INTRODUCTION

Many paradigms for system analysis and design have been proposed over the years. Early approaches have advocated the functional approach. Common methodologies that support this approach are Structured System Analysis (SSA) and Structured System Design (SSD) (DeMarco, 1978; Yourdon & Constantine, 1979). SSA is based on the use of data flow diagrams (DFD), which define the functions to be performed by the system, the data stores within the system, the external entities, and the data flows that connect the above components. SSA and similar methodologies of that age emphasize the functional aspects of system analysis, neglecting somehow the structural aspect, namely that of modeling the data structure. Concurrently, Entity-Relationship (ER) became a popular model for designing the data structure, namely the database schema. SSD is based on the use of Structure Charts (SC), which describe the division of the system to program modules as well as the hierarchy of the different modules and their interfaces. Certain techniques have been proposed to create SCs from DFDs (see Yourdon & Constantine, 1979). The main difficulty of the above approach, that is, functional analysis followed by structured design, lies in the transition from DFDs to SCs. The translation is problematic because a DFD is a network structure, whereas a SC is a hierarchical structure. In spite of various guidelines and rules for conversion from one structure to the other, the problem has not been resolved by those methodologies (Coad & Yourdon, 1990).

Shoval (1988) developed the ADISSA methodology to solve the problem. It uses hierarchical DFDs during the analysis stage (similar to other functional analysis methodologies), but the design centers on transaction design. A transaction is a process that supports a user who performs a business function, and is triggered as a result of an event. The transactions of the system are derived from the DFDs: a transaction consists of elementary functions that are chained through data flows and of data-stores and external-entities that are connected to those functions. The process logic of each transaction is defined by means of structured programming techniques. In addition to transactions design, ADISSA methodology includes the stages of interface design, inputs/outputs design, and database design.

The interface design stage results in a menu-tree, which enables users to find and fire desired transactions. The menu-tree is derived from the hierarchy of DFDs in a semi-automatic fashion. The design of inputs and outputs of each transaction is based on data flows from external entities to functions and from functions to external entities. The data-
base schema is designed as based on the DFD data stores, resulting in a normalized relational schema. The data flows from functions to data stores and from data stores to functions serve as a basis for defining access-steps from each transaction to appropriate database relations. The products of the design stages can be easily prototyped or implemented using various programming environments.

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. In the OO approach the world is composed of objects with attributes (defining its state) and behavior (“methods”) constituting the only way by which the data included in the object can be accessed. When using the OO approach, a model of the system is usually created at the analysis stage in terms of objects - an object schema (or OO schema). An OO schema consists of different object classes with various structural relationships between them (e.g. inheritance), and each object class having its attributes and behavior (functionality). During the design stage, implementation considerations are added and the result is a model, which is supposed to enable OO programming.

Many advocates of the OO approach claim (with no substantial proof) that it is more natural to begin the analysis of a system by defining its object structure rather than by defining its functions. They support this with the claim that the real world is not composed of functions but rather of objects. They claim that the OO approach simplifies the transitions in system development stages, enhances communication between users and developers, encourages code reuse and enables the creation of robust information systems that can be easily upgraded and maintained.

While there are no doubts about the advantages of the OO approach in programming, as it supports information hiding (encapsulation), software reuse and maintenance, there are doubts with respect to the effectiveness of the approach for analyzing business-oriented information systems (as opposed to real-time systems). OO methodologies tend to neglect the functionality aspect of system analysis, and do not show clearly and systematically how to integrate the application functions (transactions) with the object schema. On the other side, users usually express their information needs in terms of functions that the system ought to perform. Another difficulty with many OO methodologies is that they involve many types of diagrams and notations with no specific guidelines which to use and in what order. The multiplicity of diagram types in the OO approach has been a major motivation for developing the Unified Modeling Language (UML) (see, for example, Clee & Tepfenhart, 1991; Larman, 1998; UML-CD, 1998; 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. But UML does not offer a development methodology; it does not guide a developer in how to do or what development process to follow.

Nowadays, the convention is a separation between the system development approaches: either one follows a “pure” object-oriented, or a “pure” process-oriented approach. Our approach is to integrate the two paradigms. In our view, processes are as fundamental as objects, and the two complement each other. We propose a system analysis and design methodology that combines the two approaches and its result is an object model consisting of object and behavior schemas.

**RESEARCH MOTIVATION AND GOALS**

Information systems development is a multi-phase process, of which the analysis and design are of primary importance. Therefore it is vital to examine which methods are appropriate to perform each of these phases. If we take a look on the exists methods for performing each of these developing phases, we can conclude:

For the **analysis phase** there exist two major approaches: the functional/data approach and the OO approach, each one having advantages and disadvantages. On the one hand, those who advocate 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 said, there is no research (that we know of) that proves that the OO approach is more effective compared to 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 show clearly how to integrate the systems functions, or transactions, with the object schema. Sometimes the impression is that all the functionality of the system is expressed by means of methods that are encapsulated within objects, not paying enough attention to functional requirements that cannot be met by simple methods. On the other hand, over many years we have learned and practiced to perform functional analysis with DFDs and there were no problems with them as a mean to expression the functionality of the system; the 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 most suitable to describe the functionality of the system. ERD or OO schemas are most suitable to model the data structure. Since OO approach (as will be discussed later) is most appropriate to perform the design
phase, 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. We suggest performing data modeling by creating an initial OO schema, or by creating first an ERD, and then translating it (quite automatically) to an OO schema. This way was found easy and simple to implement and preferable by designers (Shoval & Frumermann (1994); Shoval & Shiran, 1997).

For the design phase we must 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 process- and object-oriented approaches. Dori’s Object-Process Methodology, OPM (Dori, 1995; Dori, 1996; Dori & Goodman, 1996), indeed integrates the two approaches. It utilizes a single graphic tool, Object-Process Diagram (OPD), to all development phases. However, since OPD defines a new notation that combines DFD and OO diagrams, it includes many symbols and rules. In our opinion, such diagrams are not so easy to construct and comprehend for large-scale systems. We believe that reality has to be modeled by means of simple notations, which are easy to 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 misunderstood model of reality. On the other hand, multiplicity of models and corresponding diagramming tools, as in UML, may be too complicated either. Too many diagram types may 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. We believe that since users express their information needs in a functional and data manner - not by means of an object structure and their 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 proved 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 transaction design. This enables the transition from functional analysis DFDs to an OO model that consists of object and behavior schemas.

OUTLINE OF FOOM METHODOLOGY

We present briefly FOOM, the integrated methodology for development of business-oriented information systems, which combines the functional approach and the OO approach. The functional approach is used at the analysis stage (where its advantage lies) to define users needs, and this is done by creation of a hierarchical DFD model and an ERD (in some cases) as interim product to produce an initial OO model. The OO approach is used at the design stage (where its advantage lies) to define the structure and behavior of the system, and this is done by creation of full object and behavior schemas. The essential steps of the proposed methodology are as follows:

The Analysis Phase

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

When the performing order starts from functional analysis, the analyst constructs “traditional DFDs”, which include (beside other components) data-stores. Then he/she creates an ERD from those data-stores and maps the ERD to an initial OO schema (not including “methods”).

When the performing order starts from data modeling, one possibility is to construct an ERD and map it to an initial OO schema, and then create “OO-DFDs” that include classes (rather than data-stores). Another possibility is to construct directly an initial OO schema (not from an ERD), and then create DFDs that include those classes (instead of data-stores).

According to any of the two performing orders, ERD can be used only as an interim tool to produce an initial OO schema. The justification to this indirect approach is that ERD was found to be more easy and simple to create and preferable by designers (Shoval & Frumermann (1994); Shoval & Shiran, 1997). Furthermore, there exist automatic algorithms to map an ERD to an OO schema (e.g. Kornatzky & Shoval, 1994).

Note that the initial OO schema, created at the analysis phase, does not include methods; these will be added at the design phase. It also does not include some other class types that will be added at the design phase.

The Design Phase

Defining basic methods

Basic methods of the object classes are defined as based on the initial OO schema. We distinguish between two types of basic methods: elementary methods and relationship methods. (More methods, which enable to perform the various user needs, will be added at the next stage). Elementary methods include a) construct object, b) retrieve object, c) change attributes of object, and d) delete object. Elementary methods actually belong to a general (basic) class, which all the classes inherit. Relationship methods are derived from the structural relationships between the object classes. They are intended to perform referential integrity checks, depending on the rela-
relationship types between the objects and on cardinality constraints on those relationships. For each relationship, the classes that are involved include an appropriate relationship method, which perform proper checks when respective objects are added, deleted or changed.

**Top-level design of application transactions**

This stage is performed according to ADISSA methodology, where the application transactions are derived from DFDs (see details in Shoval, 1988). But here, the transactions refer to the respective object-classes rather than to datastores. The products of this stage are structured descriptions of the transactions. These descriptions will be used in the stage of input/output design, and in the stage of system behavior design – providing the behavior schema (in addition to the above basic methods).

**Interface design – the Menus class**

This stage is performed according to ADISSA methodology. A menu-tree interface is derived semi automatically from the hierarchy of DFDs (as described in Shoval, 1988). This menu-tree is then translated to a class – the “menus class”. The instances (objects) of the class are the individual menus and the attributes of each object are the various selection options. Some of the selections may call (trigger) other menu objects, while other selections may trigger application transactions, which may be implemented as application programs or as class methods. 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 an attribute that will cause the presentation of another menu object, or invoke an application program or a method of some object class - depending on how the transaction is implemented (as will be detailed later on).

**Design of the inputs and outputs**

This stage is also performed according to ADISSA methodology, based on the input and output lines appearing in each of the transaction descriptions. As a result, two new classes are obtained: “Forms class” for the inputs, and “Reports class” for the outputs. The instances (objects) of each of these classes are the input screens or reports, respectively. Such classes are usually defined in OO programming languages and can be reused.

**Design of system behavior**

In this stage we have to convert the top-level descriptions of the transactions (created earlier; see Section 3.2.2) into application programs or methods, attach them to proper classes, and specify the message passing. The user will trigger a transaction by selecting a proper attribute in a respective menu object. The designer has to advice the programmer how to implement each transaction: either to refer it to one or more methods, which belong to one or more classes or to refer it to an application program of a special class (called “transaction” class).

The top-level description of a transaction consists of one or more Input and Output lines, one or more Read and Write lines, and one or more Execute Function lines (namely functions calls). The mapping from the top-level descriptions of the transactions to programs consisting of method calls is made in two sub-stages.

In the first one, the top-level description of the transaction is translated into a more detailed description, which refers to respective class and method names. More specifically: an Input/Output line in the top-level description is translated to a call (message) to an appropriate method of a Form/Report object; a Read/Write line is translated to a message to a basic method such as “Display”, “Construct”, “Get” or “Set” of the appropriate class. An Execute-Function line may be translated to messages to one or more existing basic methods of appropriate classes, or to new methods or procedures (e.g. algorithms) that will be attached to proper classes. At the end we get a detailed description of the transactions, including their process-logic.

In the second sub-stage, each detailed description is translated to pseudo-code, which is a more precise and structured description. It lists method names and their parameters, and the program flow (process logic) of the transaction, namely the order and conditions for sending messages to methods. For example, display of a form or a report is expressed as a call to “Display” method of the appropriate Forms or Reports class. At the end we get a pseudo-code descriptions of the transactions, which is very close to and can be easily translated to an OO programming language.

We can categorize the transactions of the system according to their level of complexity – depending on their components (the methods they consist of). For example: a simple transaction (e.g. one that finds a certain object and displays its attributes, or that updates attributes of an object) may be translated to a simple method that is already defined as a “basic method” of the respective 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 retrieves and updates various objects) may be translated to several methods, each being attached to a different object class. Here the design issue is to define the methods (if not already defined among the existing basic methods), decide to which class to attach each method, and make sure that there are proper messages from one method to the other. One method (that it is part of this transaction) may be triggered by the user from the respective object menu, and in turn may trigger (namely send message) to a subsequent method (in the same or another class) – depending on the process logic of the transaction. A more complex transaction, which involves many classes and complex process logic, may be translated to a separate program (or transaction method) that will be attached to a special
“Transactions class”. This class will not contain objects - only transaction methods.

The end products of the design phase are the complete OO schema, the menus, inputs and outputs classes, and the behavior schema. The complete OO schema is a diagram that consists of the classes and their relationships, attributes and method (actually the method interfaces, namely their names and parameters). The Menus, Forms and Reports classes include the respective objects of menus, input forms and outputs/reports. It should be noted that the behavior schema is not another diagram; it consists of detailed transaction descriptions and their pseudo-code.

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.

EXAMPLE: THE RADIO MUSIC PROGRAMS SYSTEM

To demonstrate the application of our methodology through the analysis and design stages, we use a small example: Radio Music Programs. It calls for the development of an information system for a radio station that specializes in broadcasting music programs. The example is a small enough to be worked out in some detail, yet it contains elements that are essential to demonstrate the methodology.

The Radio Music Programs System - Description of Requirements

The radio station has a store of musical pieces, which is updated when cassettes and compact discs arrive from distributors. The data on all musical pieces is stored in database. Each musical piece gets a unique ID number. In addition it has a title, owner of the creator rights, composer, performer, and music type (e.g. pop, classic). The system enables to produce various catalogs of the music pieces inventory (i.e. by music types, composers, performers, etc). The catalogs are distributed to the music manager, program editors and distributors.

Some of the music programs are intended to broadcast musical pieces according to listeners’ requests. An interested listener sends a postcard or an email message that contains the required program name, a musical piece name, dates interval for broadcasting the requested piece, and some narrative describing his request. Each request is stored and waits to be treated by the editor of the required program.

An essential task of the system is to aid program editors preparing the music programs. The music manager of the radio plans the music programs of the season. Each program has a title, duration (in hours/minutes), day(s) of the week and hours that it will be on the fly, music types to be scheduled, editor-in-charge, technician, and broadcast date. The planned music programs for the season will be distributed to the music manager, the program editors, the technicians and the listeners.

A program editor schedules the musical pieces for each of the programs he/she is in charge of. He/she inputs the program identification to be scheduled, and the system retrieves and presents information that helps the editor on scheduling. The system enables the editor to review the database of existing musical pieces according to certain criteria (e.g. music type), see previously heard pieces in past programs, and review listeners’ requests which are relevant to this program. Based on all that information, the editor selects the pieces to be heard. For each listener whose request is answered, a printed message about the broadcasting time is sent.

The radio station pays royalties to the owners of the creator rights in exchange for each hearing. Therefore, the technician in charge of a program marks, at the actual broadcasting time, the pieces actually heard. Different reports of the heard musical pieces, according to specified parameters (e.g., time intervals, programs, creator rights owners, composers, performers, editors, etc) are produces and distributed to the music manager and to the musicians association (which represents the owners of the creator rights). The financial accounts with the owners of the creator rights are made by the financial system of the radio. Therefore, at the end of each month, the system provides a summary report of the heard musical pieces through the month. It summarizes the numbers of hearings for each musical piece, sorted by the owners of the creator rights, programs and musical pieces. The report is sent to the musicians association and a corresponding file is sent to the finance department of the radio station, which take care of the payments.

The Analysis Phase

In this example we demonstrate a performing order that starts with a data model- an initial OO-schema, followed by a functional model. Figure 1 presents the initial class model, which is the result of the data modeling of the user requirements. Figure 2, 3 and 4 show the top-level DFD (DFD-0) and two sub-DFDs (DFD-4 and DFD-5) - result of functional analysis.

Data modeling

Figure 1 (without the method names listed at the bottom part of each class diagram) presents the initial OO schema - result of data analysis of the above description. It consists of four classes: “Music Program”, “Scheduled Program”, “Musical piece” and “Listener’s Request”. “Scheduled Program” class consists of actual music program objects, scheduled or unscheduled yet. It inherits the “Music Program” class, whose objects include general details on each program. “Musical piece” class contains all data on the musical pieces stored in the radio station. “Listener’s Request” class stores the requests arriving from listeners. (The “Employee” class, which contains data about program editors and technicians, is not shown here, assuming it is part of a separate/existing information system).
Functional analysis

Figure 2, 3 and 4 are the "OO-DFDs" of the system. As can be noted, they consist of classes, instead of data-stores in "traditional DFDs".

Figure 2 is the top-level DFD of the system. It consists of five functions, seven external/user entities (E1-E7) that are sources of inputs or target of outputs of the system, and the above four classes. Three functions (1, 2 and 3) are "elementary" while two (4 and 5) are "general" functions, each of which is decomposed into sub-functions and detailed in a separate DFD. Function 1, "Input Musical Pieces Details", enables to input data about musical pieces arriving (in cassettes and CDs) from distributors, and store them as musical pieces objects. Function 2, "Produce Musical Pieces Catalogs", enables producing various catalogs, according to reporting parameters entered by the music manager. The catalogs are distributed to the music distributors, program editors and the music manager. Function 3, "Input/Update Listener's Request", handles requests arrived from listeners. Before saving a request, the system checks if the required music program exists and if the required musical piece is in the radio store. Function 4, "Schedule Music Program" and Function 5, "Update/Report Broadcast Pieces" are a general function 4. It consists of eight elementary functions. Function 4.1 inputs and updates details of the music programs for the season. It reads from the external class "Employee", details on editors and technicians, and writes to "Scheduled Program" and "Music Program" classes the details that are entered by the music manager. Function 4.2 displays or prints the details of music programs to the corresponding entities. (Note that listeners may obtain the information perhaps by the newspapers or the Internet.) Functions 4.3-4.7 belong to one major transaction (because they are linked one to the other), which treats the scheduling of musical pieces. Once the program editor triggers the transaction, and inputs the program’s name and broadcast date and time the system reads details from "Music Program" and "Scheduled Program" classes and displays the relevant information to the editor. Then the system reads from "Musical piece" class and displays information about appropriate pieces for the program, and also reads from "Listener's Request" class and displays appropriate requests. Function 4.7 enables the editor to mark the pieces he/she decided to schedule and the respective listener...
requests. (Note that it is not necessary to complete the program scheduling at once; the program editor may fire the same transaction any time, till the task is complete). Function 4.8, which belongs to a separate transaction, enables producing reports about any scheduled program for the editors and technicians (to be used by them during program broadcasting).

Figure 4 is a DFD that describes general function 5. It consists of four elementary functions. Functions 5.1 and 5.2 belong to one transaction that records the musical pieces actually heard. The details about these pieces and the scheduled programs are read from “Musical Piece” and “Scheduled Program” classes, respectively. Function 5.3 produces reports for the music manager and the musicians association about the musical pieces that actually broadcast - according to parameters that are entered by the music manager. It retrieves the relevant data from the “Music Piece” and “Scheduled Program” classes. Function 5.4 produces periodic reports about the musical pieces that were heard each month. It is triggered at the end of every month. It reads from “Musical Piece” class details about broadcast pieces, composers and performers, reads from “Scheduled Program” details about musical pieces that were heard this month, and reports to the musician association and the financial system (that will use it for paying royalties).

The Design Phase

Figures 6-12, as well as the methods part of Figure 1, are the results of the design phase.

Design of basic methods

Figure 1 shows the class model after adding the elementary methods (For example: Construct, Delete, Display, Set/Get) and the relationship methods (For example: ScheduledTo, RequestedBy). The “elementary” methods actually belong to a general class that all the classes inherit from, but for demonstration purpose we show them within the specific classes. The relationship methods are added to the proper classes depending on the relationship type each class is involved. These methods are overloaded according to the passed parameter: “Cons” for constructing (inserting) an object, “Del” for deleting, “Disp” for displaying, “Set” for setting a value and “Get” for getting a value.

Top-level design of the application transactions

The application transactions are derives from the DFDs. In DFD-0 (Figure 2) there are three transactions, each consisting of one elementary function, and the related classes, external-entities and data-flows. In DFD-4 (Figure 3) there are four more transactions: one consists of function 4.1; one consists of function 4.2; one consists of functions 4.3-4.7; and
one - of function 4.8. In DFD-5 (figure 4) there are three transactions: one consists of functions 5.1 and 5.2; one consists of function 5.3; and one - of function 5.4.

Figure 5a shows the transaction that consists of function 1 and Figure 5b shows the transaction that consists of functions 4.3-4.7 (taken from DFD-4). Figures 6a and 6b demonstrate the top-level descriptions of these transactions, respectively. (The line numbers are not part of the descriptions; we will refer to them later on). Note the “Input/Output” lines from/to external-entities, the “Read/Write” lines from/to object classes, and the “Execute” lines for the elementary functions. The process logic of a transaction is expressed by means of “standard” programming patterns, namely sequences, conditions and iterations (loops).

Interface design - the Menus class

Figure 7 demonstrates three menus created for this (small) system, according to ADISSA methodology. The main menu contains five lines/items, three marked by “T” (for “trigger”), indicating lines that trigger transactions, and two marked by “S” (for “selection”) indicating lines that call other menus. The “Schedule a Music Program” menu consists of four “T” lines, each triggering a different transaction. (The numbers next to each line/item indicate the functions included in each of the transactions being triggered by the respective menu line/item). The “Update/Report Broadcast Pieces” menu consists of three “T” lines, each triggering a different transaction.

Figure 8 presents the corresponding Menus class and its three menu objects. Note that each line/item in a menu becomes an attribute of a respective menu object. Each attribute has two possible values: “Selected” or “Not Selected” (initially all are set as “Not Selected”). The class has three main methods: “Display”, “Set” and “Choose”. The method “Display” presents the respective menu object to the user. “Set” enables the user to set the value of the corresponding attribute to “Selected” and trigger the method “Choose” with the selected attribute. “Choose” gets the user’s selection (attribute) as a parameter and either sends a message to another menu object - if a “S” line is selected, or sends a message to a method of a class - if a “T” line is selected, meaning that it fires a transaction.

Input/output design

In this stage the forms and the reports of each transaction are designed, based on the data items that flow on the data flows from external-entities to functions or from functions to external-entities. Then we add two object classes “Forms” class and “Reports” class. Figure 9 presents these classes. Note that each input screen and output report is an instance (object)

![Figure 5a. Transaction 1](image1)

![Figure 5b. Transaction 4.3-4.7](image2)
of one of these classes, respectively. The “Forms” class has three methods: “Display”, “Input_Data” and “Help”. The “Display” methods gets a form number as parameter and presents to the user the appropriate form. “Input_Data” enables the user to set values on the corresponding fields and saves them in a buffer. The “Help” method displays help on how to fill out the form.

Figure 10 presents an object of the Inputs class, the form “Musical Piece”, which enables the user to input data on musical pieces. It consists of three text boxes (for the musical piece ID number, name and length of play), four combo boxes (for the music type, composer, performer and owner of rights), and two command buttons (“OK” and “HELP”).

Design of system behavior

This stage involves (mainly) detailed descriptions and pseudo-code of the transactions, as based on their top-level descriptions. We demonstrate two transactions: Figures 11a and 11b provide detailed descriptions of the transactions following the first sub-stage as detailed in Section 3.3.5, and Figures 12a and 12b provide their respective pseudo-code following the second sub-stage.

Figures 11a and 12a refer to the simple transaction – Transaction 1 - which diagram was presented in Figure 5a, and top-level description – in Figure 6a. In essence, it involves methods of Menus, Forms and Musical_Piece classes.

Figures 11b and 12b refer to a more complex transaction – Transaction 4.3-4.7 – which diagram was presented in Figure 5b, and top-level description – in Figure 6b. Since it involves several classes and a complex process logic, the transaction can be attached to various classes, or translated to an application program that will be attached to the special Transactions class - a better idea for this case. The numbers within comment lines in Figures 11a and 11b refer to corresponding line numbers in the top-level description of the transaction (see Figures 6a and 6b, respectively). For example, in Figure 11b we can see that an Input/Output line translates to a Form/Report call (e.g. /*1*/, /*7*/), and a Read/Write line translates to a basic method call (e.g. /*2*/, /*3*/). The function calls that refer to functions 4.3-4.6 (lines 4, 6, 9 and 12 in Figure 6b) are not specified explicitly in this figure because they are already included in the program flow, while the function call which refers to function 4.7 (line 16 in Figure 6b, /*16*/) here is translated to a schedule algorithm.

Figures 11a and 11b refer to the simple transaction – Transaction 1 - which diagram was presented in Figure 5a, and top-level description – in Figure 6a. In essence, it involves methods of Menus, Forms and Musical_Piece classes.
Using the above design products, the programmers can code the required software, using any common OO programming language. Details are beyond the scope of this paper.

**SUMMARY**

The advantages of the integrated methodology are: (a) 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. (b) 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 DFDs. Inputs and Outputs classes are also derived from the input forms and the outputs of the system, earlier products of the design stage. Basic methods are derived from the OO schema structure, and application methods are defined based on the transaction descriptions. The Menus class enables the users to access and trigger any application transaction, whether implemented as a class method or as a separate program. (c) The end products of the design phase - a detailed OO schema and a behavior schema, can be easily implemented with any OO programming environment.

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; examination of the alternative performing orders of the analysis phase; experimentation and evaluation of the methodology. We plan to perform controlled experiments that will compare our methodology with other hybrid methodologies (e.g. OPM). Comparisons are possible 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 may be students in relevant programs, or professionals in relevant development organizations.

**REFERENCES**


Figure 12a. Pseudo-Code of Transaction 1

```
Transaction 1()
{
   Menu-0.Display(); Menu-0.Set(A1); Menu-0.Choose(A1);
   Piece:Input_Piece_Display; /* Input_Piece is a form, which enables the user to fill in details on musical piece. When the user finishes to fill the form it returns the filled piece object */
   Musical_Piece.Construct(Piece);
}
```

Figure 12b. Pseudo-Code of Transaction 4.3-4.7

```
Transaction 4.3-4.7()
{
   Menu-0.Display(); Menu-0.Set(A4); Menu-0.Choose(A4);
   Menu-1.Display(); Menu-1.Set(A4); Menu-1.Choose(A4);
   Input_Program_Display; /* Input_Program is a form, which displays the whole list of musical pieces */
   SELECT_PROGRAM is a function, which has a loop until the user selects the program to schedule from the Input_Program form. When the user finishes to select, the function returns a program object */
   Name = Input_Program.Program.name; Date = Input_Program.Program.date;
   Time = Input_Program.Program.time;
   MusicTypes = Music_Program.GetNames(SuitableMusicalTypes); /* MusicTypes is a function that gets a list of MusicTypes that suitable to the program and returns a list of musical Pieces suitable to those types and signs them as ’s’ if they were previous scheduled or as ’*’ if they were previous broadcast. It also prepares a report of those musical pieces, called ’Output_Pieces’. */
   Output_Pieces.Display(”Suitable Musical Pieces”, SuitablePieces);
   SuitableRequests = FIND_REQUESTS(Input_Program.Program, SuitablePieces);
   /* FIND_REQUESTS is a function that get two parameters, the program details (Name, Date and Time) and details about suitable musical pieces. It returns a list of listeners’ requests that are suitable. It also prepares a report of those listeners’ requests, called ’Output_Requests’ */
   Output_Requests.Display(”Suitable listeners’ requests”, SuitableRequests);
   Schedule_Programʙputer = ”Scheduled”;
   SELECT_pieces_REQUESTS(); /* PiecesRequests */
   SelectedPieces = PiecesRequests;
   SelectedPieces = PiecesRequests;
   For each selected piece
   {
      ScheduleProgram(Piece); /* an algorithm that find the time that it broadcast */
      Schedule_Program.InsertSchedulePiece(Piece);
      Request = Input_Pieces_Requests_Display(Piece);
      For each Request
      {
         Find the request in Listeners’ Requests and set the pointer on Listeners’ Request; SetStatus = ”Answered”
         Output_Liiste.Request.Construct(REQUEST);
         Selected.Requests += Request;
         End for
      }
   }
}
```

Peretz Shoval is Professor of Information Systems and head of the Department of Information Systems Engineering at Ben-Gurion University, Israel. He earned his Ph.D. in Information Systems (1981) from the University of Pittsburgh, where he specialized in expert systems for information retrieval. In 1984 he joined Ben-Gurion University, where he started the Information Systems Program at the Dept. of Industrial Engineering & Management. Prior to moving to academia he held professional and managerial positions in computer companies and in the IDF.

Prof. Shoval’s research interests include information systems analysis and design methods, data modeling and database design, and information retrieval and filtering. He has published numerous papers in journals and presented his research in various conferences. Shoval has developed various methodologies and tools for systems analysis and design, and for conceptual and logical database design.

Judith Kabeli is currently a Doctoral student at the Department of Information Systems Engineering at Ben-Gurion University, Israel. She earned her B.Sc and M.Sc. degrees in Mathematics and Computer Science (1994) from Ben-Gurion University. Prior to Ph.D. studies she held professional and managerial position at the Israel Aircraft Industries. Kabeli’s research interests including data modeling, information systems analysis and design, and intelligent web-based user interfaces. Her dissertation deals with the development and testing of FOOM methodology that is described in this paper - supervised by Prof. Shoval.