Integration of Object-Oriented Software Development and Prototyping: Approaches and Consequences

Wolfgang Pree
Department of Computer Science, Washington University
One Brookings Drive, St. Louis, Missouri 63130, U.S.A.
wolfgang@amadeus.wustl.edu

C. Doppler Laboratory for Software Engineering
Johannes Kepler University of Linz, Austria

Abstract. Although object-oriented application frameworks like MacApp [13], AppKit [8] and ET++ [12] substantially ease the building of graphic, direct-manipulation user interfaces, the level of abstraction is considered to be too low to support prototyping such interfaces in a comfortable way. Thus we implemented a user interface prototyping tool based on an object-oriented application framework.

The most important part of a software prototype is its dynamic behavior. On the basis of the tool mentioned above we discuss several ways in which means of adding dynamic behavior to a user interface prototype can be smoothly combined in one tool, in particular combining conventional and object-oriented software. Finally, we categorize user interface prototyping tools available today according to the concepts they offer for dynamic behavior specification.

Keywords: graphic direct-manipulation user interfaces, prototyping, object-oriented programming, application frameworks, multi-paradigm systems, C++

INTRODUCTION

We presuppose that the reader is familiar with object-oriented concepts (independent of a specific language): encapsulation, data abstraction, inheritance, polymorphism and dynamic binding, as well as with principles of graphic user interface application frameworks like MacApp, AppKit and ET++.

Such user interface frameworks offer several advantages: User interface look–and–feel standards are "wired" into the framework components. Furthermore, experience has proven that writing a complex application based on an application framework can result in a reduction in source code size of 80% and more compared to software written with the support of conventionally implemented libraries.

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. Implementing 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[1] [9, 10] (**D**ynamic **I**nterface **C**reation **E**nvironment) for/with the application framework ET++ in order to extend this tool in the direction of prototyping. The subsequent section describes several ways to specify dynamic behavior as offered by DICE. What sets DICE apart from other available prototyping tools is that it elegantly combines commonly used concepts to add dynamic behavior to a prototype. Furthermore, due to its object-oriented implementation DICE's specification component is extensible in a straightforward fashion.

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

---

[1] This project was supported by Siemens AG Munich

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 [4, 11, 12].

ADDING FUNCTIONALITY TO A DICE PROTOTYPE

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., [2, 3, 9]). User Interface Prototyping in particular is important for the development of applications that have graphic direct-manipulation user interfaces by providing better requirement definitions. Prototyping this kind of user interfaces with proper tools can significantly reduce the implementation effort (especially if the prototype can be enhanced to the final product).

It is not enough to just describe screen layouts, since the most important aspect of a user interface prototype is its dynamic behavior. In order to support evolutionary prototyping it should be possible to portray the dynamic behavior of a system and at the same time to enhance the prototype to an accomplished application. For this purpose most tools available today provide interfaces to procedural languages or some kind of an integrated procedural language.

DICE supports the graphic specification of the (static) user interface layout similar to other available tools: User interface elements offered in a palette (e.g., action button, labeled radio/toggle button, editable text field, non-editable text field, menu, text subwindow—a subwindow containing a full-fledged text-editor, list subwindow—a subwindow containing a list of selectable text items) are placed into windows simply by dragging them from a palette to the appropriate window. Attributes of interface elements (like the text displayed inside an action button) are defined in dialog boxes. For example, Figure 1 shows the attribute specification of an action button labeled "Stop".

In order to enhance a prototype's functionality DICE offers three possibilities:

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

DICE either operates in a specification mode or a test mode. DICE lets the user transform the specification of a prototype (its static and dynamic behavior) into an operational one within a neglectable amount of time (a fraction of a second on a SUN Sparc Station 1+).

Figure 1: Cash Dispenser prototype (in specification mode)

Predefined Messages

Each user interface element has certain messages assigned that it "understands": For instance, the messages "Open" and "Close" are assigned to a window. All other interface elements understand at least "Enable" and "Disable". In addition, text subwindows, non-editable 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. Labeled radio and toggle buttons alter their state depending on the parameter value of a "SetState(...)" message.

DICE realizes *state transitions* (in finite automata terminology) in the following way: 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 below). If the prototype is tested (i.e., the prototype specification is transformed into an operational prototype) 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) (=state transition(s)) in the user interface. Thus rudimentary dynamics are realized without programming effort.

Let us take a simple cash dispenser prototype (see Figure 1) as an example. We want the display (¿ in Figure 1) to show the text "Oops—Stop Button Pressed" when the button labeled "Stop" is pressed. To specify this functionality, one presses the "Link..." button in the attribute sheet (= the dialog box where attributes of the selected user interface element can be edited) of the "Stop" button (see Figure 1). (We assume that the component name of the display field is "Display" and that the button labeled "Stop" has the component name "STOP".) By means of DICE's Message Editor (see Figure 2), the desired dynamic behavior can then be defined for the "Stop" button (i.e., that the message "SetText(...)" is to be sent to the non-editable text field "Display" when the "STOP" button is pressed—the button with the component name "STOP" as its sender (see ¿ in Figure 2)). After the button "Set Up Link" of the Message Editor (see Figure 2) is pressed the appropriate text string has to be provided as parameter of the message "SetText(...)" by means of a text editor.

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 (in our example the message "Disable", which is to be sent to the button with the component name "OkButton", is already defined, the button with the component name "STOP" being the sender). After the "Set Up Link" button is pressed as demonstrated in Figure 2 and the appropriate text string is specified, the message "SetText(...)" (to be sent to the component named "Display") will be added to the list of already defined messages.

Connection of a Prototype with Other UNIX Processes

Algorithmic components of a DICE prototype can be implemented in any formalism and communicate with the user interface prototype specified with DICE by means of a simple protocol that is described below. The integration requires *no code generation* for the user interface part and thus no compile/link/go cycles. An arbitrary number of components implemented in different formalisms can be connected with a user interface prototype that is specified and tested within DICE.

Figure 2: DICE's Message Editor

Communication Concept: Since DICE is implemented on UNIX systems, the UNIX Interprocess Communication mechanisms (e.g., sockets, shared memory) are used for interprocess communication of independent processes (see Figure 3). The interface specified with DICE and the process(es) interacting with the interface form a UIMS (User Interface Management System) with mixed control [1, 5]. This means that an application's "work" is accomplished by various loosely coupled parts of a software system. In case of DICE a DICE user interface prototype forms all visible parts of the user interface and maybe some basic functionality specified by means of predefined messages. Other functionality may be spread over several system parts that are coupled with the user interface by a simple protocol as described below.

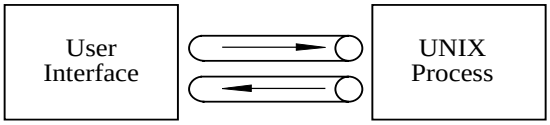

Figure 3: Connection between a user interface prototype and an arbitrary process

Communication Protocol

We illustrate this protocol as far as it is necessary to understand DICE's interprocess communication concept.

User Interface Prototype -> Connected Process: If a user interface element of a protoype is activated in DICE's test mode (activatable user interface elements are all kinds of buttons, text items in a list subwindow, and menu items), an element identifier and its value are sent to the connected process(es) in the following format: identifier=value The identifier is usually the component name of the activated element. If a menu item is selected, the identifier is the component name of the user interface element the menu is part of (e.g., 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 of the selected text item.

Activated action buttons, menu items, and text items in list subwindows always send TRUE as their value. Labeled radio and toggle buttons send either TRUE or FALSE as value (depending on their state).

Connected Process -> User Interface Prototype: 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 empty the list). 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 specified in Figure 1 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, even without switching from the test mode to the specification mode and back to the test mode.

The development of software systems that are to be connected with the interface prototype can be supported by available methods and tools. Pomberger [9], for instance, describes a tool that allows prototyping-oriented incremental software development. Due to DICE's Communication Protocol it was easy to combine this tool with DICE.

On the other hand, it is, of course, possible to connect a user interface prototype specified and tested in DICE with object-oriented systems developed by means of any domain-specific class libraries that might be available.

Generating Application Framework Subclasses

DICE simulates the static and dynamic behavior of a 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 by means of the application framework ET++, DICE allows the creation of subclasses of ET++ classes. The compilation of the generated classes results in an application which works exactly like the specified prototype.

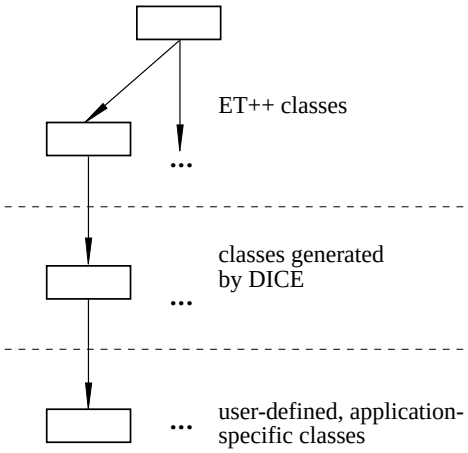

Figure 4: Code generation concept

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 overriding or extending the corresponding dynamically bound methods (see Figure 4).

Let us look at the cash dispenser interface (Figure 1) again: When the “Ok” button in the window titled “Chase Manhattan Bank N.Y.” is pressed, the correctness of the displayed amount should be checked. This functionality could not be provided by DICE’s prototyping facilities. Therefore we would like to add special code in order to implement this behavior.

DICE uses the component names of user interface elements in the generated code. Component names can be defined for each user interface element in the corresponding attribute sheet (see, for example, Figure 1: the component name of the button labeled “Stop” is “STOP”). We assume that the button labeled “Ok” has the component name “OkButton” and that the window titled “Chase Manhattan Bank N.Y.” has the component name “CashDispenser”.

So DICE generates a class CashDispenser. DICE reuses behavior implemented in the ET++ class Document by generating CashDispenser as subclass of it. Document, for example, manages a window in which the appropriate contents is displayed. Furthermore, the ET++ class Document has a dynamically bound method Control which is called each time a user interface element is activated inside a window associated with a Document object. Thus the method Control is used in the generated code to implement the behavior of user interface elements specified by means of predefined messages. Since no behavior was specified by means of predefined messages for the button with the component name “OkButton” the code generated by DICE is the following:

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

In order to check the correctness of the amount, we implement a class ExtCashDispenser (stands for “Extended Cash Dispenser”). The presented code fragment is simplified in order to stress the essential idea of adding functionality in subclasses of generated classes.

```
class ExtCashDispenser: public CashDispenser {
    …
    void Control(int id) {
        …
```

```
                case OkButton:
                        int  disp=Display->Val();
                        if  (AmountOk(disp))
                                ...
                        break;
                ...
                CashDispenser::Control(id);
        }
};
```

To sum up, this kind of code generation separates changes of the user interface from hand-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-defined classes that have been derived from the originally generated classes are not concerned. Changes of these classes only become necessary if interface elements are removed (which would result in extrenous code) or switched between windows of the prototype.

CATEGORIZATION OF USER INTERFACE PROTOTYPING TOOLS

Prototypes built with DICE (i.e., prototypes that are executable within DICE in the test mode as well as ET++ applications generated from the prototype specification) are *finite* automata consisting of a finite number of states (the static layout of user interfaces) and state transitions (the dynamic behavior). We call this basic structure of a prototype its *application model*.

Applications built with state-of-the-art application frameworks are typically *infinite* automata: states and state transitions are described in classes from which an arbitrary (and theoretically unlimited) number of instances can be created. So the number of states and state transitions is not limited. For instance, a text editor application may have an arbitrary number of documents (= windows) in which text can be edited. Though the windows of one such text editor can be specified with DICE (e.g., by means of the text subwindow), the prototype as well as the eventually generated application have only the specified windows—the text editor application is not instantiable.

Thus the underlying application model of DICE prototypes and the application model of typical applications that are built on top of state-of-the-art application frameworks differ considerably. Since DICE's application model is a subset of the application model of a modern user interface framework, it is easy to generate subclasses of such a framework (ET++ in case of DICE), so that the transformation of the generated classes into an executable program results in an application which works exactly like the prototype specified with DICE. In order to project DICE's application model to an application framework, the generated classes have to eliminate many mechanisms provided by the framework classes: in ET++, for example, the complete document management done in class Application becomes superfluous.

Abstraction Level of Dynamic Behavior Specification

In general, the abstraction level of the specification of dynamic behavior determines whether the application model of the specified prototype can correspond to the application model of typical framework applications. User interface prototyping tools known today that allow the specification of dynamic behavior on an abstraction level higher than that of a programming language rely on the concept that applications with graphic, direct-manipulation user interfaces are finite automata—an application model that does not match that of modern application frameworks. The main reason for this fact is that the application model represented by finite automata can be specified with graphic editors in an easy and intuitive way.

The more sophisticated application models of modern application frameworks would require other graphic-oriented specification techniques. Such *visual programming* editors have not reached the maturity to allow use in this context [7]. NeXT Interface Builder [8] supports the building of applications that adhere to the application model of a modern application framework (AppKit) at the cost of specifying dynamic behavior on the programming language level (At first glance, the possibility offered by NeXT Interface Builder seems to be identical with predefined messages in DICE, but there is one crucial difference: In NeXT Interface Builder message connections between objects (called *sender* and *target* in this context) are method calls of the target object issued by an activated sender object. The messages that objects "understand" must be implemented in classes.)

Supported Application Area

Another important issue of user interface prototyping has to be taken into consideration, too: Many commercial data processing applications heavily rely on database management systems. Evolutionary prototyping of applications belonging to this category could benefit a lot if the user interface prototyping tool or the generated executable prototypes could be integrated with a (relational or object-oriented) database management system. Tools that allow user interface prototyping and the development of a database management system are often called fourth generation systems [6]. Though the term fourth generation system has not been standardized yet, we give a possible definition of such a system: fourth generation systems are built around a database management system and enable the developer to specifiy/implement not only the user interface layout but also data models, reports and consistency rules on a high abstraction level. They typically provide standard search and sort facilities and procedural languages for implementing dynamic behavior.

If a user interface prototyping tool is used within a fourth generation system the kind of code generation (based on a conventionally implemented toolkit or an application framework) is almost irrelevant because the user interface of commercial data processing applications (often called *information systems*) can be completely specified with available user interface prototyping tools in most cases: text fields, buttons, lists and text editors are sufficient for this application category. The system developer usually does not need (user interface) application framework classes in order to enhance a prototype. Moreover, the finite application model of almost all user interface prototyping tools available today meets the requirements of information systems: it is, for example, not desirable to instantiate an arbitrary number of input masks that are used to enter data into a database.

SUMMARIZING REMARKS

Depending on the level of abstraction of the specification of dynamic behavior we can divide high-level user interface prototyping tools into two categories: tools which support prototyping of information systems and tools that help to reduce the implementation effort if an application framework is used. All tools which are based on the finite automata application model are especially suited for prototyping information systems and thus belong to the first category. Their application model is only a subset of the infinite automata application model of user interface application frameworks. Thus the development of software systems with the infinite application model of user interface application frameworks is not supported.

An example of a tool that belongs to the second category is NeXT Interface Builder. Research (especially in visual programming) is necessary in order to allow the specification of dynamic behavior on an abstraction level higher than that of a programming language and to retain the application model of a state-of-the-art user interface application framework.

REFERENCES

1. Betts B., et al.: Goals and Objectives for User Interface Software; in: Computer Graphics, Vol. 21, No. 2, April 1987.
2. Budde R. et al.: Approaches to Prototyping; in Proceedings of the Working Conference on Prototyping, Namur, October '83, Springer 1984.
3. Floyd, C.: A Systematic Look at Prototyping; in: Approaches to Prototyping, Springer, 1984.
4. Gamma E., Weinand A., Marty R.: Integration of a Programming Environment into ET++: A Case Study; Proceedings of the 1989 ECOOP, July 1989.
5. 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.
6. Holloway S.: Background to Forth Generation; in Fourth Generation Languages and Application Generators, The Technical Press, 1986.
7. Myers B.: User-Interface Tools: Introduction and Survey; IEEE Software, 6(1), January 1989.
8. NeXT, Inc.: 1.0 Technical Documentation: Concepts; NeXT, Inc., Redwood City, CA, 1990.
9. 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.
10. Pree W.: Object-Oriented Versus Conventional Construction of User Interface Prototyping Tools; PhD thesis, Johannes Kepler University of Linz, 1991.
11. 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.
12. 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.
13. Wilson D.A., Rosenstein L.S., Shafer D.: Programming with MacApp; Addison-Wesley, 1990.