IDEA GROUP PUBLISHING

701 E. Chocolate Avenue, Hershey PA 17033-1117, USA
Tel: 717/533-8845; Fax 717/533-8661; URL-http://www.idea-group.com

ITP4206

FOOM–An Integrated Methodology for Analysis and Design of Information Systems

Judith Kabeli and Peretz Shoval
Department of Information Systems Engineering, Ben-Gurion University, Israel
Tel: +972-8-6477003, Fax: +972-8-6477527, {Kabeli, Shoval}@bgumail.bgu.ac.il

ABSTRACT

FOOM is a Functional and Object-Oriented Methodology for analysis and design of information systems, which combines the two essential software-engineering paradigms: the functional- (or process-) oriented approach and the object-oriented (OO) approach. In FOOM, system analysis includes both functional and data modeling activities, thereby producing both a functional model and a data model. These activities can be performed either by starting with functional analysis and continuing with data modeling, or vise versa. FOOM products of the analysis phase include: an initial object schema, which can be created from the user requirements and a hierarchy of OO-DFDs (object-oriented data flow diagrams). System design is performed according to the OO approach. The products of the design phase include: a) a complete object schema, consisting of the classes and their relationships, attributes, and method interfaces; b) object classes for the menus, forms and reports; and c) a behavior schema, which consists of detailed descriptions of the methods and the application transactions, expressed in pseudo-code and message diagrams. The seamless transition from analysis to design is attributed to ADISSA methodology, which facilitates the design of the menus, forms and reports classes, and the system behavior schema, from DFDs and the application transactions.

INTRODUCTION

Many paradigms for system analysis and design have been proposed over the years. Early approaches have advocated the functional approach (DeMarco, 1978; Yourdon & Constantine, 1979). The development of object-oriented (OO) programming languages gave rise to a new approach, which maintains that in order to develop information systems in such languages, it is recommended to perform object oriented analysis and design. Many OO methodologies were developed (e.g. Booch, 1991; Coad & Yourdon, 1990; Coad & Yourdon, 1991; Jacobson, 1992; Martin & Odell, 1992; Rumbaugh, Blaha, Premerlani, Eddy & Lorensen, 1991; Shlaer & Mellor, 1988; Shlaer & Mellor, 1992; Wirfs-Brock, 1990), and the area is still evolving. The multiplicity of diagram types in the OO approach has been a major motivation for developing the Unified Modeling Language (UML) (see, for example, Boosh, Rumbaugh & Jacobson, 1999, Clee & Tepfenhart, 19971; Larman, 1998; Maciaszek, 2001; UML-Rose, 1998). UML was developed in order to produce a standard ("unified") modeling language. It consists of several types of diagrams with well-defined semantics and syntax, which enable presenting a system from different point of views.

Information systems development is a multi-phase process in which the analysis and design are of primary importance. Therefore it is vital to examine which approaches and methods are appropriate to perform each of these phases. On the one hand, those who adopt the OO approach claim that using data abstraction at the analysis phase, producing a model of reality by means of classes, is preferable to producing a functional model, because the real world consists of objects. However, as far as we know no such study has shown that the OO approach is more effective than the functional/data approach in the development of business-oriented information systems.

OO methodologies tend to neglect the functionality aspect of system analysis, and do not clearly show how to integrate the systems functions, or transactions, with the object schema. One sometimes gets the impression that the functionality of the system is expressed by means of methods that are encapsulated within objects, thus disregarding functional requirements that cannot be met by simple methods. On contrast, based on vast experience in performing functional analyses with DFDs, we have met with no problems as a means to express the functionality of the system; the only problem was how to continue from them to the following phases of development.

In our opinion, since process and object are both fundamental building blocks of reality, the analysis phase must cover both the functional and the data aspects. The functional approach, using DFDs, is suitable for describing the functionality of the system, while ERD or OO-schemas are suitable for modeling the data structure. Since the OO approach is the one most appropriate for performing the design phase, we suggest performing data modeling by creating an initial OO-schema. It seems more effective to produce an initial OO-schema already at the analysis phase, and then to use it as input to the design phase. (The term initial OO-schema will be clarified later on).

For the design phase it is crucial to provide a smooth and seamless transition to system implementation. Since there is an agreement on the advantages of OO programming, it is also desirable to design the system with methods and tools that belong to the OO family. Therefore, we conclude that in order to perform each of those development phases with its most appropriate method, there is a need to integrate the functional- and object-oriented approaches.

Dori's Object-Process Methodology - OPM (Dori, 1996; Dori, 2001), indeed integrates the two approaches. It utilizes a single graphic tool, Object-Process Diagram (OPD), at all development phases. However, since OPD defines a new notation that combines DFD and OO diagrams, it includes a great many symbols and rules. It seems to us that such diagrams are not so easy to construct and comprehend for large-scale systems, and that reality has to be modeled by means of simple notations, which are easy to learn, comprehend and utilize. A single hybrid notation, like OPM, must be very rich in order to elicit all those points of view, thus leading to a complex, perhaps distorted model of reality. On the other hand, multiplicity of models and corresponding diagramming tools, as found in UML, may also be too complicated. Too many diagram types (even standard) can hamper coherent understanding and lead to the production of erroneous models and systems.

We are looking for an optimal way to integrate the process and object approaches. Since users express their information needs in a functional and data manner, and not by means of an object structure and behavior, an appropriate (natural) method to carry out the analysis task is by functional/data analysis. On the other hand, the design should be made through the OO approach to facilitate the transition of the design to OO programming, which has proved itself to be a better approach to implement software. The integration of the two approaches is made possible because it applies principles and techniques taken from the ADISSA methodology, especially transactions design. A **transaction** is a process that supports a user who performs a business function, and is triggered as a result of an event. A transaction in a DFD consists of elementary functions that are chained trough dataflows, and of data-stores and external-entities that are connected to those functions. The transactions design enables the transition from

This paper appears in *Issues and Trends of Information Technology Management in Contemporary Organizations,* the proceedings of the Information Resources Management Association International Conference.

functional analysis DFDs to an OO model that consists of object and behavior schemas (More details can be found in the ADISSA references, Shoval 1988, Shoval 1990, and in Shoval, 1991).

OUTLINE OF FOOM METHODOLOGY

We represent a briefly description of FOOM, along with an example – the *IFIP Conference* (Mathiasssen and el., 2000). A more detailed description of FOOM can be found in (Shoval and Kabeli, 2001). Parts of the analysis specifications of the *IFIP Conference* example are present below, in Figures 1-3. We show the initial OO schema (Figure 1), the root OO-DFD-0 (Figure 2), and OO-DFD-1 (Figure 3), which describes the *Program Management* function.

The Analysis Phase

The analysis phase consists of two main activities: data modeling and functional modeling. They can be performed in any order. The products of this stage are a data model, in the form of an initial OO-schema, and a functional model, in the form of hierarchical OO-DFDs (supported by a data-dictionary).

The initial OO-schema consists of "data" classes (also termed "entity" classes), namely classes that are derived from the application requirements and contain "real world" data. (Other classes will be added at the design stage.) Each class includes attributes of various types (e.g. atomic, multi-valued and tupels of attributes, keys, sets, and reference attributes). Association types between classes include "regular" (namely 1:1, 1:N and M:N) relationships, with proper cardinalities, generalization-specialization (is-a, or inheritance) links between super and subclasses, and aggregation-participation (is-part-of) links. Note that in our model, relationships are signified not only by links between respective classes, but also by reference attributes to those classes. However the initial OO-schema does not include methods; these will be added at the design phase. An example of the OO-schema is shown in Figure 1.

The OO-DFDs specify the functional requirements of the system. Each OO-DFD consists of general or elementary functions, external entities – mostly user-entities, but also time and real-time entities, object-classes (instead of the "traditional" data-stores), and the data flows among them. Examples are shown in Figures 2-3. Note that a general function is represented as a double circle, meaning that its sub-functions are described in a separate sub-OO-DFD. Classes within the OO-DFDs correspond to classes, which exist in the initial OO-Schema.

In order to determine which of the two alternative orders to perform the analysis stage, namely start with data modeling or with functional modeling, we conducted a comparative experiment. Subjects were undergraduate students of Information Systems who studied FOOM. They were randomly divided into two groups; members in each group received a literal description of the *IFIP Conference* system and were asked to produce the analysis specifications. Subjects in one group were asked to produce functional model first, and then data model and subjects in the other group did the opposite.

For each subject we checked the correctness of specifications, using a grading scheme that was defined. Then we computed the average grades of the each subject per category, model and overall (namely the two models combined). Based on that we computed the differences of the average grades between the groups.

Figure 1: Initial OO-Schema of IFIP Conference

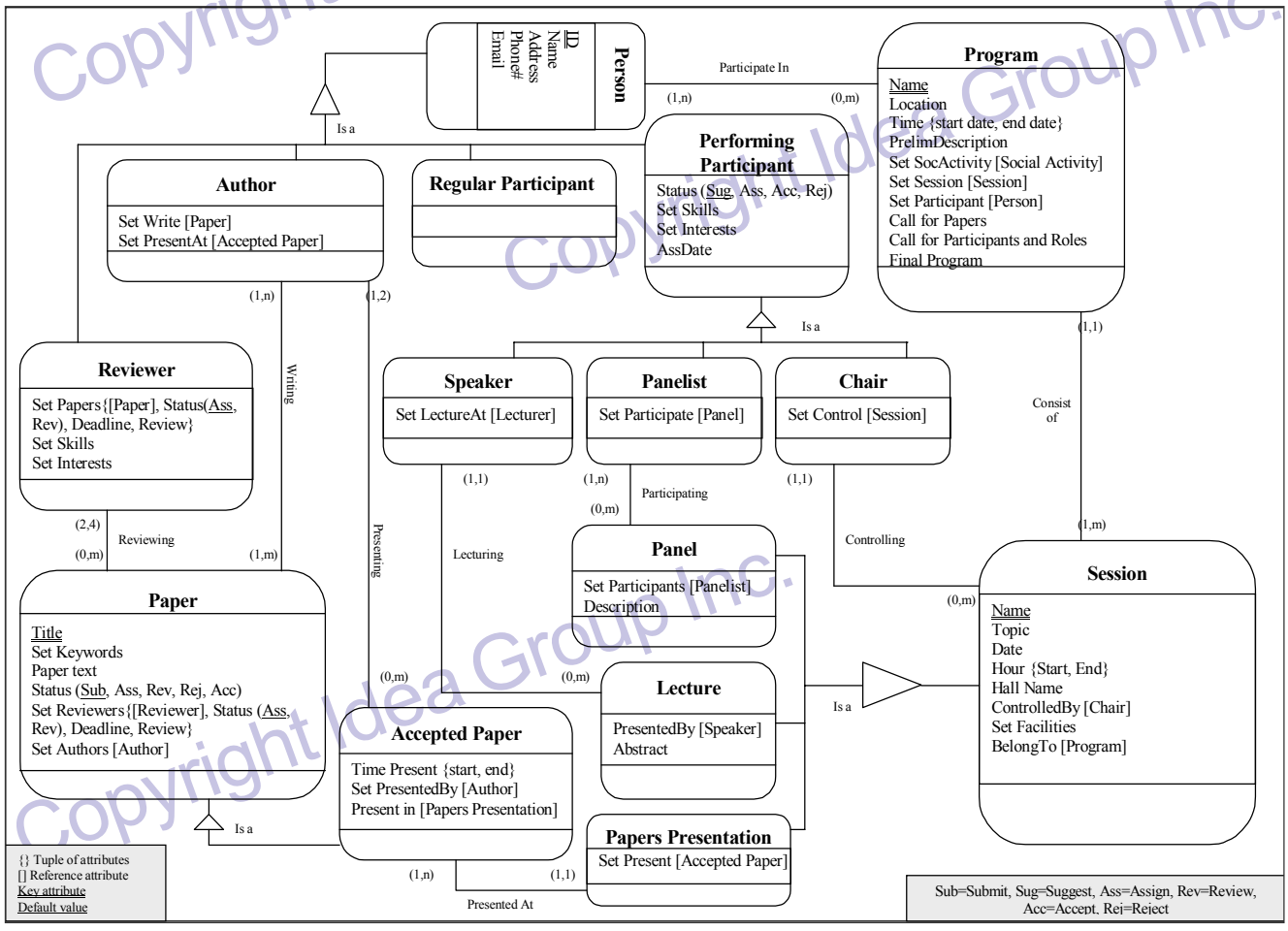


Figure 2: OO-DFD-0–The IFIT Conference

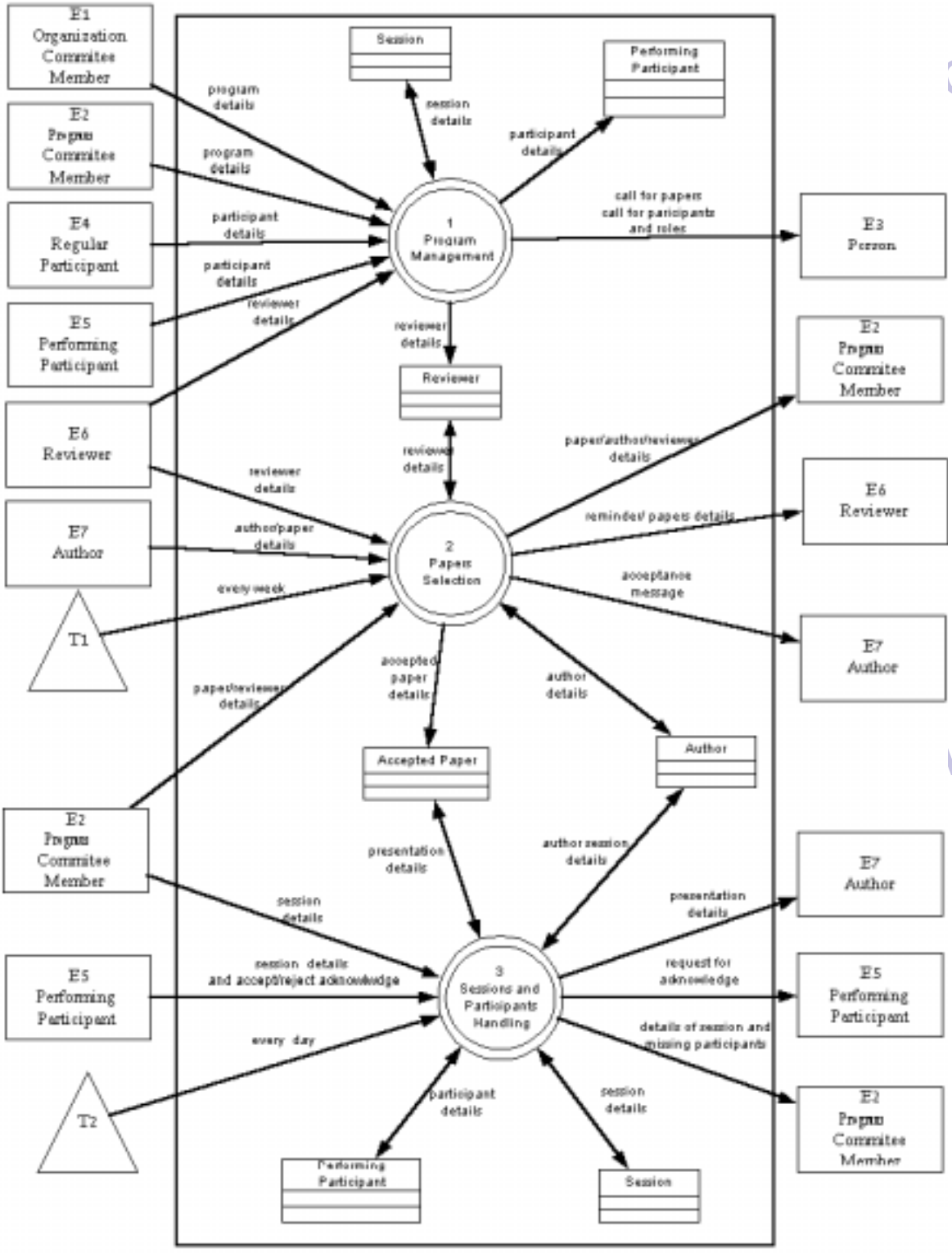
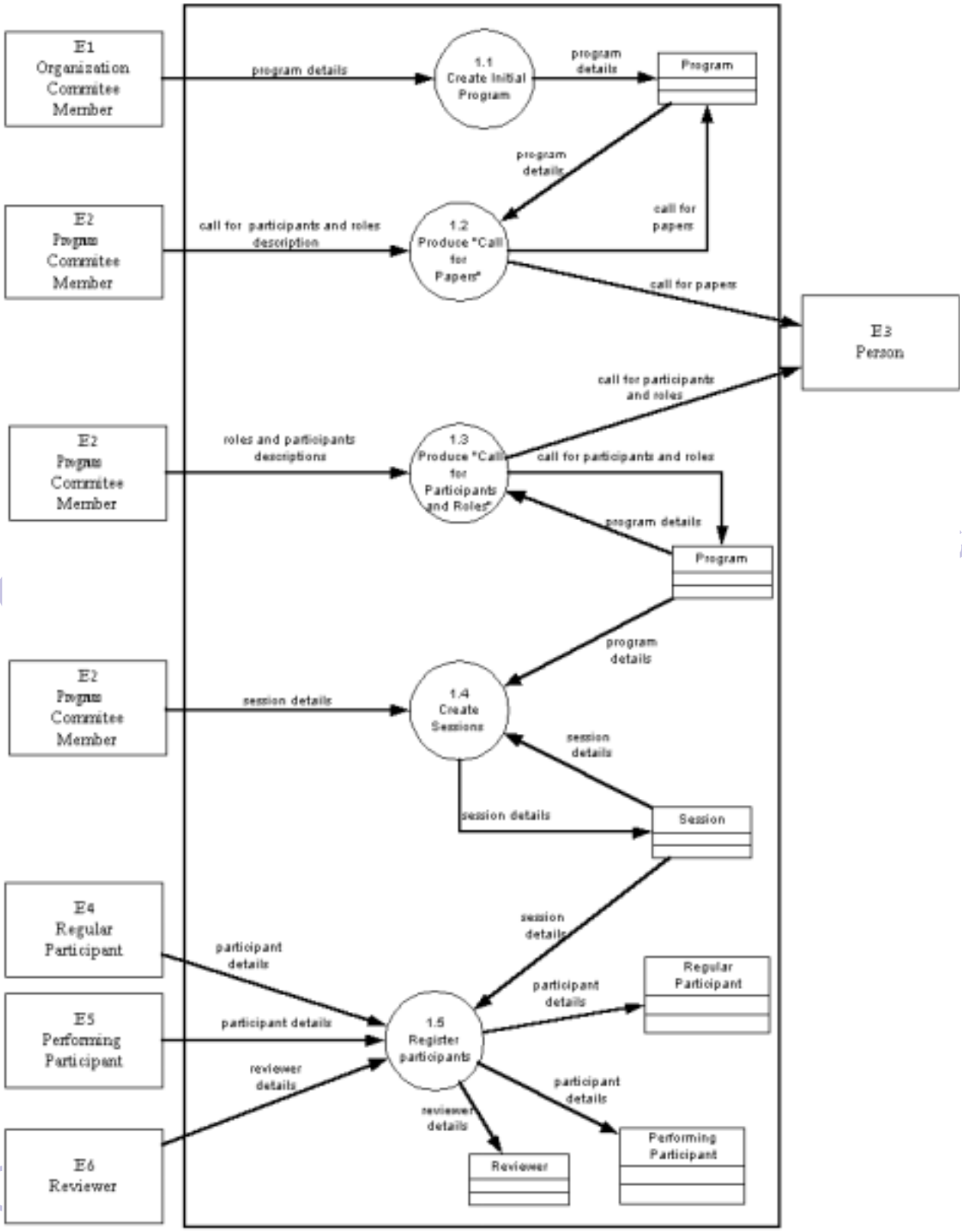


Figure 3: OO-DFD-1–Program management



We found that the order, which starts with data modeling and continues with functional analysis, provides significantly better specifications. At any rate, the analysis activities must not be carried out in sequence, as could perhaps be understood from the above discussion. Rather, they should be performed iteratively. For example, the analyst can start by creating a partial class diagram, continue by providing a few OO-DFDs which relate to those classes, return to the first activity (updating and extending the class diagram), and so on – until the analyst feels that the two products are complete, consistent and satisfy the user's requirements.

The Design Phase

Due to space limit, we describe the design activities without showing examples.

Defining Basic Methods

Basic methods of classes are defined according to the initial OO-schema. We distinguish between two types of basic methods: *elementary methods* and *relationship/integrity methods*. (More methods, for performing various users' needs, will be added at the next stage).

Elementary methods include a) construct (add) object, b) delete (drop) object, c) get (find) object, and d) set (change) attributes of object. Elementary methods actually belong to a general (basic) class, from which all the classes inherit.

Relationship/integrity methods are derived from structural relationships between classes. They are intended to perform referential integrity checks, depending on the relationship types between the classes and on cardinality constraints on those relationships. For each relationship, which is also expressed in terms of reference attributes, the involved classes include appropriate integrity methods, which will fire whenever an object of a respective class is added, deleted or changed. Generally, for each relationship between classes we can define an integrity method for operations of add, delete, connect, disconnect and reconnect. (For more details on integrity methods see Balaban & Shoval, 1999). Recall that additional, application-specific methods will be added in the stage of behavior design - see Section 2.5.

Top-Level Design of the Application Transactions

This stage is performed according to ADISSA methodology, where the application transactions are derived from DFDs (for more details see Shoval, 1988). Note that here the transactions include **classes** rather than to data-stores.

The products of this stage include transactions diagrams, which are extracted from the OO-DFDs, top-level descriptions of the transactions, and a new class - "Transactions class". This virtual class will not contain objects – only the transaction methods (as will be elaborated in Section 2.5.)

A top-level transaction description is provided in a structured language (e.g. pseudo-code or flowchart), and it refers to all components of the transaction: every data-flow from or to an external entity is translated to an "Input from..." or "Output to..." line; every data-flow from or to a class is translated to a "Read from..." or "Write to..." line; every data flow between two functions translates to a "Move from... to..." line; and every function in the transaction translates into an "Execute function.." line. The process logic of the transaction is expressed by standard structured programming constructs (e.g. if... then... else...; do-while...) The analyst and the user, who presents the application requirements, determine the process logic of each transaction. This cannot be deducted "automatically" from the transaction diagrams alone, because a given diagram can be interpreted in different ways, and it is up to the user to determine the proper interpretation.

The top-level transaction descriptions will be used in further stages of design, namely input/output design, and behavior design, to provide detailed descriptions of the application-specific class methods (which are in addition to the basic methods), as well as of the application programs.

Design of the Interface–The Menus Class

This stage is performed following the ADISSA methodology (Shoval, 1988, 1990). A menu-tree interface is derived in a semi algorithmic way from the hierarchy of OO-DFDs. Note the correspondence of the menus and menu items to the respective general functions and elementary functions in the OO-DFDs. The menu-tree is translated into a new class – the "Menus class". The instances (objects) of the Menus class are the individual menus, and the attribute values of each object are the menu items. Note that some of the selections within a given menu may call (trigger) other menu objects, signified by S (selection) while other selections may trigger transactions, signified by T. Transactions will be implemented as methods of the Transactions class (as will be detailed later). Hence, at run time, a user who interacts with the menu of the system actually works with a certain menu object. He/she may select a menu item that will cause the presentation of another menu object, or invoke a transaction, which is a method of the Transactions class.

Design of the Inputs and Outputs–The Forms and Reports Classes

This stage is also performed according to ADISSA methodology and is based on the input and output lines appearing in each of the transaction descriptions. Hence, for each "Input from.." line, an input screen/form will be designed, and for each "Output to.." line an output screen/report will be designed. Depending on the process logic of each transaction, some or all of its input or output screens may be combined. Eventually, two new classes are added to the OO-schema: "Forms class" for the inputs, and "Reports class" for the outputs. Obviously, the instances (objects) of each of these class types are the input screens and output screens/reports, respectively. Such classes are usually defined in OO programming languages and can be reused.

Design of the System Behavior

In this stage we have to convert the top-level descriptions of the transactions into detailed descriptions of the application programs and application-specific methods. A detailed description of a transaction may consist of procedures, which can be handled as follows: A certain procedure may be identified as a basic method of some class. Another procedure may be defined as a new, application-specific method, to be attached to a proper class. Remaining procedures (which are not identified as basic methods or defined as specific methods) will become a Transactions method, which is actually the "main" part of the transaction's program. Hence, every transaction is represented as a Transactions method of the Transaction class; which, once triggered by the user via proper menus selections, may call (namely, send messages to) other methods of respective classes – depending on the process logic of the transaction.

We can categorize the application transactions according to their complexity – depending on how many classes and methods they refer to. For example, a simple transaction (e.g. one that finds a certain object and displays its state, or that updates attributes of an object) may be implemented as a small procedure (namely a Transactions method) that simply sends a message to a basic method of a certain class. A complex transaction (e.g. one that finds and displays objects that belong to different classes, or that updates objects that belong to different classes or that both retrieve and update various objects) may be implemented as a more complex procedure that sends messages to basic methods and specific methods of various classes. Generally, an application may consist of many transactions, with different levels of complexity. Note that at run-time, when a user wants to "run" any transaction, he/she actually approaches the Menus class and makes proper item selections within the menu objects, until a menu item that actually fires the desired Transactions method is selected. From here, the execution of a transaction depends on the process logic of the Transactions method and the other methods it calls.

The detailed description of a transaction is expressed in two complementing forms: Pseudo-code and Message Diagram. A pseudo-

code is a structured description that details the process logic of the Transactions method as well as any other class method. The transition from a top-level description of a transaction to its detailed pseudo-code description is done as follows: Every "Input from..." and "Output to..." line in the top-level description is translated to a message calling an appropriate method of the Forms/Reports class. Every "Read from..." or "Write to..." line is translated to a message calling a basic method (e.g. "Get", "Const", "Set", and "Del") of the appropriate class. Every "Execute-Function..." line is translated to messages calling one or more basic methods of certain classes, or to new, specific methods that will be attached to proper classes, or to procedures that remain as part of the Transactions method

A Message Diagram shows the classes, methods and messages included in a transaction, in the order of their execution. A message diagram is actually a partial class diagram that shows only the classes involved in the transaction (including Data, Menus, Forms, Reports and Transactions classes), the method names (and parameters) included in that transaction, and message links from calling to called classes. Message diagrams supplement the pseudo-code descriptions of transactions.

To summarize, the products of the design phase are: a) a complete class diagram, including Data, Menus, Forms, Reports and Transactions classes, each with various attribute types and method names (and parameters), and various associations among the classes; b) detailed menu objects of the Menus class, each menu listing its items (selections); c) detailed form and report objects of the Forms and Reports classes, each detailing its titles and data fields; d) detailed transactions descriptions in pseudo-code; e) message diagrams at least for non-trivial transactions.

System Implementation with OO Programming Software

At the implementation stage, the programmers will use the above design products to create the software with any common OO programming language, such as C++ or Java. (More details on this stage are beyond the scope of this paper.)

SUMMARY

The advantages of FOOM presented in this paper are: System analysis (i.e., specification of user requirements) is performed in functional terms via OO-DFDs - a natural way for users to express their information needs, and in data terms via an initial OO-schema, or an ERD which is easily translated into an initial OO-schema. System design follows the analysis and uses its products. The OO-schema is augmented with a Menus class which is derived from the menu-tree that was designed earlier from the OO-DFDs. Inputs and Outputs classes are also derived from the input forms and the outputs of the system (earlier products of the design stage). The application programs are generated from the transaction descriptions, and are attached to the Transactions class. The Menus class enables the users to access and trigger any application transaction.

Our further research and development agenda includes: development of a set of CASE tools to support the methodology; demonstration of the methodology in use by means of several real-world case studies; and evaluation of the methodology by means of experimental comparisons with other methodologies on various dimensions, e.g. comprehension of schemas by users, quality (i.e. correctness) of designed schemas, ease of learning the methods, etc. Participants in the experiments might be students in relevant programs, or professionals in relevant development organizations.

REFERENCES

Balaban, M. and Shoval, P. (1999). Enhancing the ER Model with Integrity Methods. *Journal of Database Management,* 10 (4), 14-23.

Booch, G. (1991). *Object-Oriented Design With Applications.* Benjamin/ Cummings.

Booch, G., Rumbaugh, J. & Jacobson, I. (1999). *The Unified Modeling Language User Guide*. Addison-Wesley.

Clee, R. & Tepfenhart, W. (1997). *UML and C++ A practical guide to Object-Oriented development*. Prentice Hall.

Coad, P. & Yourdon, E. (1990). *Object-Oriented Analysis.* Prentice Hall, Englewood Cliffs, NJ.

Coad, P. & Yourdon, E. (1991). *Object-Oriented Design.* Prentice Hall, Englewood Cliffs, NJ.

DeMarco, T. (1978). *Structured Analysis and System Specification*. Yourdon Press, NY.

Dori, D. (1996). *Object-Process Methodology: the analysis phase*. Proceedings of TOOLS USA'96.

Dori, D. (2001). Object-Process Methodology applied to modeling credit card transactions. *Journal of Database Management,* 12 (1), 4-14.

Jacobson, I. (1992). *Object-Oriented Software Engineering: A Use Case Driven Approach*. ACM Press.

Larman, C. (1998). *Applying UML and Patterns- an Introduction to Object Oriented Analysis and Design.*

Maciaszek, L.A. (2001). *Requirements Analysis and System Design – Developing Information Systems with UML*. Addison-Wesley.

Martin, J. & Odell, J. (1992). *Object-Oriented Analysis & Design.* Prentice Hall, Englewood Cliffs, NJ.

Mathiasssen L., Munk-Madsen A., Axel Nielsen P. and Stage J. (2000), *Object Oriented Analysis and design*, Marko Publishing ApS, Aalborg, Denmark.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. & Lorensen, W. (1991). *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, NJ.

Shlaer, S. & Mellor, S. (1988). *Object-Oriented Analysis: Modeling the World in Data*. Yourdon Press, Englewood Cliffs, NJ.

Shlaer, S. & Mello,r S. (1992). *Object Life Cycles: Modeling the World in States*. Yourdon Press, Englewood Cliffs, NJ.

Shoval, P. (1988). ADISSA: architectural design of information systems based on structured analysis, *Information System*, 13 (2), 193-210.

Shoval, P. (1990). Functional design of a menu-tree interface within structured system development. *Int'l Journal of Man-Machine Studies*, 33, 537-556.

Shoval, P. (1991). An integrated methodology for functional analysis, process design and database design, *Information Systems*, 16 (1), 49-64.

Shoval, P & Kabeli, J. (2001). FOOM: Functional- and Object-Oriented Analysis & Design of Information Systems - An Integrated Methodology, *Journal of Database Management,* 12 (1), 15-25.

UML Rose (1998). *http://www.rational.com*