# Building user interfaces for database applications: The $O_2$ experience

P. Borras, J.C. Mamou, D. Plateau, B. Poyet and D. Tallot
$O_2$ Technology
BP 105, 78153 Le Chesnay Cedex
France

## 1 User interfaces for database applications

There is more to database application user interface than building a main window with a menu bar at the top and a push buttons column on the left. Users want to display, edit, browse, and manipulate database *objects* in a variety of ways. These objects can be large, multimedia, and complex.

The interactive manipulation of a database object is a three step process: *(i)* displaying: read the state of the object and build a graphical presentation of it; *(ii)* modifying: use cut/copy/paste, editing functions as well as application functions to modify the state of the object; *(iii)* saving: write the new state of the object back into the database.

To support user interface development, two kinds of tools are commercially available: *toolkits* and *user interface editors*. A toolkit consists of an extensible set of predefined graphical components or *widgets* such as menus, buttons, scrollbars, and text editors. Almost anything can be done with a graphical toolkit, but its usage requires an extensive learning of the system and a significant programming effort.

A user interface (UI) editor is dedicated to a particular toolkit. It allows to specify graphically a user interface. For instance, one uses the mouse to pick a widget (a menu, a scrollbar, ...), to define its position on the screen as well as its graphical attributes (color, fonts, ...). When the specification is over, the UI editor automatically generates the toolkit code that produces this user interface.

The programmer needs neither to learn the toolkit nor to write any code. User interface development is thus significantly speeded up. Unfortunately, there are many things that cannot be specified with a mouse. A programming language and direct, hard toolkit coding are often needed to do the job.

In the rest of this paper, we describe two systems. The first system, $O_2$Look [Altaïr 1990, $O_2$ 91, Plateau 90] is a graphical toolkit. Unlike traditional graphical toolkits, $O_2$Look does not provide functions to manipulate widgets but database objects. The programmer does not create a scrollbar but a presentation for a city (see Figure 1). Given an object, $O_2$Look queries the database to know about this object type and state, automatically combines and initializes widgets to represent the object on the screen. When the screen presentation is modified, it checks with the database that the modification is correct and then writes the new state in the database. This is possible because $O_2$Look is fully integrated with the database.

It should be clearly understood that $O_2$Look is not better than Motif but an extension of Motif to meet database requirements. On one hand, displaying a complex object with $O_2$Look is a simple function call when it would take thousands of lines with Motif and when it would be impossible with a user interface editor. On the other hand, $O_2$Look is not the appropriate tool to build the

user interface of a mailing system.

The second release of $O_2$Look was available in 1989 as a prototype while the third release was launched as a product in 1990.

The second system, ToonMaker [Mamou 91], a prototype, is a user interface editor. It complements $O_2$Look in the same way a traditional user interface editor complements a traditional toolkit. With ToonMaker, one can graphically specify how an object should be displayed or what modifications are allowed. The corresponding $O_2$Look code is then generated. Unlike traditional user interface editors, ToonMaker does not simply generate the code to assemble widgets but also the code which reads the object state in the database and initializes the widgets, the code which manages cut/copy/paste operations and the code which writes the state into the database. Problems here do not lie in the combination of widgets but in the definition of the mapping between the database object and the graphic widgets.

Sections 2 describes $O_2$Look while section 3 focuses on ToonMaker. The last section summarizes the lessons that we learned from these two experiences and outlines some promising directions for future experimentations.

## 2 $O_2$Look, the user interface generator

The $O_2$Look user interface generator is a Motif toolkit extension designed to meet the $O_2$ programmer requirements in terms of end-user interaction. $O_2$Look provides the programmer with two main capabilities:

*(i)* It allows the programmer to create graphical presentations of the application objects. No matter how complex, large or multimedia the objects are, $O_2$Look can build a default presentation which the end-user can cut/copy/paste to modify the database.

*(ii)* It provides functions to customize these default presentations to match the requirements of specific applications. This is usually not supported in traditional database programming environments.

$O_2$Look provides functions to manipulate *presentations* of $O_2$ objects. It is implemented in C++ on top of Motif and the X Window system. Its major features are:

### Ready-Made presentations

No matter the complexity of the objects, $O_2$Look can build a ready-made presentation of it that can be edited and saved into the database. Ready-Made presentations are built from widgets. For instance, the tuple widget is used to display a tuple and the integer widget is used to display an integer. Widgets are nested so that composition links between objects can be expressed.

Figure 1 shows ready-made presentations for the Roma City and the Ritter Hotel objects.

### Cut/copy/paste editing

Presentations
can be edited. The cut/copy/paste operations play an important role in presentation editing. For example, the address of the Ritter hotel has a City field that could be filled by copying the Roma object and pasting it in the City field of the Ritter address.

To avoid inconsistencies that could occur if one attempts to paste a country in the City field, dynamic type checking is automatically performed by $O_2$Look. Cut/copy/paste operations are implemented by widgets.

### Interactive method activation

Every presentation displaying an object has a menu: each menu item corresponds to a method of the object which is activated when selecting the item. If the method has arguments, they are interactively collected by $O_2$Look. At any time, the application can disable a method in a menu, in which case the corresponding item cannot be selected.

### Layout management

The position of a presentation can be defined by an application. It can be specified explicitly (providing the coordinates on the screen), by the end-user (moving a ghost with

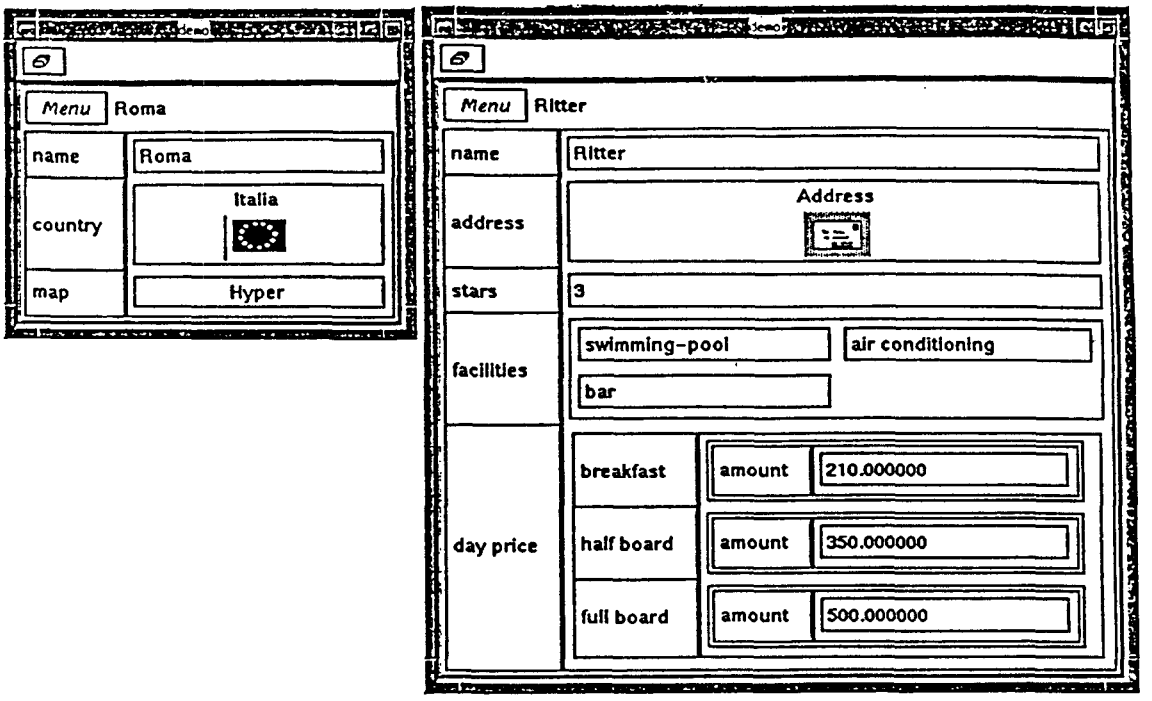


Figure 1: Ready-Made presentations of City and Hotel objects

the mouse), or relatively to another presentation (tiling, stacking). In the latter case, the position link between the presentations can be persistent: if so, the link is maintained after any move, resize or iconify operations.

### Screen consistency

The presentations on the screen constitute a view of the database. This view may temporarily be inconsistent with the database. $O_2$Look provides functions to enforce consistency either by forcing groups of presentations to be saved in the database or by refreshing groups of presentations based on the database contents.

### Masks and resources

Masks and resources are used to customize ready-made presentations. First, each widget has graphical resources such as colors, fonts and editing resources. An editing resource is a $O_2$C function to be triggered when/before/after a user action occurs.

For instance, each time a facility is added or removed from the set of facilities of a hotel, the editing resource is a function that recomputes the price of the rooms. Resources can be statically defined in resource files or dynamically changed. In the latter case, a mask is used to match resources and widgets.

Second, several widgets might be available to display a given object. For instance, a list of strings can be displayed with the list and string widgets (the ready-made widgets) or with the text widget. Masks are used to choose the widgets.

Last, an object can be viewed either as a reference (an icon object) or as the data that it contains (an opened object). For instance, on Figure 1, the "Roma" object is opened while the Italia sub-object is an icon. Masks are used to force an object to be opened.

## Extensibility

$O_2$Look provides a set of predefined widgets. An $O_2$Look widget is very similar to a standard toolkit widget such as Motif widgets except that:

*(i)* it knows, when created, how to read the

objects in the database. This is mandatory to make the widget usable.

*(ii)* it supports cut/copy/paste operations. This is an option since some objects are not to be modified.

*(iii)* it knows how to write the objects into the database. If the object can be modified by the end-user, the state of the object has to be modified also in the database. If the object cannot be modified, the write feature is optional.

New widgets can be added to $O_2$Look at any time. As mentioned above, the mask will be used to refer to the new widget. From a programmer point view, an $O_2$Look widget is a C++ class.

Figure 2 shows a specific widget dealing with bitmaps. Notice that a bitmap is stored in the database as a tuple with three attributes: width, height and bitmap.

The $O_2$Look system has been used by many programmers. They enjoyed $O_2$Look for its ready-made, quick to build and easy to use presentations. However, it was stressed that the presentation customization based on masks and resources should be improved. This is why we started the development of ToonMaker, which is described in the next section.

# 3 Interactive customization of database presentations with ToonMaker

ToonMaker [Mamou 91] addresses the problem of the mapping between a database object and a graphical structure. When $O_2$Look maps an object into an isomorphic widget structure, ToonMaker allows a programmer to change interactively and incrementally this ready-made mapping. During this process, tree structures (detailed below) are built by ToonMaker to memorize the correspondence between a widget tree and a part of an object. This memory makes ToonMaker able to preserve the read and write capabilities of $O_2$Look widgets.

A ToonMaker session involves modifying a ready-made presentation of an object in a particular class, to create a presentation template that can be reused at will by a programmer to display any instance of the class. A programmer can customize a presentation by performing any of the following:

- restricting the information shown in the presentation,
- adding adornments (e.g., labels),
- choosing graphical resources (for instance, color and fonts),
- defining triggers or callbacks on end-user events,
- specifying the graphical structure of the presentation.

These features appear to the user interface designer as part of an editor and can be used to edit presentations by direct manipulation (see Figure 3). The starting point for this editing process is a ready-made presentation that can be changed at will.

When using the editor, the screen is divided into five areas. At the top of the screen is the menu bar that offers some classical operations e.g., save or load and some more specific ones on which we will elaborate later on. Just below are two windows. Each one contains a tree. The tree on the left represents the type structure of the object (**the object tree**) and the one on the right focuses on the graphical widget construction of the selected node of the object tree (**the toon tree**). At the bottom is the current presentation, which changes as the editing goes forward. Finally, at the left side, is a column of graphical constructors (column, row, button, text, etc.) that will be used during the editing process.

The customization process consists of building and modifying a toon tree associated with each node of the object tree, using the available graphical constructors called **Toons**. For each toon of a toon tree, the designer can display a resources sheet in which the graphical resources of the toon can be modified. In the same form, the designer

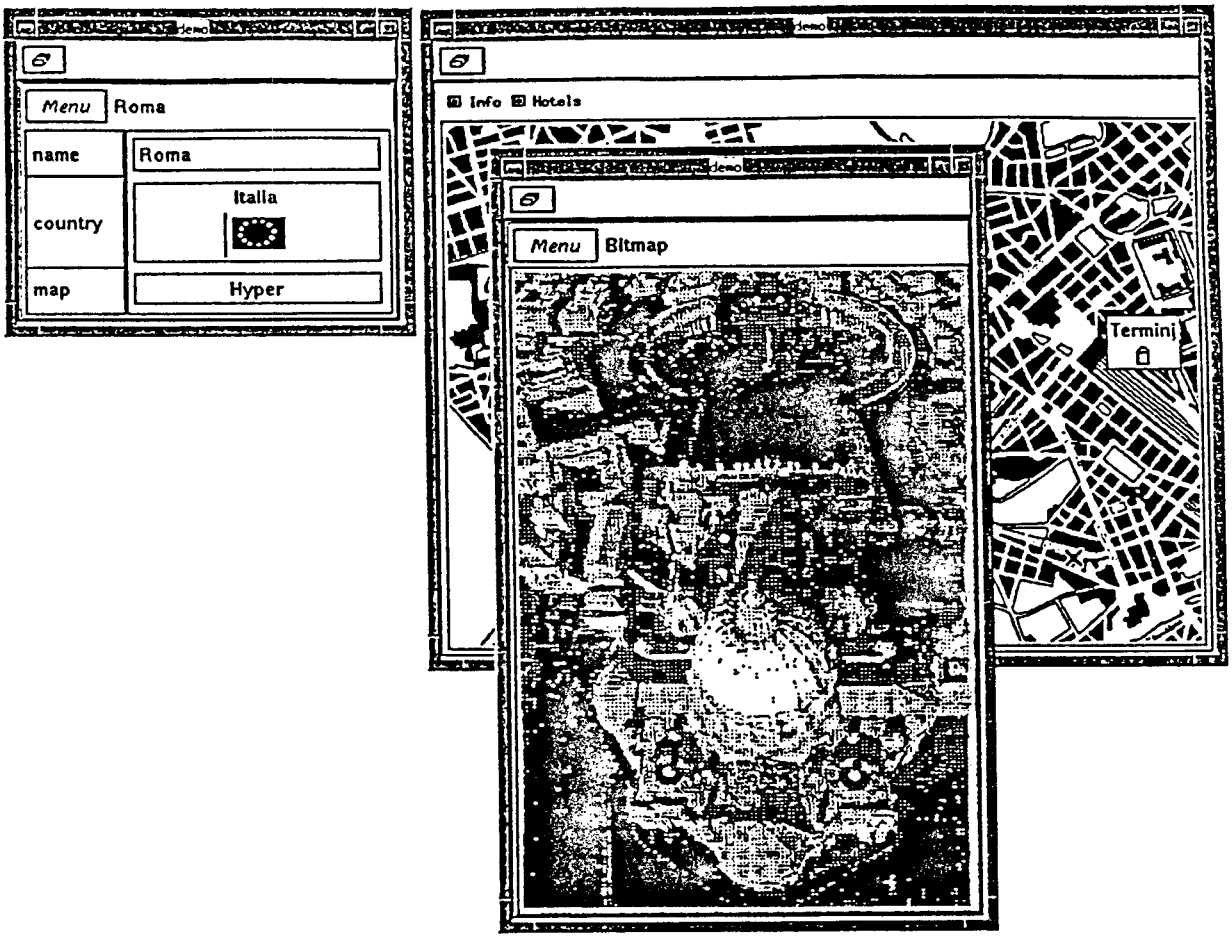


Figure 2: Roma, its map and Santo Pietro Basilic

can also define triggers. A trigger is an $O_2C$ method that will be triggered in response to an event in a toon (i.e. click on a ToonButton, insertion in a ToonColumn, etc.).

Figure 4 shows the object edited in Figure 3, except that Figure 3 shows the beginning of the process (the ready-made presentation), whereas Figure 4 presents the object after it has been edited.

ToonMaker complements $O_2$Look the same way a user interface editor complements a toolkit: ease of use, interactivity, and productivity increase. The major benefit of this experience is the integration between a graphical toolkit dedicated to the manipulation of database object and a user interface editor.

# 4 Lessons and perspectives

The development of $O_2$Look started in 1986. $O_2$Look has now reached commercial quality. Over 100 users have tested the system, written applications and given feedback on its good and bad points. ToonMaker was developed between 1988 and 1991 but is still in a prototype state.

Below, we present the main lessons from these 2 experiences:

### Toolkits evaluation

$O_2$Look and ToonMaker are built on top of toolkits. In both cases, the toolkits have been useful but not completely adequate. There are two main features missing. First, the widgets protocol does not support cut/copy/paste operations. For in-

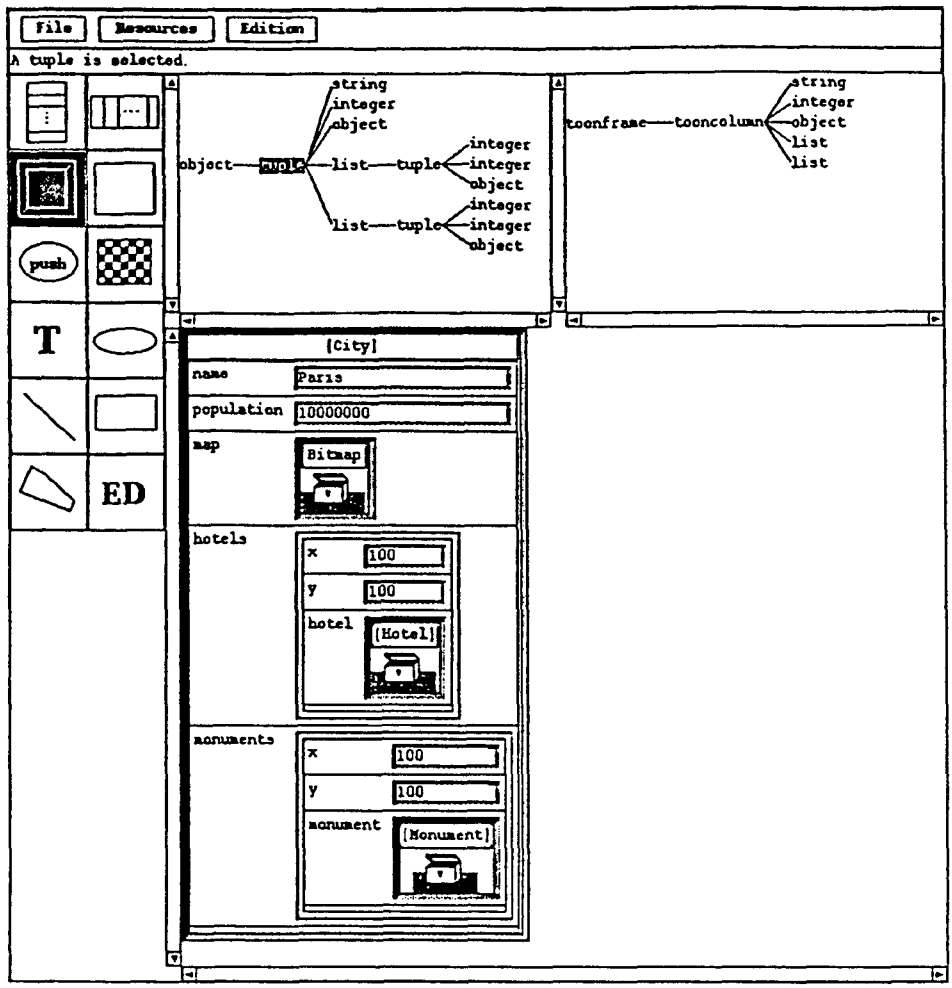


Figure 3: The interactive presentation editor

stance, with X / Motif widgets, widgets cannot be reparented dynamically. Secondly, there are no widgets to graphically represent data and data structures. Graph, table or chart widgets are badly missing.

An interesting work for the future is to define a widgets set that supports the interactive manipulation of data structures (for instance C++ data structures).

### User Interface Editors

User interface editors hide the complexity of toolkit programming. However, when developing software to display database information, this advantage is lost because most of the code has to be written at the toolkit level. Therefore, toolkit expertise is required with its high learning and programming costs.

### Mapping data structures and widgets

$O_2$Look and ToonMaker help programmers to define the mapping between the database objects with the widgets. The mapping has to be powerful enough to work both ways (objects to widgets and widgets to objects) and incrementally, i.e one should be able to address part of the data or part of the widgets for selection, refresh or update operations. When building tools that help in defining the mapping, the important trade-off is between power and usability. It is easy to map an object to an isomorphic widget structure. This is what $O_2$Look does with ready-made presentations. Mapping completely different structures is still possible but the complexity of the process makes the tool unusable.

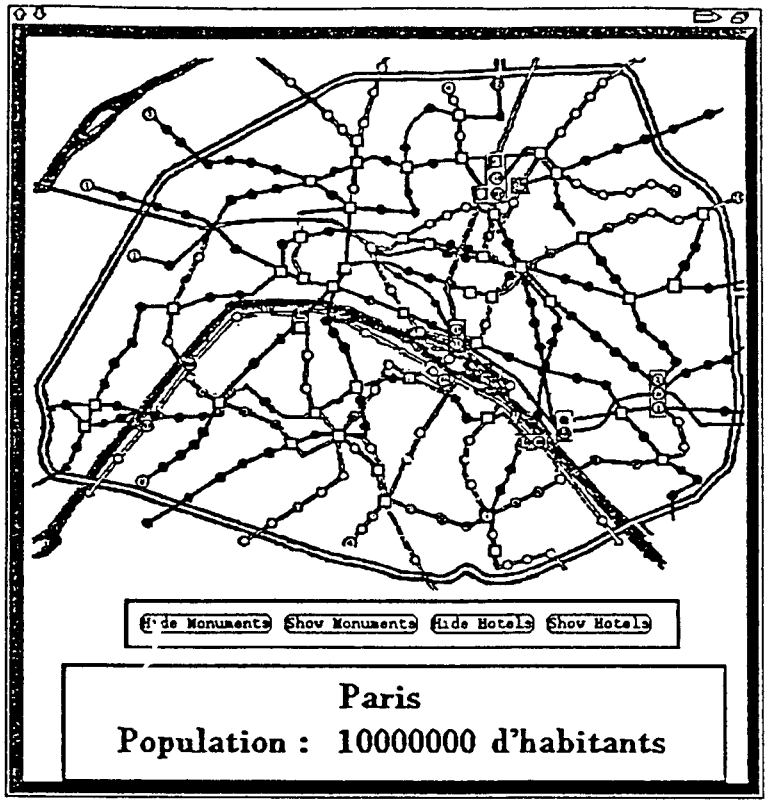


Figure 4: The edited presentation

**Presentation versus widgets programming**

Toolkits are libraries for manipulating widgets. Systems like $O_2$Look and ToonMaker are useful for handling presentations. A presentation is basically an editor: it shows an object, allows to modify it interactively and to save it into the database. A presentation is using many widgets and therefore provides a higher level of abstraction. The advantage is that programming is simpler and faster. The disadvantage is that it forces a style of interaction that does not fit all the applications. Furthermore, the application looses the control on the widget level.

Mixing the two programming abstractions to get the benefits of both is an interesting issue for future investigation.

## References

[Altaïr 1990] "The story of $O_2$", *IEEE Transactions on Knowledge and Data Engineering, Vol. 2, No. 1, March 1990*

[$O_2$ 91] "The $O_2$ system", *Communications of ACM, October 1991*

[Plateau 90] D. Plateau, P Borras, D. Lévêque, J.C. Mamou, B. Poyet and D. Tallot. "Building User Interface with the $O_2$Look Hyper-Object System", *EuroGraphics on Object Oriented Graphics, Koenigwinter 1990.*

[Mamou 91] J.C. Mamou. "Du Disque à l'Ecran : Génération d'Interfaces Homme-Machine pour Objets Persistants", *PhD thesis, May 1991.*