Construction Techniques of Graphic, Direct-Manipulation User Interfaces

Wolfgang Pree[a], Gustav Pomberger[a], Hermann Sikora[b]

[a]Institut für Wirtschaftsinformatik, University of Linz, A-4040 LINZ, Austria; E-mail: {pomberger,pree}@swe.uni-linz.ac.at

[b]Institut für Informatik, University of Linz, A-4040 LINZ, Austria; E-mail: K390170@ALIJKU11.BITNET

Abstract

This paper deals with human-computer interaction in several ways. On the one hand it presents the roots of interactive, graphic user interfaces and how such interfaces are implemented on the abstraction level of programming languages: The construction of graphic, direct-manipulation interfaces with conventional programming techniques is compared with an object-oriented approach based on powerful class libraries (called user interface application frameworks).

Although application frameworks substantially ease the building of highly interactive applications the abstraction level is considered to be too low to support prototyping such interfaces in a comfortable way. Hence we portray DICE[1] (**D**ynamic **I**nterface **C**reation **E**nvironment), a tool for prototyping graphic user interfaces implemented itself in an object-oriented manner. In particular this paper discusses the question of how dynamic behavior can be added to a user interface prototype. It also presents a useful and powerful way to combine conventionally developed and object-oriented software systems.

Keywords: human/computer interaction, graphic direct-manipulation interfaces, toolkits, object-oriented programming, class libraries, application frameworks, prototyping, C++

1 Graphic User Interfaces—Typical Object-Oriented Systems

Object-oriented programming is defined in [Knu90] as follows: “A program execution is regarded as a physical model, simulating the behavior of either a real or imaginary part of the world. Instead of describing a part of the world by means of mathematical equations or other abstraction mechanisms, a physical model is literally constructed. Objects are just the computerized material, used to construct the computer based physical model”. This view of programming was supported for the first time by Simula-67, a *simulation* language, which was restricted to a narrow range of application (i.e., mere simulation of real parts of the world). Simula-67 was initially described in [Dah70]. [Nyg81] gives a survey of Simula’s history.

Imaginary things (like the computer “desktop”, which mimics a real desktop but does not map to the physical world) were first modelled in an object-oriented manner by a research group at the Xerox PARC (Palo Alto Research Center), which started to investigate effective

[1] This project was supported by Siemens AG Munich

human/computer interaction in the early seventies. They tried to find concrete metaphors—imaginary things which give the user a certain *illusion*—from the real world, so that users would have a set of expectations to apply to computer environments. Smalltalk-80 [Gol84, Gol85]—an object-oriented programming language together with a convenient environment—was developed in order to realize these concepts. Smalltalk as programming language has served as a model for object-oriented extensions of existing languages and the definition of new languages. Smalltalk's interactive programming environment provided the foundation for window-based graphic user interfaces. This kind of user interface is often called an *object-oriented user interface* since its elements are metaphors of real world objects. PARC, for instance, supplied the first explicit expression of the computer desktop. Icons, typically representing familiar objects, appeared on the computer desktop to provide direct and visible access to files, operations and so on.

It appears that object-oriented programming techniques have drastically reduced the effort of programming such user interfaces. The class libraries developed for that purpose together with their underlying concepts gave rise to great hopes that these ideas would make it possible to rationalize software development in general. The subsequent chapter compares conventional implementation techniques based on procedural programming languages with the object-oriented way of programming such user interfaces.

2 "Low-Level" Implementation Techniques

Applications with an object-oriented graphic user interface do not carry out a sequence of steps in a predetermined order. They are driven primarily by unpredictable events (e.g., moving and clicking the mouse, typing). Thus the central activity of an application with an object-oriented user interface is to constantly look for input events (mouse actions, keystrokes) that occur in any order. This application structure is often called an *event loop*. The application is *event-driven*: input events are accepted asynchronously from different devices and gathered in an event queue. As long as the application is running, events from that queue are read and processed in a specific way. This approach contrasts with programs that systematically limit the alternatives available to the user. It allows the user the widest possible range of activities (this characteristic is often called "modelessness"), since the emphasis is on responding to each local request the user makes. Possible implementation techniques of the event loop are discussed in the subsequent sections.

2.1 Conventional Solutions

We speak of "conventional solutions", since they rely on routine libraries which are implemented in conventional programming languages, i.e., languages which are based on the concepts of procedures and (eventually) modules, but which do *not* support the basic object-oriented concepts: class hierarchies, encapsulation, data abstraction, inheritance, polymorphism and dynamic binding.

Toolboxes: Simple Routine Libraries

Simple routine libraries which support the programming of object-oriented interfaces implement the low-level components of the user interface like windows and menus. The programmer has to implement the event loop (i.e., the whole application structure) himself. The library just supports the programmer providing a set of routines that every application calls to implement the common behavior (see Figure 1). Such routine libraries are often called *toolboxes*. Since the terms are not yet standardized, we define a routine library to be a *toolbox* if it has the following properties:

(1) It serves for the construction of graphic user interfaces.
(2) The functions of the toolbox are grouped in several parts which are related to the objects of the user interface they help to implement.
(3) The routines, although they might be quite powerful, do not provide any application structure.

The major drawback of a toolbox is described above in (3). Hence, the application structure is often given as a program skeleton which can be copied and modified to fit the specific application's requirements. This is not an optimal solution, of course, since code is always duplicated—a fact that indicates that this code should go into the library. Furthermore the programmer has to determine the order of routine calls, which allows a great deal of flexibility. But this flexibility is not desirable, because it increases the overall complexity and prevents from "wiring" standards of the user interface into the toolbox. This is why "look-and-feel" standards of the user interface have to be explained textually. Examples of toolboxes are the Macintosh Toolbox [App85], and the X-Window System [Rao87].

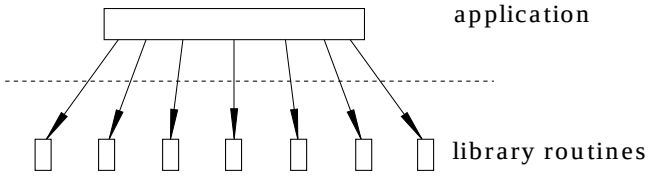

Figure 1. Use of a simple routine library

Toolkits: Routine Libraries Based on the "Hollywood Principle"

A first step to reduce the programming effort of graphic user interfaces with conventional libraries is the so-called *Hollywood Principle*: "Don't call us, we'll call you". Let us illustrate this principle by taking the typical event loop of applications with an interactive, graphic interface as an example: The whole event loop including the initializations could go into a library if the application-specific parts could be added later. This is possible if such a more powerful library routine calls routines the application programmer provides. This is why the principle used in such libraries is called *Hollywood Principle* or *Callback Style of Programming*. The difference to the toolbox approach is illustrated in Figure 2. The main control loop resides in the library routines, not in the application. The library routines read events and *call out* to various procedures which the application has previously registered with the library routines. Libraries which have the properties (1) and (2) of a toolbox as described above, but rely on this style of programming are often called *toolkits*. The advantage of this callback style of programming is that it takes over the burden of managing a complex, event-driven environment.

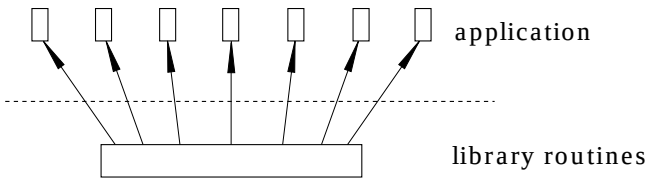

Figure 2. Callback style of programming (= Hollywood Principle)

The striking drawback of toolkits is their lack of fexibility and extensibility, since functionality can only be added where the developer of toolkit parts provides that possibility. In other words, there are only certain spots in toolkit routines where functions are "called back". Changes to the behavior of toolkit routines not forseen by its designer can only be realized by adapting the source code of the toolkit. Another disadvantage of toolkits is the fact that the callback style of

programming takes some getting used to. Examples of toolkits are SunView (a toolkit for the Sun Window System, [Sun90]), XView [Sun91] and the like.

Libraries with an "Object-Oriented Flavor"

The so-called *X-Toolkit* (Xt [McC88]) is based on the X window system. It implements the class concept of object-oriented languages in the not-object-oriented C language by means of conventions (the *Xt Instrinics*). Thus implementations that use Xt have to take care of these conventions: they mimic object-oriented concepts in C. This is rather complicated compared to the real object-oriented solutions described in the subsequent Section 2.2. The main "class" of Xt is called *widget*. All specific user interface elements (buttons, text, scrollbars, etc.) are derived from that abstract building block, which is based on one X window. Weinand [Wei89] states that such a class design can be very inefficient: "It is questionable whether the implementation of a spreadsheet, for example, could use a widget for each cell without substantial performance penalty."

2.2 Object-Oriented Solution – Application Frameworks

We presuppose that the reader is familiar with the basic object-oriented concepts (independent of a specific language): class hierarchies, encapsulation, data abstraction, inheritance, polymorphism and dynamic binding. One of the main advantages of object-oriented programming is that it "actively" supports software-reuse. An application framework is a collection of abstract and concrete classes and the interfaces between them, and is the design for a subsystem [Wir90].

The first widely used framework was Model/View/Controller, the Smalltalk-80 user interface framework [Gol84]. It showed that object-oriented programming is ideally suited for implementing highly interactive, graphic user interfaces. As described in [Wir90] a framework is the design of a subsystem, just as an abstract class is the design of a concrete class. Like a subsystem, a framework is a mixture of abstract and concrete classes. It differs from a subsystem by being designed to be refined. It can be refined by changing the configuration of its components or by creating new kinds of components (i.e., new subclasses of existing classes) without affecting the existing classes.

A mature framework will have a large class library of concrete subclasses of each abstract class, so that most of the time an application can be "plugged" from existing components. Even when new subclasses are needed, they are easily produced because the abstract superclasses provide their design and much of their code. While the framework approach is useful for the development of any software, it is especially attractive if a user interface should be encouraged, for it is possible to completely define the components that implement this standard and to provide these reusable components as building blocks to other developers [Wei89]. In an application with a graphic user interface these components are, for example, documents, windows, commands, and the application itself.

Such user interface frameworks offer several advantages: Compared to the toolbox and toolkit approach where user interface look–and–feel standards are explained textually, such standards are "wired" into the framework components. Furthermore, experience has proven that writing a complex application based on powerful frameworks (such as MacApp [App86, Wil90], AppKit [NeXT89] and ET++ [Wei88, Gam89, Wei89]) can result in a reduction in source code size of 80% and more compared to software written with the support of a conventional toolbox.

Apart from this enormous code reduction, application frameworks have other important benefits: the abstraction level is raised, and a standardization is achieved in terms of both the user interface and the code structure. However, the abstraction level of an application framework is considered to be too low to support prototyping in a comfortable way. Imple-

menting applications with a framework absolutely requires specialized programming ability (especially in object-oriented programming). Furthermore, the programmer must become familiar with the particular application framework—a time investment that cannot be neglected.

This fact is contrary to the philosophy of prototyping. Therefore we implemented DICE (**D**ynamic **I**nterface **C**reation **E**nvironment) for/with an application framework in order to extend such a tool in the direction of rapid prototyping. The subsequent section describes DICE. We implemented DICE with the application framework ET++ for the following reasons: Compared to other available application frameworks, ET++ was the cleanest object-oriented implementation based on a small set of basic mechanisms. ET++ provides a homogenous object-oriented class library that integrates user interface building blocks, basic data structures and high level application components. ET++ was implemented in C++ and runs under UNIX and either SunWindows, NeWS, or the X11 window system. The design and implementation of ET++ is described in detail in [Wei88, Wei89].

3 DICE – An Object-Oriented Tool for Rapid Prototyping

Prototyping is a paradigm that is well established in research and practice for enhancing the Software Life Cycle and improving software quality. There are various publications discussing definitions of prototyping in depth (e.g., [Bud84, Flo84, Pom87, Bis90, Pom91]). User Interface Prototyping in particular is important for the development of applications that have non-trivial user interfaces by providing better requirement definitions.

This is especially true for applications with graphic-oriented interfaces. Prototyping this kind of user interface with proper tools can significantly reduce the implementation effort (especially if the prototype can be enhanced to the final product). Furthermore, the acceptance of the software system is improved since the interface can be discussed with the later user at an early stage of the development. DICE supports the graphic specification of the user interface layout (see Section 3.2). In order to enhance the functionality of the prototype, most comparable tools available today provide interfaces to procedural languages or some kind of an integrated procedural language. DICE offers three possibilities for enhancing the functionality (see Section 3.3):

- Without programming: Interface elements communicate with one another by sending predefined messages.
- With object-oriented programming: Subclasses of ET++ classes can be generated. Application-specific behavior is added in subclasses of the generated classes.
- With conventional programming: A protocol was developed that allows the prototype to be connected with other UNIX processes using one of UNIX's interprocess communication mechanisms.

The simulation of the constructed interface (screen layout together with its dynamic behavior) is at the designer's disposal.

3.1 Prototype Management

Experience has proven that user interface prototypes developed for different software systems often consist of similar or even identical parts. This is why DICE's Control Panel (see Figure 3) was implemented. It manages an arbitrary number of prototypes so that several prototypes can be viewed and compared at the same time. Thus prototype parts may be interchanged, too (see Section 3.2). A prototype in DICE is a *set of windows* which contain specific interface

elements. Section 3.2 focuses on how these windows and their contents are specified and modified.

After DICE is started, DICE's Control Panel (see Figure 3) and DICE's Tool Palette (see Figure 4) appear on the screen. The left part displays a list of currently opened prototypes (¿). If a prototype is selected in the prototype list (¿), all windows belonging to that prototype are displayed in the window list (¡). As Figure 3 shows, the prototype "Accounting", for example, consists only of one window, "Overview".

Figure 3. DICE's Control Panel

Until now we have used the term "window" as a generic term for "prototype window", "window title", and "window editor". To be exact, we have to define what we mean: an operational prototype consists of several *prototype windows* (rectangular areas on the screen that contain user interface elements (such as buttons, menu, lists, etc.) and have a specific *window title*. For example, the window title of DICE's Control Panel in Figure 3 is "DICE").

Windows of a prototype are specified in a *window editor* in DICE. Each window has its own dedicated window editor that is itself a window and mimics the prototype window which is specified by means of it as far as possible (WYSIWYG principle): the window title in the window editor is the same as in the window that is specified by means of that editor—this window title is also displayed in the window list (¡ in Figure 3). The position of the editor window is identical with the position of the prototype window; the window contents of the window editor almost looks like the contents of the prototype window.

The only difference is the behavior of the window contents: a window editor allows us to edit components of a window (e.g., to insert, move or resize constituent interface elements, to edit the window title, etc.—see Section 3.2 "Static Layout"), but these user interface items do not work (e.g., action buttons cannot be pushed, text cannot be edited in a text subwindow, etc.). If a prototype is tested (by pressing the "Test Prototype" button), "real" prototype windows are generated out of their specification in corresponding editor windows so that the behavior of constituent user interface elements can be tested. For each window displayed in the window list (see ¡ in Figure 3), a checkmark at the left indicates that its corresponding window editor is open. The possibility to close window editors of a particular prototype helps to prevent the user from having a mess of open windows on the screen.

3.2 Static Layout Aspects

DICE provides graphic window editors that allow comfortable specification of prototype windows. Since these window editors mirror the way windows are specified in ET++, we need

to explain one ET++ concept first: ET++ offers two specific interface items, *Cluster* and *Expander*, together with a number of layout operators, that allow the grouping of interface elements. Both, Cluster and Expander elements have the following attributes:

- alignment of contained interface elements
- distance between contained interface elements

Cluster and Expander elements follow the principle of recursion: these objects may contain "single" interface elements (e.g., text subwindows, buttons) as well as Clusters and Expanders.

Figure 5. Sample user interface specified with DICE

Figure 4. DICE's Tool Palette

This concept fits the needs of all complex layouts without the user having to position interface elements explicitly. Furthermore, these two grouping elements allow layout specification of user interface elements that copes even with window resize operations: Interface elements that are grouped in an Expander grow and shrink proportionally to the size of their surrounding Expander, the distance between the constituent elements remains unchanged. If a Cluster is resized, the distance between the constituent elements changes, but not the size of the elements. The outermost interface element of each prototype window is an Expander: when the "Add

Window" button in DICE's Control Panel (see Figure 3) is pressed, a window editor that contains an empty Expander is opened.

Insertion of Interface Elements

To insert user interface elements, one simply chooses the proper element in DICE's Tool Palette (see Figure 4) and mark the position in a group (Expander or Cluster) inside a window editor where the item is to be placed.

Editing Functions

DICE's window editor provides *undoable* editing functions as known from state-of-the-art graphic editors. The editing functions described below all require that the Pointing Tool be chosen in DICE's Tool Palette (see Figure 4):

- Elements are *selected* with a mouse click. (A selected element has handles on the corners of its boundary box.)
- A selected element is *resized* by dragging its handles with the mouse.
- A single element or a group (i.e., a Cluster or Expander) can be *moved* with the mouse within a window editor. If an element is dragged outside the outmost Expander, it is thrown away.
- *Cut/copy/paste* works within a window editor, between window editors of a prototype and between window editors of prototypes that are opened in DICE's Control Panel (see Figure 3). A group item must be selected in order a cut or copied element can be pasted as the last element of a selected group.

The parameters of interface elements are defined in specific attribute sheets. Figure 5 shows attributes for an action button.

To sum up, DICE's Window Editor component together with DICE's Tool Palette represent a *graphic specification language for graphic user interfaces*. The specification of a prototype is transformed within a neglectable amount of time (a fraction of a second on a SUN Sparc Station 1+) into an operational prototype simply by pressing the "Test Prototype" button in DICE's Control Panel (see Figure 3). The same button (whose label changes from "Test Prototype" to "Edit Prototype" during a prototype test) has to be pressed again in order to further modify a prototype—so DICE either operates in a *specification mode* or a *test mode*. All operations described in this section for modifying the static layout of a prototype are only available in the specification mode.

3.3 Adding Functionality to the Prototype

The behavior of a system is above all determined by its dynamics. Therefore it is not enough just to describe screen layouts. Rather, it should also be possible to portray the dynamic behavior of a system and at the same time to develop the prototype to an accomplished application. For this purpose, DICE offers the possibilities described below.

3.3.1 Predefined Messages

Each user interface element has certain messages assigned that it "understands": the messages "Open" and "Close" are assigned to windows; Clusters, Expanders, and Empty Subwindows react to no message; all other interface elements understand at least "Enable" and "Disable". In addition, Text Subwindows, Static Text Fields and Editable Text Fields change their text if they

receive a “SetText(...)” message. A List Subwindow switches its list if it receives a “SetList(...)” message. Enumeration Items display another integer value according to the integer number sent in a “SetValue(...)” message. (Labeled) Radio and Toggle Buttons alter their state depending on the parameter of a “SetState(...)” message.

Figure 6. Message Editor

From each element that can be activated (buttons and menu items), any number of messages to other elements can be specified by means of DICE’s Message Editor (see Figure 6). If the prototype is tested and an interface element is activated in the test mode, the messages specified for that element are sent to their receivers. They effect the corresponding change(s) in the user interface. Thus rudimentary dynamics are realized without programming effort.

Let us take a simple “Cash Dispenser” prototype (see Figure 5) as an example. The “Cash Dispenser” window is to be closed when the “Stop” button is pressed. To specify this functionality, one presses the “Link...” button in the attribute sheet of the “Stop” button (see Figure 5). By means of DICE’s Message Editor (see Figure 6), the desired dynamic behavior can then be defined for the “Stop” button (i.e., that the message “Close” is to be sent to the window “Cash Dispenser” when the “Stop” button is pressed—the button with the component name “StopButton” being the sender). The left list (“Target Objects”) in the Message Editor displays component names of already existing user interface elements. After a component name is selected in the left list, all messages that are understood by the selected user interface element are displayed in the list “Possible Messages”. The right list of already defined messages shows message names together with the component names of their receivers. Pressing the “Set Up Link” button as demonstrated in Figure 5 will add the message “Close” (to be sent to the component “CashDispenserWindow”) to the list of already defined messages.

One of the criteria for judging prototyping tools is their extensibility (e.g., adding new messages to DICE’s interface elements). The classes of DICE have been designed in such a way that this adaption can be achieved by means of object-oriented techniques (i.e., without modifying the source code of the tool).

3.3.2 Generating Classes

DICE simulates the static and dynamic behavior of the specified prototype when that prototype is tested. Thus no code generation and no compile/link/go cycles are necessary for testing. In order to enhance the prototype based on the application framework ET++, DICE creates subclasses of ET++ classes (i.e., C++ classes) when the “Generate Code” button is pressed

(Figure 3). The compilation of the generated classes results in an application which works exactly like the specified prototype.

The generated classes need not (and should not) be changed when further functionality is added in the sense of evolutionary prototyping. Additional functionality can be implemented in subclasses of the generated classes by overwriting or extending the corresponding dynamically bound methods (see Figure 7). The basic idea of code generation is the following: A subclass of the ET++ class Document is generated for each window specified with DICE. The dynamically bound method Control of this class is called if an interface element is activated.

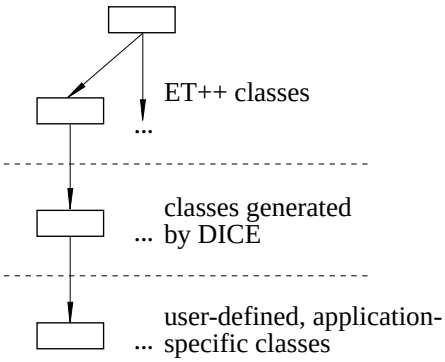

Figure 7. Code generation

Let us look at the cash dispenser interface (Figure 5) as an example: When the "Ok" button (component name: "OkButton") is pressed, a check of the correctness of the displayed amount (in the *Text Field* called "Display") is necessary. The generated class CashDispenser implements no special behavior if the "Ok" button is pressed; this behavior could not be specified with the predefined messages:

```
class CashDispenser: public Document {      …
        void Control(int id) {
            …
            case OkButton:
                break;  // no action
                …
        }
};
```

For this reason the class ExtCashDispenser (stands for "Extended Cash Dispenser") is implemented in order to check the correctness of the amount.

```
class ExtCashDispenser:  public CashDispenser {                 …
        void Control(int id) {
                …
                case OkButton:
                        int disp=Display->Val();
                        if (AmountOk(disp))    …
                        break;
                …
                CashDispenser::Control(id);
        }
};
```

This kind of code generation separates changes of the user interface from coded functionality as far as possible. For instance, if the user interface layout is changed, code (i.e., ET++ subclasses) must be generated again. The user-designed classes that have been derived from the originally generated classes are not concerned. Changes of these classes become only necessary if interface elements are removed (which would result in extraneous code) or switched between windows of the prototype.

3.3.3 Integration of Conventional and Object-Oriented Systems

Writing applications based on frameworks can result in an enormous reduction in source code size (see Section 2.2) since they provide powerful building blocks for the kind of applications they support. Unfortunately, the only powerful frameworks available today are for user interface programming. Therefore it is very important for a user interface prototyping tool to facilitate communication with other systems. Since ET++ is implemented on UNIX systems, the socket mechanism is used for interprocess communication of independent processes. The interface specified with DICE and the process interacting with the interface form a so-called UIMS (User Interface Management System) with mixed control [Bet87, Hay85].

Communication between the application and user interface (see Figure 8) is based on a simple protocol: If a user interface element is activated in DICE's test mode (i.e., all kinds of buttons, Enumeration Items, Text Items in a List Subwindow, and Menu Items), an exact element identifier and its value are sent to the connected process in the following format: identifier=value. The identifier is usually the component name of the activated element. If a menu item is selected, identifier is the component name of the user interface element the menu is part of (either a Pop-up Item, a Text Subwindow, or a List Subwindow) concatenated with a dot (".") and the text of the selected menu item. If a text item in a List Subwindow is selected, the identifier consists of the component name of the List Subwindow concatenated with a dot (".") and the text of the selected text item. Activated Action Buttons, menu items, and text items in List Subwindows always send TRUE as their value. The value of an Enumeration Item is its current integer value, (Labeled) Radio and Toggle Buttons send either TRUE or FALSE as value (depending on their state).

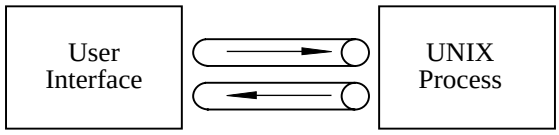

Figure 8. Communication between application and user interface

A connected process can ask for the value of an interface element by sending identifier ? to the user interface prototype. If a user interface element exists that matches identifier, it "answers" as if it had been activated using the format described above. Values of user interface elements can be changed from the connected process by sending identifier=value to it. This allows some special changes in the user interface, too: windows, for example, can be opened or closed using the value OPEN or CLOSE. A List Subwindow accepts EMPTY as value (to make the list emtpy). A text string sent to a List Subwindow as value means that this text is to be appended as a list item in the correspondent List Subwindow.

The communication protocol is the precondition that a user interface developed with DICE can be connected with any conventional or object-oriented software system. E.g., the functionality of the cash dispenser depicted in Figure 5 was implemented in C. (It could also be implemented in Cobol or Fortran or what else is available.) Necessary modifications or enhancements of the functionality are implemented in a C program. Immediately after compiling and starting this

program, the modified functionality can be tested together with the user interface prototype (in test mode) without restarting DICE.

The development of software systems that are to be connected with the interface prototype can be supported by available methods and tools. Bischofberger and Pomberger [Bis90, Pom91], for instance, describe a paradigm called "Prototyping-Oriented Incremental Software Development" and a tool that supports this paradigm, the System Construction Tool. It is, of course, possible to connect the interface prototype with object-oriented systems developed by means of any domain-specific class libraries that might be available.

4 Conclusion

We are convinced that application frameworks available today are well suited as a basis for the programming of applications with graphic, direct-manipulation interfaces. System design decisions are partially prescribed. A number of object-oriented ideas, as offered in an application framework, significantly reduce the implementation effort for such systems. DICE demonstrates how currently available user interface application frameworks can be enhanced in order to serve as powerful prototyping tools. DICE was evaluated in several small and medium-scale projects. Parts of the tool (especially attribute sheets and DICE's Control Panel) were generated with DICE itself. The next step will be an extension of available user interface elements and the integration of a simple programming environment for C++ so that application specific behavior can be programmed in an object-oriented manner within DICE.

Extensions of DICE (e.g., adding new interface elements, modifying the attributes to be defined for an interface element, and adding new messages that are "understood" by the interface elements) can be achieved by means of object-oriented techniques (i.e., without modifying the source code of the tool) since the tool itself is implemented in an object-oriented way based on the application framework ET++. Thus DICE can be extended in an incremental manner by the user.

References

[App85] Apple Computer: *Inside Macintosh, Vol. I-V,* Addison-Wesley, 1985-1988

[App86] Apple Computer: *MacApp Programmer's Manual,* Apple Computer, Inc., Cupertino, CA, 1986

[Bis90] Bischofberger W.R.: *Prototyping-Oriented Incremental Software Development: Paradigms, Methods, Tools and Implications*, doctoral thesis, University of Linz, 1990.

[Bet87] Betts, B. et al.: *Goals and Objectives for User Interface Software*; in: Computer Graphics, Vol. 21, No. 2, April 1987, pp. 73-78.

[Bud84] Budde R. et al.: *Approaches to Prototyping*; in Proceedings of the Working Conference on Prototyping, Namur, October '83, Springer 1984.

[Dah70] Dahl O.-J., Myrhaug B., Nygaard K: *Simula 67, Common Base Language*; Publication S-22, Norsk Regnesentral, Oslo, October 1970.

[Flo84] Floyd, C.: *A Systematic Look at Prototyping*; in: Approaches to Prototyping, Springer, 1984, pp. 1-18.

[Gam89] Gamma E., Weinand A., Marty R.: *Integration of a Programming Environment into ET++ A Case Study,* Proceedings of the 1989 ECOOP, July 1989.

[Gol84] Goldberg A.: *Smalltalk-80, The Interactive Programming Environment* Addison-Wesley, Reading, Mass., 1984.

[Gol85] Goldberg A., Robson D.: *Smalltalk-80 / The Language and its Implementation*; Addison-Wesley, 1985.

[Hay85] Hayes, P.J., Szekely, P.A., Lerner, R.A.: *Design Alternatives for User Interface Management Systems Based on Experience with COUSIN*; in: Human Factors in Computing Systems: CHI'85 Conference Proceedings, Boston, Mass., April 1985, pp. 169-175.

[Knu90] Knudsen J.L., Madsen O.L., Norgaard C., Petersen L.B., Sandvad E.S.: *Teaching Object-Oriented Programming Using BETA*; presented at the 6th Annual Apple European University Consortium Conference, Salamanca, Spain, April 18-20, 1990.

[McC88] McCormack J., Ascente P., et al.: *Using the X Toolkit*; in: Proceedings of USENIX '88, 1988

[NeXT89] NeXT, Inc.: *1.0 Technical Documentation: Concepts*; NeXT, Inc., Redwood City, CA, 1989

[Nyg81] Nygaard K., Dahl O.-J.: *Simula 67*; in "History of Programming Languages", ed. R. W. Wexelblat, 1981.

[Pom87] Pomberger G., Bischofberger W.R., Keller R., Schmidt D.: *Prototyping-Oriented Software Development – Theoretical, Technical and Organizational Aspects, Part I* (in German); Research Report Nr. 87.05, Institut für Informatik, University of Zürich, 1987.

[Pom91] Pomberger G., Bischofberger W., Kolb D., Pree W., Schlemm H.: *Prototyping-Oriented Software Development, Concepts and Tools*; in Structured Programming Vol.12, No.1, Springer 1991.

[Rao87] Rao R., Wallace S.: *The X Toolkit: The Standard Toolkit for X Version 11*. USENIX Association Conference Proceedings, USENIX Association, El Cerrito, CA, 1987.

[Sun90] Sun, Inc.: *SunView Programmer's Guide*; Sun Microsystems, Inc., Mountain View, CA, 1990.

[Sun91] Sun Microsystems, Inc.: *Writing Applications for Sun Systems: A Guide for Macintosh Programmers*; Addison Wesley, 1991.

[Wei88] Weinand A., Gamma E., Marty R.: *ET++ - An Object-Oriented Application Framework in C++*; OOPSLA 88, Special Issue of SIGPLAN Notices, Vol. 23, No. 11 (1988)

[Wei89] Weinand A., Gamma E., Marty R.: *Design and Implementation of ET++, a Seamless Object-Oriented Application Framework*; in: Structured Programming Vol.10 No.2, Springer 1989.

[Wil90] Wilson D.A., Rosenstein L.S., Shafer D.: *Programming with MacApp*; Addison-Wesley, 1990.

[Wir90] Wirfs-Brock R.J., Johnson R.E.: *Surveying Current Research in Object-Oriented Design;* in Communications of the ACM, Vol. 33, No. 9, Sept. 1990.

Trademarks: