Flexidata: Using Metadata Liberally

David Warner
Kevin Wilkinson

Massey University at Wellington
Wellington, New Zealand

D.Warner@massey.ac.nz

Proceedings of the 15th Annual NACCQ, Hamilton New Zealand July, 2002 www.naccq.ac.nz

ABSTRACT

Two innovative and contrasting ways of using metadata are described. One involves keeping metadata in an Excel spreadsheet during the design phase of an application system for a government agency so that a complex and volatile set of features can be managed. The other involves a museum collections case study in which metadata is made available to the end users as operational data so that the functionality of the system can be expanded after implementation. The advantages and limitations of each approach are discussed.

Keywords: Metadata, case studies.

1. INTRODUCTION

In this paper, we use 'flexidata' to refer to metadata that is stored and used in the same way as ordinary operational data.

Flexidata can be thought of as either a lower level of metadata or a higher level of operational data.

Normally, we hold operational data on tables and the data that describes this data, the metadata, is looked after by the operating system or database management system. With flexidata, there is an intermediate level. We hold some of the metadata on tables and allow it to be seen and manipulated as though it were operational data.

The point of doing this is to defer decisions about what the operational data will really look like.

Two examples are described. The first involves using flexidata to defer decisions during the design phase of a large and complex development project in which the user requirements are unusually volatile. The second involves a system in which tables of metadata are provided to the user to allow the operational data to be stored in new formats after implementation.

Metadata is an active research area, but the current focus is on mark-up languages, where text and formatting instructions are bundled together in the same document only to be channelled to different destinations: the text to the end-user, the metadata to the processing software. The concerns of this paper, which are to displace metadata from its usual context in order to automate the production of forms or to expose it to the user directly in the interests of flexibility, do not seem to be recognised as design strategies.

2. DEFERRING PRE-IMPLEMENTATION DECISIONS

Flexidata was used to ease the problems in a particular case where new legislation very substantially increased the requirements for an existing risk management system application that was deemed to be un-maintainable. The new requirements involved a large increase in the scope and complexity of the system. The business had to develop new processes in parallel with the development of the supporting application.

2.1 REQUIREMENTS AND DESIGN APPROACH

The design approach was required to allow end-users to contribute to the design of the application in a situation where the user requirements and methods of calculation were being developed in parallel with the application.

The implementation was to use SQL Server with a front-end using Visual Basic.

The form layout was to be consistent across all of the forms. Although the forms were all different, any similarities were to be exploited to give a uniform look and feel to the application.

Further requirements that influenced the design but were not directly associated with the decision to use the flexidata approach were that the database be temporal and that, within a session, there should be limitless undo and re-do. As the database is used to support New Zealand legislation there is a need to track all changes made in the data and to examine and compare the state of the database at points in the past.

2.2 SCALE

39 forms are required to define the input information to the application, 667 fields are used to contain that information, and 155 of these fields are memo fields (Further fields in the database are used for derived information and status information).

2.3 APPROACH

Users and the application developer jointly examined the data requirements and the calculation processes and defined the forms required and the layout for each form.

The forms were implemented and the results reviewed by the team. An iterative process of review and change was followed.

2.4 FORESEEN PROBLEMS

Keeping the form layouts consistent and changing the forms was expected to be a major undertaking. It transpired that, over the development period, the number of fields and the number of forms doubled and it was necessary to split or combine many of the forms.

It was expected to be difficult to keep the look and feel of the application consistent, (e.g. all combo boxes should work in the same way) and difficult if a change of behaviour for a control was required.

2.5 MANAGING THE PROBLEMS

To manage the foreseen problems it was decided to automate the generation of forms. Some thought was given to containing the form descriptions in a database, but it was decided that for ease of management it would be sensible to use Excel to allow the use of formulae and cut-and-paste functions.

An example of the use of formulae is that in many cases the height and spacing of controls is calculated from a knowledge of the number of controls and the space into which they need to be fitted, often itself a function of the form's height. If a control is added or deleted the others are automatically re-positioned to take account of the change.

The solution developed documents all parameters associated with the forms. These include, for each form:

- The name
- The title
- The menu items available
- The controls on the form and their positioning.

The information stored for each control varies according to the type of control and is a sub-set of the following:

- Type (list follows)
- Caption or field name
- Size
- Position
- Tool tip
- Template (i.e. allowable data formats).

The controls used are all standard VB controls, but several formatting variations may be used. The control types used include:

- A normal right-justified 8 point label
- A normal left-justified 8 point label
- A bold left-justified 8 point label
- A normal right-justified 12 point label
- A normal left-justified 12 point label
- A bold left-justified 12 point label
- Text box
- Read-only text box
- List box
- Combo box (combo box entries may be limited to those appearing in the list, or not.)
- Check box
- Read-only check box
- Editable rich text box
- Non-editable rich text box
- Button
- Microsoft flexgrid.

Controls may be specified to be loaded with literal data (defined in the spreadsheet) or from the database.

2.6 EXTRACTION OF METADATA

An Excel VBA program scans the form definition spreadsheet and builds a set of tables to allow the application to dynamically generate the forms, with controls, as required. The form definition data is stored in the same SQL database as the application's data.

The VBA program also generates a list of all the property fields used to define the field IDs in the database.

2.7 USE OF METADATA.

A module within the application reconfigures a single form according to its description retrieved from the database. The user perceives this form as being one of some 39 forms.

Navigation between these perceived forms is via the form's menu bar. The normal menu keyboard short cuts are also available to the user.

Menu items are pre-configured but may be hidden or shown under instruction from the form description, allowing a context-controlled menu bar.

When a form image is generated, the application retrieves the parameters required for the fields on that form, including references to the field names, where appropriate. Field data is accessed from the database via a cache, held within the application. The cache implements the undo and re-do feature and ensures that only modified data is written back to the database.

2.8 DOES IT WORK?

The tools used are shown in figure 1.

Does the approach work? Yes, it does.

It was gratifying to see changes reflected automatically across the entire application. This allowed the designer to respond quickly to the needs of the user and readily demonstrate the effects of proposed changes.

Use of the flexidata approach encouraged a uniformity which flows through to the appearance of the final forms.

For example, when it was decided that all rich text fields should be implemented so that when double-clicked an editing window would open as soon as the change was made, all rich text boxes behaved identically.

Quite often, it was possible to make substantial changes to a form by updating its metadata, aided by the use of calculated values within the spreadsheet, and to see the results within a couple of minutes.

The names for the Visual Basic controls were initially generated from labels (or captions) in the metadata, as were the database field names. This allowed the reconciliation of field names between forms and eased data migration from the old database. It has also been much more fun developing software this way: developing tools is more enjoyable than performing repetitive tasks.

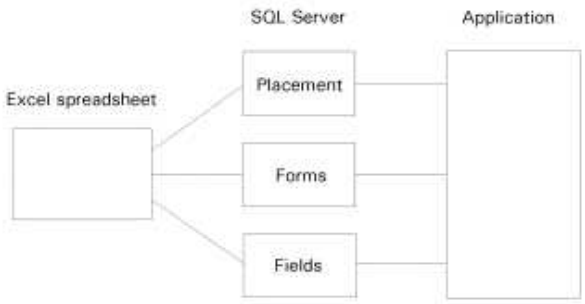

Figure 1: The tools used in this flexidata approach

2.9 CAN IT BE IMPROVED?

The format of the metadata, as stored by the database, could be enhanced to support the dynamic re-sizing of forms. In this case, a decision was taken to restrict the number of forms used at the expense of placing a large number of controls on each form and mandating the use of full-screen forms at 1024 x 768 resolution on a 17" screen.

A "macro" facility would be useful within the spreadsheet to allow commonly occurring configurations of VB controls to be defined once only, then re-used as required. The present spreadsheet achieved the same effect by using formulae and cut-and-paste techniques.

3. DEFERRING POST-IMPLEMENTATION DECISIONS

So far, we have seen how flexidata can be used to regenerate design elements during system development and thus reduce the impact of volatile user requirements. In the remainder of this paper, we examine some of the ways in which flexidata can be incorporated into a completed system to make it more versatile.

3.1 A SIMPLE EXAMPLE

A simple example of this is to allow the user to change the way in which data elements are described. For example, in a system that maintains contact details, a contact can belong to any of five different categories. On the database, these are named Category1 through to Category5. Users provide their own descriptions of these categories and the system uses the descriptions, not the field names, when displaying the contents. Figures 2 and 3 show how the category names might be set up by two different users of the same software.

3.2 A SOPHISTICATED EXAMPLE

A more sophisticated example of making metadata available to the users without requiring them to master a full-blown data definition language is provided by COLLECTION, the PC-based museum package developed by Vernon Systems Limited in Auckland.

Each of the system's main tables includes twenty pre-loaded fields of varying types which the user can then name and maintain as needs arise.

In addition, a field name override facility allows any field in the system to have its displayed description changed by the user.

Twenty symbolic fields are provided whose contents are calculated as the result of an expression and displayed on a report. This allows, for example, an age to be calculated as the difference between the current date and a birth date.

3.3 TRYING FOR EVEN GREATER FLEXIBILITY

While the features in COLLECTION described above are sophisticated, they do not allow the number of available fields to be altered by the user. Rather, they allow existing fields to be variably described and new results to be reported that are calculated from previously stored values.

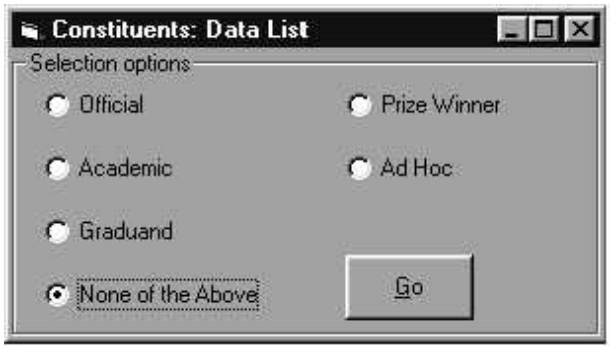

Figure 2: User-defined field descriptions — category descriptions for a graduation committee

Figure 3: User-defined field descriptions — the same categories as described by an events promoter

Is it possible to design systems in such a way that the user can change some of the basic design elements without the intervention of a software developer?

The answer depends on the toolset available, but even if it is "yes", one should not assume that doing so would be a good idea. At its limit, the strategy of allowing users to modify their own metadata results in developing DBMS and CASE tool features for every installed system, features that are triggered by the user but which act automatically to modify the system's data structures and generate the necessary maintenance and reporting routines.

Sensible decisions will always involve a tradeoff between the flexibility required by the user and the extent to which sophisticated software should be provided to cater for possible future extensions. At a certain point, it becomes cheaper and easier to wait until the demand for a new feature arises and then ask the developers to prepare a new version of the software. Totally future-proof systems, even if they are possible, are not likely to be good value for money.

A new case study developed for third-year degree students at Massey University requires them to make just such a tradeoff. The scenario is coincidentally similar to the COLLECTION system mentioned above. In a museum collections management system, the user would like the ability to define and implement new sub-collections without having to commission a new version of the software each time.

The problems the student must address include:

- What data types are available and how are they selected? This is very dependent on the toolset being used. The programming language has its own set of recognised data types, the database may have another, and the technique used to create new data structures from within the program code, such as embedded SQL commands, may cater for some of these but not others.
- How will fields in a new sub-catalogue be defined and allocated? Presumably, the same field may occur in more than one sub-catalogue, so field definitions should be reusable.
- How will the system validate the name of each new sub-catalogue? Names must be unique, so the system must have some way of knowing whether a name has been used already.
- Once a new data structure has been created, how will its contents be maintained? This is probably the most challenging issue and is sure to influence the answer to the next question, which is:
- What limits, if any, will be placed on how many sub-catalogues can be generated and how many fields each one can contain?
- How will the menu structure accommodate the new software routines associated with each new sub-catalogue?

A model answer has been developed to help anticipate the extent to which students can be expected to solve these problems. A toolset of Visual Basic and Access was used.

Writing the software for adding new tables dynamically proved relatively straightforward. Because SQL statements can be embedded in Visual Basic code, changes to the data structure can be made in response to user preferences.

Some difficulties were encountered, however. Firstly, the data environment did not seem to tolerate variable table names in a CREATE TABLE statement, so the SQL commands had to be embedded in the code rather than stored in the environment. Secondly, the INTEGER keyword produced a **LongInt** field in the database instead of an **Integer**, and thus had the same effect as LONG. Thirdly, although a length could be specified for a TEXT field, neither length nor precision could be specified for numeric fields.

Fig. 4: The user-definition of a new sub-catalogue

Figure 4 shows how the user assigns fields to a new sub-catalogue. The fields themselves are defined by the user too, but separately to allow for reuse.

Providing customised tables on demand is not much use, however, unless they are accompanied by software for maintaining data on them, and this is likely to be the most challenging part of the job.

Visual Basic provides two techniques for dynamically adding and removing controls at runtime: using control arrays and using the methods of a form's Controls collection (Bradley and Millspaugh 2001, ch. 2). The second of these is more elegant because new controls can be created *ex nihilo*, as it were, without having to include a hidden template for each type of control that is added. This does not work for forms or menus, however. They need a template from which new instances are cloned as required, each identified by its own index value.

These techniques allow the feature of user-defined sub- catalogues to be coded tolerably easily, but only as long as the maintenance requirements of each can be inferred from the metadata of its table. If this could not be relied upon, hard-coded exceptions would be needed and the benefits of flexidata would begin to erode.

4. CONCLUSIONS

We have considered two ways in which metadata can be used as though it is operational data and have termed this "flexidata".

The pre-implementation use allows form layouts to be regenerated each time a change is requested by the design team, and its benefits are uniformity and speed. Desirable extensions of this approach were identified, but it suffers from no inherent disadvantages.

The post-implementation use provides the end user with flexibility as its big advantage, but the method is inherently limited. Maintenance forms, although they are generated automatically, have to follow the same pattern of layout and controls. To avoid the feature becoming a patchwork of special cases, no special processing features are provided for particular sub-collections. Also, the data types available are limited to those common to the programming language and the database management system.

ACKNOWLEDGMENTS

We would like to thank our colleagues in the Information Systems department of Massey University at Wellington, especially Errol Thompson and Tony Powell, for their help in exploring the dynamic creation of menus, forms, and controls in Visual Basic.

REFERENCES

Bradley, Julia Chase and Millspaugh, Anita C. (2001). *Advanced Programming in Visual Basic 6.0*, McGraw-Hill.