A Brick Construction Game Model for Creating Graphical User Interfaces: The Ubit Toolkit

Eric Lecolinet

Ecole Nationale Supérieure des Télécommunications & CNRS URA 820
Dept. INFRES, 46 rue Barrault, 75013 Paris, France
EMail: *elc@enst.fr*
URL: *http://www.enst.fr/-elc*

This paper presents "Ubit", a new graphical toolkit that is based on the "brick construction game" model. This approach makes it possible to create sophisticated application-specific components by combining simple "basic bricks". All bricks can be shared in order to simplify GUI control and to reduce memory cost. This model supports the the concept of ubiquitous GUI components that are inherently able to display several representations of their content on the screen. At last, Ubit provides a simple and flexible C++ API that makes it possible to specify GUIs in a pseudo-declarative style.

Keywords: User interface software, graphical toolkits, declarative GUI language, brick object model, hyperdocument, interaction control, ubiquitous components, multiple views.

1 INTRODUCTION

It is a well known fact that user interfaces are not only hard to design but are also hard to implement (Myers, 1995). As a consequence, most people prefer using tools (such as interactive interface builders or other kinds of user interface management systems) rather than programming directly with a GUI toolkit. Thus, in recent years, attention has rather been focused on tools than on the principles of GUI toolkit design. We believe we now need to reconsider the underlying ideas that are at the basis of the implementation of graphical user interfaces. There are several reasons for that:

1. UIMS are very useful tools for creating "static" GUIs that are mainly made of forms, menus and dialog boxes. But they generally provide rather limited help for creating application-specific components that evolve dynamically at run time (for instance a graph editor where nodes are application-specific objects that can change dynamically.)

2. New interaction and visualization concepts such as zoomable interfaces, magic lenses and other information visualization techniques are now coming to maturity. But such techniques are generally out of the scope of current tools. Moreover, most tools do not even fully support the implementation of classical but highly interactive GUIs that make an extensive use of direct manipulation techniques.

It may seem reasonable to think that the implementation of sophisticated GUIs will always require a certain amount of textual programming at the toolkit level (although some interesting research has been performed on interactive model-based UIMS able to produce advanced GUIs (Szekely et al., 1993)). Most tools just do not provide the appropriate level of abstraction for dealing with loops, dynamic creation and deletion of objects, highly interactive behaviors, etc. Textual programming may just be more appropriate in such cases.

Unfortunately, programming at the toolkit level is often quite a difficult task that is reserved to experienced programmers. GUI toolkits are generally quite complex and hard to use. As a result, GUI source code tends to be verbose and cumbersome. Besides, creating application-specific components may be a non trivial task as toolkit design often makes it difficult to deeply customize the set of standard components.

This paper presents a new GUI toolkit, called **Ubit** (for "Ubiquitous Brick Interaction Toolkit") that is based on the "brick construction game" model. In this model, GUI objects are simple **basic bricks** that can easily be combined together. This model makes it possible to create highly customized components in a simple way by combining (or deriving) these bricks.

Besides, it also improves GUI source code legibility and simplifies GUI control. Ubit provides a flexible C++ API that supports a generic *adding mechanism*. This feature favors code compactness and makes GUI source code resemble mark-up language text. The brick model also simplifies GUI control thanks to a generalized *sharing mechanism*. This design feature ensure the implicit control of multiple views coherency. Ubit also supports the new concept of **ubiquitous** GUI components: in this model any user interface object is inherently able to manage an arbitrary number of synchronized representations of its data on the screen.

The next sections will describe and compare classical toolkit architectures, introduce the conceptual principles of Ubit model and describe its properties. The last sections will then detail some implementation aspects and present the current status of the system and future work.

2 GUI ARCHITECTURES

Many graphical toolkits consist in a set of "fat objects" that implement quite a large variety of functions. As a consequence, GUI objects are often hard to learn and to use because they handle so many different different aspects. Moreover, objects attributes often inadequately fit the real needs of a given application. Paradoxically, most objects attributes are useless for most applications, while GUI objects often lack "this specific feature" that would be useful for a given application. For instance, the push button widget of the Motif 1.2 toolkit has more than 60 different "resources", most of them being rarely used. However, in its standard version, this object can *either* display a character string *or* a pixmap image, but can not display both simultaneously.

This points highlights an important conception problem: there is no possible "best choice" in defining object properties while conceiving a GUI toolkit because applications are simultaneously so similar and so different. They are similar because they use standardized GUI components that should remain similar from one application to another in order to preserve a consistent look and feel. But application are also very different because they deal with specific domains and thus require customized representations and interaction styles. Trying to anticipate all possible GUI component uses is an impossible task. As a consequence, we believe GUI objects should be dynamically adaptable instead of providing a static set of predefined features that can not fit all application needs. Besides, for the sake of simplicity and memory efficiency, standard objects should not systematically provide a large number of generally useless features. This last point should not be neglected as applications may require quite a large number of GUI components.

The "fat object" architecture also tends in multiplying similar object classes. For instance, the Java AWT toolkit introduces two different push button classes (depending on whether buttons are located inside or outside menus), while the Motif toolkit even defines six different classes (including "widget" and "gadget" variants). This problem clearly shows the limitations of such architectures: fat objects are so complex that they are difficult to derive or to combine together in order to fit application needs. Thus, many objects variants must be provided in order to compensate for the lack of flexibility of standard components.

This lack of generic design leads to many ambiguities and increases the toolkit complexity. Besides, it makes iterative development quite laborious as small UI design refinements may lead to heavy programming changes. Some of these aspects have already been addressed in previous research. MVC based (or derived) systems, such as the Java Swing toolkit (Fowler) provide more flexible object sets that are easier to customize and to extend. However, these systems generally lead to an increased level of complexity as they require extended knowledge about various components architecture and how they should precisely interact.

3 THE "UBIT" BRICK MODEL

The Ubit toolkit proposes a new approach that is based on the concept of **generic basic bricks**. Basic bricks are specialized C++ objects that only implement and control one specific functionality. These bricks can easily be combined together through a standard and dynamical *adding mechanism*. The model is recursive so that composite objects resulting from brick combinations are brick themselves that can be further combined with other bricks.

Brick combination is always performed in a standardized way through a generic interface: the **Box** object. The Box brick acts as a general container that can contain any possible brick combination, including other boxes. The idea of using container objects is not new: this principle was for instance used by most X-Window (or derived) toolkits (e.g. Motif, Athena, OpenLook, InterViews, Tcl/Tk, Java AWT...) With such systems, GUIs are made by creating instance trees where container objects are able to layout and display their children in an appropriate way. Containers and interactors are generally not dealt with in an homogeneous way: most toolkits do not allow interactive components (such as buttons, text fields, lists, etc.) to contain other interactor objects. However, certain systems offer extended containment capabilities

(for instance: Gtk (Mattis, 1998), Fresco (Linton, 1994), Self/Morphic (Maloney, 1995)). Applications constructed with Morphic are composite "morphs" whose submorphs can handle user input events. The Fresco toolkit is based on an advanced composition mechanism that allows the mixture of user interface components and structured graphics objects. Fresco objects derive from a primitive type called "glyph". While UI components are organized in a strict hierarchy, primitive glyphs may be shared. Fresco thus supports the redisplay of a direct acyclic graph (DAG) of objects. At last, object composition is also related to the field of Visual Programming (Glinert 1986). Some VP systems include a special-purpose toolkit layer based on a component combination metaphor (Esteban 1995).

The **Ubit** toolkit is based on a drastic generalization of the containment principle. In this model, a Box is not seen as a window object that can display child widgets, but rather as a generic interface that let other bricks cooperate. By opposition with classical systems boxes do not have graphical properties of their own. Instead, it will be up to the programmer to add all the necessary "ingredients" to boxes in order to obtain the appropriate effect. A Box brick basically is a "pure container" which role is to contain children of various types. Depending on their respective classes, these children will dynamically change the appearance and the behavior of their Box parent. This design principle is related to the Design Pattern concepts of "Container" and "Decorator" objects.

Character strings, images, pixmaps, decorations, borders and graphical symbols are first class objects (called **Item** bricks) that can be Box children. A Box can thus contain any combination of Items, manage their layout and display them on the screen. This makes it possible to produce a wide range of customized components by combining a limited set of basic objects (Fig. 1). For instance, a push button or any other interactor could contain an arbitrary combination of pixmap images, character strings and various symbols (such as arrows, checking indicators and other special marks.) Decorations (the borders, shadows, etc.) could also be dynamically specified in the same way. This may be seen as an elegant solution to the "no possible best choice" dilemma that was presented in the previous section, as object attributes are actually chosen by toolkit users, not by toolkit designers.

This flexibility is a direct consequence of the reification of all GUI objects. For instance, the Box container does not need to know how to display any specific child: this service is always provided by the child itself, even if it is a low-level GUI object. Besides, the layout capabilities of the Box container also depends on a separate "Layout" brick that can be dynamically changed. As said before, a Box is indeed a pure container whose appearance and behavior depends on more specialized bricks.

The same principle applies for specifying graphical properties such as background and foreground colors, character fonts, background tiling and so on. Graphical properties are not predefined Box attributes. Instead the toolkit provides an extensible set of **Property** bricks that can be dynamically added to Box interactors in the same way as other components.

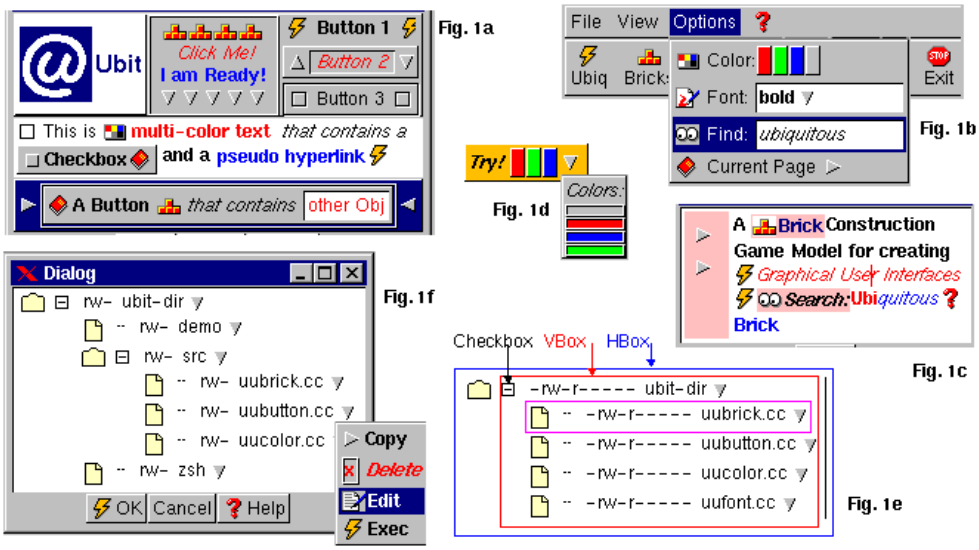

Figure 1: GUI object composition

3.1 Ubit Metaclasses

Box children can derive from five distinctive metaclasses: other Boxes, viewable Items, graphical Properties, Callback bricks and State objects. **Callback** bricks specify that a given call-back function or object method will be invoked when a certain condition is verified on the containing Box (typically, when a certain event occurs on this Box, although more sophisticated conditions may also be specified.) **State** bricks makes it possible to change to the graphical state of the Box or to modify its behavior. Quite an important point is that *child order* matters when meaningful. A Box child list could for instance contain a callback (brick), a pixmap image, a red color (brick), a first character string, a bold font, a blue color, a second string and an arrow Symbol. As a result, the containing box would display all viewable elements in sequence, the two strings being drawn by using different colors and fonts (Fig. 1a). This box would be sensitive and a given function would be called according to the callback brick specification.

The combination of object reification, list ordering and extended composition capabilities make it possible to create multifont and multicolor text in an easy way. Besides, it also allows for mixing up text and other GUI components and make them appear in a sequence. This ability of sequencing text, images and interactors and parameterizing their graphical attributes is somewhat similar to the capabilities of hypertextual languages. This feature makes it easy to create GUIs that resemble hyperdocuments (fig. 2a) and that follow the same logic (especially for what concerns lay out management and the propagation of graphical properties among components.) Besides, GUI source code will resemble mark-up language text as will be shown in the next section. Ubit thus proposes an unified framework that attempts to bring together classical GUIs and hyperdocuments.

3.2 Styles and Gadgets

For the sake of simplicity, Ubit also offers a set of predefined **Gadgets** that simulate the usual “widgets” or “controls” that can be found in other toolkits (e.g. buttons, menus, text fields, dialog boxes...). Most Ubit gadgets are mere derivations of the Box brick. They are just provided for convenience and mostly act as “shortcuts”.

Default Gadget appearance is determined by **Style** bricks. By construction, each Gadget class is associated to (at least) one Style object. A Style object can be seen as a collection of default Properties. Style and customized specifications are cascaded so that Box Property children can override default Style settings. Styles provide a convenient way for parameterizing the default appearance of GUI components. This ability of configuring the “look” of the GUI is somewhat similar to the notion of “plugable look and feel” developed in the Java Swing toolkit.

Gadget instances do not store graphical attribute in a static way but point to specialized objects (the Style and Property bricks) when needed. This model is very efficient in terms of memory management as objects do not maintain useless data and share most physical resources. Moreover, this specification mechanism works in a dynamical way and does not require to create new objects classes. Colors, fonts and other graphical attributes are determined at display time by scanning the Box child list and propagating them along the Box hierarchy.

This model is coherent with the fact that most GUI components are highly standardized, but that very specific objects are also often required. Moreover, such components often quite differ one from another. Thus, the ability of customizing objects without having to create new classes is a feature that is especially well suited for GUI design. This point was formerly addressed in some research systems through the concept of prototype-instance object systems such as Amulet (Myers 1995).

4 THE PSEUDO-DECLARATIVE API

Ubit provides two compatible C++ APIs. The first one is a classical object-oriented API: object are created by invoking the “new” primitive and are added to parents by using their “add” method. One could for instance write the following source code to create a push button that contains a pixmap and a character string:

```
UButton *b = new UButton();
UPix *p = new UPix("my_image.xpm");
UStr *s = new UStr("Click Me!");
b->add(p);
b->add(s);
```

This first API presents two remarkable points: object creation does not depend on parents (this will avoid useless a priori dependencies in the code) and objects can possibly be added to *several* parents. This important design principle will be detailed in section 5. The second API makes it possible to specify GUIs in a **pseudo-declarative** style. It is based on a generic *adding mechanism* that favors code legibility and compactness:

```
ubutton(upix("my_image.xpm") + "Click Me!");
```

This simplified API only makes use of standard C++ features:

1. ubutton() and upix() are just intermediate functions

that call the "new" primitive with the appropriate class,
2. The "Click Me!" character string is implicitly converted into a UStr brick through the C++ standard conversion mechanism,
3. The "+" adding operator has been overloaded in order to create brick lists in a convenient way.

Any brick can be added to a Box Gadget in a similar way:

```
utext( ucallback(foo, UState::action)
       + upix("ubit.xpm")
       + UColor::red
       + "Click Me!"
       + UFont::bold + UColor::blue
       + "I am Ready!"
       + ubutton( USymbol::down
                  + umenu( ucheckbox("Mode")
                           + ubutton("Do it")
                           + utext("Ubit...")
                           )
                  )
       )
```

Property and Item bricks can either be constants or variables in which case they can be dynamically modified. The containing box will be automatically updated when one of its children is modified:

```
   UColor &col = ucolor(UColor::red);
   UStr &str = ustr("Click Me!");
   ubutton( col + str );
then:
   col.set(UColor::blue);
   str.set("I am Ready!");
```

This programming style roughly resemble Lisp programming or special-purpose specification languages such as Forms VBT (Avrahami, 1989). It is also conceptually similar to mark-up languages (Boxes could be seen as tags whose behavior and appearance is specified by Property bricks.)

4.1 Conditional Specifications

Property and Item bricks can also be specified in a **conditional** way. These bricks will only be active (or visible) when a given condition is verified. For instance the following code specifies that the background color and the character font will change (respectively) when the mouse enters this text area and when it is pressed:

```
utext( ufont(UFont::bold,UState::entered)
       + ubgcolor(UColor::red,UState::pressed)
       + "Click Me!"
       )
```

Conditions may also depends on timer values or on the State value of another gadget. This makes it possible to program animations or to enforce simple coherency constraints among several GUI objects:

```
UButton *close = null;
udialog( .......
        + ubutton(&close, "Close")
        + ushow(false,
                close->when(UState::pressed))
        )
```

The "ushow" Property will close its dialog parent when the "close" button is pressed. This example also shows how brick pointers can be initialized by giving them as the first argument of uxxx() functions.

4.2 Customization and Ghost Gadgets

Application-dependent components may be created by adding Decoration or Layout bricks or by including Gadgets Boxes into other Gadgets Boxes. Standard Decoration bricks provide various kinds of borders, shadows, etc. for customizing Box objects. New Decoration bricks can easily be derived from standard ones (the corresponding code being quite simple and limited in size.) Decoration bricks may be active and dynamically change according to the current Box State. They may also be specified in a conditional way.

Layout bricks makes it possible to display Box children vertically, horizontally, in a flow or in a table. The "flow" mode mimics HTML standard layout. Text can be warped and can be combined with other GUI components (Fig. 1c).

The gadget composition mechanism is especially powerful because it has been designed in a way that gives the illusion of perfect illusion. For instance, Fig. 1d shows a button that contains four sub-buttons. Clicking on each subpart will provoke different actions (such as invoking a call back function or opening a pop-up menu.) The last button has no visible border and only displays a down arrow. This is not a specific "ArrowButton" object but a standard button that contains a single arrow Symbol brick (a predefined brick that could easily be subclassed for other purposes.)

This visual effect is made possible by using the **Ghost** feature. Ghosts are invisible gadget boxes whose children remain visible. They do not interfere with the propagation of graphical Properties: ghost children are drawn accordingly to the graphical attributes of ghost parents. Ghost Gadgets are not specific new classes. They are just standard Gadgets whose Style specification mechanism has been inactivated (by adding a Ghost State brick) but still behave in the expected way (a ghost button can be clicked, a ghost checkbox can be set, etc.). So, ghosts are not mere sensitive areas but actually act as invisible versions of standard interactors. It should be noted that the same result could be achieved by directly adding the appropriate bricks to an empty Box brick. The Ghost feature produces the same effect, but rather works in a

"subtractive" way. Both techniques are equivalent and can be used according to programmers preferences.

Fig. 1f shows a file manager example that is entirely made of standard components. Each directory line is an horizontal box that contains a mixture of items and gadgets (Fig. 1e). This GUI component could (almost) be entirely written in pseudo-declarative style. The +/- indicator is a Ghost Checkbox that contains two conditional pixmaps that depends on its on/off state (the "+" pixmap appears when the checkbox is in the "off" state while the "-" pixmap appears in the opposite case.) Clicking on this checkbox will also open (or close) a vertical box that contains the directory subfiles (each subfile line being made in the same way.) This basic behavior is also specified in a declarative way by linking a box "ushow" brick to the checkbox on/off State. Just one call-back function is needed for searching directory subfiles in the file system.

5 BRICK SHARING AND UBIQUITOUS COMPONENTS

An important consequence of the brick model is that it implicitly transfers GUI control from gadgets to Property and Item bricks. In classical architectures, UI objects store and control the values of their own attributes. Heterogeneous architectures based on the MVC model improve this basic architecture by clearly separating GUIs components into "model" and "rendering" objects. Among other characteristics, this makes it possible to create multiple-view GUIs in a simplified way. Besides, some systems also introduce the notion of object groups that makes it possible to handle a collection of primitive objects in a more abstract way (Ousterhout, 1994).

Both aspects are handled in a different way in Ubit interfaces. First, all Primitive and Item bricks can be shared (i.e. have multiple gadget parents). As these bricks control the GUI appearance, they implicitly act as groups. So, changing the current value of a String brick would automatically update all the gadgets that contain this string. This mechanism is completely generic and does not make assumptions on object precise types.

This feature is somewhat similar to the active value mechanism that can be found in certain systems. For instance the UStr and UInt bricks can also be seen as generic data types for representing character strings and integer values. Such objects are not necessarily related to graphical aspects and can be used by non GUI application parts for notification purpose. Similarly, Box objects can be seen as generic data containers.

Data sharing optimizes memory cost and simplifies the synchronization of multiple views. This feature can also be used for parameterizing graphical interfaces. Because graphical properties are reified, Colors, Fonts and Decorations can be shared by an arbitrary number of Box gadgets. Thus, modifying a simple Property brick will automatically update all related UI components. Moreover, common graphical properties such as Fonts and Colors are automatically propagated along the GUI DAG (Ubit interfaces having a DAG structure rather than a tree structure.) The combination of both features offers quite a powerful way for controlling GUI aspect.

5.1 Implicit Behaviors

This data sharing principle also applies to all other bricks, including Gadget Boxes. Gadgets are said to be shared when they have multiple parents. Two different cases must be distinguished depending on whether the shared Gadget is a Box or a Window subclass. A Window is a Box subclass that owns a physical window on the screen. The Window class is the base class for making Menus, Dialog boxes and the main window of the application.

Childhood relationships do not denote physical inclusion when the shared child is a Window subclass but lead to various implicit behaviors that depends on parent/child combinations. So, a Menu parent will for instance automatically open this Menu when it is activated in an appropriate way. The activation condition and the child behavior depend on context. A Menu Gadget will behave as a pull-down menu if its parent is a Button that is located inside a Menu Bar. But it will behave as a contextual pop-up menu if its parent is an isolated Button. It will only be opened by pressing on the right button of the mouse if its parent is a Label or a Text field. At last, some parent/child combinations will not perform any implicit behaviors. Similar rules applies for Dialog boxes (except that parents must be clicked instead of being pushed.)

Implicit behaviors work in a dynamical way. The same Window child may behave in different ways depending on which parent was actually activated. So, the same Menu could either behave as a pop-up or as a pull-down menu depending on activation context.

Implicit Behaviors make it possible to encode menu systems and dialog boxes in a quasi-procedural style, all basic behaviors being automatically deduced from structural relationships among components. Besides, by opposition to most other toolkits, there is no need to use different specific kinds of button or menu classes as instances combinations automatically lead to the appropriate behavior.

5.2 Ubiquitous Gadgets

The second case concerns childhood relationships when the shared child is not a Window subclass. This case is quite different as this type of parent/child

combination do imply physical inclusion of children into parents. Child Gadgets are then *visually replicated* into all parents. However, data is not duplicated, only physical representations are. A single gadget can thus have **ubiquitous** representations of the screen. Moreover, this mechanism is able to virtually replicate an entire instance subtree (all subtree gadgets being then implicitly ubiquitous.)

This feature is quite useful when implementing multiple views. It could for instance be used to display a set of checkboxes or composite text (including complex combination of items or other gadgets as in Fig. 2a) in different parts of the GUI. All views would be automatically updated and data would alway be displayed in a consistent way.

The ubiquity mechanism ensures logical coherency but does not impose all views to be strictly identical. As seen before, the Ubit toolkit works in a totally dynamical way. Thus, different graphical properties can simultaneously apply on a shared component, depending on which component parent these properties where added to (each view will recursively use the properties specified by its corresponding parent.) This feature can for instance be used to display the same view at different scales, with another layout or with different colors, etc.

This feature could be extended to the case of distributed interfaces, so that ubiquitous gadgets could be represented on several screens. This could be done in a simple way in the case of X-Window applications by using the standard networking capabilities of the X protocol. A single program could then display identical windows on several remote machines in a transparent way (the code being almost identical when displaying GUIs on one or several machines).

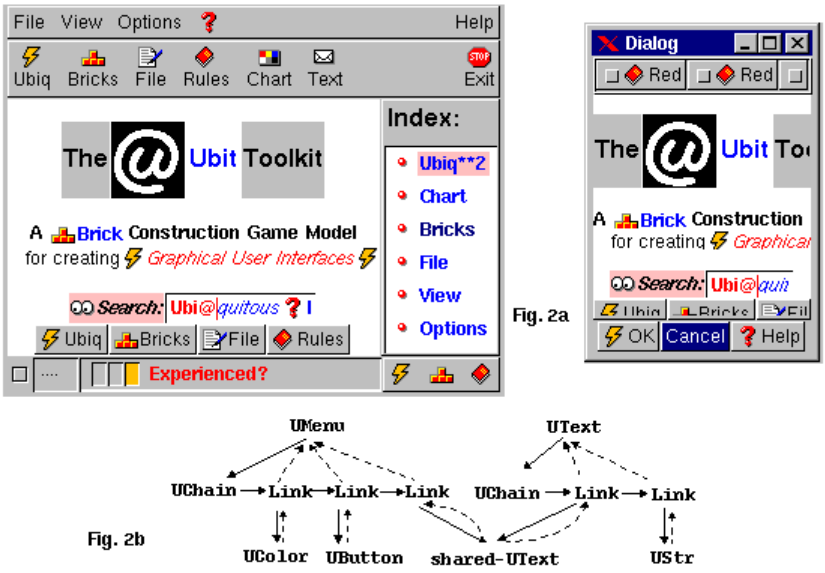

Figure 2: Ubiquitous components and ULinks

6 IMPLEMENTATION

6.1 Box anatomy and visual polymorphism

All gadgets derive from, and are very similar to, the Box brick (or to the Window brick for menus and dialogs.) Boxes are conceptually divided into three separate (but logically related) parts: the Style, the Renderer and the State/Behavior part.

Styles are interchangeable parts that specify all possible default properties (such as colors, fonts, borders, shadows...) that a given gadget class may need for being drawn in all possible states (i.e. when this gadget is pressed, activated, disabled...). Styles are defined in a hierarchical way by further customizing of the Default Style object, so that most data is actually shared between style instances. This feature simplifies Style customizing (for instance for implementing application-defined or native “look-and-feels”). It also improves data management and makes it possible to optimize certain drawing routines.

Styles can be inactivated, as in the case of Ghost objects. They also allow for *visual polymorphism* as several Styles can be associated to the same gadget

class. The appropriate style is then dynamically chosen by the gadget instance depending on its structural context. For instance Button gadgets use different visual Styles whether they are located inside or outside menus.

Behaviors are also defined in a generic way. Typical GUI behaviors are virtually designed at the Box brick level. In most Box subclasses (i.e. the Gadgets) just declare which specific behaviors they will require. This is done by changing the Box State characteristics. Box State can also be changed dynamically by adding appropriate State bricks. So, all gadgets are for instance virtually able to deal with text, can be activated and can have an ”on/off” state.

These design principles simplifies the toolkit architecture and improves memory management efficiency. First, code duplication is avoided as object features are never implemented twice. Moreover, the system does not require programmers to learn a large number of complex object classes that may differ in a subtle way.

6.2 Ubiquitous objects

Brick sharing and ubiquity are tightly related to the way objects are internally stored. Box objects maintain three different lists: the parent, behavior and child lists. The behavior list includes the Callback and State bricks that are relative to this Box while the child list contains its Property, Item and (child) Box bricks. The bricks are indirectly chained through intermediate objects called ULinks. Each list consists in a chain of ULink objects that both point to their corresponding brick and to the next ULink in the list (Fig. 2b). This design allows for object sharing as a single brick may belong to several Box child lists.

ULink objects do not only point to brick objects. They can also contain specific data that is used in combination with the brick instance they point to. There are several ULink subclasses that correspond to the main brick metaclasses (Property, Item, Box, Callback and State.) This feature is at the base of ubiquitous gadgets: Box sizes and coordinates are not stored in Box instances but in their corresponding links. Thus, Gadget Boxes that have multiple parents can deal with separate coordinate systems. This mechanism is quite general and is transparent to the user. Link handling routines are part of their counterpart brick classes so that correspondence between link and brick objects is performed in an implicit way.

7 CURRENT STATUS AND FUTURE WORK

The current version of the Ubit system has been implemented on the top of the X-Window system. It has been tested on several Unix/Linux operating systems and is freely available at URL: http://www.enst.fr/-elc/ubit. A MS-Windows version of the system is currently under development.

Future work will first focus on Information Visualization. We plan adding standard IV capabilities (such as zooming interfaces and miscellaneous focus+context techniques) in the toolkit design. We also plan to adapt the XXL builder (a visual programming tool that was based on textual + visual equivalence and sketch drawing (Lecolinet, 96, 98)) to the Ubit toolkit.

ACKNOWLEDGMENTS

We would like to thank J-D. Fekete, L. Robert, D. Verna, S. Pook and the anonymous reviewers for useful comments.

REFERENCES

Avrahami G., Brooks K., Brown M. (1989) “A Two-View Approach to Constructing User Interfaces.” *Computer Graphics*, **(23)**3.

Esteban O., Chatty S., Palanque P. (1995) “Whizz’ed: a visual programming environement for building highly interactive software”. *Proc. INTERACT’95*, 121-127.

Fowler A. “A Swing Architecture Overview.” http://www.javasoft.com/products/jfc/tsc (archive.)

Glinert, E.P., (1986), “Towards ”Second Generation” Interactive, Graphical Programming Environments.” *Proc. IEEE Workshop on Visual Languages*, 61-70.

Lecolinet E. (1996) “XXL: A Dual Approach for Building User Interfaces.” *Proc. ACM UIST*, 99-108.

Lecolinet E., (1998) “Designing GUIs by Sketch Drawing and Visual Programming.” *Proc. Int. Conf. on Advanced Visual Interfaces (AVI)*, 274-276.

Linton M., Vlissides J.M., Calder P.R. (1989) “Composing User Interfaces with InterViews. *Trans. IEEE Computer*, **22**6, 8-22.

Linton M., Tang S., Churchill S. (1994) “Redisplay in Fresco”. *The X Resource*, **9**, 63-69.

Maloney J.H., Smith R.B. (1995) “Directness and Liveness in the Morphic User Interface Construction Environment.” *Proc. ACM UIST*, 21-28.

Mattis P. at al. (1998) “The GIMP Toolkit.” http://www.gtk.org/docs/gtk.html.

Myers B.A. (1995) “User Interface Software Tools.” *ACM Trans. on Computer-Human Interaction*, **2**(1), 64-103.

Myers B, McDaniel R., Mickish A., Klimovitski A. (1995) “The Design for the Amulet User Interface Toolkit.” *Proc. Human-Computer Interaction Consortium meeting.*

Ousterhout J.K. (1994) *Tcl and the Tk Toolkit.* Addison Wesley.

Szekely P., Luo P., Neches R. (1993) “Beyond Interface Builders: Model-Based Interface Tools.” *Proc. INTERCHI’93*, 383-390.