THE THIRD MANIFESTO:

FOUNDATION FOR

OBJECT / RELATIONAL DATABASES

by

Hugh Darwen *and* **C. J. Date**

Note: *A book with the same title by the same authors is due to be published by Addison-Wesley in mid 1998. The article that follows is basically a late draft of one chapter from that book, though it has been edited somewhat to make it more self-contained. A brief description of that book can be found at the end of this article. For more information, contact either of the authors (contact information is also given at the end of the article).*

ABSTRACT

The Third Manifesto [3] is a detailed and rigorous proposal for the future of data and database management systems. The present article consists of an informal discussion of certain of the key technical ideas underlying the *Manifesto*, including in particular the idea that domains in the relational world and object classes in the object world are the same thing.

Go back to the top Previous page Next page

INTRODUCTION

There is much current interest in the database community in the possibility of integrating objects and relations. However (and despite the fact that several vendors have already announced-in some cases, even released-"object/relational" products), there is still some confusion over the question of the right way to perform that integration. Since part of the purpose of *The Third Manifesto* [3] is to answer this very question, the idea of bringing the *Manifesto* to the attention of a wider audience than hitherto seems timely.

The *Manifesto* is meant as a foundation for the future of data and database management systems (DBMSs). Because of our twin aims in writing it of comprehensiveness and brevity, however, it is-unfortunately but probably inevitably-rather terse and not very easy to read; hence this introductory article (which might be characterized as "the view from 20,000 feet"). Our aim is to present some of the key technical ideas underlying the *Manifesto* in an informal manner, thereby paving the way for a proper understanding of the *Manifesto* itself. In particular, as already indicated, we would like to explain what we believe is the right way to integrate objects and relations. More precisely, we want to address the following question:

What concept in the relational world is the counterpart to the concept "object class" *in the object world?*

There are two equations that can be proposed as answers to this question:

1. domain = object class

2. relation = object class*

In what follows, we will argue strongly that the first of these equations is right and the second is wrong.

-----

* More correctly, *relvar* = object class. See the section "Relations *vs*. Relvars," later.

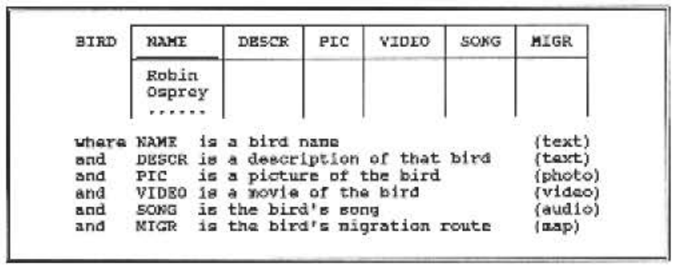
Go back to the top Previous page Next page

WHAT PROBLEM ARE WE TRYING TO SOLVE?

Databases of the future will contain much more sophisticated kinds of data than current commercial ones typically do. For example, we might imagine a biological database that includes a BIRD relation like that shown in Fig. 1. Thus, what we want to do is extend-dramatically-the range of possible kinds of data that we can keep in our databases. Of course, we want to be able to manipulate that data, too; for example, we might want to find all birds whose migration route includes Italy:

```
SELECT NAME, DESCR, VIDEO
 FROM BIRD
 WHERE INCLUDES ( MIGR, COUNTRY ( 'Italy' ) ) ;
```

Note: We use SQL here for familiarity, though in fact the *Manifesto* expressly proscribes it (see the next section).



BIRD	NAME	DESCR	PIC	VIDEO	SONG	MIGR
	Robin					
	Osprey					
	......					

```
where NAME  is a bird name                  (text)
and   DESCR is a description of that bird   (text)
and   PIC   is a picture of the bird        (photo)
and   VIDEO is a movie of the bird          (video)
and   SONG  is the bird's song              (audio)
and   MIGR  is the bird's migration route   (map)
```

Fig. 1: The BIRD relation

Thus, the question becomes: How can we support new kinds of data within the relational framework? Note that we do take it as axiomatic that we want to stay in the relational framework!-it would be unthinkable to walk away from nearly 30 years of solid relational R&D. We mustn't throw the baby out with the bathwater.

Go back to the top   Previous page   Next page

WHY THE *THIRD* MANIFESTO?

Before going any further, we should explain that "third" in our title. In fact, the *Manifesto* is the third in a series (of a kind). Its two predecessors are:

1. *The Object-Oriented Database System Manifesto* [1]

2. *The Third Generation Database System Manifesto* [7]

Like our own *Manifesto,* each of these documents offers a proposed basis for future DBMSs. However:

1. The first essentially ignores the relational model! In our opinion, this flaw is more than enough to rule it out immediately as a serious contender.

2. The second does agree that the relational model must not be ignored, but assumes that SQL (with all its faults) is an adequate realization of that model and hence an adequate foundation for the future. By contrast, we feel strongly that any attempt to move forward, if it's to stand the test of time, must *reject SQL unequivocally*. Our reasons for taking this position are many and varied, far too much so for us to spell them out in detail here; in any case, we've described them in depth in other places (see, e.g., references [2] and [4]), and readers are referred to those publications for the specifics.

A major thesis of *The Third Manifesto* is thus that we must get away from SQL and back to our relational roots. Of course, we do realize that SQL databases and applications are going to be with us for a very long time-to think otherwise would be quite unrealistic. So we do have to pay some attention to the question of what to do about today's SQL legacy, and *The Third Manifesto* does include some proposals in this regard. Further details are beyond the scope of this article, however.

Without further preamble, let's take a look at some of the key technical aspects of our proposal.

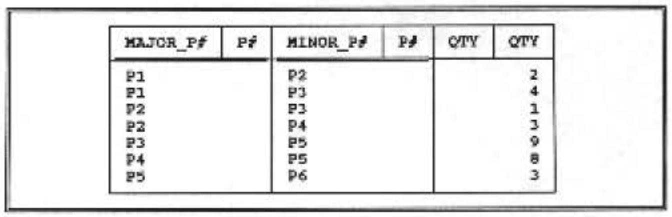
RELATIONS *vs.* RELVARS

The first thing we have to do is clear up a confusion that goes back nearly 30 years. Consider the bill-of-materials relation shown in Fig. 2. As the figure indicates, every relation has two parts, a *heading* and a *body;* the heading is a set of column-name/domain-name pairs, the body is a set of rows that conform to that heading. For the relation in Fig. 2:

- The column names are MAJOR_P#, MINOR_P#, and QTY (where P# means *part number*);
- The corresponding domain names are P#, P#, and QTY, respectively;
- Each row includes a MAJOR_P# value (from the P# domain), a MINOR_P# value (also from the P# domain), and a QTY value (from the QTY domain).

Informally, of course, we often ignore the domain-name components of the heading (as indeed we did in Fig. 1).

Go back to the top Previous page Next page



MAJOR_P#	P#	MINOR_P#	P#	QTY	QTY
P1		P2			2
P1		P3			4
P2		P3			1
P2		P4			3
P3		P5			9
P4		P5			8
P5		P6			3

Fig. 2: A bill-of-materials relation

Now, there's a very important (though perhaps unusual) way of thinking about relations, and that's as follows. Given a relation *R,* the heading of *R* denotes a certain *predicate* (or truth- valued function), and each row in the body of *R* denotes a certain *true proposition,* obtained from that predicate by substituting certain domain values for that predicate's parameters ("instantiating the predicate"). In the case of the bill-of-materials example, the predicate is

part MAJOR_P# contains QTY of part MINOR_P#

(the three parameters are MAJOR_P#, QTY, and MINOR_P#, corresponding of course to the three columns of the relation), and the true propositions are

part P1 contains 2 of part P2

(obtained by substituting the domain values P1, 2, and P2);

part P1 contains 4 of part P3

(obtained by substituting the domain values P1, 4, and P3); and so on. In a nutshell:

- *Domains* comprise the things we can talk about;
- *Relations* comprise the truths we utter about those things.

It follows that:

- First, domains and relations are both essential (without domains, there's nothing we can talk about; without relations, there's nothing we can say).
- Second, they aren't the same thing (beware anyone who tries to tell you otherwise!).

Now we can get back to the main theme of the present section. Historically, there's been much confusion between relations *per se* (i.e., relation *values*) and relation *variables*. Suppose we say in some programming language:

```
DECLARE N INTEGER ...
```

N here isn't an integer *per se,* it's an integer *variable* whose *values* are integers *per se*-different integers at different times. Likewise, if we say in SQL:

```
CREATE TABLE T ...
```

R here isn't a relation (or table) *per se,* it's a relation *variable* whose *values* are relations *per se*-different relations at different times. And when we "update R" (e.g., by "inserting a new row"), what we're really doing is *replacing the old relation value of R* en bloc *by an entirely new relation value*. Of course, it's true that the old value and the new value are somewhat similar-the new one just has one more row than the old one-but conceptually they *are* different values.

Go back to the top Previous page Next page

Now, the trouble is that, very often, when people talk about relations, they really mean relation variables, not relations *per se*. This distinction-or, rather, the fact that this distinction is usually not clearly made-has been a rich source of confusion in the past. For example, the overall value of a given relation, like the overall value of a given domain, doesn't change over time, whereas of course the value of a relation variable certainly does. Despite this obvious difference, some people-we suppress the names to protect the guilty-have proposed that domains and relations (meaning relation variables) are really the same kind of thing! See the section "Relvars *vs*. Object Classes," later.

In *The Third Manifesto,* therefore, we've tried very hard to be clear on this point (and the same goes for the rest of the present article). Specifically, we've introduced the term *relvar* as a convenient shorthand for *relation variable,* and we've taken care to phrase our remarks in terms of relvars, not relations, when it's really relvars that we mean.

DOMAINS *vs.* OBJECT CLASSES

It's an unfortunate fact that most people have only a rather weak understanding of what domains are all about; typically they perceive them as just conceptual pools of values, from which columns in relations draw their actual values (to the extent they think about the concept at all, that is). This perception is accurate so far as it goes, but it doesn't go far enough. The fact is, a domain is really nothing more nor less than a *data type*-possibly a simple system-defined data type like INTEGER or CHAR, more generally a user-defined data type like P# or QTY in the bill-of-materials example.

Now, it's important to understand that the data type concept includes the associated concept of the *operators* that can legally be applied to values of the type in question (values of that type can be operated upon solely by means of the operators defined for that type). For example, in the case of the system-defined INTEGER domain (or type-we use the terms interchangeably):

- The system defines operators "=", "<", and so on, for comparing two integers;
- It also defines operators "+", "*", and so on, for performing arithmetic on integers;
- It does *not* define operators "||", SUBSTRING, and so on, for performing string operations on integers (in other words, string operations on integers aren't supported).

Likewise, if we had a system that supported domains properly (but most of today's systems don't), then we would be able to define our own domains-say the part number domain P#. And we would probably define operators "=", "<", and so on, for comparing two part numbers. However, we would probably *not* define operators "+", "*", and so on, which would mean that arithmetic on part numbers would not be supported.

Go back to the top Previous page Next page

Observe, therefore, that we distinguish very carefully between a data type *per se* and the *representation* or *encoding* of values of that type inside the system. For example, part numbers might be represented internally as character strings, but it doesn't follow that we can perform string operations on part numbers; we can perform such operations only if appropriate operators have been defined for the type. And (in general) the operators we define for a given domain will depend on that domain's intended *meaning,* not on the way values from that domain happen to be represented or encoded inside the system.

By now you might have realized that what we've been talking about is what's known in programming language circles as *strong typing*. Different writers have slightly different definitions for this term; as we use it, however, it means, among other things, that (a) everything *has* a type, and (b) whenever we try to perform an operation, the system checks that the operands are of the right type for the operation in question. And note carefully that-as already indicated-it's not just *comparison* operations that we're talking about here (despite the emphasis on comparisons in much of the database literature). E.g., suppose we're given the well-known suppliers-and-parts database, with relvars S (suppliers), P (parts), and SP (shipments), and consider the following expressions:

1. P.WEIGHT + SP.QTY /* part weight plus shipment quantity */
2. P.WEIGHT * SP.QTY /* part weight times shipment quantity */

The first of these expressions makes no sense, and the DBMS should therefore reject it. The second, on the other hand, does make sense-it denotes the total weight for all parts involved in the shipment. So the operators we would define for weights and quantities would presumably include "*" but not "+".

Observe now that so far we've said nothing at all about the nature of the values that can belong to a domain. In fact, those values can be *anything at all!* We tend to think of them as being very simple (numbers, strings, and so forth), but there's absolutely nothing in the relational model that requires them to be limited to such simple forms. Thus, we can have domains of sound recordings, domains of maps, domains of videos, domains of engineering drawings, domains of legal documents, domains of geometric objects (and so on, and so on). The only requirement is that (to say it one more time) the values in the domain must be manipulable solely by means of the operators defined for the domain in question.

The foregoing message is so important that we state it again in different words:

THE QUESTION AS TO WHAT DATA TYPES ARE SUPPORTED
IS ORTHOGONAL TO THE QUESTION OF SUPPORT FOR THE
RELATIONAL MODEL

To sum up, therefore: What we're saying is that, in the relational world, a domain is a data type, probably user-defined, of arbitrary internal complexity, whose values are manipulable solely by means of the operators defined for the type in question. Now, if we turn to the object-oriented (OO) world, we find that what is arguably the most fundamental OO concept of all, the *object class,* is a data type, probably user-defined, of arbitrary internal complexity, whose values are manipulable solely by means of the operators defined for the type in question ... In other words, domains and object classes are *the same thing!* Thus, we have here the key to integrating the two technologies-and, of course, this position is exactly what we espouse in *The Third Manifesto*. Indeed, we believe that a relational system that supported domains properly would be able to deal with all of those "problem" kinds of data that (it's often claimed) OO systems can handle and relational systems cannot: time-series data, biological data, financial data, engineering design data, office automation data, and so on. Accordingly, we also believe that a true "object/relational" system is nothing more than a true *relational* system -- which is to say, a system that supports the Relational Model, with all that that entails.

Go back to the top   Previous page   Next page

RELVARS *vs.* OBJECT CLASSES

In the previous section we equated object classes and domains. Many people, however, equate object classes and *relvars* instead (see reference [6] for an example). We now argue that this latter equation is a serious mistake. Indeed, the *Manifesto* includes a categorical statement to the effect that *relvars are not domains*.

Consider the following example. First, here's part of a simple object class definition, expressed in a hypothetical OO language (the keyword PUBLIC is meant to indicate that the specified items are "public instance variables"):

```
CREATE OBJECT CLASS EMP
 PUBLIC ( EMP# CHAR(5),

          ENAME CHAR(20),
          SAL NUMERIC,
          HOBBY CHAR(20),
          WORKS_FOR CHAR(20) ) ... ;
```

And here's part of a simple relational-or at least SQL-table (relvar) definition:

```
CREATE TABLE EMP

     ( EMP# CHAR(5),
     ENAME CHAR(20),
     SAL NUMERIC,
     HOBBY CHAR(20),
     WORKS_FOR CHAR(20) ) ... ;
```

It's very tempting to equate these two definitions!-which is in effect what certain systems (both prototypes and commercial products) have already done. So let's take a closer look at this equation. More precisely, let's take the CREATE TABLE just shown, and let's consider a series of possible extensions that (some people would argue) make it more "OO"-like.

First, we allow column values to be *tuples from some other relvar* ("tuple" here being just another word for *row,* loosely speaking). In the example, we might replace the original CREATE TABLE by the following collection of definitions:

```
CREATE TABLE EMP

        ( EMP# CHAR(5),
        ENAME CHAR(20),
        SAL NUMERIC,
        HOBBY ACTIVITY,
        WORKS_FOR COMPANY ) ;

CREATE TABLE ACTIVITY

        ( NAME CHAR(20),
        TEAM INTEGER ) ;

CREATE TABLE COMPANY

        ( NAME CHAR(20),
        LOCATION CITYSTATE ) ;

CREATE TABLE CITYSTATE>

        ( CITY CHAR(20),
        STATE CHAR(2) ) ;
```

Go back to the top   Previous page   Next page

Fig. 3 shows the structure of relvar EMP at this point.

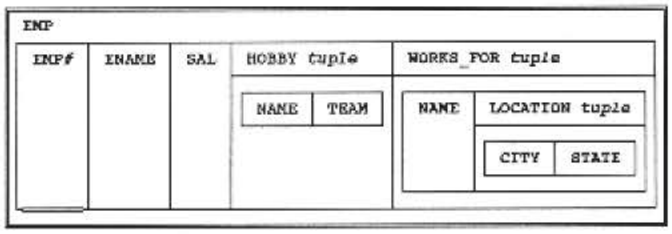


Fig. 3: Columns containing (pointers to) rows -- deprecated

Explanation: Column HOBBY in relvar EMP is declared to be of type ACTIVITY. ACTIVITY in turn is a relvar of two columns, NAME and TEAM, where TEAM gives the number of players in the corresponding team-for instance, a possible "activity" might be (Soccer,11). Each HOBBY value is thus actually a *pair* of values, a NAME value and a TEAM value (more precisely, it's a pair of values that currently appear as a row in relvar ACTIVITY). Note that we've already violated the dictum that relvars aren't domains!

Similarly, column WORKS_FOR in relvar EMP is declared to be of type COMPANY, and COMPANY is also a relvar of two columns, one of which is defined to be of type CITYSTATE, which is another two-column relvar, and so on. In other words, relvars ACTIVITY, COMPANY, and CITYSTATE are all considered to be *types* as well as relvars (as is relvar EMP itself, of course).

This first extension is thus roughly analogous to allowing objects to contain other objects, thereby supporting the concept sometimes known as a *containment hierarchy*.

Note: As an aside, we remark that we have characterized this first extension as "columns containing rows" because that is the way advocates of the "relvar = class" equation themselves characterize it. It would be more accurate, however, to characterize it as "columns containing *pointers to* rows" -- a point that we will be examining in a few moments. (In Fig. 3, therefore, we should really replace each of the three appearances of the term *row* by the term *pointer to row*.) Analogous remarks apply to the second extension also.

That second extension, then, is to add *relation-valued columns*. E.g., suppose employees can have an arbitrary number of hobbies, instead of just one (refer to Fig. 4):

Go back to the top Previous page Next page

```
CREATE TABLE EMP

        ( EMP# CHAR(5),
        ENAME CHAR(20),
        SAL NUMERIC,
        HOBBIES SET OF ( ACTIVITY ),
        WORKS_FOR COMPANY ) ;
```

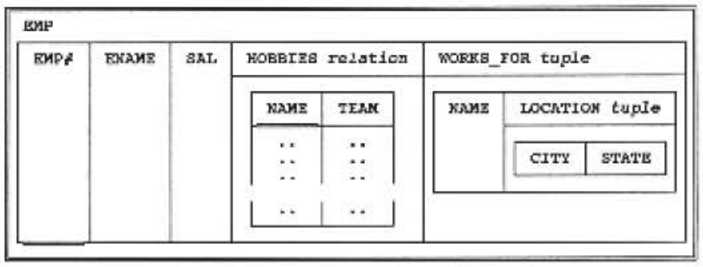


Fig. 4: Columns containing sets of (pointers to) rows -- deprecated

Explanation: The HOBBIES value within any given row of relvar EMP is now (conceptually) a *set* of zero or more (NAME,TEAM) pairs-i.e., rows-from the ACTIVITY relvar. This second extension is thus roughly analogous to allowing objects to contain "aggregate" objects (a more complex version of the containment hierarchy).

The third extension is to permit relvars to have associated *methods* (i.e., operators). E.g.:

```
CREATE TABLE EMP

        ( EMP# CHAR(5),
        ENAME CHAR(20),
        SAL NUMERIC,
        HOBBIES SET OF ( ACTIVITY ),
        WORKS_FOR COMPANY )

 METHOD RETIREMENT_BENEFITS ( ) : NUMERIC ;
```

Explanation: RETIREMENT_BENEFITS is a method that takes a given EMP instance as its argument and produces a result of type NUMERIC. The code that implements the method is written in a language such as C.

The final extension is to permit the definition of *subclasses*. E.g. (refer to Fig. 5):

```
CREATE TABLE PERSON

        ( SS# CHAR(9),
        BIRTHDATE DATE,
        ADDRESS CHAR(50) ) ;
```

Go back to the top   Previous page   Next page

```
CREATE TABLE EMP

        AS SUBCLASS OF PERSON
        ( EMP# CHAR(5),
        ENAME CHAR(20),
        SAL NUMERIC,
        HOBBIES SET OF ( ACTIVITY ),
        WORKS_FOR COMPANY )

METHOD RETIREMENT_BENEFITS ( ) : NUMERIC ;
```

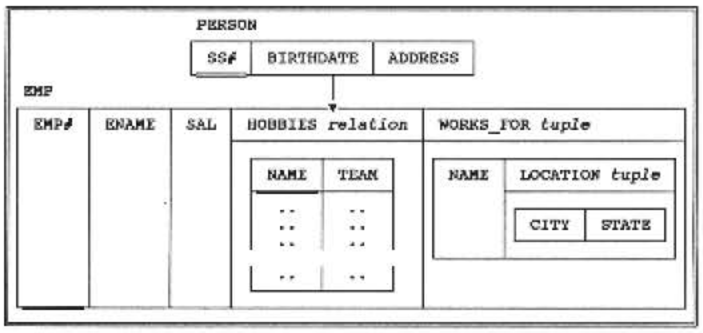


Fig. 5: Relvars as superclasses and subclasses -- deprecated

Explanation: EMP now has three additional columns (SS#, BIRTHDATE, ADDRESS) inherited from PERSON. If PERSON had any methods, it would inherit those too.

Along with the definitional extensions sketched above, numerous manipulative extensions are required too, of course-for instance:

- Path expressions-e.g., EMP.WORKS_FOR.LOCATION.STATE (and note that such expressions can return scalars or rows or relations, in general; note further that in the latter two cases the components of those rows or relations might themselves be rows or relations in turn, and so on)
- Row and relation literals (possibly nested)-e.g.,

```
( 'E001', 'Smith', $50000,
          { ( 'Soccer', 11 ), ( 'Bridge', 4 ) },
                ( 'IBM', ( 'San Jose', 'CA' ) ) )
```

- Relational comparison operators-e.g., SUBSET, SUBSETEQ, and so on
- Operations for traversing the class hierarchy
- The ability to invoke methods within expressions-e.g., in WHERE clauses
- The ability to access individual components within column values that happen to be rows or relations

Go back to the top   Previous page   Next page

So much for a quick overview of how the "relvar = class" equation is realized in practice. What's wrong with it?

Well, first of all, a relvar is a variable and a class is a type; so how can they *possibly* be the same thing? (We showed in the section "Relations *vs*. Relvars" that relations and domains aren't the same thing; now we see that relvars and domains aren't the same thing either.)

The foregoing argument should be logically sufficient to stop the "relvar = class" idea dead in its tracks. However, there is more that can usefully be said on the subject, so let us agree to suspend disbelief a little longer ... Here are some more points to consider:

- The equation "relvar = class" implies that "objects" are *rows,* and the corresponding (public) "instance variables" are *columns*. It follows that, whereas a pure OO class has methods and no public instance variables, a relvar "class" has public instance variables and only optionally has methods. So, again, how can the two possibly be the same?

- There's a major difference in kind between the column definitions (e.g.) SAL NUMERIC and WORKS_FOR COMPANY. NUMERIC is a true data type (equivalently, a true-albeit primitive-domain); it places a time-independent constraint on the values that can appear in column SAL. By contrast, COMPANY is *not* a true data type; the constraint it places on the values that can appear in column WORKS_FOR is *time-dependent* (it depends, obviously, on the current value of relvar COMPANY). In fact, as pointed out earlier, the relvar *vs.* domain distinction has been muddied.
- As we saw, row "objects" can contain other such "objects"; e.g., EMP "objects" (apparently) contain COMPANY "objects." But they don't!-not really; instead, they contain *pointers* (object IDs, to use the OO term) to those "contained" objects, and users must understand this point clearly. E.g., suppose the user updates one particular COMPANY row in some way (refer back to Fig. 3). Then that update will immediately be visible in all EMP rows that correspond to that COMPANY row.

  It follows that we're not really talking about the relational model any more. The fundamental data object isn't a relation containing values, it's a "relation" (actually not a proper relation at all) containing values *and pointers*.

- Suppose we define view V to be the projection of EMP over (say) just ENAME. V is a relvar too, of course (a *derived* one, whereas EMP is a *base* relvar). Thus, if "relvar = class" is a correct equation, V is also a class. *What class is it?* Also, classes have methods; *what methods apply to V?*

  Well, "class" EMP had just one method, RETIREMENT_BENEFITS, and that one clearly doesn't apply to V. In fact, it hardly seems reasonable that *any* methods that applied to "class" EMP would apply to V-and there certainly aren't any others. So it looks as if (in general) *no methods at all* apply to the result of a projection; i.e., the result, whatever it is, isn't really a class at all. (We might *say* it's a class, but that doesn't make it one!-it will have public instance variables and no methods, whereas we've already observed that a true class has methods and no public instance variables.)

Go back to the top   Previous page   Next page

  In fact, it's clear that when people equate relvars and classes, it's specifically *base* relvars they're referring to-they're forgetting about the *derived* ones. (Certainly the pointers discussed above point to rows in base relvars, not derived ones.) As we've argued elsewhere [5], to distinguish base and derived relvars in this way is a mistake of the highest order, because the question as to which relvars are base and which derived is, to a very large degree, arbitrary. For further discussion of this important issue, see that same paper [5].

- Following on from the previous point: Suppose we have a relvar *R* and we project it over *all* of its columns. In a typical relational language, the syntax for such a projection is simply *R*. So the expression "*R*" is now ambiguous! If we think of it as referring to relvar *R,* then it's a class, with methods. If we think of it as referring to the projection of relvar *R* over all of its columns, then it's not a class and it has no methods. How do we tell the difference?
- Finally, *what domains are supported?* Those who espouse the "relvar = class" equation never seem to have much to say about domains, presumably because they cannot see how domains fit into their overall scheme. And yet (as we saw in the section "Relations *vs*. Relvars" earlier) domains are essential.

A NOTE ON INHERITANCE

You might have noticed that we did briefly mention the possibility of *inheritance* in the previous section but not in the earlier section "Domains *vs*. Object Classes." And you might therefore have concluded that support for inheritance does constitute at least one point in favor of the "relvar = class" equation. Not so, however; we do indeed want to include inheritance as part of our "domain = class" approach, and thus (e.g.) be able to define domain CIRCLE as a "subdomain" of "superdomain" ELLIPSE. The problem is, however, there doesn't seem to be a clearly defined and generally agreed *model* of inheritance at the time of writing. As a consequence, *The Third Manifesto* includes *conditional* support for inheritance, along the lines of "if inheritance is supported, then it must be in accordance with some well defined and commonly agreed model." We do also offer some detailed proposals toward the definition of such a model.

CONCLUDING REMARKS

We have discussed the question of integrating relational and object-oriented (OO) database concepts. In our opinion, OO contains exactly one unquestionably good idea: user-defined data types (which includes user-defined operators). It also contains one *probably* good idea: type inheritance (though the jury is still out on this one, to some extent). A key technical insight underlying *The Third Manifesto* is that these two ideas are *completely orthogonal to the relational model*. In other words, the relational model needs no *extension,* no *correction,* no *subsumption*-and, above all, no *perversion!*-in order for it to accommodate these orthogonal ideas.

To sum up, therefore: What we need is simply for the vendors to give us *true relational DBMSs* (and note that "true relational DBMSs" does *not* mean SQL systems) that include *proper domain support*. Indeed, an argument can be made that the whole reason OO systems (as opposed to "O/R" systems) look attractive is precisely the failure on the part of the SQL vendors to support the relational model adequately. But this fact shouldn't be seen as an argument for abandoning the relational model entirely (or at all!).

ACKNOWLEDGMENTS

This article is an updated version of "Introducing *The Third Manifesto*" by Hugh Darwen and C. J. Date, which appeared in *Database Programming & Design 8,* No. 1 (January 1995); it appears here by permission of Miller Freeman Inc. We would also like to thank the many people who have reviewed drafts of *The Third Manifesto* and offered constructive criticism and helpful comments on those drafts.

Go back to the top Previous page Next page

REFERENCES

1. Malcolm Atkinson *et al*. "The Object-Oriented Database System Manifesto." Proc. First International Conference on Deductive and Object-Oriented Databases, Kyoto, Japan (1989). New York, N.Y.: Elsevier Science (1990).

2. Hugh Darwen. "Adventures in Relationland." In C. J. Date and Hugh Darwen, *Relational Database Writings 1985-1989*. Reading, Mass.: Addison-Wesley (1990).

3. Hugh Darwen and C. J. Date. "The Third Manifesto." *ACM SIGMOD Record 24,* No. 1 (March 1995). Version Two of this document is due to be published in book form by Addison-Wesley in 1998.

4. C. J. Date. *An Introduction to Database Systems* (6th edition). Reading, Mass.: Addison- Wesley (1995).

5. C. J. Date. "Objects and Relations: Forty-Seven Points of Light." *Data Base Newsletter 23,* No. 5 (September/October 1995).

6. Won Kim. "On Marrying Relations and Objects: Relation-Centric and Object-Centric Perspectives." *Data Base Newsletter 22,* No. 6 (November/December 1994).

7. Michael Stonebraker *et al*. "Third Generation Database System Manifesto." *ACM SIGMOD Record 19,* No. 3 (September 1990).

THE THIRD MANIFESTO: A NOTE REGARDING THE BOOK-LENGTH VERSION

The book's full title is the same as that of the current article -- *viz., The Third Manifesto: Foundation for Object/Relational Databases*. And there's a subtitle too: *a detailed study of the impact of objects and type theory on the relational model of data, including a comprehensive proposal for type inheritance*. As noted in the abstract to the present article, *The Third Manifesto* is a detailed and rigorous proposal for the future of data and database management systems; it consists of a precise, formal definition of an abstract model of data, to be considered as a blueprint for the design of a DBMS and a database language. In particular, it provides a rock-solid foundation for integrating relational and object technologies, a foundation conspicuously lacking in current approaches to such integration.

The proposed foundation represents an evolutionary step, not a revolutionary one. It builds on Codd's relational model of data and on the research that sprang from Codd's work. Most notably, it incorporates a precise and comprehensive specification for a method of defining data types, including a comprehensive model of type inheritance, to address a lack that has been observed by many authorities; thus, it also builds on research in the field of object orientation. With a sound basis in both camps of the object/relational divide, therefore, the **Manifesto** is offered as a firm foundation for true object/relational DBMSs.

Go back to the top Previous page Next page

The book is arranged into four parts and a set of appendixes:

I. Preliminaries: Background and overview; objects and relations

II. Formal Specifications: The *Manifesto* proper; a new relational algebra; and a language called **Tutorial D,** a concrete realization of the ideas of the *Manifesto*

III. Informal Discussions and Explanations: A careful point-by-point examination and exposition of the *Manifesto,* with copious examples in **Tutorial D**

IV. Subtyping and Inheritance: A detailed and comprehensive proposal for a model of type inheritance, with (again) numerous examples

Appendixes: Annotated references and bibliography; comparisons with SQL3 and ODMG; database design considerations; and many other topics

The authors combine precision and thoroughness of exposition with the approachability that readers familiar with their previous publications will recognize and welcome. The book is essential reading for the database student or professional. *Author information:* **Hugh Darwen** (darwen@vnet.ibm.com, +44 (0) 1926-464398 voice, +44 (0) 1926-410764) is a database systems specialist at IBM United Kingdom Limited; **C. J. Date** (+1 707/433-6523 voice, +1 707/433-7322 fax) is an independent author, lecturer, researcher, and consultant, specializing in relational database systems.

***** End *** End *** End *****

Go back to the top Previous page Next page