Accounting the OO & RDB Way, Part I: Concepts

Lincoln Stoller, Ph.D.

I. Introduction

This three-part article is about designing architecture-driven accounting applications using a combination of object oriented (OO) and relational database (RDB) methods. This first part deals with concepts, the second with method, and the third with the application to accounting.

In this article I describe relational database and object oriented design, each in terms of three basic concepts. Together, these six concepts give a fairly complete description of OO and RDB design principles, and I use them in the next article to show how both can be employed when designing a 4D application.

According to Grady Booch, a leader in object oriented design, the term "architecture-driven" applies to software that is scalable, extensible, portable, and reusable. Not all designs produce this kind of application. In particular, simply using a 4th generation language, building relational applications, or assembling software using objects does not make your software architecture-driven. Because I believe that the next great step in software will come when designers and programmers understand and internalize the architecture-driven approach, the ideas presented here aim to help you design architecture-driven 4D applications.

II. OO & RDB Design in 4D

You can break a database application into two parts: the data stored on disk and the processes that operate on those data. Relational design principles apply to the stored data, and most readers of this article will be familiar with them to some extent. OO principles, on the other hand, are not widely understood because:

- OO principles are new and complex
- 4D is not an OO programming language
- Database design is "data centered" while OO design has its roots in "process centered" programming.

While 4D is not an OO programming language, there is still a large part of the OO program that can be implemented, and great advantages to doing so.

Object oriented principles apply to the processes which, although they involve data, do not apply to the structure of the permanently stored data. OO design is concerned with the structure of "objects." The storage of these objects is something of an afterthought because for the most part these objects are temporary.

The idea of marrying RDB and OO principles is appealing because the two sets of ideas help to solve different software design problems. One would think that the two methods should be compatible. However, people in the OO and the RDB worlds speak different languages, which makes it difficult for the two worlds to understand, much less resolve, their differences. One of my main purposes is to close the gap between the two by using a common terminology.

III. The Major Components of OO Design

I reduce object oriented design to three basic concepts:

1.) Modularization
2.) Encapsulation
3.) Inheritance

This breakdown may sound unfamiliar to those used to the building blocks of class, object, and method, which are a prominent part of OO programming languages. However, what I am addressing are the concepts that lie behind these building blocks. Class, object and method are programming tools that support the more fundamental ideas of modularization, encapsulation, and inheritance. This distinction is important for 4D programmers because the 4D language has limited support for these OOP tools. On the other hand, if we understand the underlying concepts, we can begin to use them in our designs.

1.) Modularization

Modularization is what you get when you take the divide-and-conquer approach to problem-solving and apply it to programming. Modular code is not just a kind of code, it is a style of software problem-solving. Before code can be made modular you must first divide the problem that you are solving into modules.

It is imperative to distinguish here between good and bad modularization. After all, you could just chop your code into blocks, segregate them into groups, and call each group a module. Random or poorly thought-out modularization as such is not an effective strategy. To mobilize an effective strategy, you must first understand the raison d'être behind modularization. Modularization is valuable because it better enables you

to find all the rules that affect subsets of data. When you can locate all the rules it becomes much easier to make sure that they are all consistent and complete. Modularizing also makes it possible to isolate all the rules that affect subsets of data. This makes it possible to divide your program into semi-independent areas. Dividing your code into interacting areas endows your whole application with greater flexibility, scalablity, and stability under changing conditions.

Imagine your code modules acting like countries that engage in international commerce. Each country has its own rules and resources, but also interacts and exchanges resources with other countries operating under different rules. Similarly, each of your code modules has its own set of data for which it makes all the rules, but it also shares data in collaboration with other modules. Changes in the rules and structure of data — an inevitable occurrence in the evolution of an application — will be limited in their effect. For modularization to be done well each module must have a large degree of logical and functional independence. It is only when this occurs that modules are a powerful problem-solving tool. This is the manner in which modules are used in object oriented programming.

In the following discussion I signify a module by putting its name in curly brackets. Each module is a set of procedures whose operations share enough commonalty to warrant grouping them together. For example, the collection of all procedures used to handle clients is written as {Clients}. This is meant to reflect the syntax of the 4D language, where the data of all client records are signified by the name of the file in square brackets [Clients].

In order to be effective a module scheme must adhere to the following principles:

a.) It must exhibit limited overlap between modules.

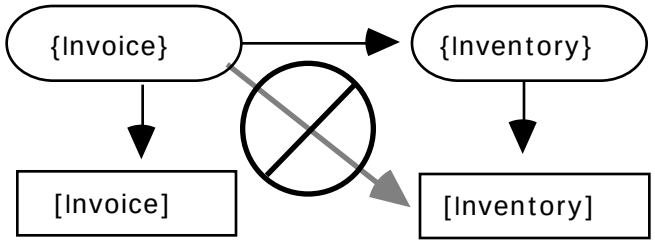
Example #1:

In the average client-invoice-inventory database, changes in client information do not affect, and are not affected by, changes in inventory or invoice data. In contrast, each invoice affects inventory though the invoice line items. Even though the invoice and inventory processes interact, we can still consider them to be processes in separate modules.

b.) The data that are changed by one module should not be changed by other modules.

Example #2:

In Example #1 the processes that update client information will not affect the client invoices. The interaction between inventory and invoices is more subtle. Here a change in a line item will affect inventory levels. In order to preserve the boundary between inventory and invoice processing, this change should be handled by coordinating one process from the invoice module, to update the line item, and one process from inventory, to update the item levels. The invoice process calls an inventory process, which in turn effects the change in inventory.



c.) Each module consists of procedures, functions, or other code objects that either do similar things, or act upon the same type of data.

Example #3:

The module {Client} might consist of functions named ClientAddRecord, ClientSalesBalance, ClientDelete, etc. All the client module functions would start with the string "Client", and as a set they would cover all the data management tasks associated with client data.

OOP languages support a construct called an “object,” which is an extension of the procedure concept. You invoke an object by making a call to it along with an indication of a particular action, called a "method," that it is to perform. A fairly good analogy to this is calling a procedure with a passed parameter that tells the procedure which of a variety of actions to perform. What distinguishes this from any other use of parameters is that a procedure's methods should constitute the complete set of operations that can be done to, or done by, that object.

If you understand that a single procedure performs all the operations that you ever need to do to a set of data, then you can start to write programs with objects and methods in 4th Dimension. But in order to gain any advantage in doing this you must start looking at a procedure as a collection of actions and not just as a block of code created for convenience, reuse, or improved readability. And in order to understand the benefit of this perspective you need to think of modularizing the structure of the code.

Example #4:

We could take the previous example of the client module and realize it by combining the actions into a single function. Each of the functions can be invoked by calling one function with an action parameter. Call the client function "Client" and pass it any of the following parameters: "AddRecord", GetSalesBalance", "DeleteRecord", etc.

The client function itself would consist of a Case Of statement that trapped the action parameter. Each block of code in the Case Of statement would execute independently. I've provided an example below with pseudo code indicating what might be involved in each operation.

```
`Function Client
`Handles actions related to client and client information.
`$1= action code
`$2= >> text array that supplies values and is set by the action
`$0= error code, < 0 for error condition, >= if operation succeeded.
C_Longint  ($0;$Err)
C_Pointer  ($3)
$Err:=0

Case of
:($2="AddRecord")
`$3=>>array containing initial values
`convert passed values from text to appropriate type
`check values completeness and consistency
`create record, assign ID, initialize fields, save record

:($2="GetSalesBalance")
`$3=>>array with two elements that give a date range
`convert string array elements to date values
```

```
`search for sales entries within the date range
`add the entries to get the total sales in the period
`convert the sales balance to text
`store the text value in the first array element

:($3="DeleteRecord")
`check that the current record is unlocked and deletable
`make changes to related records where necessary
`delete the current record

etc.
End case
$0:=$Err
```

In this discussion of code modularization I've given two ways to group the actions of the code into modules. One is by using a naming convention that makes it easy to identify related procedures, and the other is a parameter passing technique in which related actions are performed by a single procedure.

I have argued that there is a benefit to modularizing, but I have not said how to determine the appropriate modules for your application. The question of how to divide up the tasks to be handled within the module is a subject of its own. In the object oriented literature it is referred to as the problem of good class design (see Booch 1996, Chapter 4).

2.) Encapsulation

The idea of encapsulation is closely related to that of modularization. It means that the inner workings of the module, the code that performs the module's actions, are self-contained. Consider the following comparison: a module consists of parts whose operations are limited in their scope; code that is encapsulated operates without depending on outside information, and its operation is shielded from the settings, values, or other operating conditions that obtain outside of this code. An encapsulated object operates like a black box — it accepts values before it starts and it effects a change when it is finished, but the process itself remains hidden. Encapsulation itself implies some degree of modularity, but only in a limited sense. Modularity requires a set of code elements that handle a family of events relating to a certain task. Encapsulation only requires a single task.

In languages other than 4D, encapsulation is accomplished by careful scoping. This requires being able to define local variables whose values persist and are accessible by all procedures that run within the procedure that define the local variable. In 4D this might mean that you could define a local variable that retained its value for all phases of a layout procedure.

4D currently supports local variables whose scope lies within a single procedure or within a single layout phase. The scope of global variables is the range of the process in which they are defined. Neither of these kinds of scoping is sufficient to fully support encapsulation, or to support encapsulation in general cases.

Good coding practice requires that you think about situations in which your code will not operate as designed. The most common problem that one encounters in enforcing this practice is ensuring that the data you operate upon, and the variables that you operate with, cannot be changed unexpectedly by some other event. The most common symptom of code that is not encapsulated occurs when global variables used in the code are reset in the middle of their being used by some other action.

In 4th Dimension we can use naming conventions, local variables, pointers, and other tricks to build what amounts to a "firewall" around our code. Unfortunately, this firewall is easily breached — it is sometimes breached even when you are coding carefully.

Modularity must be joined with the concept of encapsulation before the notion of independently operating code makes sense. A full implementation of modularity relies on a large dose of encapsulation. This means that we must strive to encapsulate our code if we want to gain the full benefits of modular coding.

Example #5:

A block of code is encapsulated when the manner in which the code operates can only be changed from within that code. This means the code's variables are protected from outside influences; the global variables cannot be redefined and the data being acted upon cannot be modified by any other person or process until the current code's operations are complete. Consider a function call that takes the following form:

```
$ErrorCode:=SALECheckEntry     (“Cash”; SalesID; SalesDate; >>ErrorMsg)
```

In this example a function in the "Sale" module is called to check an entry. The value "Cash" is passed to the function to tell it to check the entry according to those criteria associated with "Cash." The SalesID, and SaleDate values are provided to further direct the function's operation. These passed parameters will be necessary for the function to complete its task.

When the function finishes it will have assigned a value to the $ErrorCode field. Some rule is needed to establish how to interpret error code value — which returned values constitute serious failure, complete success, or some conditional success. The value stored in the variable ErrorMsg could be set by the function to inform you of the nature of the problem that was encountered.

3.) Inheritance

Inheritance is a method supported by object oriented programming languages that makes it easier to reuse code. The concept is simple and can be roughly defined as reassembling existing pieces of code (procedures, scripts, or layouts) so that they can be used to accomplish slightly different tasks in different parts of the application.

This is a rather crude translation of an inherently object oriented concept into non-object oriented terminology. To describe inheritance more precisely would require a discussion of how OOP languages define and use objects. Since 4th Dimension is not an OOP language, this would serve no purpose. Suffice it to say that inheritance is much more than putting some actions in a procedure that can be called from different places.

One of the most vexing problems for developing a unified OOP and RDB framework is that inheritance allows for the program to procedurally define a multitude of different but related data structures. All the ways I know of to resolve this problem impose limits on how the data structure can be redefined.

4D does allow files to records in one file to be procedurally linked to records in a range of other files. That is to say that 4D is not limited to file structures that are defined by drawing relations (lines) between related fields in the structure editor. This means that there is a degree to which a fixed file structure can store "objects" whose records are related to records in other files, but that the particular related files are defined procedurally. This is a topic best reserved for another article. Let's just say that if the relationship of records in files is totally determined by the way lines are

drawn in the structure editor, then the problems that can arise from inheritance don't arise here. We can find common ground between OO and RDB concepts without difficulty.

IV. The Major Components of RDB Design

I reduce relational database design to three basic concepts:

1.) Modularization
2.) Consolidation
3.) Nonduplication

These three concepts apply to the file structure that is embodied in the fields, the files, and the relations between them. These concepts can be turned into rules governing how to correctly define the relationships between files, or what we are now calling tables. These are the familiar rules of data normalization, or at least the basic rules embodied in what is known as "Boyce-Codd Normal Form."

I'll describe each of these straightforward concepts below. They are equivalent, for the most part, to the more cryptic concepts used to describe what is required to put a structure in Boyce-Codd form. But by using these intuitive concepts we'll be able to talk about RDB and OO ideas with the same terminology. This way we can integrate all six concepts using the same set of ideas.

1.) Modularization

Modularizing the data structure is not the same as modularizing the code. The RDB concept of modularizing has been precisely defined, which is not true of the OO concept. This means placing data that are always recorded as having the same structure in one table, and putting other data that are linked, but varied in relationship, to the first in other tables.

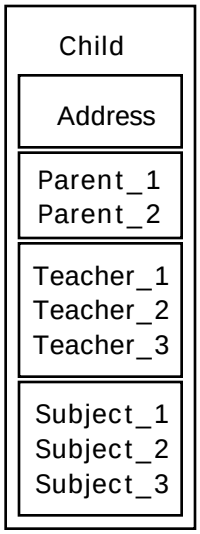
Modularization is a structural concept, and not a functional one. That is, it assumes that there is some constant, unchanging picture that represents all of the data. This is not to say that the data cannot undergo changes through "data processing." Rather, it means that a picture exists of the whole life-cycle of the data, and that you can stand back and say "the preliminary data are stored in Table A, once they have been processed they will go to Table B, and after they are archived they will be stored in Table C." Modularization is so basic to RDB design that we are almost unaware when we are doing it.

Example #6:

Consider a database to track the children in a first grade class. The elements we must include are children, parents, subjects, teachers and addresses. Using a "flat file" model we could establish one table whose fields are:

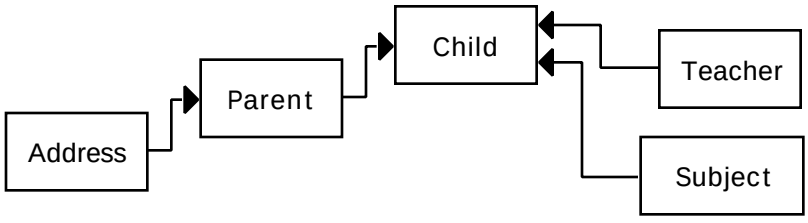
Child
Address
Parent_1, Parent_2
Teacher_1, Teacher_2, Teacher_3
Subject_1, Subject_2, Subject_3

We can draw this as:



This would be an acceptable relational design if every child had three teachers, three subjects, and two parents who lived at the same address. If this scenario is not true, then this is a poor relational design.

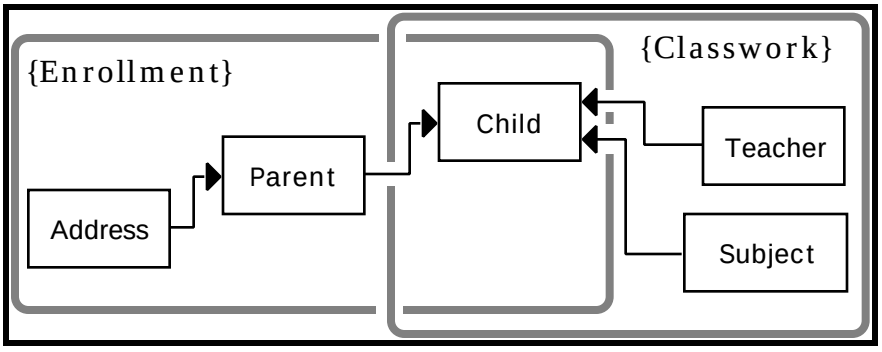
When the relationship between each of these items has more flexibility, then each of the items should be placed in a separate table. So if a child can have one or more teachers, one or more subjects, a single parent, one or more step-parents, or two parents who live at different addresses, then the following table structure better suits the problem:



I want to clarify that creating a relational table structure — what I refer to as "modularizing" the data — is not the same as creating modular program

code. The former applies to how the data are stored, while the latter applies to the actions that are performed.

In this same example, according to the actual situation, we might create an "enrollment" module for handling the application, review, acceptance, and registration process. We might also create a "classwork" module with methods for tracking assignments, problems, and performance in different subjects. We could superimpose these "process" modules on top of the data structure by drawing round-cornered boxes around those items affected by these processes:



2.) Consolidation

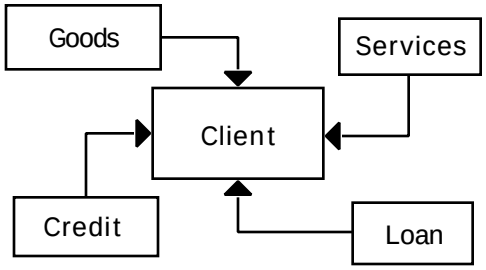
By consolidation I mean enlarging the set of fields contained in closely related files to reach a single field structure common to all. When you take this "least common denominator" approach, the data types can be distinguished using a tag that is stored in a special field. Consolidation has several advantages:

- New classes of data can be added without changing the file structure.
- Information is stored in a more compact file structure.
- Large-scale logical relationships become more evident.
- It is easy to change data from one type to another.

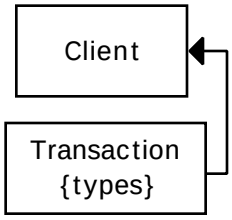
Consolidation is a process of generalization. The number of fields that need to be defined in order for the file to hold records of various related types is larger than the number of fields that would be needed for records of only one type. Fields are defined that may be used by one type of record but not another. The file structure becomes more compact at the expense of the field structure which, in the case of the file used to store several types of records, becomes less compact. I like to describe this process as "removing the content from the data structure," by which I mean making the structure more abstract.

Example #7:

Consider the example where a client can be involved in four different kinds of business transactions: purchasing goods, purchasing services, receiving credit, or getting a loan. These are four different types of transactions that represent some financial involvement between the client and your business. You could store this information in a data structure that used five different files that can be linked to the client, as shown below:



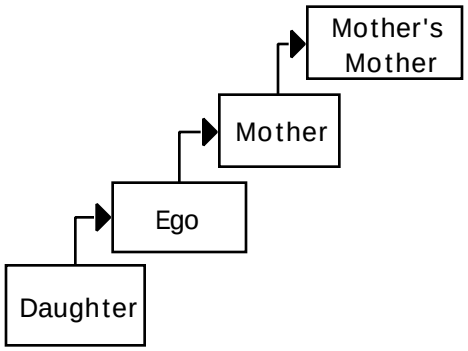
This relationship might also be expressed as having the client become involved in a transaction of some kind. If we define a general Transactions file that has enough information to represent any of the four specific transaction types, then each record of a particular type can be stored in this file, and different types can be stored in separate fields.



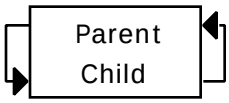
The benefit in this example is that new kinds of transactions could be added simply by adding records of a new type. The consolidated structure is more flexible.

Example #8:

Consider an anthropological database tracking kinship that starts with the reference point of a particular individual (ego) and includes members of her lineage (mother, father, mother's mother, mother's father, daughter, son, etc.). Every person fits into one of these categories, and the same kind of information is stored for each person. Each generation can be stored in a separate file, and every person could be linked to ego as shown below:



This could also be represented with a much simpler structure containing only one file. The records in this file are lineage members and each one points to lineage members of previous and subsequent generations, that is, to parents and children:

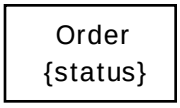


The consolidated structure is simpler.

Example #9:

In most sales order systems the order can be preceded by a quote. A quote records much the same information as an order, but it does not represent any commitment. The order then turns into an invoice once it has been executed. The invoice also contains the same information as the quote and the order, but now represents a financial obligation. These three items could be stored in separate files. As a record evolves through these phases, it is removed from one file and added to the next.

The relationship could also be stored in one file that contained a status field to indicate whether a record was a quote, order, or invoice. In this compact representation the evolution of data through a series of phases is effected simply by changing the type.



This consolidated structure enables the records to change their type without having to move to different files.

3.) Nonduplication

The proscription against storing duplicate values of the same data is the most familiar rule of database normalization. The reasoning behind it is fairly obvious as well: when you store the same value in multiple locations it takes extra work to keep the values synchronized. There is also the risk that as the database evolves someone will write a procedure that changes the value stored in one location without updating the other value or values.

This seems like a simple rule that is easy enough to adhere to. But in reality avoiding duplicate data is not so simple. The rule is frequently violated unintentionally, and occasionally violated out of necessity.

It is clear enough that if a check is deposited in the bank and we store the value of the deposit in two places, then we violate the rule. But what if we store the deposit in one place, and the current account balance in another — does that constitute duplication? The answer is yes if the balance can be calculated as a sum of deposits and withdrawals, and if it is logically identical to the current sum of these entries. In this case, storing the balance violates the rule against duplicating data.

However, the necessities of speed, network traffic, and load on the server prevent us from performing a sum over a potentially huge number of entries every time we want to know the current balance. The current balance must be maintained as a separate field. The risk in this case is that the current balance and the actual total will disagree. And since the actual total will rarely, if ever, be recalculated, there is little chance that anyone would notice the discrepancy.

This is an obvious example of "denormalizing" the data. There are also less obvious situations that introduce similar risks over the short term. Whenever we copy a value from the database and store it in memory we duplicate data. In fact, we do this every time we load a record! For the most part this is not a serious infraction since we know, as developers, that the correct value is the value stored in the datafile. When the record is next loaded the correct value will be known. However the user may not know this, and may not reload the record for a long time. In the meantime, they may have taken various actions based on the assumption that the value they were seeing was correct.

This is actually a difficult problem that leads to the different approaches of "optimistic" and "pessimistic" data management in multi-user systems. About all we can do most of the time is to make sure the user knows that

values are only accurate for an instant after they are loaded. After that the value can change due to the actions of another user.

There are also those cases where we cache information locally. This is frequently done with stable data that changes infrequently. Cached information lies somewhere between temporary and permanent storage. It can be done by creating arrays of values, sets, or named selections. The caches are created when the user starts a session, and are not updated until the user logs on again.

The risk in using caches is greater than the risk associated with relying on the values of a record stored in memory. This is because in addition to storing the wrong values, a cache might also refer to the wrong elements.

For example, consider storing all bank deposits in a cache that is created when the user logs on. These values can be modified by other users who can also add and delete deposits. As a result, the cache can become invalid in both size and content.

Another example is two caches created to store the same information that is being viewed simultaneously by one user in different windows. It's easy to imagine that the windows would display inconsistent information.

In all of these cases there are steps that you can take to alert users to changes, and to periodically refresh cached information. Before taking any of these steps the developer must consider:

- How frequently these inconsistencies appear;
- How large these inconsistencies are;
- Whether the inconsistencies have serious consequences;
- Whether the benefit of guarding against them is worth the cost.

The object here is not to detail the problems and solutions associated with duplicated values, but to note that nonduplication is a fundamental component of database design, and that it has ramifications at many levels of our applications.

Most importantly you should note that the issue of duplication is not relevant to the object oriented design issues we considered earlier. We can address nonduplication issues and OO design issues without the solutions to one problem having an effect upon the solutions to the other.

IV. Summary

I've expressed OO and RDB design in terms of six basic concepts. OO concepts focus on the program's function while the RDB principles address issues of information storage. Consequently these concepts are relatively independent.

However the OO and RDB paradigms overlap when it comes to modularization and encapsulation. These are the concepts that address the problem of breaking down the design problem into a set of smaller, semi-independent parts. These concepts have different meanings in these two paradigms. Reconciling their differences is essential if you are to design architecture-driven applications.

In subsequent articles I'll discuss how modularization and encapsulation can be simultaneously applied to both the function and structure of an application. I'll then apply these methods to the design of accounting applications.

Bibliography

Booch, G. 1996. Object Solutions, Menlo Park, CA: Addison-Wesley.

Booch, G. 1994. Object-Oriented Analysis and Design with Applications. Redwood City, CA: Benjamin/Cummings.