STRUCTURED OBJECT-ORIENTED SYSTEM DECOMPOSITION

INFORMATICA 1/91

Stevan Mrdalj
Eastern Michigan University
Ypsilanti, Michigan 48197, USA
Vladan Jovanović
University of Detroit
Detroit, Michigan 48221, USA

Keywords: object-oriented design, structured system design, object-oriented models

ABSTRACT: The aim of this paper is to present a structured method for object-oriented system design with special emphasis on discovering objects within the system and creating an object-oriented model of the system. First, a technique for decomposing an object into its components according to the actions required by the operations of that object is introduced. Second, system decomposition is presented as a coherent top-down process based on an object-oriented model. Lastly, reusability and extensibility of such an approach to system design are discussed.

REZIME: U ovom radu je predstavljen strukturan metod za objektno-orijentisano projektovanje sistema sa posebnim naglaskom na način na koji se objekti otkrivaju u sistemu i kako se kreira objektno-orijentisani model sistema. Prvo je dat opis tehnike za dekomponovanje objekata na komponente u zavisnosti od akcija neophodnih za realizaciju njihovih operacija. Nakon toga je predstavljen proces dekompozicije sistema kao koherentan proces od vrha nadole koji se bazira na upotrebi objektno orijentisanog modela. Na kraju je diskutovana mogućnost ponovne upotrebe objekata i mogućnost nadgradnje objekata izmenom, odnosno dodavanjem operacija.

1. INTRODUCTION

The object-oriented paradigm has broader impact on system development than the traditional, functional or data oriented paradigms. The objective of object-oriented design is not only to create a model of the system, but to do so by reusing existing objects. Thus, object-oriented design requires more than just choosing objects and arranging them in class hierarchies. It needs a structured technique to: (1) avoid confusion in defining objects, (2) arrange objects into a system model, and (3) reuse already defined objects from the object library. Hence, here we propose a formalized approach for structured object-oriented system decomposition (SOOSD).

In considering the context of creating the object-oriented model of the system, most of the existing methods [13,1,3,8,5,15,16,2] are intuitive. They provide some informal rules for identifying the objects and their operations and can be categorized as direct decomposition methods. Also, they place little emphasis on the object's complexity and decomposition, and do not support different levels of either object or system abstractions.

On the contrary, SOOSD focuses on the discovery and arrangement of the objects of interest in the real world and creating an object-oriented model of reality. It allows us to develop an object-oriented model of a system as a leveled and incremental top-down decomposition process in which already existing objects can be reused. SOOSD is based on: (1) a specification using an object-oriented model and (2) a structured object decomposition technique.

2. OBJECT-ORIENTED MODEL

We use an object-oriented model called Abstract Object Model (AOM) as a formal specification tool to naturally and efficiently design the structure of a complex system. In AOM all things or concepts, in the designer's work environment, that are visible or otherwise tangible to the designer, are modeled as abstract objects. An abstract object encapsulates the problem space inside a set of pre-defined operations that manipulate and access that space. We refer the reader to [10] for a full description of AOM. Here we concentrate only on the following characteristics that are used in the object decomposition process:

1. AOM recognizes two kinds of objects: simple and composite. Simple objects occupy coherent space which cannot be further decomposed into meaningful objects, or one is not interested in their further decomposition. Conversely, composite objects are an *aggregation* of simple and/or other composite objects. Figure 1 illustrates a graphi-

cal representation of the object aggregation structure.

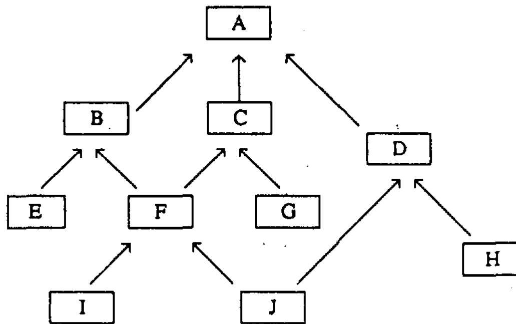

Figure 1: Object Aggregation Diagram.

2. The state of the composite object is a collection of its components' states. Thus, Figure 2 illustrates the assumption that the composite object state can be changed or accessed only by changing or accessing the state of at least one of its components.

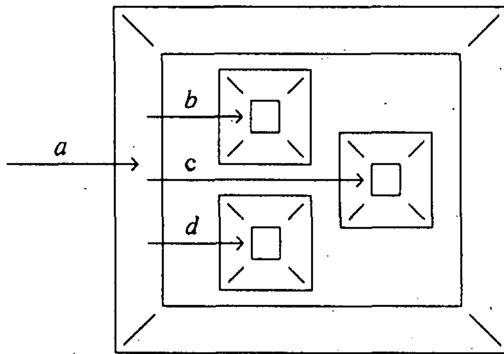

Figure 2: Access to the object space.

3. As a consequence of the previous characteristics, the operations of the composite objects are composed of the messages sent only to the components of that object. Figure 2 depicts that the response to message *a* are messages *b*, *c*, and *d*.

3. STEPWISE OBJECT DECOMPOSITION

We use the technique called Stepwise Object Decomposition (SOD) to discover "new" objects and to start the specification of such discovered lower level abstract objects [11]. SOD is based on the following two principles:

1. Actions required to construct operations of the given object determine components of that object.
2. The operations of an object are determined by the needs of the objects in which the given object is aggregated.

This is illustrated in Figure 3 where the actions required for the operations of object *X* determine that objects *Y* and *Z* become components of object *X*, while messages received by objects *Y*, *Z* and *W* determine their own operations.

The usage of these principles in the process of object decomposition is summarized into the following algorithm:

1. List all operations *O* of object *X*.
2. For each operation from *O*:
 2.1. List actions *A* required for that operation;
 2.2. For each action *a* from *A*:
  2.2.1. Associate *a* with corresponding object *Y*;
  2.2.2. Assign *Y* to the list *C* of *X*'s components;
  2.2.3. Assign *a* to *O* of *Y*.
3. For each object from *C* repeat steps 1 through 3.

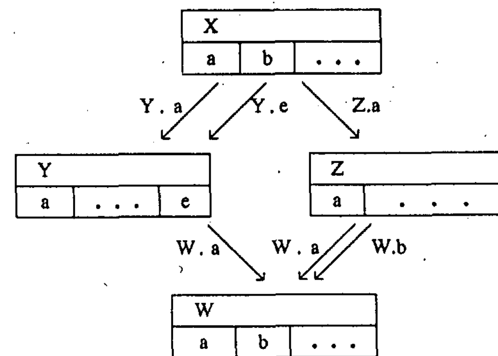

Figure 3: Message Flow Diagram.

The object that the designer chooses to decompose with this algorithm becomes the context of the decomposition process. Obviously, if the starting object is the system itself, the result will be the entire system specification.

4. SYSTEM DECOMPOSITION

AOM allows us to view any individual level of system abstraction as an abstract object, as well as to view the entire system as an abstract object on the highest level. For example, consider a system which maintains customers' requests for transportation and the assignment of taxis to customers as an abstract object called TAXI-DISPATCH.

Decomposition of the TAXI-DISPATCH object starts from the messages that it receives from its environment. Let us say that these messages define the following list of operations {AnswerCall, BuyCar, Hire}. These operations can be compared with basic functions of the TAXI-DISPATCH object that are required by its environment.

In order to discover components of the TAXI-DISPATCH object, one needs to define actions required to accomplish operations of the TAXI-DISPATCH object. Let us start from the AnswerCall operation for which the list of all required actions is {FindAvailable, ReceiveRequest, MonitorTransport}. Next, one associates all these actions with the objects which should be responsible to perform them. For example, by associating the ReceiveRequest action with the object which has to perform it, one

discovers the DISPATCHER object as one of the TAXI-DISPATCH's components. Of course, at the same time ReceiveRequest becomes the DISPATCHER's operation. After specification of all TAXI-DISPATCH's operations one may have its aggregation structure as it is displayed in Figure 4.

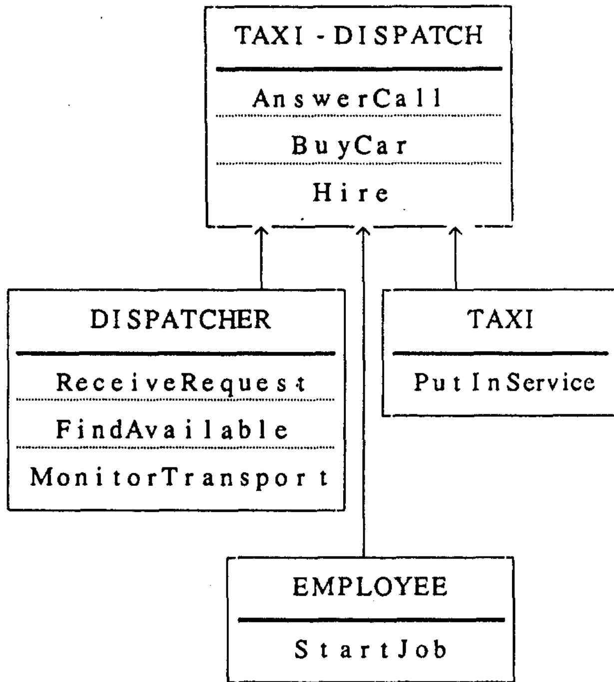

Figure 4: TAXI-DISPATCH's components.

Next, we decompose the system components into their sub-components. For example, let us decompose the DISPATCHER object with the following list of actions required for operation ReceiveRequest: {GiveName, GiveRoute, FindAvailable, Assign}. Now one has to assign again all these actions to the corresponding objects. In that way one discovers the CUSTOMER, TAXI, and DISPATCH objects and their operations. Just as before, one may keep decomposing components of the previously discovered composite objects until they are composed only of simple objects.

After specification of all TAXI-DISPATCH's components one may have the complete aggregation structure of the system as it is shown in Figure 5. Notice that the system has been simplified and its structure is partially presented so that the basic ideas of the decomposition process can be emphasized.

5. REUSABILITY AND EXTENSIBILITY

Within a framework of developing complex and long-lasting systems, we identify two objectives that SOOSD should provide support for: (1) Reusability - ability of objects to be reused, in whole or in parts, for the construction of an existing system or a new system; (2) Extensibility - changes of the objects to accommodate modifications of their requirements.

The simplest kind of reusability is the use of an object type as it already exists. For example, CUSTOMER object in Figure 5, gets its specification through the specification of DISPATCHER object. Later on, it is reused in the aggregation of the DISPATCH object. This kind of reusability is limited because object types can only be reused in the construction of the system if there is a need for exactly the same behavior as provided by that object type. Thus, it does not solve the reusability problem in general.

Very often object types need to be extended or modified to fit in a new aggregation. Object types

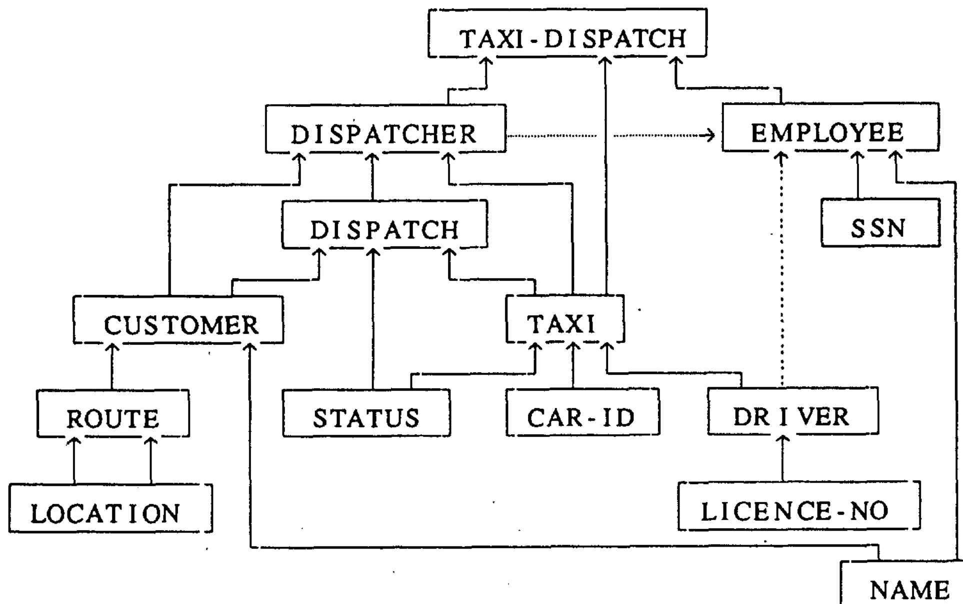

Figure 5: TAXI-DISPATCH's aggregation diagram.

can be extended and modified by means of incremental design and inheritance.

Incremental design. Since an object can be aggregated into more then one composite object, it can obtain its specification according to the needs of more than one object. For example, TAXI object from Figure 5, gets its initial specification through the specification of the TAXI-DISPATCH object. But then, by decomposing the DISPATCHER object, action FindAvailable is required from the TAXI object. That operation has to be added to the TAXI object prior to its aggregation into the DISPATCHER object. Later on during the decomposition of DISPATCH object, action AssignDrive is required from the TAXI object. That operation again has to be added to the previous specification of the TAXI object.

Inheritance. Another case of reusability is the usage of inheritance to define new object types out of existing ones by adding new components and operations. For example, EMPLOYEE object from Figure 5 gets its initial specification through the specification of the TAXI-DISPATCH object. But then one may discover the operation Drive for EMPLOYEE object which is required by TAXI object. Instead of adding that operation to the EMPLOYEE object, one defines a new object DRIVER as a subtype of the EMPLOYEE object shown in Figure 5 by the dashed line. The newly required operation Drive can now be attached to the DRIVER object as its specialized behavior. At this point we may also reconsider the DISPATCHER object and make it a subtype of the EMPLOYEE object. In this case the DISPATCHER and DRIVER objects inherit EMPLOYEE's components SSN and NAME, as well as its operation StartJob.

Conceptually lower-level objects could be the subtypes with specialized functionalities or even the special cases of the more abstract objects. An object library consisting of a set of object type hierarchies in the background may significantly reduce effort in the decomposition process.

As can be seen, an open design architecture with open-ended sets of extensions to an existing design is promoted. This is important for long-lasting systems because a system's functionality changes over time. However, the types of objects from which the system is composed will probably be more or less the same over time. The changes are most likely to occur when an object gets a new operation, an existing operation changes behavior, or a new object arises. At that time, it will only be necessary to add new operations, modify existing ones, or aggregate existing objects into new ones. Therefore, SOOSD enables maintenance of a system model as a process of reusing objects across time, and not only across applications.

6. RELATION TO OTHER WORK

There are a few works which explore the merging of structured system analysis and design techniques [4,17] and object-oriented design [12,14]. But none of them view object-oriented system design as a leveled process that starts from an entire system as an object or from any high complexity object and that decomposes them into lower level objects as it is in our case. This is the essential difference between the existing object-oriented design methods and our object-oriented top-down system decomposition. Although we use a top-down decomposition approach, the solution to system decomposition is a digraph that combines one object aggregation diagram and many object inheritance diagrams.

We also concentrate on making object types reusable through aggregation, incremental design and inheritance, three powerful mechanisms for sharing specification and promoting their reuse. That is what makes our decomposition approach distinct from most object-oriented designs which use only inheritance as a tool for reusability [9].

Finally, SOOSD is build upon the abstraction mechanisms (such as encapsulation, classification, aggregation, and generalization/specialization) from semantic data models [7] and object-oriented programming languages.

6. CONCLUSION

Structured object-oriented system decomposition represents a refinement of the structured system analysis and design using object-oriented principles. It allows us to start system decomposition from the system-object and work top-down to the complete design solution using the objects' operations to discover their components.

We have now had about three years of successful experience in using SOOSD to design and implement vastly large systems. We have found SOOSD to be extremely useful in the initial design of the systems. That is because it provides system decomposition according to currently existing object operations. But as with any software project, we have done as much re-design as design. In such cases, SOOSD continues to play an important role in reusing object types through aggregation and inheritance.

7. REFERENCES

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

[2] P. Coad and E. Yourdon, *Object-Oriented Analysis*, Prentice Hall, 1990

[3] W. Cunningham and K. Beck "A Diagram for Object-Oriented Programs", *Proc. of the OOPSLA'86*, Sep. 29 - Oct. 2, 1986, pp. 361-367.

[4] T. DeMarco *Structured Analysis and System Specification*, Prentice-Hall, 1979.

[5] R.M. Ladden "A Survey of Issues to be Considered in the Development of an Object-Oriented Development Methodology for ADA", *Software Engineering Notes*, Vol.13, No.3, 1988, pp. 24-31.

[6] P. Lyngbaek and W. Kent "A Data Modeling Methodology for the Design and Implementation of Information Systems", *Proc. of the Inter. Workshop on Object-Oriented Database Systems*, September, 1986, pp. 6-17.

[7] R. King and D. McLeod "Semantic Data Models", in S.B. Yao (ed.) *Principles of Database Design*, Vol.I, Prentice-Hall, 1985.

[8] B. Meyer "Reusability: The Case for Object-Oriented Design", *IEEE Software*, March 1987, pp. 50-64.

[9] J. Micallef "Encapsulation, Reusability, and Extensibility in Object-Oriented Programming Languages", *Journal of Object-Oriented Programming*, Vol.1, No.1, April/May 1988, pp. 12-35.

[10] S. Mrdalj "Abstract Object Model: Data Model for Object-Oriented Information System Design", *Informatica*, Vol.14, No.2, April 1990, pp. 1-11.

[11] S. Mrdalj "Stepwise Object-Oriented System Design", *Proc. of the IEEE International Conf. on Computer Systems and Software Engineering - CompEuro'90*, May 7 - 9, 1990, pp. 520-521.

[12] E. Seidewitz and W. Stark "Towards a General Object-Oriented Software Development Methodology," *SIGAda Ada Letters*, July/Aug. 1987, pp. 54-67.

[13] R.F. Sincovec and R.S. Wiener "Modular Software Construction and Object-Oriented Design Using Ada," *J. of Pascal, Ada & Modula-2*, March/April 1984, pp. 29-34.

[14] P.T. Ward, "How to Integrate Object Orientation with Structured Analysis and Design," *IEEE Software*, March 1989, pp. 74-82.

[15] A.I. Wasserman, P,A. Pircher and R.J. Muller "An Object-Oriented Structured Design Method for Code Generation", *ACM SIGSOFT*, Vol.14, No.1, January 1989, pp. 32-55.

[16] R. Wirfs-Brock and B. Wilkerson, "Object-Oriented Design: A Responsibility-Driven Approach," *Proc. of the OOPSLA'89*, October 1-6, 1989, pp. 71-75.

[17] E. Yourdon and L. Constantine *Structured Design: Fundamentals of a Discipline of Computer Program Design*, Prentice-Hall, 1975.