DICE
AN OBJECT-ORIENTED TOOL FOR RAPID PROTOTYPING

Wolfgang Pree
Institut für Wirtschaftsinformatik
Johannes Kepler University of Linz
A-4040 LINZ, Austria
E-mail: K2G0190@AEARN

Abstract

This paper deals with object-oriented programming in several ways. On the one hand it presents DICE (**D**ynamic **I**nterface **C**reation **E**nvironment), a tool for rapid prototyping implemented in an object-oriented manner. In particular it 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. On the other hand, the possibilities and limits of currently available application frameworks for prototyping-oriented software development are examined on the basis of DICE's implementation.

Keywords:
prototyping, integration of conventional and object-oriented systems, application frameworks, class libraries, ET++, C++, reusability, extensibility

1 Introduction

The object-oriented programming paradigm gave rise to great hopes of drastically improving the extensibility and reusability of software components. The assertion that in object-oriented programming a linear increase in coding effort can produce an exponential rise in functionality is reinforced by such user interface application frameworks as MacApp [Wil90], AppKit [NeXT89] and ET++ [Wei88, Gam89, Wei89]. Experience has proven that writing a complex application based on frameworks such as those mentioned above can result in a reduction in source code size of 80% and more compared to software written with the support of a conventional graphic toolbox.

Apart from this enormous code reduction, application frameworks offer other important advantages: 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.

DICE was implemented for/with an application framework in order to extend such a tool in the direction of rapid prototyping.

We chose 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]. Furthermore, the author (and implementor of DICE) had the opportunity to work directly with the developers of ET++, E. Gamma and A. Weinand, and thus got to know ET++ intimately.

One of our basic premises was not to change the application framework itself. Only this restriction would allow us to test the suitability of the object-oriented approach to the extension and modification of given software building blocks.

This report portrays DICE as a rapid prototyping tool based on an application framework (Section 2) and describes case studies around the prototyping-oriented design and implementation of DICE's components (Section 3). The suitability of object-oriented programming for prototyping is summed up in the conclusion (Section 4).

2 Prototyping with DICE

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., [Flo84, Jör84, Bisch90]). User Interface Prototyping in

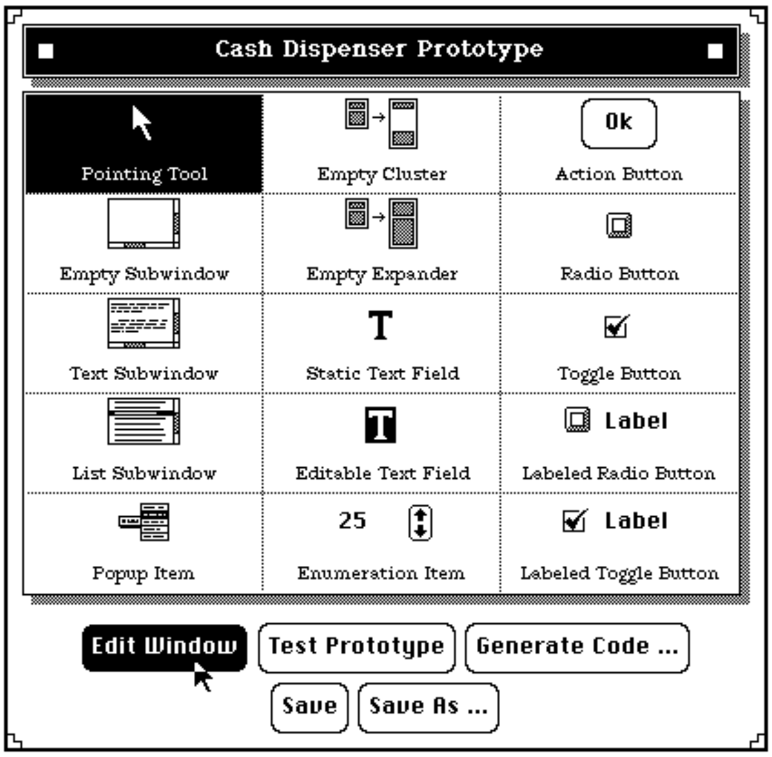


Figure 1. DICE's Control Panel

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 highly interactive 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 graphical specification of the user interface layout (see Section 2.1). In order to enhance the functionality of the prototype, most comparable tools available today provide interfaces to third generation languages or some kind of an integrated third generation language. DICE offers three possibilities for enhancing the functionality (see Section 2.2):

- 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 the UNIX Interprocess Communication mechanism.

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

2.1 Static Layout Aspects

The main building block of a DICE application is a *Window*. When the *"Edit Window"* button in DICE's Control Panel is pressed (see Fig. 1), an empty window is opened. To insert user interface elements, one has to choose the proper element in the control panel and mark the position where the item is to be placed in the window.

Besides the usual interface elements (*Buttons*, *Text Fields*, *Popup Items* and scrollable *Subwindows*, as shown in Fig. 1) there are two elements for grouping interface items – *Cluster* and *Expander*. They allow a comfortable specification of the window layout and of the behavior of the contained elements if a window is resized.

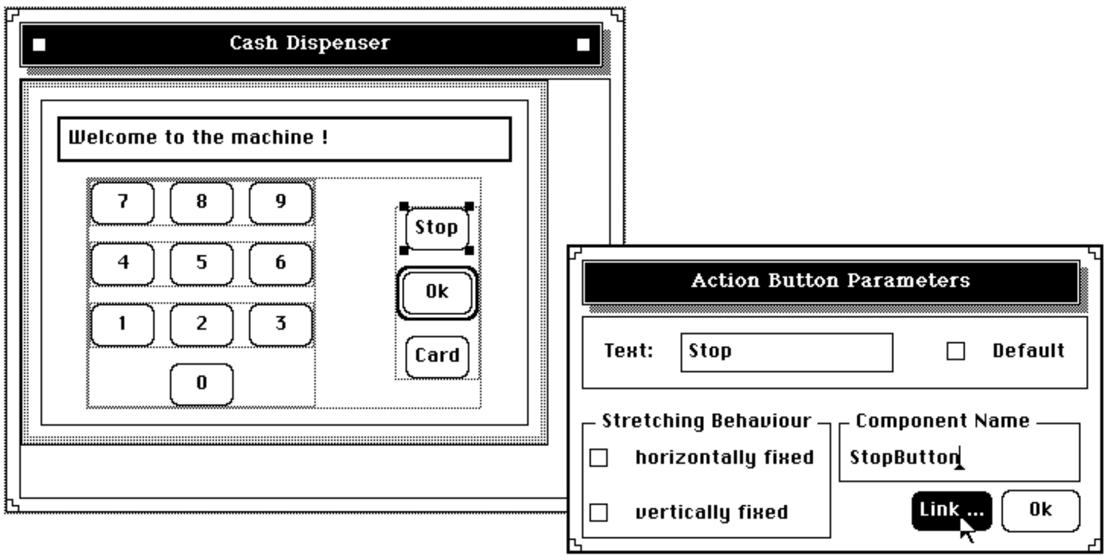


Figure 2. Sample user interface specified with DICE

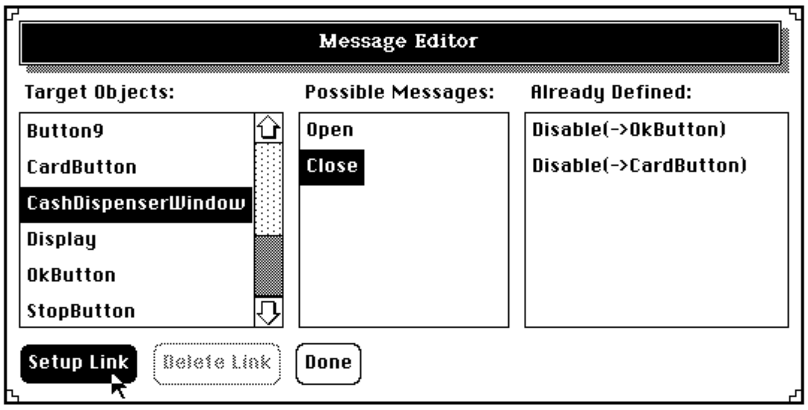


Figure 3. Message Editor

DICE provides undoable editing functions as known from state-of-the-art WYSIWYG editors: moving and resizing of the elements, cut/copy/paste between windows and different DICE prototypes, etc. The parameters of interface elements are defined in specific parameter sheets. Figure 2 shows the parameters for an action button.

2.2 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.

2.2.1 Predefined Messages

Certain messages are assigned to each user interface element (e.g., the messages “Open” and “Close” to a window, the messages “Enable”, “Disable” and “SetText(...)” to a text field, etc.). Let us take the “Cash Dispenser” interface (see Fig. 2) 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 parameter sheet of the “Stop” button (see Fig. 2). By means of DICE’s Message Editor (see Fig. 3), the desired dynamic behavior can then be defined for the “Stop” button (i.e., that the message “Close” is sent to the window “Cash Dispenser” when the “Stop” button is pressed).

From each element that can be activated (*Buttons* and *Menu Items*), any number of messages to other elements can be specified. If you test the prototype (by pressing the “Test Prototype” button in DICE’s Control Panel – see Fig. 1) and activate an interface element, the messages specified for that element are sent to their receivers. They effect the corresponding change in the user interface. Thus rudimentary dynamics can be realized without programming effort.

One of the criteria for judging prototyping tools is their extensibility (e.g., adding new messages to DICE’s interface elements). This aspect is discussed in Section 3.

2.2.2 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 by pressing the “Generate Code” button (Fig. 1). 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 sub-classes of the generated classes by overwriting or extending the corresponding dynamically bound methods (see Fig. 4).

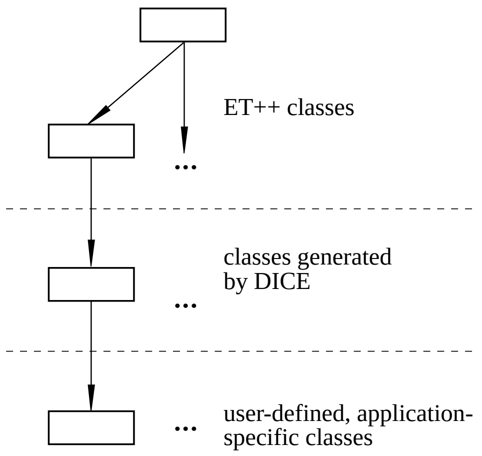

ET++ classes
...
classes generated by DICE
...
user-defined, application-specific classes
...

Figure 4. Code generation

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. Let us look at the cash dispenser interface (Fig. 2) 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 couldn’t 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-defined 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 (this would result in redundant code) or switched between windows of the prototype.

2.2.3 Integration of Conventional and Object-Oriented Systems

Writing applications based on frameworks can result in an enormous reduction in source code size (see Section 1) since they provide powerful building blocks for the kind of applications they support. Unfortunately, the only frameworks available 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]).

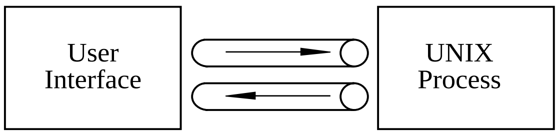


Figure 5. Communication between application and user interface

Communication between the application and user interface (see Fig. 5) is based on a simple protocol: If a user interface element is activated, the name and (if meaningful) its value are sent to the connected process. The value of a *Radio Button*, for instance, is “TRUE” or “FALSE”; an *Action Button* and a *Menu Item* do not send a value. On the other hand, the connected process can ask for the value of each interface element by sending its name and a “?” to the interface.

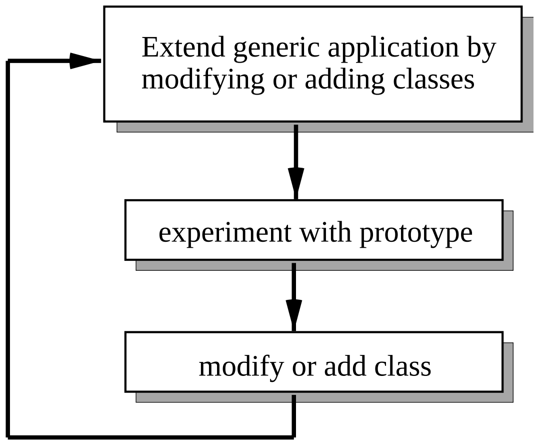
Thus the user interface developed with DICE can be connected with any conventional software system. E.g., the functionality of the cash dispenser depicted in Fig. 2 was implemented in C. Necessary modifications or enhancements of the functionality are implemented in a C program. Immediately after compilation and starting of this program, the modified functionality can be tested together with the user interface prototype 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. [Bisch90], for instance, describes a new paradigm called “Prototyping-Oriented Incremental Software Development”. On the other hand 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.

3 Implementation of DICE

An application framework can be classified as a simple prototyping tool: The abstraction level is raised and the implementation effort can be drastically reduced due to powerful building blocks. If we assume that a software developer is familiar with object-oriented programming and a specific application framework, the prototyping cycle as depicted in Fig. 6 results.

The suitability of an application framework for prototyping depends on the reusability and extensibility of the available building blocks. DICE itself is implemented on the basis of ET++. Short case studies drawn from the implementation of DICE components show that the desired reusability of software components often cannot be achieved even by using state-of-the-art application frameworks or object-oriented programming in general.


Extend generic application by modifying or adding classes
experiment with prototype
modify or add class
Figure 6. Prototyping with an application framework

First the Good News

One of the most important advantages associated with object-oriented software building blocks is that they naturally lend themselves to bottom-up reuse, extension and combination [Mey88, Mey89]. Application frameworks in particular provide a structure in their “empty” application, which is to be extended accordingly for an application that is based on it. Certain extensions are necessary for almost every application. These extensions are prescribed by the framework and forseen by the designer of the framework classes. Thus it is easy to reuse and extend those building blocks.

Generally speaking, classes can be reused and extended without problems if the designer has forseen the wishes of future users. The classes of DICE, for example, have been designed in such a way that the following adaptations can be achieved by means of object-oriented techniques (i.e., without modifying the source code of the tool):

- adding new interface elements,
- modifying the parameters to be defined for an interface element, and
- adding new messages that are “understood” by the interface elements.

Modifications not forseen by the designer can be pretty hard to realize, as the following case studies show.

Then the Bad News

[Mey88] states that reusability does not work automatically for object-oriented software building blocks: Disregarding the economical, political and psychological problems involved, software will only be reused if the potential beneficiaries know about available components. Unfortunately, this is not the only problem, as two examples will demonstrate.

We must note in advance that ET++ is not yet construed by its designers as a finished class library. DICE was implemented on ET++ Version 1.1. The conclusions drawn about application frameworks are not, however, weakened by the argument that ET++ is continually being improved. In our opinion complex software systems will always be inadequate for certain problems. Object-oriented programming is intended to eliminate precisely this weakness of software systems. The general reasons why this is not always possible in a clean way even with the object-oriented programming paradigm can be discerned from the ET++ specific case studies.

Missing Hook Methods

In order to be able to modify the behavior of a class in subclasses, *hook methods* need to be called at the respective locations in the methods of the base classes. By overwriting these dynamically linked methods in subclasses, class behavior can be modified.

The above procedure is possible only at locations where the class designer provided hooks. In order to make a more general extensibility possible, the following (theoretical) solutions could be suggested:

- A hook method is invoked between every pair of statements of a method.
- Every class method consists of exactly one statement.

The idea of a narrow inheritance interface that is usually implemented in application frameworks [Wei89] contradicts these suggestions: Behavior that is spread over several methods in a class should be based on a minimal set of dynamically bound methods. This allows a client deriving subclasses from an existing class to override just a few methods in order to adopt its behavior. Not adhering to this narrow inheritance principle often means that too many methods have to be overridden, resulting in ugly and bulky code.

For extensions that were not foreseen, the source code of the respective superclass must be copied and complemented by the desired statements. This procedure was unavoidable, for example, in the modification of the ET++ class VObjectStretcher (which stands for “Visual Object Stretcher”), which is outlined below:

The ET++ class VObjectStretcher is a subclass of the ET++ class Command and can be used for stretching VObject objects. Thus it serves as a basis for resizing interface elements in DICE. The direct use of this class without adaptation is undesirable for the following reason:

If the size of an interface element has changed, the entire window contents must be recalculated (while the mouse is being moved) in order to maintain the interface as *WYSIWYG*. If the class VObjectStretcher is used directly, only the respective outline rectangle of the interface element to be changed is drawn when the mouse is moved (with mouse button pressed). The window contents itself are recalculated and redrawn only after the mouse button is released.

Thus the class ItemStretcher was created as a subclass of VObjectStretcher in order to implement the behavior described above. There was no dynamically linked hook method in the class VObjectStretcher that is invoked with each mouse movement (with mouse button pressed). In the missing hook method, the desired behavior could have been implemented in two statements.

Since the superclass VObjectStretcher provided no other possibility, the source code of the relatively extensive method TrackMouse was copied into the method TrackMouse of the class ItemStretcher and extended by the necessary statements at the required position. This procedure was only possible because

- the source code of the class to be modified was available; and
- the instance variables used in the method TrackMouse of the class VObjectStretcher can be accessed in their subclasses.

These facts directly contradict the promises of object-oriented programming.

Correcting Framework Errors

In order to correct errors or to eliminate restrictions in the library classes by creating subclasses, the source code is often necessary in order to understand class behavior and find the methods to be modified. The ET++ class MatrixView serves as an example that an error couldn't have been removed without the source code. MatrixView is a basic building block for user interface elements that include a matrix of selectable objects (e.g., menu items, graphically depicted tools in a toolpalette, etc.). If one object is selected, a class method returns the row and column of that object in the matrix.

Since the objects in such lists are usually ordered vertically in only one column, an error in the implementation of the method mentioned above was not detected. The modifications of the class MatrixView by creating a subclass would not have been possible without the source code since the relationships between the class methods could not have been discerned otherwise.

The comprehension of a class to be modified can become more difficult due to the "Yoyo Problem" ([Taen89]). This issue is related to objects sending themselves messages, which may cause the execution of methods up and down the class hierarchy.

4 Conclusion

We are convinced that application frameworks available today are well suited as a basis for object-oriented programming. System design decisions are partially prescribed. A number of object-oriented ideas, as offered in an application framework, significantly reduce the implementation effort for systems.

DICE demonstrates how currently available 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 dialog sheets) were generated with DICE itself. The next step will be the integration of a simple programming environment for C++ so that application specific behavior can be programmed in an object-oriented manner within DICE.

References

[Bisch90]
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.

[Flo84]
Floyd, Ch.: *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.

[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.

[Jör84]
Jörgensen, A.H.: *On the Psychology of Prototyping*; in: Approaches to Prototyping, Springer, 1984, pp. 278-289.

[Mey88]
Meyer B.: *Object-Oriented Software Construction*; Prentice Hall, 1988.

[Mey89]
Meyer B.: *From Structured Programming to Object-Oriented Design: The Road to Eiffel*; in: Structured Programming Vol.10 No.1, Springer 1989.

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

[Taen89]
Taenzer D., Ganti M., Podar S.: *Problems in Object-Oriented Software Reuse*, Proceedings of the 1989 ECOOP, July 1989.

[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.