Accounting the OO & RDB Way,

Part II: Methods

Lincoln Stoller, Ph.D.

> "Competitive edge can only be achieved by organizations which are not only willing to adapt, but use their computers to enable previously unimaginable changes. Reusable, and extensible code and specification is not a pious wish but a life and death necessity."
>
> — Ian Graham, in "Object Oriented Methods," Addison-Wesley, 1991

I. Introduction

This is the second in a three-part series about designing applications using a combination of object oriented and relational database methods. In the first installment I defined the six main concepts in these two leading programming paradigms. I reduced object oriented (OO) design to the three basic concepts of modularization, encapsulation, and inheritance, and compared the meaning of these terms with three of the most basic concepts of relational database (RDB) programming, namely modularization, consolidation, and non-duplication.

In this article I describe a method that joins object oriented (OO) concepts of modularization and encapsulation with the relational concepts of modularization and consolidation. The six basic design concepts have certain overlaps, as illustrated in Figure I.

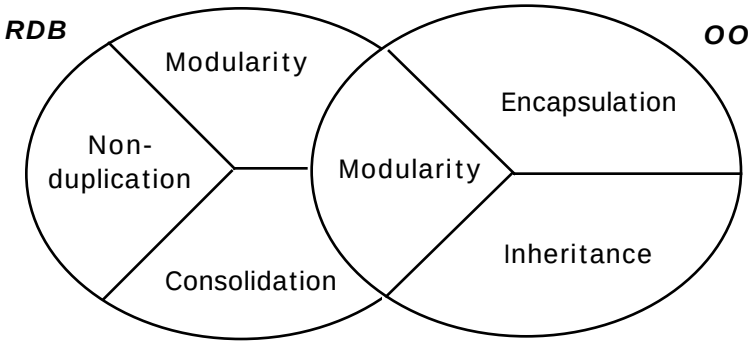

Figure I.

The concepts of non-duplication of data, encapsulation of objects, and object inheritance are largely independent. That is, they can be applied to database design independently and without interference. In contrast, the concepts of relational modularity, relational consolidation, and object modularity are interdependent, and must be applied together. While each represents positive design objectives in itself, when applied together they generate both complementary and conflicting priorities. Our objective is to find a parsimonious approach.

II. Definitions

Relational Modularity: The segregation of data storage within an application (in files or tables) such that:

A. There is limited overlap between, or little sharing of data in, different modules;
B. The structure and quantity (or "plurality") of data in one module should not depend on the structure or quantity of data in other modules.

Relational Consolidation: Making the structure more abstract by moving elements that distinguish data types from the structure and into the data themselves. This consists in enlarging the set of fields contained in closely related files to reach a single field structure common to all.

Object Oriented Modularity: The segregation of rules into classes that apply to different portions of the data. This results in the creation of semi-independent functional areas. Each module must exhibit:

A. A limited interaction between the rules in different modules;
B. Independence such that the actions of one module do not change the data in other modules.

Relational modularity expresses the benefit of segregating logically distinct data into different tables (or files). "Logically distinct" means that there are sets of similar elements whose number can be varied without altering the data in other sets. In other words, the number of items in a set can grow or shrink without violating the integrity of the rest of the data. In relational database design the objective is to create "modules" whose boundaries are distinct and whose members can vary in number.

Object oriented modularity is also concerned with independent or semi-independent data, but it takes a much more functional approach. The OO concept of modularity is formalized into rules for the formation of "object

classes". From an OO standpoint, what is important is how the objects in these classes interact both within their class (similar objects) and with objects in other classes (dissimilar objects). The OO rules have to do with the creation of classes of similar objects. The objects in these classes contain their own data and control the operations performed on these data. OO design has little concern for the segregation of classes into distinct realms, and even less concern with issues of plurality (how many objects might exist within a class).

Relational consolidation plays a complementary role to relational modularity. Where modularity tells us to recognize differences in our data and to divide our structure accordingly, consolidation tells us to recognize similarities between related modules and combine modules that exhibit minor differences. To use the example of the previous article, where modularity would tell us to distinguish the females of different generations in a kinship diagram by placing each generation in a separate file, consolidation tells us to store all generations in one file (Figure II). In this single parent/child file the generation relationship between different people would be encoded in the data rather than in the way the data were stored.

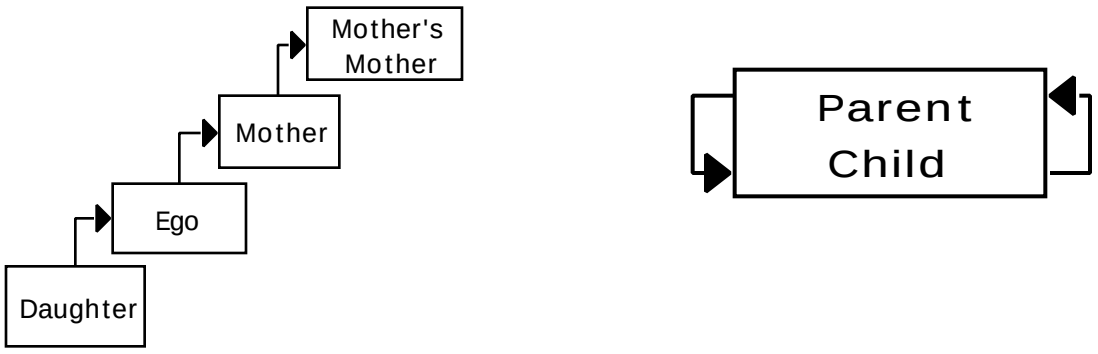

Figure II.

It may seem that these relational and object oriented concepts are irreconcilably different, or that they are so different as to make any attempt to join them a mere academic exercise. But I hope to show you that these ideas fit together easily, like jigsaw puzzle pieces, when viewed from the right perspective.

III. Structures that Work Together

> "When crafting an object-oriented system for which data dominates, focus upon the creation of a domain model first, and then treat any associated underlying database as simply a mechanism that provides transparent persistence for the objects in the domain model."

> — Grady Booch, "Object Solutions, Managing the Object-Oriented Project", Addison-Wesley, 1996, p.258

A database application can be broken into two parts: the storage of data, and the processes that operate on the data. Relational design principles apply to this first part, the stored data. Object oriented principles apply to the processes that involve data, but that do not apply to the structure of the permanently stored data. OO design is concerned with the structure of "objects," and the storage of these objects is essentially an afterthought. The idea of marrying RDB and OO principles is appealing because the two sets of ideas solve different software design problems. While one might hope for an integrated method, none exists. Engineers in the OO and the RDB worlds speak different languages, making it difficult for the two worlds to understand, let alone resolve, their differences. My approach to uniting RDB and OO concepts is based on finding harmony between the file structure and the code structure.

Returning to the notation of the previous article, I represent the division of the data structure into modules with labeled boxes. I'll refer to these modules simply as boxes. The procedural modules, referred to as brackets, are given by labels enclosed in curly brackets. Arrows represent the interaction of the brackets with the boxes. Using this notation we can catalog different approaches to modularizing data and code. In the simple examples that follow we'll place the boxes and brackets on opposite sides of the figures and connect them with the arrows. The heads on the arrows indicate the direction when the brackets (functions) draw information from boxes (data storage), when they place information in boxes, and when they do both.

Corresponding

This simplest structure involves a one-to-one correspondence between the boxes and brackets. There is little functional overlap between data and code modules.

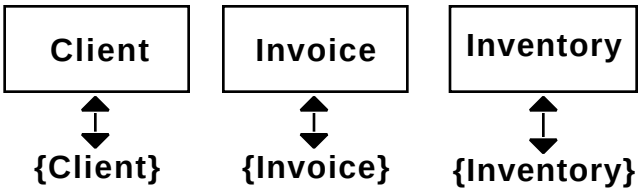

Figure III.

Each bracket corresponds to its own box. The functions represented by each bracket are distinct, and there is no sharing of code between these modules.

Supplementary

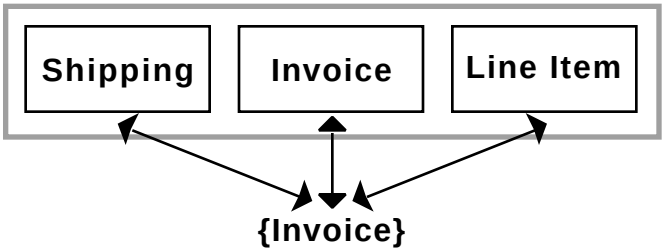

Figure IV.

This structure is a variation of the corresponding structure in which the data modules are related parts of an integrated whole. The bracket draws from and affects information in all of the boxes.

Complementary

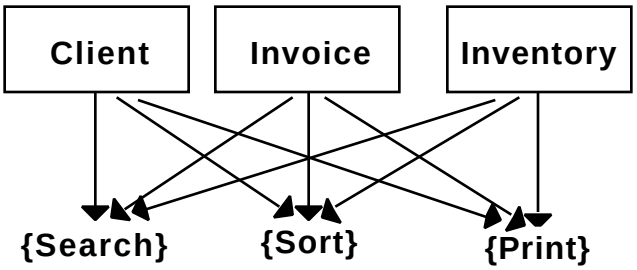

Figure V.

In this structure each bracket draws on information stored in a combination of boxes. Each bracket performs similar functions applied to different information, but none of the brackets modify the data in the boxes.

Overlapping

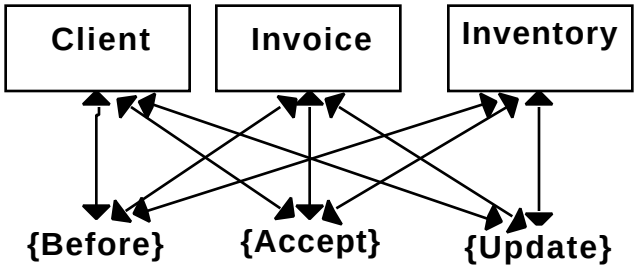

Figure VI.

Here each bracket both draws on and affects the contents of all the boxes. In this case actions within each functional area affect all the data.

IV. Evaluating the Alternatives

Corresponding and supplementary structures constitute good software design because they allow the related box and bracket units to evolve independently.

Consider the extension of the client data to include outlets. Outlets are the physical locations from which clients conduct their business. The outlets are an extension of the clients, and can be represented by a box labeled "outlet." This is connected to the client by a line, which represents the logical link between the two. An outlet bracket, representing actions that draw on and affect outlet data, can be added next to the client bracket. Since outlet functions depend upon client functions, the outlet and client brackets are connected with a line.

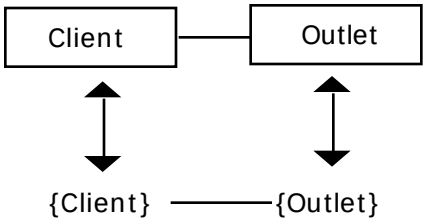

Figure VII.

As another example of extending the system we can add a product category. Assuming that product data and functions are independent from those of the client, we can represent the former by a separate box and bracket. Since they are independent, this extension can be implemented without affecting the existing application.

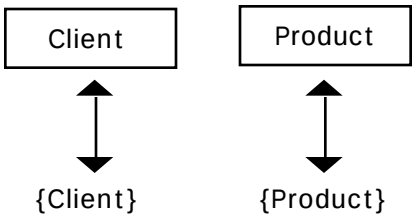

Figure VIII.

Designs based on complementary relationships also exhibit the characteristics of good software design. Here the brackets (software functions) draw upon various information modules without affecting the data in those modules. The actions of one bracket will not conflict with the actions of any other.

Functionally and structurally independent areas can be changed without effects propagating to other areas of the system. Such changes may affect the way in which the functions operate on data (for example, removing a field will make it impossible to perform a search on that field) but they will not affect the functions themselves (i.e., generic search procedures will remain unchanged).

To further illustrate how a complementary design can be modified without losing its integrity, consider the following functional modifications:

Adding Query Functionality

{Search} {Sort} —> {Search} {Sort} {Print}

Adding GUI Functionality

{Window} {Button} —> {Window} {Button} {Array}
{Popup} {Checkbox}

Expanding Display Functionality

{Print} —> {Display on-screen} {Save as file}
{Send to printer}

Figure IX.

The brackets in both the original versions on the left, and the expanded versions on the right do not modify data. As a result these extensions can be implemented without affecting the data structure. That is not to say the data structure could not be modified to take advantage of these new

functions, but that changes to the data structure can be implemented independently from the functional changes.

Contrast these benefits with the drawbacks that arise from overlapping designs. In an overlapping design the function and data structure constrain each other. In this regard the "modules" are dependent and thus display limited actual modularity. The data structure may be modular when considered by itself, but the operations performed on and with the data create functional relations between these modules.

Consider the example in Figure X where the client, invoice, and inventory information is managed by sets of operations that apply to the Before, Accept, and Update phases of data processing shown below. Here you have sets of related procedures that handle the presentation of all data (the Before module), procedures that handle the entry of all information (the Accept module), and procedures that handle the updating of all files (the Update module).

To add another data module, one dealing with contracts, for example, requires the modification of all functions modules — a change to one box requires a change to all brackets. Similarly, a change in one of the brackets will affect all of the boxes, as shown in Figure XI.

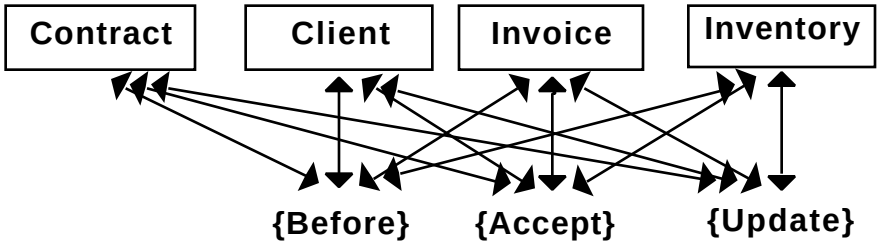

Figure X

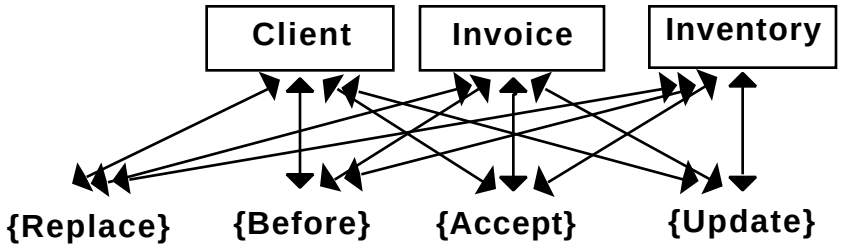

Figure XI.

In the overlapping model there is no way to create a "firewall" between modules. A coding mistake in one area can introduce errors throughout the

datafile. Any such change would require retesting all areas of the application.

Overlapping designs, which are sometimes advocated for rapid application development, lead to fragile applications that are difficult to maintain. These designs reflect indiscriminate reuse of code, and the failure to recognize the difference between good and poor code reuse. Poor reuse of code leads to applications that are not "reusable"!

Consider an architectural analogy. An architect notices that her design has many doors. She knows that doors are expensive and that structurally they are all basically the same. Following the precept of reuse the architect removes all but the front door. She then redesigns the interior spaces so that each room has adequate access to this single door. This design also has implications for the structure of the foundation and the placement of load-bearing walls.

The resulting design may make a good design for a single-purpose structure, but it is an inflexible design that is inadequate in other contexts. If there were ever a need to alter the function of this building, much of it would have to be rebuilt.

In this analogy doors correspond to an "egress" function while rooms and other interior spaces correspond to the storage of data. Reducing the design to a single door corresponds to forcing all data to be handled by a single function. This has the same diagramatic structure as using a single update module to update all data.

Figures XII and XIII show two different ways that code and data might be combined in a complex system. The first uses the overlapping model, while the second uses the corresponding model.

Overlapping

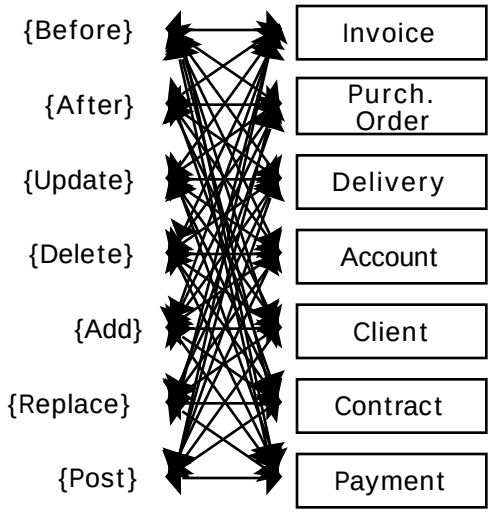

Figure XII.

Corresponding

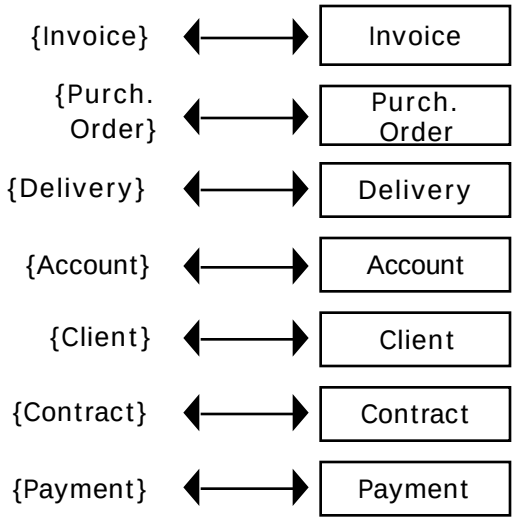

Figure XIII.

Most real systems are either complex to start with, or will become complex over time. As the complexity of a system grows, the number of lines connecting boxes and brackets in an overlapping design grows geometrically. In the system based on a correspondence between boxes and brackets the number of lines grows roughly linearly with the number of boxes and brackets. The upshot is that if you are working with a limited set of resources, both in terms of time and money, any benefits that might initially derive from the use of an overlapping design will ultimately be erased by the complexity that develops.

V. Combining the Paridigms

RDB design is primarily concerned with the structures represented by boxes in the previous figures. The identity of these structures is largely determined by issues of independence, plurality, and storage. OO design is largely concerned with the function, less concerned with data, and almost indifferent to issues of storage. The identification of object classes, whose functional aspects are represented by the brackets, is an object oriented design issue. However, we cannot simply draw a line between data and function and say that the former is the province of relational theory and that the latter is the province of object oriented design. Object classes impose requirements on the data handled by the object classes even though these implications are more limited in their scope than the relational design criteria. Object design does have implications for how we identify the boxes. This is illustrated in the Venn diagram of Figure XIV.

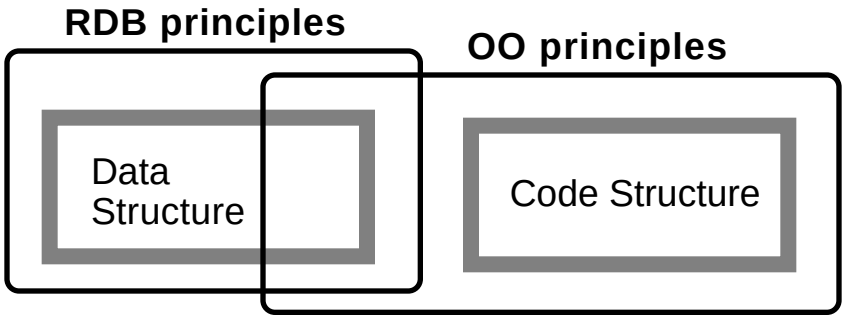

Figure XIV.

The combination of RDB and OO concepts requires three steps:

1 — apply RDB concepts to the data,
2 — apply OO concepts to the code,
3 — apply OO concepts to the data.

These steps correspond to the three elements in our diagrams: the boxes, the brackets, and the arrows.

1 — Boxes: Represent the modularity that results from applying RDB concepts.
2 — Brackets: Derive from a functional analysis of the class structure.
3 — Arrows: Represent the relation between the functions and the data that express the OO principles of class independence.

The designs that succeed are those whose structures are consistent with relational and object oriented principles. These are the approaches referred to as corresponding, supplementary and complementary. The overlapping design fails because the arrow structures create dependencies between the modules represented by boxes and those represented by

brackets. The overlapping model fails because it leads to conflicts between the functional and structureal organization of the application.

VI. Summary

I've distinguished three interdependent software design concepts that come from the relational and object oriented perspectives. These are data modularity, data consolidation, and code modularity. Data modularity and consolidation focus on data structure and the interrelations between data structures. These considerations are represented by the arrangement of boxes in my diagrams.

Code modularization has a more complex effect on design. It leads us to consider both the functional organization and how these functions affect the data. Different approaches to code modularization are represented using brackets and arrows.

Four types of structures were identified. The advantages and drawbacks of these structures were shown to correspond to the degree to which they integrate relational and object oriented design concepts.