32

Some O-O techniques for graphical interactive applications

*Famous Designer has recently designed an automobile. It has neither a fuel gauge, nor a speedometer, nor any of the idiot controls that plague other modern cars. Instead, if the driver makes a mistake, a large "***?***" lights up in the middle of the dashboard. "The experienced driver", says Famous, "will usually know what went wrong".*

Unix folklore. (Instead of "*Famous Designer*", the original names one of the principal contributors to Unix.)

*E*legant user interfaces have become a required part of any successful software product. Advances in display hardware, ergonomics (the study of human factors) and software have taken advantage of interaction techniques first pioneered in the seventies: multiple windows so you can work on several jobs, mouse or other fast-moving device so you can show what you want, menus to speed up your choices, icons to represent important notions, figures to display information visually, buttons to request common operations.

The acronym GUI, for Graphical User Interfaces, has come to serve as a general slogan for this style of interaction. Related buzzwords include WYSIWYG (*What You See Is What You Get*), WIMP ("Windows, Icons, Menus, Pointing device") and the phrase "direct manipulation", characterizing applications which give their users the impression that they work directly on the objects shown on the screen.

These impressive techniques, not long ago accessible only to users of a few advanced systems running on expensive hardware, have now become almost commonplace even on the most ordinary personal computers. So commonplace and popular, in fact, that a software developer can hardly expect any success from a product that uses just a line-oriented interface, or even one that is full-screen but not graphical.

Yet until recently the construction of interactive applications offering advanced graphical facilities remained so difficult as to justify what may be called the *Interface Conjecture*: the more convenient and easy an application appears to its users, the harder it will be for its developers to build.

One of the admirable advances of the software field over the past few years has been to start **disproving** the interface conjecture through the appearance of good tools such as interface builders.

More progress remains necessary in this fast-moving area. Object technology can help tremendously, and in fact the fields denoted by the two buzzwords, GUI and O-O, have had a closely linked history. Simply stated, the purpose of this chapter is to disprove the Interface Conjecture, by showing that to be user-friendly an application does not have to be developer-hostile. Object-oriented techniques will help us concentrate on the proper data abstractions, suggest some of these abstractions, and give us the ability to reuse everything that can be reused.

A complete exploration of O-O techniques for building graphical and interactive applications would take a book of its own. The aim of the present chapter is much more modest. It will simply select a few of the less obvious aspects of GUI building, and introduce a few fundamental techniques that you should find widely applicable if your work involves designing graphical systems.

32.1 NEEDED TOOLS

What tools do we need for building useful and pleasant interactive applications?

End users, application developers and tool developers

First, a point of terminology to avoid any confusion. The word "user" (one of the most abused terms in the computer field) is potentially misleading here. Certain people, called **application developers**, will produce interactive applications to be used by other people, to be called **end users**; a typical end user would be a dentist's assistant, using a system built by some application developer for recording and accessing patient history. The application developers themselves will rely, for their graphical needs, on tools built by the third group, **tool developers**. The presence of three categories is the reason why "user" without further qualification is ambiguous: the end users are the application developers' users; but the application developers themselves are the tool developers' users.

An **application** is an interactive system produced by a developer. An end user who uses an application will do so by starting a session, exercising the application's various facilities by providing the input of his choice. Sessions are to applications what objects are to classes: individual instances of a general pattern.

This chapter analyzes the requirements of developers who want to provide their end users with useful applications offering graphical interfaces.

Graphical systems, window systems, toolkits

Many computing platforms offer some tools for building graphical interactive applications. For the graphical part, libraries are available to implement designs such as GKS and PHIGS. For the user interface part, basic window systems (such as the Windows Application Programming Interface, the Xlib API under Unix and the Presentation Manager API under OS/2) are too low-level to make direct use convenient for application developers, but they are complemented by "toolkits", such as those based on the Motif user interface protocol.

All these systems fulfil useful needs, but they do not suffice to satisfy developers' requirements. Among their limitations:

- They remain hard to use. With Motif-based toolkits, developers must master a multi-volume documentation describing hundreds of predefined C functions and structures bearing such awe-inspiring names as *XmPushButtonCallbackStruct* — with the *B* of *Button* in upper-case, but the *b* of *back* in lower-case — or *XmNsubMenuId*. The difficulties and insecurities of C are compounded by the complexity of the toolkit. Using the basic Application Programming Interface of Windows is similarly tedious: to create an application, you must write the application's main loop to get and dispatch messages, a window procedure to catch user events, and other low-level elements.

- Although the toolkits cover user interface objects — buttons, menus and the like — some of them offer little on graphics (geometrical figures and transformations). To add true graphics to the interface is a significant effort.

- The toolkits are incompatible with each other. Motif, the Windows graphics and Presentation Manager, although based on essentially similar concepts, differ in many ways, some significant (in Windows and PM creating a user interface object displays it immediately, whereas under Motif you first build the corresponding structure and then call a "realize" operation to display it), some just a matter of convention (screen coordinates are measured from the top left in PM, from the bottom left in the others). Many user interface conventions also vary. Most of these differences are a nuisance to end users, who just want something that works and "looks nice", and do not care whether window corners are sharp or slightly rounded. The differences are an even worse nuisance to developers, who must choose between losing part of their potential market or wasting precious development time on porting efforts.

The library and the application builder

To answer the needs of developers and enable them to produce applications that will satisfy their end users, we must go beyond the toolkits and provide portable, high-level tools that relieve developers from the more tedious and repetitive parts of their job, allowing them to devote their creativity to the truly innovative aspects.

The toolkits provide a good basis, since they support many of the needed mechanisms. But we must hide their details and complement them with more usable tools.

The basis of the solution is a library of reusable classes, supporting the fundamental data abstractions identified in this chapter, in particular the notions of window, menu, context, event, command, state, application.

For some of the tasks encountered in building an application, developers will find it convenient to work not by writing software texts in the traditional fashion, but by relying on an interactive system, called an application builder, which will enable them to express their needs in a graphical, WYSIWIG form; in other words, to use for their own work the interface techniques that they offer to their users. An application builder is a tool whose end users are themselves developers; they use the application builder to build the parts of their systems that may be specified visually and interactively. The term "application builder" indicates that this tool is far more ambitious than plain "interface builders", which only cover the user interface of an application. Our application builder must go further into expressing the structure and semantics of an application, stopping only where software text becomes the only reasonable solution.

In defining the library and the application builder, we should be guided, as always, by the criteria of reusability and extendibility. This means in particular that for every data abstraction identified below (such as context, command or state) the application builder should provide two tools:

- For reusability, a **catalog** (event catalog, context catalog, state catalog…) containing predefined representatives of the abstraction, which developers can include directly into their applications.
- For extendibility, an **editor** (context editor, command editor, state editor…) enabling developers to produce their own variants, either from scratch or more commonly by pulling out an element from a catalog and then modifying it.

Using the object-oriented approach

In the object-oriented approach to software construction, the key step is to find the right data abstractions: the types of objects which characterize applications in the given area.

To advance our understanding of graphical user interfaces and devise good mechanisms for building applications, we must explore the corresponding abstractions. Some are obvious; others will prove more subtle.

Each of the abstractions encountered below will yield at least one class in the library. Some will yield a set of classes, all descending from a common ancestor describing the most general notion. For example, the library includes several classes describing variants of the notion of menu.

We will first examine the overall structure of a portable graphics library; then consider the main graphical abstractions covering the geometrical objects to be displayed, and the "interaction objects" supporting event-driven dialogues; finally we will study the more advanced abstractions describing applications: command, state, application itself.

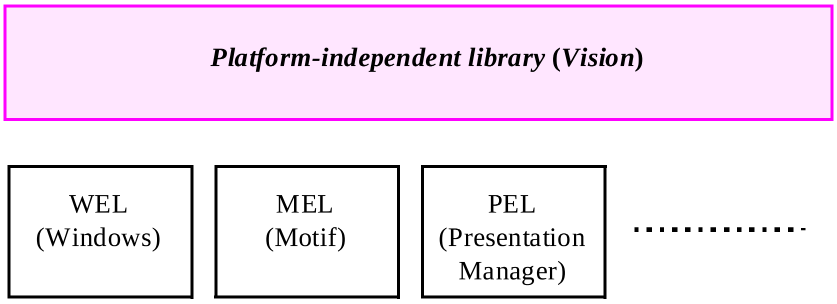
32.2 PORTABILITY AND PLATFORM ADAPTATION

Some application developers want a portable library, which will enable them to write a single source text that will then adapt automatically to the look-and-feel of many platforms, at the price of a recompile but without any change. Others want the reverse: to gain full access to all the specific "controls" and "widgets" of a particular platform such as Microsoft Windows, but in a convenient fashion (rather than at the typically low level of the native libraries). Yet others want a bit of both: portability as the default, but the ability to go native when needed.

With a careful design, relying on a two-layer structure, we can try to satisfy all of them:

Graphical libraries architecture

(*See a similar archi tecture for concur-rency: page 970.*)



To make things more concrete the figure shows the names of the corresponding components in ISE's environment, but the idea is applicable to any graphical library. At the top level (*Vision*) there is a portable graphical library; at the bottom level you find specialized libraries, such as *WEL* for Windows, adapted to one platform only.

See "AN APPLICATION: THE HANDLE TECHNIQUE", 24.3, page 817.

WEL and other bottom-level libraries can be used directly, but they also serve as the platform-dependent component of the top level: *Vision* mechanisms are implemented through WEL on Windows, MEL on Motif and so on. This technique has several advantages: for the application developers, it fosters compatibility of concepts and techniques; for the tool developers, it removes unneeded duplications, and facilitates the implementation of the top level (which relies on clean, abstract, assertion-equipped and inheritance-rich O-O libraries such as WEL, rather than interfacing directly with the C level, always a dangerous proposition). The connection between the two levels relies on the *handle* design pattern developed in an earlier chapter.

Application developers have a choice of level:

- If you want to ensure portability, use the higher layer. This is also of interest to developers who, even if they work for a single platform, want to benefit from the higher degree of abstraction provided by high-level libraries such as *Vision*.
- If you want to have direct access to all the specific mechanisms of a platform (for example the many "controls" provided by Windows NT), go to the corresponding lower-layer library.

The last comment touches on a delicate issue. How much platform-specific functionality do you lose by relying on a portable library? The answer is necessarily a tradeoff. Some early portable libraries used an *intersection* (or "lowest common denominator") approach, limiting the facilities offered to those that were present in native form in all the platforms supported. This is usually not enough. At the other extreme the library authors might use the *union* approach: provide every single mechanism of every supported platform, using explicit algorithms to simulate the mechanisms that are not natively available on a particular platform. This policy would produce an enormous and redundant library. The answer has to be somewhere in-between: the library authors must decide individually, for every mechanism present on some platforms only, whether it is important enough to warrant writing a simulation on the other platforms. The result must be a consistent library, simple enough to be used without knowledge of the individual platforms, but powerful enough to produce impressive visual applications.

For application developers, one more criterion in choosing between the two layers is performance. If your main reason for considering the top layer is abstraction rather than portability, you must be aware that including the extra classes will carry a space penalty (any time penalty should be negligible with a well-designed library), and decide whether it is worthwhile. Clearly, a one-platform library such as WEL will be more compact.

Finally, note that the two solutions are not completely exclusive. You can do the bulk of your work at the top level and provide some platform-specific goodies to users working on your top-selling platform. This has to be done carefully, of course; carelessly mixing portable and non-portable elements would soon cancel any expected benefits, even partial, of portable development. An elegant design pattern (which ISE has applied to some of its libraries) relies on assignment attempt. The idea is this. Consider a graphical object known through an entity *m* whose type is at the top level, say *MENU*. Any actual object to which it will become attached at run time will be, of course, platform-specific; so it will be an instance of a lower-layer class, say *WEL_MENU*. To apply platform-specific features you need an entity, say *wm*, of this type. You can use the following scheme:

```
wm ?= m
if wm = Void then
    … We are not on Windows! Do nothing, or something else …
else
    … Here we may apply any WEL_MENU (i.e. Windows-specific)
      feature to wm …
end
```

We can picture this scheme as a way to go into the Windows-only room. The room is locked, to prevent you from claiming, if someone finds you there, that you just wandered into it by accident. You are permitted to enter, but you must ask for the key, explicitly and politely. For such official and conditional requests to enter a special-purpose area, the key is assignment attempt.

32.3 GRAPHICAL ABSTRACTIONS

Many applications will use graphical figures, often representing objects from an external system. Let us see a simple set of abstractions that will cover this need.

Figures

First we need a proper set of abstractions for the graphical part of an interactive application. To keep things simple, this discussion will assume two-dimensional graphics.

Geographical maps provide an excellent model. A map (of a country, a region, a city) provides a visual representation of some reality. The design of a map uses several levels of abstraction:

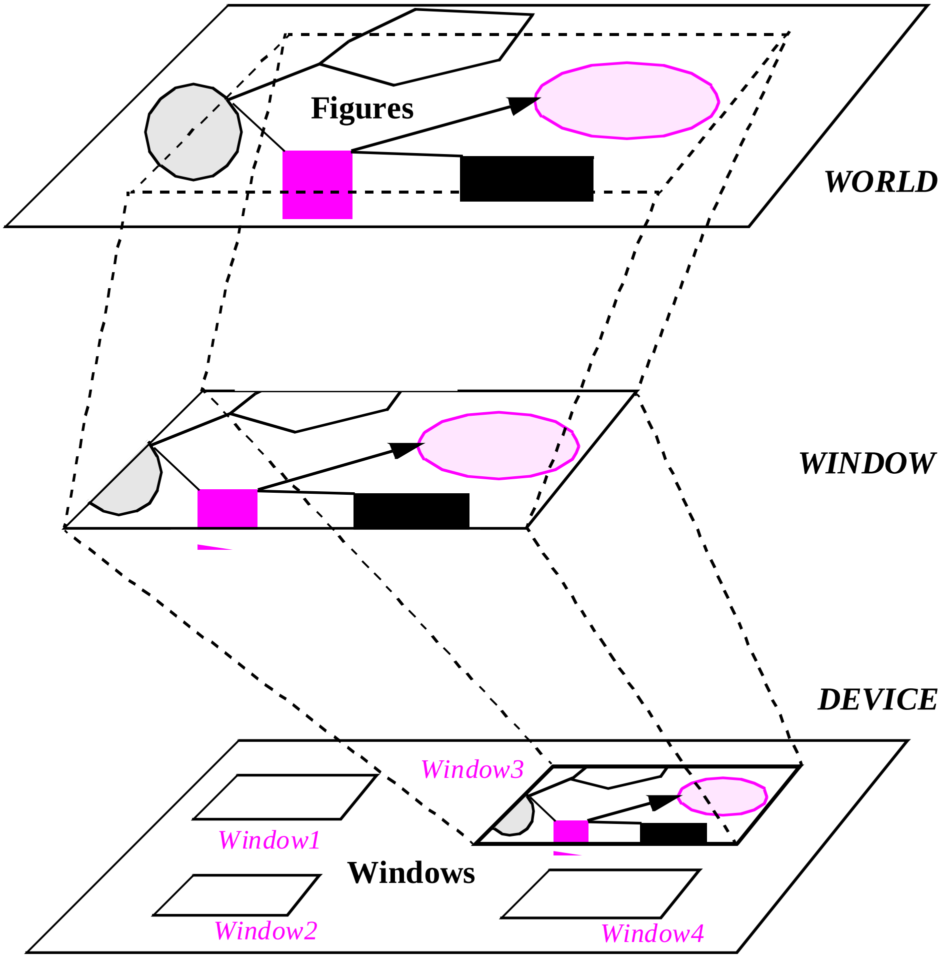
- We must view the reality behind the model (in an already abstracted form) as a set of geometrical shape or **figures**. For a map the figures represent rivers, roads, towns and other geographical objects.

- The map will describe a certain set of figures, which may be called the **world**.

- The maps will show only a part of the world — one or more areas which we will call **windows**, and assume to be rectangular. For example a map can have one main window devoted to a country, and subsidiary windows devoted to large cities or outlying parts (as with Corsica in maps of France or Hawaii in maps of the USA).

- Physically the map appears on a physical display medium, the **device**. The device is usually a sheet of paper, but we may also use a computer screen. Various parts of the device will be devoted to the various windows.

The graphical abstractions



The four basic concepts — *WORLD*, *FIGURE*, *WINDOW*, *DEVICE* — transpose readily to general graphical applications, where the world may contain arbitrary figures of interest to a certain computer application, rather than just representations of geographical objects. Rectangular areas of the world (windows) will be displayed on rectangular areas of the device (the computer screen).

The figure on the previous page shows the three planes: world (bottom), window (middle) and device (top). The notion of window plays a central role, as each window is associated both with an area of the world and with an area of the device. Windows also cause the only significant extension to the basic map concepts: support for hierarchically nested windows. Our windows will be permitted to have subwindows, with no limit on the nesting level. (No nesting appears in the figure.)

Coordinates

We need two coordinate systems: device coordinates and world coordinates. Device coordinates measure the positions of displayed items on the device. On computer screens, they are often measured in pixels; a pixel (picture element) is the size of a small dot, usually the smallest displayable item.

There is no standard for the unit of world coordinates, and there should not be since the world coordinate system is best left for application developers to decide: an astronomer may wish to work in light years, a cartographer in kilometers, a biologist in millimeters or microns.

Because a window captures part of a world, it will have a certain world position (defined by the *x* and *y* world coordinates of its top left corner) and a certain extent (horizontal and vertical lengths of the parts of the world covered). The world position and the extent are expressed in world coordinate units.

Because the window is displayed on part of a device, it has a certain device position (defined by the *x* and *y* device coordinates of its top left corner) and a certain size on the device, all expressed in device coordinate units. For a window with no parent, the position is defined with respect to the device; for a subwindow, the position is always defined relative to the parent. Thanks to this convention, any application that uses windows may run with the whole screen to itself as well as in a previously allocated window.

Operations on windows

To take care of the hierarchical nature of windows we make class *WINDOW* an heir of class *TWO_WAY_TREE*, an implementation of trees. As a result, all hierarchical operations are readily available as tree operations: add a subwindow (child), reattach to a different enclosing window (parent) and so on. To set the world and device positions of a window, we will use one of the following procedures (all with two arguments):

	Set absolute position	Move, relative to current position
Position in world	*go*	*pan*
Position on device	*place_proportional* *place_pixel*	*move_proportional* *move_pixel*

The *_proportional* procedures interpret the values of their arguments as fractions of the parent window's height and width; arguments to the other procedures are absolute values (in world coordinates for *go* and *pan*, in device coordinates for the *_pixel* procedures). Procedures are similarly available to set the extent and size of a window.

Graphical classes and operations

All classes representing figures are descendants of a deferred class *FIGURE*; standard features include *display*, *hide*, *translate*, *rotate*, *scale*.

It is indispensable to keep the set of figure types extendible, allowing application developers (and, indirectly, end users of graphical tools) to define new types. We have seen how to do this: provide a class *COMPOSITE_FIGURE*, built by multiple inheritance from *FIGURE* and a container type such as *LIST* [*FIGURE*].

32.4 INTERACTION MECHANISMS

Let us now turn our attention to how our applications will interact with users.

Events

Modern interactive applications are **event-driven**: as the interactive user causes certain events to occur (for example by entering text at the keyboard, moving the mouse or pressing its buttons), certain operations get executed.

Innocuous as this description may seem, it represents a major departure from more traditional styles of interaction with users. In the old style (which is still by far the most common), a program that needed input from its user would get it by repeatedly executing scenarios of the form

```
… Perform some computation …
print ("Please type in the value for parameter xxx.")
read_input
xxx := value_read
… Proceed with the computation, until it again needs a value from the user …
```

In the event-driven style, roles are reversed: operations occur not because the software has reached a preset stage of its execution, but because a certain event, usually triggered by the interactive user, has caused execution of a certain component of the software. Input determines the software's execution rather than the reverse.

The object-oriented style of software development plays an important role in making such schemes possible. Dynamic binding, in particular, enables the software to call a feature on an object under the understanding that the form of the object will determine how it will handle the feature. The feature may be associated with an event and the object to a command; more on this below.

The notion of event is important enough in this discussion to yield a data abstraction. An event object (instance of the *EVENT* class) will represent a user action; examples are key press, mouse movement, mouse button down, mouse button up. These predefined events will be part of the event catalog.

In addition, it must be possible to define custom events, which a software component may send explicitly by a procedure call of the form *raise* (*e*).

Contexts and user interface objects

GUI toolkits offer a number of predefined "User Interface Objects": windows, menus, buttons, panels. Here is a simple example, an OK button.

A button

Superficially, a user interface object is just a figure. But unlike the figures seen above it usually has no relation with the underlying world: its role is limited to the handling of user input. More precisely, a user interface object provides a special case of **context**.

To understand the need for the notion of context, we must remember that an event generally does not suffice to determine the software's response. Pressing a mouse button, for example, will give different results depending on where the mouse cursor is. Contexts are precisely those conditions which determine the responses that an application associates with events.

In general, then, a context is simply a boolean value — a value which will be true or false at any instant of the software's execution.

The most common contexts are associated with user interface objects. A button such as the one above defines the boolean condition "is the mouse cursor inside the button?", a context. Contexts of this kind will be written *IN* (*uio*), where *uio* is the user interface object.

For every context *c* its negation *not c* is also a context; *not IN* (*uio*) is also called *OUT* (*uio*). The context *ANYWHERE* is always true; its negation *NOWHERE* is never true.

Our application builder should then have a context catalog, which will include *ANYWHERE* and contexts of the form *IN* (*uio*) for all commonly useful interface objects *uio*. In addition, we may wish to enable application developers to define their own contexts; the application builder will provide a context editor for this purpose. Among other facilities, the context editor makes it possible to obtain *not c* for any *c* (in particular a *c* from the catalog).

32.5 HANDLING THE EVENTS

We now have the list of events, and the list of contexts in which these events may be significant. We must describe what to do as a response to these events. The responses will involve *commands* and *transition labels*.

Commands

"Command as a class", page 699.

Recognizing the notion of command as an important abstraction is a key step in producing good interactive applications.

This notion was studied as part of the Undoing case study. As you remember, a command object represents the information needed to execute a user-requested operation and, if undoing is supported, cancel it.

To the features defined in the earlier discussion, we will add the attribute *exit_label*, explained below.

Basic scheme

With contexts, events and commands we have the basic ingredients to define the basic operation of an interactive application, which our application builder should support: an application developer will select the valid context-event combinations (which events are recognized in which contexts) and, for every one of them, define the associated command.

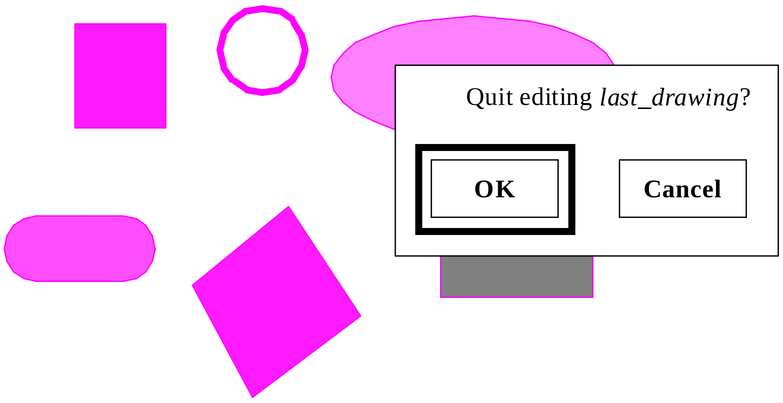
This basic idea can provide the first version of an application builder. There should be catalogs of contexts and events (based on the underlying toolkit) as well as commands (provided by the development environment, and available for application developers to extend). A graphical metaphor should make it possible to select a context-event combination, for example left-click on a certain button, and select a command to be executed in response.

States

For a fully general scheme we should include an extra level of abstraction, giving the **Context-Event-Command-State** model of interactive graphical applications.

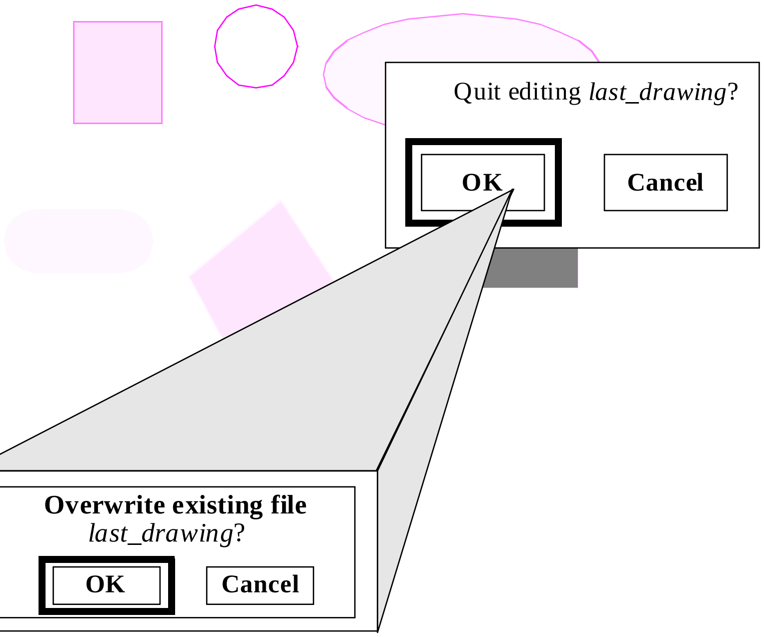
In an application a given context-event combination does not always have the same effect. For example, you might find yourself during a session in a situation where part of the screen looks like this:

An exit command



In this state the application recognizes various events in various contexts; for example you may click on a figure to move it, or request the Save command by clicking on the OK button shown. If you choose this latter possibility, a new panel appears:

Confirming a command


Quit editing *last_drawing*?
OK
Cancel
Overwrite existing file *last_drawing*?
OK
Cancel

At this stage only two context-event combinations will be accepted: clicking on the "OK" or on the "Cancel" button of the new panel. All others have been disabled (and the application has dimmed the rest of the figure as a reminder that everything but the two buttons is temporarily inactive). What happened is that the session has entered a new state of the application. States, also called *modes*, are a familiar notion in discussions of interactive systems, but are seldom defined precisely. Here we have the seeds for a formal definition: a state is characterized by a set of acceptable context-event combinations and a set of commands; for each context-event combination, the state defines the associated command. This will be restated as a mathematical definition below.

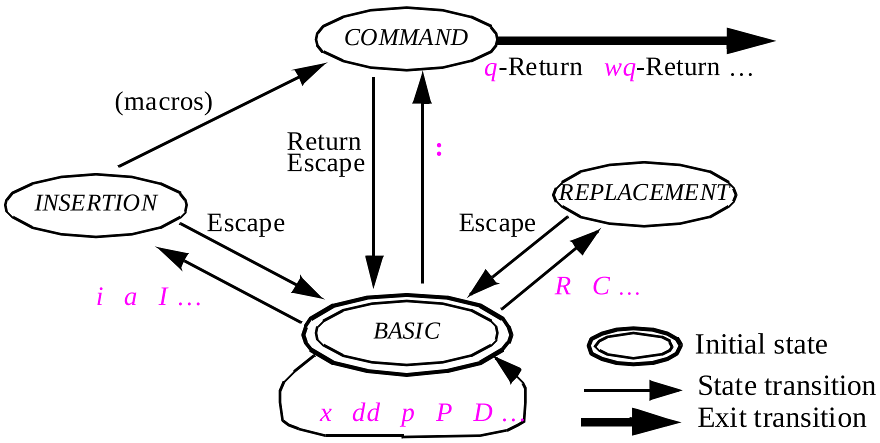
Many interactive applications, graphical or not, will have several states.

A typical example is the well-known *Vi* editor under Unix. Since this tool is not graphical, events are simply key presses (each keyboard key triggering a different event) and the contexts are various possible cursor positions (under a character, at beginning of line, at end of line etc.). A rough analysis of Vi indicates at least four states:

- In the basic state (which is also the initial one for an end user who calls the editor on a new or existing file), typing a letter key will, in most cases, directly execute a command associated with the letter. For example, typing *x* deletes the character at cursor position, if any. Some keys cause a transition to another state; for example typing a colon **:** leads to the command state, typing *i* leads to the insertion state, and typing *R* leads to the replacement state. Some letters cause unaccepted events; for example (unless it has been expressly defined as a macro) the letter *z* has no effect.

- In the command state, only one is available, at the bottom of the Vi window; it serves to enter commands such as "save" or "restart".
- In the insertion state, any key corresponding to a printable character is acceptable as event; the corresponding character will be inserted into the text, causing displacement of any existing text to its right. The ESCAPE key gets the session back to basic state.
- Replacement state is a variant of insertion state in which the characters that you type overwrite rather than displace the ones already in place.

Partial state diagram for Vi



The article was in the special Smalltalk issue of Byte, [Goldberg 1981].

The literature on user interfaces is critical of states because they can be confusing to users. An early article on the Smalltalk user interface contained a picture of the article's author wearing a T-shirt that read "Don't mode me in!". It is indeed a general principle of sound user interface design to ensure that at every stage of a session end users should have as many commands as possible at their disposal (instead of having to change state before they can execute certain important commands).

In accordance with this principle, a good design will try to minimize the number of states. The principle does not mean, however, that this number should always be one. Such an extreme interpretation of the "don't mode me in" slogan could in fact decrease the quality of the user interface, as too many unrelated commands available at the same time may confuse end users. Furthermore, there may be good reasons to restrict the number of commands in a certain situation (for example when the application needs an urgent response from its end user).

States, in any case, should be explicit for the developers, and usually for the end users as well. This is the only way to enable developers to apply the user interface policy of their choice — whether of the strongly anti-modal persuasion or more tolerant.

So our application builder will provide developers with an explicit *STATE* abstraction; as for the other abstractions, there will be a state catalog, containing states that have proved to be of general use, and a state editor, enabling developers to define new states, often by modifying states extracted from the catalog.

Applications

The last major data abstraction is the notion of application.

All the previous abstractions were intermediate tools. What developers really want to build is applications. A text processing system, an investment banking system, a factory control system will be examples of applications.

To describe an application, we need a set of states, transitions between these states, and the indication of which state is the initial one (in which all sessions will begin). We have seen that a state associates a certain response with every accepted context-event pair; the response, as noted, includes a command. To build complete applications, we may also need to include in a response some indication of the context-event pair which led to the response, so that different combinations may trigger transitions to different states. Such information will be called a transition label.

With states and transition label we may build the transition diagram describing an entire applications, such as the partial diagram for Vi shown on the preceding page.

Context-Event-Command-State: a summary

The abstractions just defined can serve as the basis for a powerful interactive application builder — not just an *interface* builder, but a tool that enables application developers to build entire applications graphically; they will explore visual catalogs of contexts, events, and, most importantly commands; selecting the desired elements graphically, they will build the desired context-event-command associations through a simple drag-and-drop mechanism until they have a complete application.

Because simple applications can often rely on just one state, the application builder should make the notion of state should as unobtrusive as possible. More advanced applications, however, should be able to use as many states as they need, and (if only for interface consistency) to derive a new state incrementally from an existing one.

32.6 A MATHEMATICAL MODEL

Some of the concepts presented informally in this chapter, in particular the notion of state, have an elegant mathematical description based on the notion of *finite function* and the mathematical transformation known as *currying*.

Because these results are not used in the rest of the book, and mostly of interest to readers who like to explore the mathematical models of software concepts, the corresponding sections are not printed here but appear in electronic form in the CD-ROM accompanying this book, as a supplementary chapter entitled "mathematical background", an extract from [M 1995e].

32.7 BIBLIOGRAPHICAL NOTES

The ideas for an application builder sketched in this chapter derive largely from ISE's *Build* application builder, described in detail in [M 1995e], which also discusses in detail the underlying mathematical model. (This is the manual from which the extra chapter on the CD-ROM was extracted.)