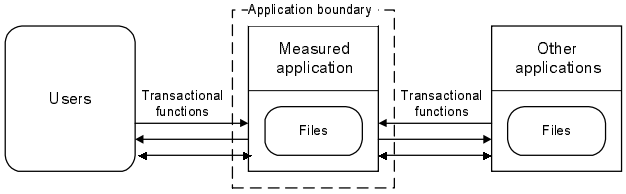

Figure 8: Step 2 – Identification of items within the counting boundary.

[1] See [5, p. 4-2].
[2] Jacobson et al. call this boundary the "system delimitation".

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[3]. Nevertheless, the set of use cases is the set of candidates for transactional functions.

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. Which of the candidates identified in these and the rules later on will be counted is due to evaluation of the actual Function Point rules.

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

The mapping is formulated explicitly in the following rules.

Proposed mapping 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. Additionally, the application boundary identified by rules 1-3 is needed as input.

The result is a set of candidates for transactions. In step 3 we will decide whether to count a candidate as a transaction, based on the original Counting Practices Manual. Furthermore, a candidate may then be counted as one or several transactions.

Step 2b: Files. The second concept of Function Point items that have to be counted is the *file*, and the corresponding object-oriented concept is the object. As pointed out in section 2, we have to consider two types of object-models.

The model of domain objects identifies the data concepts which are relevant for the application domain.

A model of domain objects is an optional part of the requirements analysis in the Jacobson approach. 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 (rules 7b and 8b).

In either case, the items to be counted will be selected from the candidates in the third step (see section 4.3 below).

As mentioned in section 2.2 and 2.3, objects can be structured with two relationships, these structures have to be analyzed in detail.

Aggregation relationships. Aggregation orders objects hierarchically in structures similar to traditional approaches. In Function Point terms the *parts of* an object do not correspond to files themselves, but to the logical structure of the file concept. *Optional* and *mandatory subgroups* of files are called record element types (RET). An object that is aggregated into (part of) another object constitutes such a subgroup. Even if the relationship is nested over several levels, the objects on various levels can be interpreted as subgroups of the top-level object.

Both [6] and [10] assign one RET for every class that is part of another class, counting these as subgroups

[3] 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.

of a single file rather than as logical files themselves. We agree to this view and formulate it in rule 9.

Inheritance relationships. Inheritance is a genuine object-orientated concept and does not have a direct representation in FPA. However, as this relationship realizes a specialization, it can be translated into terms of mandatory and optional subgroups.

In [6] the complete hierarchy is taken as a single file with RETs for each subclass.

[10] states that in general one class should represent one internal file. For classes that are part of an inheritance structure, Whitmire states: "If the generalization is truly part of the application domain, it is counted as a separate logical file." If the generalization was built for the ease of modeling, Whitmire counts the general class with each specialized class.

In our view, however, there is no general mapping for this case, different intentions have to be represented differently.

Abstract objects that represent common attributes are not visible to the user and do not relate to logical files themselves. They rather define a subgroup in each of their sub-objects.

Concrete super-objects, visible to the user, are candidates for files. Their sub-objects can be visible to the user and are then candidates for files themselves. Otherwise, they are an optional subgroup of their super-object and will be represented as a RET for the file related to the super-object.

We formulate rules 10 and 11 to represent inheritance.

Proposed mapping rules.

(a) for typed objects

7a) Select every object of entity type as a candidate for a logical file, unless rules 9-11 state otherwise.

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 logical file, unless rules 9-11 state otherwise.

8b) No other objects will be selected.

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

for aggregation relationships

9) A domain or entity object that is a part of another object (is aggregated into another object) is not a candidate for a logical file, but it is a candidate for a record element type (RET) for the file related to the aggregating top-level object.

The documentation required for this step is the object model used under (a) or (b).

for inheritance relationships

10) An abstract object is not a candidate for a logical file. It is a candidate for a RET for each object that inherits its properties.

11) Sub-objects of a concrete object are candidates for a logical file or for a RET of that object. If these sub-objects are not counted as logical files themselves, they are optional subgroups of the file related to their super-object.

The documentation required for this step is the object model used under (a) or (b).

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.

12) 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).

Results of step 2b. Rules 7-12 identify a set of candidates for files. Which of these candidates will actually be counted as a file has to be determined in step 3 with the original rules from the Counting Practices Manual. From the set of candidates for record element types, RETs are determined in step 4. The original counting rules are used in step 4 as well.

4.3. Step 3: Determination of types of the items

The items identified in step 2 will now be classified. Transactional function types are external inputs, external outputs and external inquiries. File types are internal logical files and external interface files (see Fig. 6).

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.

It should be noted that the directions of the arrows in the use case model give no indication of the transaction type, they represent a view different from the FPA model.

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".

This step requires detailed documentation for the use cases. The set of candidates identified in rules 4-5 in step 2a forms the basis for the analysis.

The result determines the set of transaction types that have to be counted in FPA.

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".

The basis for this step is the set of candidates determined by rules 7-12 in step 2b. The rules have to be evaluated for each candidate. Detailed documentation from the domain object or analysis model, respectively, is therefore needed.

The resulting set of file types forms the items to be counted with FPA.

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 element types (DET), record element types (RET) and file types that are referenced (FTR), illustrated in Figure 9. This information has to be extracted from detailed documentation of the use cases for transactions and of the domain objects for files.

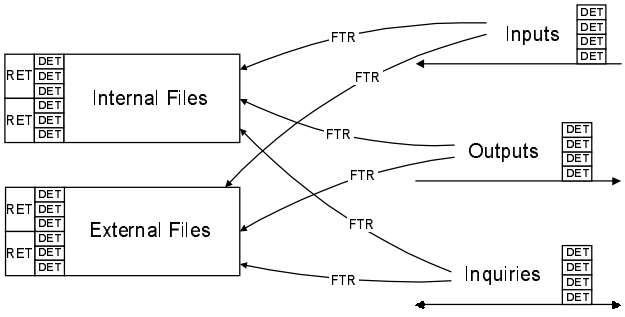

Figure 9: Step 4 – Determination of weights.

The concept corresponding to DETs is the attribute of an object. DETs for a file are the attributes of the corresponding object. RETs are determined by subgroups in these DETs and by the rules 9-11.

DETs for transactions are strongly related to the DETs maintained and/or read by the transaction in files. Additionally to the documentation of use cases, the DETs identified for files will thus support the determination of DETs for transactions. The number of file types referenced results from the number of files maintained and/or read by the transaction.

However, if the necessary level of detail in the 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.

Proposed mapping rules.

13) Attributes of objects are candidates for data element types (DET) for files and for the transactions by which it is read and/or maintained.
14) Candidates for record element types (RET) are determined by subgroups of files and by rules 9-11.
15) Each object maintained and/or read by a use case counts as a file type referenced (FTR) for the associated transaction(s), if and only if the object has been identified as a file in step 3.

Input for this step are the sets of transaction and file types determined in step 3, detailed documentation of use cases and the relevant objects as well as the results of rules 9-11 in step 2b.

DETs, RETs and FTRs are determined from their sets of candidates with the "Complexity and Contribution Rules" for the appropriate function types in the Counting Practices Manual. The results are the components of file and transaction types that determine the weighting factors.

5. Counting experiments

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

The result obtained in the four steps of section 4 is the *unadjusted* Function Point count. Function Point Analysis defines an adjustment factor that takes so called *global system characteristics* into account, e. g. data

communications, performance or end-user efficiency. This adjustment is external of and independent from the concepts of the abstract FPA model. The global system characteristics determine an adjustment factor that is multiplied with the unadjusted count. Adjustment factors were not calculated for the three sample projects.

The calculated sizes of the projects in unadjusted Function Points were:

Project 1	265
Project 2	181
Project 3	215

Compared with the approaches proposed in literature, our mapping rules have certain advantages.

- The mapping rules are based on the standard FPA defined in the IFPUG Counting Practices Manual. This widely used measure for functional size is designed to measure independent of technology and to be consistent among various projects and organizations. Measurement results can be compared between different development methods.
- The count is based on requirements models, which are the first models available in the life cycle. For the purpose of effort estimation based on Function Points, this is an essential prerequisite.
- The approach is formulated in a set of mapping rules that supports the actual counting process. The approach is thus explicit and can be understood and criticized in detail.

Our approach also has some limitations.

- We did not consider the type of count defined in FPA, i. e. whether an application or a project is counted. This concept sits on top of the FPA model presented in section 3 and does not influence it. We also left the general system characteristics out of consideration, because there determination rules are independent of any development models. They rather formulate general requirements like performance, reusability, etc.
- Additional research issues with FPA have been ignored. Those issues include real-time software characteristics and measurement of functional reuse. These areas would have exceeded the scope of the project. They are also not solely relevant with object-oriented systems.
- The main limitation is the focus on the Jacobson OOSE method. Our mapping rules are based on the requirements models of this approach and cannot be applied to methods that do not develop these models. However, it is an advantage of this focus on OOSE, that the models used are unambiguously defined in the method.

6. Summary

In this work we have demonstrated the applicability of Function Point Analysis 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 and that it can be used in the object-oriented paradigm.

Our solution is based on the standard FPA defined in the IFPUG Counting Practices Manual. Measurement results are therefore comparable with software developed with other methodologies.

The focus on the OOSE approach binds the use of our mapping to software developed in this approach. While this certainly is a limitation, it is also an advantage, because the models used are defined unambiguously in the OOSE method. Furthermore, our mapping rules support the counting from the OOSE requirements model, which no other approach in the literature allows.

Finally, the mapping rules that describe the four FPA steps directly support the actual counting procedure.

Future work in the field has to deal with the application of FPA to other object-oriented design techniques. This would make the measure available for these 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 studies.

Acknowledgments

We thank Ericsson for the support and funding of this work. This research was carried out at the Software Engineering Management Research Laboratory at the Université du Québec à Montréal. The opinions expressed in this article are solely those of the authors.

Literature

[1] Albrecht, A. J. *Measuring Application Development Productivity*. IBM Applications Development Symposium, Monterey, CA, 1979.

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

[3] Goh, F. *Function Points methodology for object oriented software model*, Ericsson Australia Pty Ltd., 1995.

[4] Gupta, R. and S. K. Gupta. *Object Point Analysis*. IFPUG 1996 Fall Conference, Dallas, Texas.

[5] Function Point Counting Practices Manual, Release 4.0. Westerville. Ohio, International Function Point Users Group, 1994.

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

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

[8] Jones, J. *FP Issues for O-O and K-B Systems*. IFPUG 1995 Spring Conference, Masville.

[9] Karner, G. *Resource Estimation for Objectory Projects*, Objectory Systems, 1993.

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

About the authors

Thomas Fetcke received his diploma (master's degree) in computer science from the Technische Universität 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 at the Otto-von-Guericke-Universität Magdeburg. In April of 1996, he joined the Software Engineering Management Research Laboratory at 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 MBA and Master of Engineering degrees from University of Ottawa, and holds a Ph.D. in software engineering from 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.