A Combined Functional and Object-Oriented Approach to Software Design

Haifeng Qian, Eduardo B. Fernandez, Jie Wu
Department of Computer Science and Engineering
Florida Atlantic University
Boca Raton, FL 33431

Abstract

Large and complex software systems contain a variety of entities (objects) and a complex control system (transformation function). The pure object-oriented design and structured design approaches concentrate on either objects or the transformation function separately. As such they may not be adequate in isolation, to deal with the design of complex systems. Therefore, it makes sense to study their combination. In this paper we propose a Combined Functional and Object-Oriented Design *approach (CFOOD) based on the extended object-oriented design method proposed by P. Jalote. The CFOOD approach makes full use of the object-oriented design and structured design techniques combining the object view and the functional view to provide a more complete view of a system. We demonstrate the use of our approach by a design example of a hospital patient monitoring system.*

Keywords: Complex systems, functional modules, object-oriented design, stepwise refinement, structured design.

1 Introduction

In the development of a software system the design phase is the bridge connecting the problem space and the solution space. It determines the major characteristics of the system and has great impact on the later phases, particularly testing and maintenance, which account for the majority of the cost of a software system over its entire life cycle. Many design techniques have been proposed, among which, structured design (SD) [17], [20] and object-oriented design (OOD) [3], [9], [15] are the most widely used.

The SD method views every system as a function that transforms the given inputs into desired outputs, the main task being designing this transformation function by functional abstraction and functional decomposition. In object-oriented design a system is decomposed into objects and associations between these objects. An *object* is an instance of an abstract data type, which encapsulates the *object data*, and provides a set of predefined *operations* to manipulate and access this data.

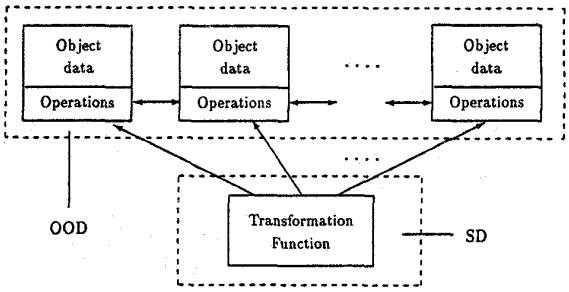

Figure 1: The model of a system, (adapted from [9])

In the SD paradigm, the basic units of a system are functions, data are secondary. In the OOD paradigm, data are primary and functions are secondary. However, systems in real life consist of complex entities with operations on them through which they interact with other entities, and of a complex transformation function that controls and operates on the entities. This transformation function requires its own method of development [16]. The entities and the transformation function are usually of equal importance. Thus a more natural way to model a real life system is to incorporate both OOD and SD techniques. Entities can be represented as objects with their operations using the OOD technique, and the transformation function can be designed by the SD approach (Figure 1).

The work of exploring integration of methods has deep significance [7], [8], [13], [18]. It is useful to explore these areas of compatibility for pragmatic rea-

0-8186-7123-8/95 $4.00

sons: contractual and/or documentation requirements may make it necessary, and also it promotes the development of software systems to a higher theoretical level, as well as facilitating the evolution of more coherent and usable approaches. This integration may also be useful as a transition phase from one methodology to another. While SD is theoretically weaker than OOD, it is still used in many places and there are many programmers familiar with its use. This makes the combination of these methods of practical value. In particular, when we are designing large and complex systems, a specific methodology may not be appropriate for all the development phases or for all the subsystems of this system.

Several studies on the integration of the OOD and SD approaches have appeared in the last few years. Ward [18] showed that there is no fundamental opposition between the two approaches, and that "real-time structured analysis/structured design can, with modest extensions to the notation and to the model-building heuristics, adequately express an object-oriented design". Jalote [9], [10] proposed an *extended object-oriented design* (EOOD) methodology, which incorporates a top-down, stepwise refinement approach in the OOD approach. Bailin [1] described a method for combining structured analysis with the object-oriented approach for requirement specifications. Similarly, Lee and Carver [12] used data flow diagrams in the analysis part of the problem. In [4], Constantine emphasized that we must get beyond the "madness of methods" and get "back to basics" by agreeing on a set of fundamental principles independent of any methodology. With sound principles being recognized as more important than specific methods, the groundwork could be laid for a coherent integration of the methods. Wasserman et al. [19] created an object-oriented structured design notation that can be used to describe not only object-oriented designs but also structured designs and even concurrent designs. Loy [13] made a comparison of the object-oriented and structured development methods and concluded that there is little evidence of the statement that the OOD is better than SD, although he might have obtained a different result if he had done his study at a later time. In [7] and [8] Henderson-Sellers and Constantine pointed out that the object-oriented and structured techniques can be seen as complementary, and could be used at different stages of the life cycle, so that the existing investments in traditional tools are preserved. There are some OOD methods based on hierarchical decomposition of objects, e.g. HOOD [6], but these methods use only objects. The OMT model [15] includes a functional model to describe the implementation of the operations of an object; however this model is used after all these operations have been defined. In summary, all these methods are used to produce only an object-oriented design or they discuss general aspects, none of them presents a specific development methodology combining the methods in the final result.

In this paper, we present a *Combined Functional and Object-Oriented Design* (CFOOD) approach based on the extended object-oriented design approach. The CFOOD approach makes full use of the SD and OOD techniques, provides stronger ability to deal with those systems whose transformation function and operations on objects are both fairly complex, and also provides adaptability to different types of problems. Of the methods discussed above, our approach is closest to EOOD. However, when applied to systems that have transformation functions and/or operations that are much more complex than the corresponding objects, our way to develop the functional refinement is more systematic and formal than the way used by EOOD.

2 Preliminaries

A *functional model* of a system shows how output values in a computation are derived from input values, without regard for the order in which the values are computed. The functional model consists of multiple *data flow diagrams* (DFD) which show the flow of values from the external inputs, through operations and internal data stores, to external outputs. Data flow diagrams do not show the logic for implementing them. In the OOD method, the design is represented by an *object model* which captures the static structure of a system by showing the objects in the system, relationships between objects, and attributes and operations that characterize each class of objects. In the SD method, the design is represented by a *structure chart* which is produced from the functional model by transform and transaction analysis. The structure of a system is composed of the *functional modules* of that system together with the interconnections between these modules.

The structured design (SD) methodology consists of four steps [10]: 1. Restate the problem as a data flow diagram. 2. Identify the input and output data elements. 3. First-level factoring, transfer the DFD to a structure chart. 4. Factoring of input, output and transform modules.

The object-oriented design (OOD) methodology consists of five basic steps [15]. From the word statement of the problem: 1. Identify the objects and their attributes. 2. Identify the operations on the objects. 3. Establish associations between objects, including generalization, aggregation and relationships. 4. Develop a dynamic model of the system. 5. Implement the operations.

This purist OOD approach is acceptable for smaller systems, but may not be suitable for complex problems [9], [10]. The extended object-oriented design (EOOD) method proposed by Jalote [9] has three phases: 1. Produce the initial design. 2. Do a functional refinement. 3. Perform object refinement.

The first phase utilizes the OOD approach. From an informal strategy, identify the objects, their attributes and operations on them. Then identify the operations that do not seem to belong to any identified object, and mark them for functional refinement in phase 2. These would typically be the operations that employ many objects or do not seem to use any objects.

In the second phase, for each of the operations marked for functional refinement, write an informal strategy, identify the objects, operations on them and operations to be further refined in the next refinement. This process is repeated until no operations for further refinement are identified. As the functional refinement finds new objects and operations on them, new operations on old objects may be uncovered. When this phase terminates, all the objects in the problem space are identified, they form the *Problem Space Object Set* (PSOS).

The third phase is to refine objects in PSOS. For each object in PSOS, write an informal strategy for all the operations on the object, and identify any new objects (and their corresponding operations) that are required to implement these operations. New operations on old objects may also be identified. The new objects should naturally be regarded as nested within the object whose refinement uncovered their existence. The process continues on the nested objects until the objects can be implemented directly.

In the EOOD method, emphasis is placed on objects. The SD technique is not brought into full play and the design of the transformation function is processed in a less systematic and coherent way than that of objects. The reason is that the initial design uses only the OOD approach and the implementation of informal strategies always consider objects and their operations first. It is fine when the method is applied to the design of an object-oriented system in which objects are primary. However, when it is used to design a function-oriented system, i.e. the transformation function is dominant over and more complex than objects, the SD technique used in the EOOD approach appears to be inadequate, full use of the SD technique is not only helpful but also required. The same thing happens to the implementation of operations on objects when the operations are very complex.

3 A Combined Functional and Object-Oriented Design Methodology

In the proposed methodology we start from two models: object model and structure chart, each resulting from the OOD and SD methods respectively. Like EOOD we employ a top-down, recursive refinement approach both for the transformation function and for the objects in the system. However, unlike the EOOD, the approach is unbiased to either objects or the transformation function, the OOD as well as the SD technique are applied as needed. The transformation function and operations are designed using mostly the SD technique. In addition, CFOOD provides a more complete view of the system by interrelating the components of the object model and structure chart. We describe below the five phases in this approach: 1. Produce an initial design by creating a combination of object model and structure chart. 2. Relate functional modules to objects. 3. Functional refinement. 4. Object refinement. 5. Produce the final design. We analyze below each one of these steps in detail.

1. *Produce an initial design*

We create an initial object-oriented design from the specifications. A reasonable way to start is to select some nouns as potential objects. We also define some basic associations. Finally, we define intuitively some appropriate operations on the defined objects (we might use a state diagram or event-trace diagram to define operations if necessary). Then we restate the strategy as a functional model (DFD) using the SD technique, identify the most abstract input and output data items, and create an structured chart from the functional model by transform and transaction analysis [10]. The outputs of this phase are an object model composed of objects and associations and the structure chart composed of functional modules.

2. *Relate functional modules to objects*

The object model and the structure chart are two different views of the same system. The modules in the structure chart may be interpreted as functions operating on objects in the object model. For each

module in the structure chart, we identify the relationship between this module and the objects on which it has actions by adding links connecting the module and the objects. If the function of the module is simply an operation already defined on the object to which it is connected, then this module needs not to be further refined. Otherwise, mark it for functional refinement in the next phase. The output of this phase is a combination of two models with a many-to-many mapping between them and marks on some functional modules that are to be further decomposed.

3. *Functional refinement*

This is an iterative process to refine the functional modules in the structure chart. Each module marked for functional refinement in the last phase, is refined by applying step 1 to it. New objects and operations on them may be uncovered during the refinement; if this happens, we add them to the object model, and identify their attributes and interfaces with other objects. Also, new operations on the objects that already exist will be identified, we attach them to their objects. When the modules in the current level are refined, apply step 2 to mark modules in the next level that still require further refinement. To keep things clear, remove the link between a module and an object if the relationship is also indicated by that between its submodule and the same object. This refinement process ends when no module is marked for further refinement, i.e. the function of each leaf module in the structure chart is either an operation on some object or can be implemented directly.

The objects and operations that are discovered during refinement at a given level are used for defining the functions that were marked for refinement at the previous level, but have no other effect on the transformation function. At the end of this phase, we have a refined structure chart and an object model consisting of problem space objects, with relationships between the modules and objects that are clearer: a leaf module is related to an object if the function of the module is an operation defined on the object.

4. *Object refinement*

In this phase we identify the objects and operations that are required to implement the operations on the objects in the PSOS. To refine an object, for all the operations defined so far on the object, apply steps 1, 2 and 3 to them, i.e. use both OOD and SD techniques to create object and functional models and a structure chart, then relate the components in the two models, and refine modules level by level. Identify any new objects and their operations required to implement these operations. The new objects and operations should be regarded as nested within the object undergoing refinement. Also identify any new operations on old objects (other objects in the PSOS). This may require the refinement of those objects reconsidered. This is repeated for each object in the PSOS. After all operations on the parent objects are identified, the refinement of nested objects starts. This phase terminates when the objects can be implemented directly. The output of this phase are two related hierarchical models where the last level can be implemented directly.

5. *Produce the final design*

A design should contain: modules and their specifications, a set of classes with their attributes and operations including their visibility, and design decisions. The problem specification can be obtained from the data flow diagram which is generated in phase 1, with the specification of data flows that occur in the DFD. Modules and their specifications are acquired from the structure chart. The specification of a module includes its interfaces, its abstract behavior, and its submodules. The design decisions should explain the choices that were available and reasons for making a particular choice.

4 A Design Example

In this section we demonstrate the use of the combined functional and object-oriented design methodology through an example (this is a variation of a classical problem [14]).

A hospital needs a patient monitoring system. Each patient is monitored by a sensor which measures pulse, temperature, blood pressure, etc. The program reads these vital values (specified for each patient) periodically and stores these values in a database. For each patient, safe ranges for vital values are specified. If a reading is outside the safe range, the alarms in the offices of the doctor and nurses who are responsible for the patient will sound. Each patient's values are constantly displayed by his bed and in the offices of the persons responsible for him. A patient has several nurses and one doctor assigned to him. Doctors and nurses have several patients assigned to them. Each doctor has his own office while several nurses share an office.

For conciseness, we will only show some major details. Readers are referred to [10] and [15] for the complete model notation.

Phase 1: Produce the initial design

Our strategy for this phase of the problem is based on [2] and [15]. We assume that nouns correspond to

potential objects. The words that are boxed represent the selected objects, and the possible operations on the objects are *italicized*. The phrases in **boldface** denote the functions of modules.

Get vital values of a patient and **check the patient's safety**. If not safe, **set alarms** for the related doctor and nurses. **Display the values on corresponding screens and store the values in a database**.

For simplicity, we assume that the frequency at which the sensor reads values from the patient is set manually. From this informal strategy, three objects: Patient, Doctor and Nurses, with their relevant attributes and operations are identified (there could be other attributes and operations that are not of interest for this application). The Doctor and Nurse classes could be generalized to a superclass such as Employee. Because this aspect is not peculiar to our methods, we leave it out but of course in a real design one should take advantage of convenient generalizations. From the statement: "A patient has several nurses and one doctor assigned to him. Doctors and nurses have several patients assigned to them", we get two relationships between these classes. We use the OOD and SD methods to get the object model and functional model, then we convert the functional model into a structure chart. The loop in the structure chart root corresponds to looping through all the patients. Figure 2 shows all three initial models. The data flow details are omitted in the functional model and the structure chart.

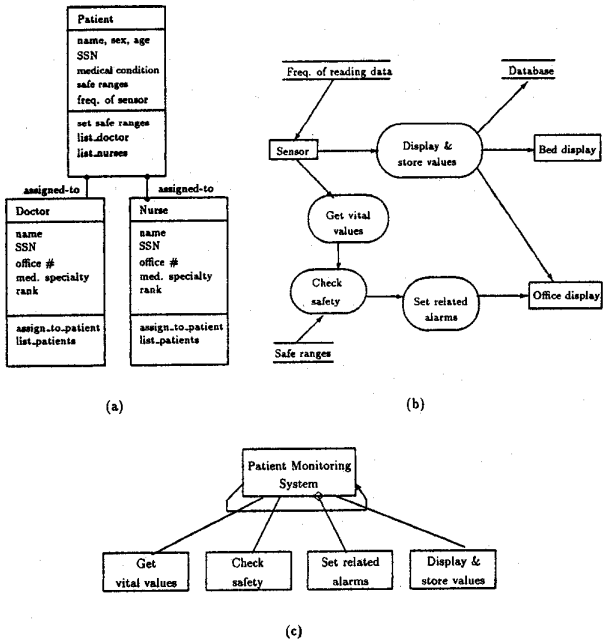

Figure 2: (a) Object model (b) Functional model (c) Structure chart

Phase 2: Relate modules to objects

Examining the four modules in the structure chart, we find that the module **Check safety** is related to the Patient object since it needs the patient's safe ranges. The modules **Set related alarms** and **Display & store values** are related to the objects Doctor and Nurse, since only in the offices of those responsible medical personnel, will the alarms sound and the patient's values be displayed. The module **Get vital values** does not belong to any existing object at this point. Clearly, all modules need to be further refined. Figure 3 shows explicitly the relationships between the modules in the structure chart and their corresponding objects.

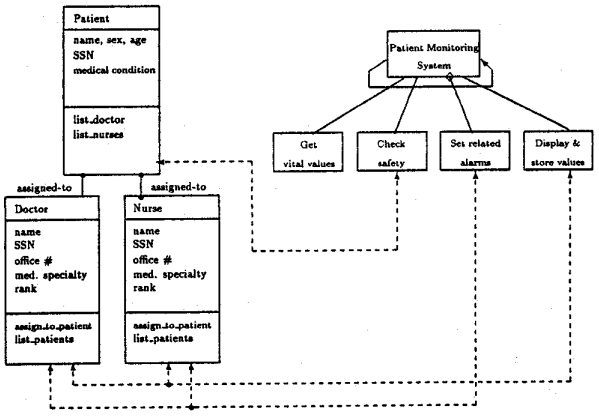

Figure 3: The combined object model and structure chart

Phase 3: Functional refinement

We perform now a second-level factoring of the modules in the structure chart. The informal strategy for the module **Get vital values** is: **Read analog values** from a sensor for a specific patient, then **Con-**

vert them to digital values. The Vital-values are collected from the patient by the sensor. The module is simple so we can do as in the EOOD method instead of recursively applying phase 1 to create two models. Here another object Vital-values is identified, with two operations *Read analog values* and *Convert A/D* on it. We add this Vital-values object to the object model, identify its attributes and one-to-one relationship with the Patient object. Two submodules are created for implementing this module (Figure 4).

The informal strategy for **Set related alarms** is: **Get the responsible person** for the patient, **Set on the alarm** in that person's office. Repeat this for the other people responsible for the patient. The decomposition of this module is indicated in Figure 4.

The decomposition of the other two modules **Check safety** and **Display & store values** can be done similarly (Figure 4).

Next we relate these modules to objects. The modules **Read analog values** and **Convert A/D** are two operations that refer to Vital-values (because these values are obtained from sensors), thus they are related to object Vital-values, and do not need further refinement. Other relationships can be adjusted similarly. Modules **set office alarm**, **display on office screen** and **store values** are marked for further refinement. Other modules are either operations on objects or can be implemented directly. For example, "**get the responsible person**" for the patient can be performed through the assignment relationship between the patient and his doctor and nurses. After the second-level factoring is done, we get the related two models of Figure 4.

In the third-level factoring, the strategy for **Set on office alarm** is: **Get** office **number, set on the alarm** in the office. Here a new object Office and the operation *set on the alarm* on it are identified. We add it to the object model, identify its attributes, and its one-to-one relationship with the object Doctor and one-to-many relationship with the object Nurse according to the problem specification. The function **Get office number** is realized through a new operation on objects Doctor and Nurse: *find office*. **Set on alarm** is an operation of object Office. The module **display on office screen** is done similarly. In the above functional refinements, we did not recursively applying phase 1 since the functions are simple. The implementation of the module **store values to database** depends on the specific structure of the database, we leave it open here. After applying phase 2, no module is marked for further refinement (except module **store values**), the output of phase 3 is shown

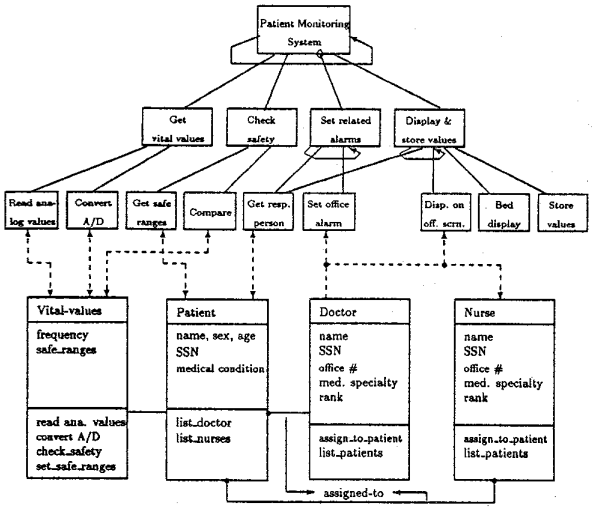

Figure 4: The combined models after 2nd-level modules have been factored

in Figure 5. At this time, the objects: Patient, Doctor, Nurse, Sensor and Office form the problem space object set.

Phase 4: Object refinement

We will not show all the refinement of objects here, we use the object Patient as an example to show the object refinement procedure. Consider the operation *set safe ranges* on the patient object. Assume the safe ranges of a patient are determined by his sex, age, medical history and current diseases contracted, medicines taken, etc. These data are processed by a function, the result of which is reviewed by the doctor, and finally the safe ranges are given by the doctor. The medical history and medicines taken are two new attributes. The medical history is composed of medical records, each of which may contain period, description of disease, treatment, responsible doctor, etc. It is referenced by the doctor and nurses and should be kept up to date by inserting new medical records. The medicines taken should have attributes such as name of medicine, the length of period the medicine was taken, dosage, side effects, etc. These two attributes are complex and have their own operations, they should be implemented as new objects nested within the object Patient.

The informal strategy for the operation of *set safe ranges* is: **read patient's data, process data, doctor review result & set the safe ranges**. Then apply steps 1, 2 and 3 to create the object model, functional model and structure chart, and refine mod-

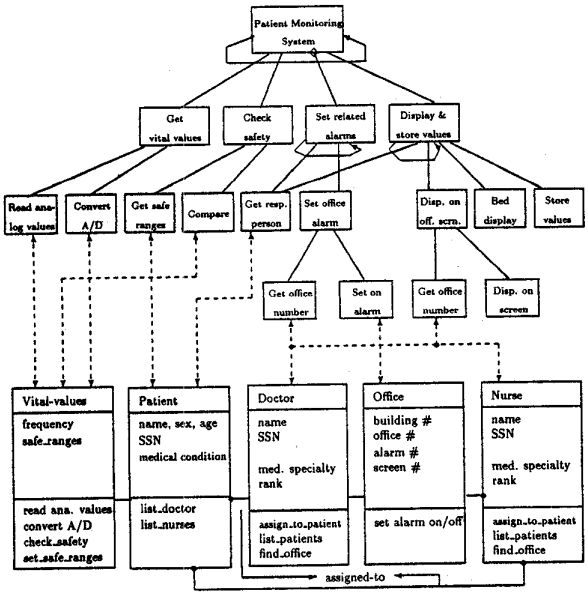

Figure 5: The result of functional refinement

ules level by level, as in functional refinement.

Repeat the process for other objects in PSOS. When this phase finishes, we get the two related models with every component in either model being implemented by 'atomic actions' and/or 'atomic objects' (Figure 6).

Phase 5: Produce the final design

The design specification can be generated as described earlier, we omit the details here.

In this example, the transformation function is not complex, thus we did not recursively apply step 1 to create the object model and structure chart while refining functional modules, the EOOD method is enough. However, some operations on objects such as *set safe ranges* are very complex, we may have to recursively apply the CFOOD method in order to implement those operations. Some attributes of objects may seem not important for this system, they are included for reusability reasons.

5 Conclusions

The pure SD and OOD methods which view a system as composed of either functions or objects may not be suitable for designing large and complex systems. The extended object-oriented design (EOOD) method uses the OOD approach with the aid of SD technique, but the emphasis is still placed on objects. The design process follows an initial design resulting

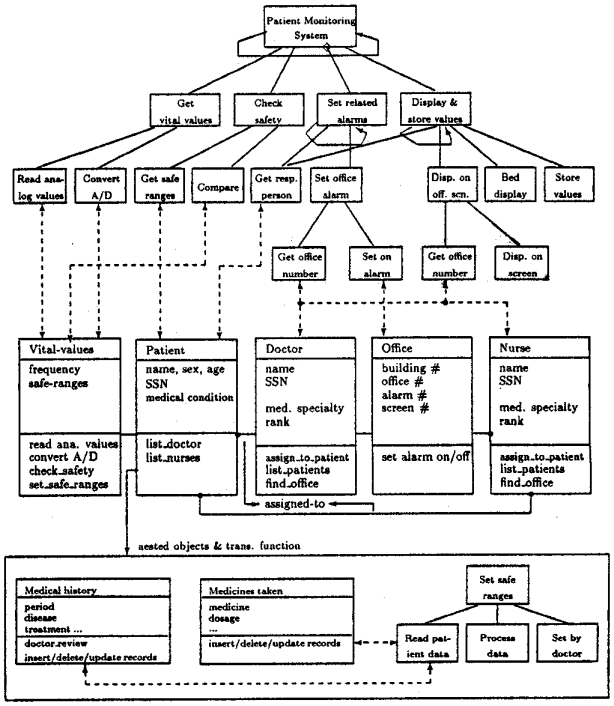

Figure 6: The output of object refinement

from the OOD approach. Its use of the SD technique may not be adequate to deal with very complex functions or operations.

We have proposed a combined functional and object-oriented design approach (CFOOD) to attack these weaknesses of the EOOD. The CFOOD approach starts from an initial design that combines an object model and a structure chart that result from the use of the OOD and SD approaches respectively. Then the two models are related together to combine the two different views of the system. These two steps are recursively applied to the complex modules and operations in the following phases of functional and object refinement. The approach provides the flexibility to use the OOD, SD, and EOOD techniques, depending on the complexity of the modules or operations undergoing refinement, and also makes it possible to design the object and the transformation function parts of a software system independently.

CFOOD makes full application of the OOD and SD techniques. The refinement of functional modules and objects applies the OOD and SD techniques recursively. This seems complicated, but it could be really necessary for a very complex module. In fact, CFOOD provides adaptability for using the OOD and SD techniques for different modules or operations. For a module or an operation that is simple, we can go through

an informal strategy as in EOOD instead of phase 1. For a module that is function dominant we may apply only the SD method. In extreme cases, CFOOD can become SD , OOD or EOOD, respectively. In addition, by incorporating the object model and structure chart, and relating the two models, the CFOOD method is more systematic and complete in designing a system than the EOOD method. A more general advantage is the possibility that the two parts of a large software system, objects and the transformation function, can be designed independently by OOD and SD experts respectively. The two submodels obtained in the initial design in CFOOD are two descriptions of the same system, are consistent with each other, and are naturally related.

Another application of the proposed approach comes from the need for controllers for an application [16]. The objects obtained through any design method, e.g. OMT, are static; to define a complete application we need a controller that implements some flow of control and makes calls to object operations according to the needs of the application. In our approach the final structure chart is in effect a controller for the application and there is no need to design any other controller. A final justification for our method is that the O-O paradigm is not the best approach for every aspect of an application. For example, some real-time procedures can be more predictable when designed procedurally than as operations on objects. In these cases, our method gives a designer the flexibility to choose the most convenient approach for different portions of the application. The value of the method should be validated through its use in real applications. We have used it in two applications: a backup system for file servers and a streaming/synchronization system (part of a multimedia user interface). While clearly more experience in its use is needed, we are convinced it is a method that has practical value.

References

[1] S. Bailin, "An object-oriented requirements specification method," *Comm. of ACM*, Vol. 32, No. 5, May 1989, pp. 608-623.

[2] G. Booch, "Object-oriented development," *IEEE Trans. on Software Eng.*, Vol. SE-12, No. 1, Feb. 1986, pp. 211-221.

[3] G. Booch, *Object-Oriented Analysis and Design (2nd Edition)*, The Benjamin/Cummings Publ. Co., 1994.

[4] L. L. Constantine, "Beyond the madness of methods: system structure modeling and convergent design," *Software Development '89: Proceedings*, Miller-Freeman Publishing Co., 1989.

[5] L. L. Constantine, "Object-oriented and structured methods toward integration," *American Programmer*, Vol. 2, No. 7/8, Aug. 1989, pp. 34-40.

[6] J. E. Cooling, *Software Design for Real-time Systems*, Chapman & Hall, London 1990.

[7] B. Henderson-Sellers, "Three methodological frameworks for object-oriented systems development," *Proceedings of TOOLS3*, Sydney, Australia, Nov. 1990, pp. 118-131.

[8] B. Henderson-Sellers and L. L. Constantine, "Object-oriented development and functional decomposition," *JOOP*, Jan. 1991, pp. 11-17.

[9] P. Jalote, "Functional refinement and nested objects for object-oriented design," *IEEE Trans. on Software Eng.*, Vol. 15, No. 3, March 1989, pp. 264-270.

[10] P. Jalote, *An Integrated Approach to Software Engineering*, Springer Verlag, New York, 1991.

[11] N. L. Kerth, "A structured approach to object-oriented design," *Addendum to the Proceedings of OOPSLA'91*, pp. 21-43.

[12] S. Lee and D. L. Carver, "Object-oriented analysis and specification: a knowledge base approach," *JOOP*, Jan. 1991, pp. 35-43.

[13] P. H. Loy, "A comparison of object-oriented and structured development methods," *Tutorial on System and Software Requirements Engineering* , IEEE Computer Society Press, CA, 1990, pp. 290-303.

[14] S. Rotenstreich and W. E. Howden, "Two-dimensional program design," *IEEE Trans. on Software Eng.*, March 1986, pp. 377-384.

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

[16] J. Rumbaugh, "Controlling code—how to implement dynamic models," *JOOP*, Vol. 6, No. 2, May. 1993, pp. 25-30.

[17] W. P. Stevens, G. J. Myers and L. L. Constantine, "Structured design," *IBM Syst. J.*, Vol. 13, No. 2, 1974.

[18] P. Ward, "How to integrate object orientation with structured analysis and design," *IEEE Software*, March 1989, pp. 74-82.

[19] A. I. Wasserman, P. A. Pircher, and R. J. Muller, "The object-oriented structured design notation for software design representation," *Computer*, Vol. 23, No. 3, March 1990, pp. 50-63.

[20] E. Yourdon and L. L. Constantine, *Structured Design*, Prentice-Hall, Englewood Cliffs, NJ, 1979.