Space Enough and Time: Accommodating Special Data Types

Dr. Kevin Wilkinson
Department of Information Systems,
Massey University
Wellington, New Zealand
K.J.Wilkinson@massey.ac.nz

ABSTRACT

A case study is described in which students are asked to include two special data types in a process design assignment, one for spatial coordinates of latitude and longitude, the other for historical dates that range from AD to BP. The problems of implementing these data types by means of a programming language that interfaces with a relational DBMS are examined. It is argued that these problems could be overcome by switching to an object-oriented DBMS or a so-called object-relational DBMS (a relational DBMS extended to include certain features of object-orientation). The case seems to support C.J. Date's argument that it is domains (data types) that should be equated with object classes, not tables.

KEYWORDS

Extensible data types

1. THE CASE STUDY AND ITS CHALLENGES

At its Wellington campus, the Information Systems Department of Massey University is developing a set of case studies each of which forms the basis of graduate diploma and 3rd year degree paper called *Process Design Application.*

As befits the aims of the paper, the assignment is practical in nature and requires students to demonstrate their understanding of the processing requirements of an application, for which they have the data model, by developing a working prototype.

One of the case studies is based on *Te Kahui*, the Collections Management system of *Te Papa Tongarewa*, the National Museum of New Zealand. Te Papa, a national icon situated within walking distance of the campus, is a Public Good institution with interesting and sophisticated information technology needs. As such, it has a specialised realism that makes it an appealing subject for educational study.

Students are asked to develop a prototype that

maintains museum catalogue records for any number of cultural and scientific collections such as Art, Birds, Crustacea, Maori, Molluscs, Philatelic, Photography, Reptiles and so on. A mature version of the system would also maintain details, both current and historical, of the acquisition, storage, insurance, restriction, deaccession, condition, treatment, movement, exhibition and archiving of items in these collections.

There are many interesting features to tax both the imagination of the designer and the features of the software tools, including the need to support data entries at any level of the Linnaean taxonomy of life forms.

Two features, though, are especially challenging because they require, in effect, the provision of special data types. Firstly, for specimens collected on field trips, the location of the find must be recorded with spatial coordinates of latitude and longitude. Secondly, cultural and natural history items alike require historical dates to be recorded that lie far beyond the scope of the date/time validation logic of normal commercial software.

2. THE COMPLEXITY OF THE TASK

The complexity of the task of validating these special data types derives from the combination of the following features.

2.1 Superposition

A superposition of formats is implied. An historical date field may contain an AD, BC, or BP date. (BP dates are before the present era. We are currently in the Cenozoic era, which began about 65 million years ago after the mass extinctions of the Cretaceous period, so BP dates begin 65 million years ago.) Each of these variants has different validation rules for its year number. Spatial coordinates may be either latitude or longitude, which also have different validation rules. As we will see, these are probably better implemented as two separate types, in which case there will be no superposition.

2.2 Tight Editing

Tight editing and formatting are important. The user needs spatial coordinates and historical dates that are strictly normalised so that range searching on these fields will deliver accurate results for a mature database that has been maintained by many different users.

2.3 Ease of Data Entry

Editing should make data entry easy and convenient. Missing values should be supplied so that, for example, if a longitude of 170W is entered, it is converted to 170 00.00 W. A number of common delimiters should be recognised so that the coordinates 170,30W, 170;30W, and 170/30W are all converted to 170 30.00 W. Dates default to AD but the AD period can be explicitly shown if required.

2.4 Smart Editing

Editing should be smart. For example, the circa qualifier should be recognised no matter where it is placed so that c1870 and 1870c are treated as equivalent. The user should not be expected to supply leading zeros. Degrees of latitude, for example, must be padded to two digits, whereas degrees of longitude require three. Minutes of latitude and longitude are optional. If entered, they may include decimal fractions but the decimal point is not required for whole numbers of minutes.

2.5 Structure

Both data types have a bit of structure that their validation functions need to decipher. Coordinates have degree and minute parts and a compass point of N or S for latitude and E or W for longitude. Dates can be AD, BC, or BP and may also be qualified as circa (c), decade or century (s), or uncertain (?).

2.6 Consistency Checking

This structure implies some consistency checking. For example, N(orth) is valid for latitude, but not for longitude. 180 is the maximum degree of longitude but 90 is the maximum for latitude. 1999 is a valid year, but is incompatible with the decade or century qualifier.

3. A MODEL ANSWER

Because of this complexity, it was thought prudent to code a solution to the task of validating these data types before asking students to attempt it. The resulting program, **checker.exe**, returns a Boolean analysis of the validity of each entry and a message describing its format. The form layouts of the program are included here as an appendix.

The task took about 400 lines of Visual Basic code to solve. This figure includes a number of blank lines added to aid legibility but does not include the code attached to the forms that allow the two validation functions to be demonstrated.

Students taking the paper are in the Information Systems stream of a business-oriented qualification and cannot be expected to have a Systems Development focus. The programming skills of the average student will be rudimentary. Including the design and implementation of these two complex data types in the prototyping task would risk turning what is already seen as an overly technical assignment into one that is intolerably so.

Consequently, it has been decided to supply the validation functions as instructive solutions and to invite the students to incorporate them in their prototype, modifying them as they see fit.

Even so, challenges remain. Entering the data is just the beginning. Values then have to be retrieved, selected, and sorted, which will not always straightforward. In a range check, for example, the decade or century qualifier itself implies a range, so a search on dates that fall between 1925 and 1935 should include items whose date shows as 1920s.

4. A BETTER TOOLSET

Given the toolset the students are expected to use (typically a system written in Visual Basic that connects with an MS Access database), there does not seem to be any feasible way of validating and normalising these data types other than by coding the sort of functions that appear in the model answer.

The format and input mask features in MS Access don't help because we would need to be able to define several different formats or masks and superimpose them. (Formats can force particular values or types of values into a character string in a nominated position. An input mask does the same but the special characters are not stored with the rest of the data).

Even if different formats could be superimposed, we want to provide normalization, not just enforce it. The input rules must be very permissive, standardizing the format and supplying missing data. This almost certainly requires program code attached to a data type that is run whenever an instance of the type is processed. Even if formats and masks could be made to work, we would not want to have to specify a complicated set of them for every occurrence of a type.

Like most languages, Visual Basic includes a Type statement that allows the user to define her own data structures. However, this merely allows a compound structure to be defined in terms of the elementary data types that come with the language. We are perfectly happy with the elementary String data type: we would just like to be able to define a number of different string patterns and superimpose them.

If we were writing in ANSI COBOL, we might be able to do the job using lots of REDFINE clauses in the data division, but we would still need to repeat all that work for each declaration of a special data type.

What we really need is a language that has extensible data types so that users can define their own special types and incorporate them into the language to be used in the same way as the built-in system types.

5. EXTENSIBLE DATA TYPES

5.1 High Hopes for Object-orientation

Most toolsets do not provide extensible data types. Powerful products such as the Oracle Designer 2000 case tool go some way towards providing the sort of features required, but even there user-defined types are little more than a named data structure that is declared as part of one table in a way that makes it

reusable in others.

A good theoretical account of the main features that are needed to implement user-defined types can be found in Date 2000, section 5.2 and chapter 19. The examples of coded type declarations in this paper are original, written to suit the data types of the case study. The language used is **Tutorial D**, which follows Date 2000.

In general, what we want is the ability to define data types that become part of the toolset used within the organization. We do not want to have to define special data types anew for each application.

Further, we need declarative integrity support so that typing errors can be detected at compile time, not just at runtime. A language that provides only procedural integrity support is not good enough. There, type error logic is coded in procedures and takes effect only when those procedures are executed [Date 2000, 250].

Object-oriented tools are the obvious candidates for acquiring this ability.

Firstly, an object class, the fundamental concept of object-orientation, is a data type that either comes with the system or can be defined by the user [Date 2000, 122-3].

Secondly, object classes can be of arbitrary complexity and are therefore well suited to instantiating problematic kinds such as "… time series data, biological data, financial data, engineering design data, office automation data, and so on" [Date 2000, 863-4].

If an object-oriented database is used, extensible typing has the effect of moving semantics from applications into the database [Loomis 1995, 40]. This provides a consistency of type checking across all applications, which is highly desirable.

But object-orientation, though sufficient, may not be necessary. Date devotes an entire chapter of his book to arguing that the relational model is capable of being extended to include the best features of object-orientation, including extensible data types. Whatever the tools, they need to provide support for the central concepts of data typing: possible representations, components of representations, type constraints, and operators. The need for type inheritance is also considered.

5.2 Possible Representations

We should be familiar with the idea of a data type being encapsulated in the sense that its physical representation is hidden from the user and where what the user sees is merely one of the possible representations of a value of that type [Date 2000, 115-9].

For example, when SQL received its first major revision about 1992, the resulting language SQL2 extended the set of built-in data types to include times and dates [Eaglestone and Ridley 1998, 341]. We can imagine the date on which William I of Normandy and Harold II the Saxon King of England fought the Battle of Hastings as being represented in various ways, such as:

October 14, 1066
14 Oct 1066
1066 287

The last of these is sometimes known as the "Julian" date format, and should not be confused with the Julian calendar. It provides a convenient way to ensure that records sort correctly into date order without requiring special formatting.

Regardless of how the date is stored on disk, it must be possible to define a user view of the value in any of the available representations. This implies that the stored type must be translatable from any of its representations to any other.

Another example, [from Date 2000, 117 and elsewhere] is that of a geometric point, which can be represented by either a pair of Cartesian coordinates (the usual convention), or by polar coordinates (where the point is arrived at by rotating a radius of length *R* through ?degrees).

Now it should be obvious that the cases we are considering of spatial coordinates and historical dates

do not really need more than one representation. Latitudes and longitudes are different domains, not alternative representations of the same domain. Similarly, BP, BC, and AD dates are not alternative ways of expressing a year number. The way our case study defines them, they are mutually exclusive domains. The year 65,000,001 BC is not the same as 1 BP: it is *invalid*.

However, if our case study were ever required to exhibit cultural sensitivity and cater for calendars other than the Gregorian, such as Hebrew, Chinese, Hindu, or Muslim, a possible representation would be needed for each. Although modern and future years could be expressed in any form, some years would be invalid under some representations. For example, year 621 in the Gregorian calendar would be invalid in the Muslim, being the year *before* Mohammed fled from Mecca to Medina. One would need to introduce BM dates (before Mohammed) or the like to the Muslim calendar to correct for this.

5.3 Components of Representations

Possible representations have components. Because our data types need only one possible representation, their declaration might look like

```
TYPE LATITUDE
      POSSREP      (DEGREE INTEGER,
                    MINUTES REAL,
                    NS CHAR);
TYPE LONGITUDE
      POSSREP      (DEGREE INTEGER,
                    MINUTES REAL,
                    EW CHAR);
TYPE HISTDATE
      POSSREP      (YEAR INTEGER,
                    PERIOD CHAR,
                    DCIND CHAR,
                    APPROX BOOLEAN);
```

In each case, the POSSREP name is omitted so that it will default to the TYPE name. Types with multiple representations must name them individually.

Variables of these types could then be declared, such as

```
      LastSeenAtLat:     LATITUDE;
      LastSeenAtLong:    LONGITUDE;
      DateOfOrigin:      HISTDATE;
```

How a value is represented is a related but separate issue from how values are displayed. The present version of the case study asks that the “AD” be optionally displayed for those dates, but other formatting might well be requested too. For example, instead of “1000000000BP” we might prefer to see “1B BP”, using “B” to abbreviate “billion”. Similarly, instead of “7000000BC” we might prefer “7M BC”, where “M” abbreviates “million”.

The “B” and “M” values do not form part of a representation. Instead, they are modifiers in the form of a mask that causes data to be transformed as it passes between the storage area and the display medium. Such transformations would be coded as part of the Get and Put operators (see 5.5 below). Similarly, comma or space separators could be supplied to make large year numbers more legible.

5.4 Type Constraints

We would use type constraints to limit the range of values to just those we want to be valid for the type in question [Date 2000, 251-2]. To be more accurate, the constraints are specified for the components of each possible representation, but we have agreed that the types in our case study will have only one such representation.

Much of the code in the functions of the **checker.exe** program will find its way into these constraint statements. For example, to ensure that years are not equal to zero we might code

```
TYPE HISTDATE
      POSSREP      (YEAR INTEGER,
            PERIOD CHAR,
            DCIND CHAR,
            APPROX BOOLEAN)
CONSTRAINT THE_YEAR (HISTDATE) <> 0;
```

As long as our language allows compound expressions that can range over all components of a representation, we should be able to code consistency checks this way as well.

Another desirable feature would be the ability to return constraint-sensitive error messages when an invalid value is detected. Instead of a general error condition, such as

```
Value conflicts with data type
```

we would like to be able to return

```
Invalid: latitude degrees > 90
Invalid: AD date > 10000
```

and so on.

5.5 Operators

Constraints can be given user-defined names that allow them to serve as operators. Together, these operators are defined on the possible representations of a domain and specify all its possible legal values.

Values are maintained solely by operators defined on their domain. That is, the physical representation of a domain's values remains hidden [Date 2000, 114].

We can distinguish between "observer" and "mutator" operators. Observers GET values for a client while mutators PUT or SET them [Date 2000, 120 footnote]. Constraints are checked whenever a mutator operator is used.

It is also possible to select a value of a particular type by specifying the values of its components [Date 2000, 117]. For example, we could speak of the value

```
LATITUDE (25, 59.99, N)
```

THE_ operators allow these components to be addressed individually. This makes components visible to the user, but the data type as a whole is still encapsulated: what is visible is the component *of a representation* [Date 2000, 115 point 4].

Selector operators can be used to achieve type conversions. For example, if the YEAR component of an historical date needs to be compared to an integer, the integer can be coerced into the same type to avoid a typing error, effectively producing, for example,

```
IF THE_YEAR (DateOfOrigin) = THE_
YEAR(2000) THEN ...
```

This selection ability provides functionality well beyond that of the model answer written in Visual Basic.

5.6 Type Inheritance

Inheritance is one of the hallmarks of object-orientation, but would it help our present cause?

Probably not. When it comes to performing the work done by the functions in our model answer, type constraints provide the most useful feature.

Assuming we would implement spatial coordinates as two separate types, there may be some advantage to be gained by splitting the work between a supertype SCOORD and its two subtypes LATITUDE and LONGITUDE. Some of the constraints on the DEGREE and MINUTE components could be specified at the supertype level, while the remaining ones and the checking of the compass point components NS and EW would be specializations at the subtype level.

However, this seems a bit clumsy and the potential gains in terms of future expansions of the hierarchy seem limited.

6. DATA TYPE BICULTURALISM

One of the most attractive prospects of using an object-oriented DBMS is that developers do not have to deal with what we can informally call data type biculturalism. A seamless transition can be made from the types available for holding transient data to those available when moving it to a persistent store. As Mary Loomis puts it,

"The extensible type system gives the object database the ability to provide persistence for any kind of data that applications need. Applications are not required to translate objects from the programming language types to a set of built-in database types." [Loomis 1995, 32]

When we add to this benefit the feature we noted in 5.1 above that object-oriented databases move the semantics of type checking from applications into the database, we have a significantly improved

programming environment.

Of course, not all object-oriented products can be expected to deliver this second benefit. As Loomis points out, “Even though a fundamental principle of object technology is the close coupling of data structure and applicable operations into objects, most object DBMS products currently manage only states, in the form of data structures …, leaving the behavioral aspects of objects to the object programming languages.” [Loomis 1995, 47]

The combination of Visual Basic and MS Access obviously fails on both counts: although it supports object classes, it does so only on the programming language side of the bicultural divide and the database side supports neither object states nor object behaviour.

7. THE EXTENDED RELATIONAL MODEL

So far, the clear favourite as a toolset for our special data types is one that includes an object-oriented DBMS that stores and executes operations in the database engine rather than in application space. Loomis identifies Servio’s *Gemstone* and Hewlett Packard’s *OpenODB* as products that do this [Loomis 1995, 47-8].

In most organizations, however, there is too valuable a legacy of application code to contemplate abandoning all the code written in, say, C++ and rewriting it in, say, an object-oriented version of SQL. It would be preferable to use a database whose engines speaks the application language.

It is still far from certain that object-oriented DBMSs will ever cross the chasm between early adopters and early majority in the market maturation model (Loomis 1995 205-8 contains an interesting discussion of this.) The dominance of the relational database model is such that a smooth conversion to a so-called “third generational” RDBMS would be more acceptable to most organizations than a jarring change to an object model.

In any case, how superior is the object model? Eaglestone and Ridley argue that the object-oriented data model is superior to the relational model because of its ability to incorporate real world meaning.

“The strength of [object-oriented technology] is that, instead of having a fixed number of different types of abstraction for different types of real world phenomena built into the model, an extensible type system allows designers to define new types to model different types of real world information.” [Eaglestone and Ridley 1998, 354].

When one compares the relational model as implemented in available products, this seems correct. However, C.J. Date has argued that the available products do not do justice to the full power of the relational model, which, like the object model, in fact allows for domains of arbitrary complexity.

This raises the possibility of a relational model extended to provide the best features of object-orientation, such as extensible data types.

In appendix B of Date 2000, an account of one of these “third generational” RDBMSs is given. The SQL3 standard is described, which includes such features as

- Composite attributes
- Relation-valued attributes
- Methods for tables
- Subclasses.

Unfortunately, SQL3 commits what Date calls “The Great Blunder” [Date 2000, 865, 907-8], which is to equate *tables* with object classes instead of equating *domains* with object classes.

Several arguments are advanced to show why this standard, though admirable in other ways, commits The Great Blunder.

One of these arguments seems pertinent here. We need time-independent integrity checks, not time-dependent ones, and implementing objects as tables rather than domains makes the outcome of integrity checks depend on table contents, which can vary over time [Date 2000, 870].

This is a problem that would need to be addressed. The object model seems to be the preferred option for implementing the special data types we have been considering here. An version of the relational model might well be another option, but we would need to be able to implement our special types in a way that encapsulated them from application programmers. The types should be an extension of the data definition language, not a set of integrity checks held on a table that anyone can modify.

8. CONCLUSIONS

Starting with a realistic case study, we have seen how two special data types in it pose problems for students and staff alike. Students working on the case study would find coding the validation and normalization logic too demanding. Staff can provide the necessary code as instructive solutions, but there is no way to provide it where it belongs: as part of the toolset.

Object-oriented systems support extensible typing because special types can be implemented as object classes and variables can then be declared as being of those classes. We would then hope to have an object-oriented database that avoids data type biculturalism. That is, we want the set of types available to the programming language to be the same as the set used when persistent data is stored.

Object-oriented databases have still not crossed the chasm between early adopters and early majority and may never do so. A database with a relational model extended to include the main benefits of object-orientation may prove the most acceptable way of obtaining the sort of features we are looking for. Any such system that treats objects as tables instead of domains may still not accommodate our special data types satisfactorily though. We need types whose rules do not rely on the time-dependent content of tables.

REFERENCES

Date, C.J. 2000, An Introduction to Database Systems, 7ed., Addison-Wesley.

Eaglestone, Barry, and Ridley, Mark 1998, Object Databases: An Introduction, McGraw-Hill.

Loomis, Mary E.S. 1995, Object Databases: The Essentials, Addison-Wesley.

APPENDIX

Visual Basic forms for the model answer

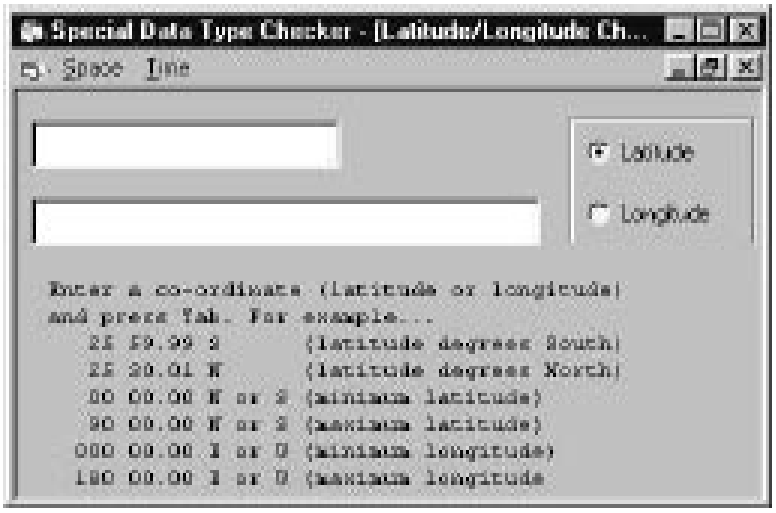

Figure 1.
Latitude/Longitude Checker

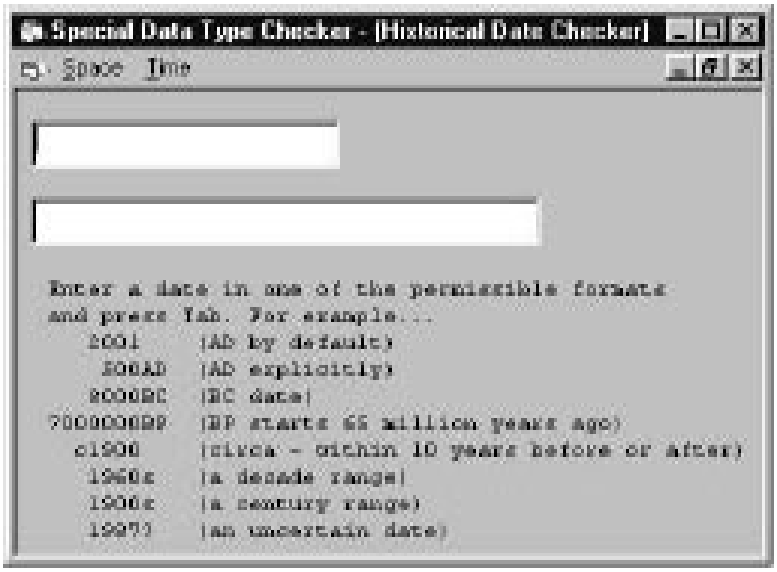

Figure 2.
Historical Date Checker