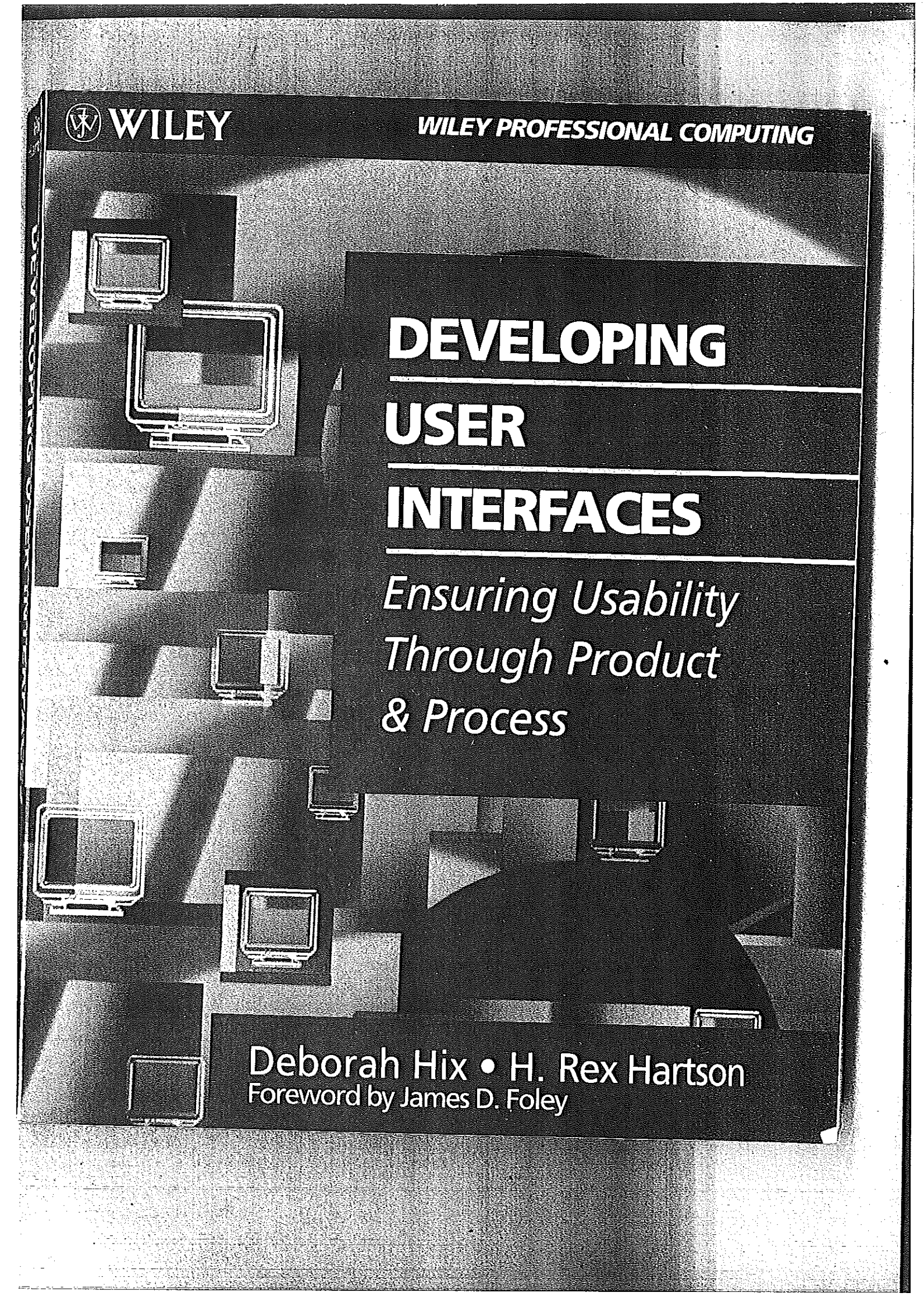
WILEY
WILEY PROFESSIONAL COMPUTING
DEVELOPING USER INTERFACES
Ensuring Usability Through Product & Process
Deborah Hix • H. Rex Hartson
Foreword by James D. Foley

6

Techniques for Representing User Interaction Designs

6.1 DESIGN REPRESENTATION AS A DEVELOPMENT ACTIVITY

Good user interaction designs invariably depend on an ability to understand and evaluate—and thereby improve—those designs during the development process. Understanding and evaluating designs depend, in part, on the techniques used to represent the designs. Design and representation are very closely related. *Design* is a creative, mental, problem-solving process; *representation* is the physical process of capturing or recording a design. *Design representations* are the means by which interaction designs are communicated and documented.

The need for effective design representation techniques is especially important with new interaction development methods that emphasize iterative refinement and involve *a multiplicity of separate but cooperating roles* for producing the interface. As already discussed (in Chapter 1), these roles include at least the following: designer, implementer, problem domain expert, evaluator, documentation specialist, marketer, customer, and user. There are also roles for developing the noninterface parts of an interactive system.

Each of these roles has its own, often different, *needs for communicating*—recording, conveying, reading, and understanding—an interaction design. This communication need translates into the need for a common set of *interaction design representation techniques*—the mechanism for completely and unambiguously capturing an interaction design as it evolves through all phases of the life cycle. Our main interest in design representation here is as a means for designers of the interaction component of a user interface to communicate their designs among all developers, and particularly to user interface software designers and implementers.

6.2 THE NEED FOR BEHAVIORAL REPRESENTATION

In software engineering, the distinction of design from implementation is an established principle. Chapter 1 established a similar distinction in user interface development, motivating the need for a behavioral domain in which to carry out and discuss the interaction development process, addressing user interaction distinctly from implementation (software) considerations. This chapter and the next introduce a design representation technique capable of specifying the *behavioral design* of an interactive system—the tasks and the actions a user performs to accomplish those tasks. Often, communication within a development group is informal, based mostly on implicit corporate culture. Typically, as Grudin (1991) claims, "for communicating a design, paper has been the preferred medium and brevity a preferred style." This book presents a more formal, tool-supportable design representation as a record of the design for all development roles.

Most representation techniques currently used for interface software development (e.g., state transition diagrams, event-based mechanisms, object orientation) are constructional—and properly so. Any technique that can be thought of as describing interaction from the system viewpoint is constructional. For example, a state transition diagram depicting interaction control flow represents the system view, looking out at the user and waiting for an input. State transition diagrams can also show the current system state and how each input takes the system to a new state. Constructional representation techniques support the designer and implementer of the interface software, but do not support design of the interaction part of the interface.

In the behavioral domain, an interaction developer gets away from software issues and (as discussed in Chapter 5) into the activities that precede software design, such as task analysis, functional analysis, task allocation, and user modeling. Consequently, behavioral representation techniques and supporting tools tend to be user centered and task oriented.

With the current emphasis on user-centered design, the interaction develop-

ment process is driven heavily by user requirements and task analysis. Early evaluation of designs is based on user- and task-oriented models. In fact, the entire interaction development life cycle is becoming centered around the evaluation of users performing tasks. Thus, the *user task* has become a common element of interest among developer roles in the behavioral domain. The set of tasks that forms an interaction design becomes a set of requirements for the design and implementation of user interface software. A formal representation of interaction designs is, therefore, needed to convey these requirements.

Behavioral representation techniques are not replacements for constructional techniques; they just support a different domain. Each behavioral design must be translated into a constructional design that is the computer system's view of how the behavior is to be supported. *Behavioral design and representation* involve physical and cognitive user actions and interface feedback—that is, the behavior both of the user and of the interface as they interact with each other. This is the subject of this chapter and the next. Before describing this though, we briefly discuss some related existing representation techniques in the following section. If you are not interested in this related work, you can skip directly ahead to Section 6.4.

6.3 SOME EXISTING REPRESENTATION TECHNIQUES

This section first presents some representation techniques that are used in the constructional development domain and then discusses some that are used in the behavioral domain.

6.3.1 Constructional Representation Techniques for Sequential Interaction

In a *sequential interaction style*, control moves in a predictable way from one part of the dialogue to another. An example is navigation through a menu network. In sequential interaction, each user action leads to a predetermined successor menu, screen, window, or dialogue box.

State transition diagrams have long been used as a graphical representation of interaction control flow in sequential interaction (e.g., Wasserman & Shewmake, 1985; Jacob, 1986). *Nodes* represent interface states or screens; *arcs* represent state transitions based on inputs. Because the nodes of these diagrams usually do not represent interface feedback or screen appearance directly, the content of a node, a representation of what happens within that state, is often described in some other form, such as an *interface representation language*. These are high-level specialized programming languages with constructs especially for representing interaction design.

6.3.2 Constructional Representation Techniques for Asynchronous Interaction

Most interfaces today include both sequential and asynchronous styles. In an *asynchronous interaction style*, many tasks are available to a user at one time, and sequencing within each task is independent of sequencing within other tasks. Asynchronous interaction can be more difficult to represent than strict sequential interaction.

One kind of constructional representation technique especially for asynchronous interaction is event handlers (e.g., Green, 1985). With these, each user action or input is viewed by the system as an event and is sent to the appropriate *event handler*, a software routine associated with a class of events that can cause output, change the system state, and call computational routines.

Concurrent state diagrams (e.g., Jacob, 1986) are also used as constructional representations for control flow in asynchronous interaction. A mutually asynchronous set of state diagrams represents the interface, offering the graphical advantage of a diagrammatic approach but avoiding the complexity of a single large diagram. Multiple diagrams are active simultaneously, with control being transferred back and forth among the diagrams, in coroutine fashion.

Another popular method for constructional representation of asynchronous interaction is based on an *object-oriented approach* (e.g., Sibert, Hurley, & Bleser, 1988). This is a very good fit to today's interaction styles because the behavior of interaction objects is naturally event driven and asynchronous. In addition, high-level object classes contain default behavior, yet they can be specialized for specific behavior. However, a drawback is that the object-oriented approach distributes flow of control, and as a result, it is difficult to understand or trace the sequencing.

Other work has involved specifying *interaction by demonstration* (e.g., Peridot—Myers, 1988). This is a novel and creative approach and is very suitable for producing rapid prototypes. However, it produces only program code, with no behavioral representation of the interaction that can be analyzed.

Another approach, the User Interface Development Environment, or UIDE (Foley, Gibbs, Kim, & Kovacevic, 1988), involves building a *knowledge base* consisting of objects, attributes, actions, and pre- and post-conditions on actions that form a declarative description of an interface. From these descriptions, alternative interfaces are generated for the same underlying functionality.

6.3.3 Behavioral Representation Techniques for Asynchronous Interaction

The representation techniques discussed so far are all constructional, taking the system view of an interface—the view of the user interface software designer and implementer. Stop for a minute and think about how you write down your user

interaction designs. You may describe the interaction objects and widgets and how they work, but how do you describe what the *user* does? Do you use just prose descriptions for this, perhaps with some state diagrams to keep track of major state changes? And, of course, you use screen pictures. What about behavioral ways of representing how it all works? This requires a task-oriented approach because tasks are how the user sees an interface.

The next section introduces a technique for representing user interaction designs from the behavioral view of the user: the *User Action Notation*, or *UAN*. Before describing this technique, however, we will briefly mention some other techniques used in the behavioral domain and the intended users of these techniques.

The existence of these other behavioral techniques might cause you to wonder why we are introducing yet another behavioral representation technique. Design of interactive systems, as with most kinds of design, involves an alternation of analysis and synthesis activities (Hartson & Hix, 1989). Most of these task-oriented models were originally oriented toward analysis; that is, they were not intended to capture a design as it is being created, but rather to build a detailed representation of an existing design with the purpose of predicting user performance for evaluating usability. In contrast, the UAN directly supports synthesis. That is, the UAN addresses the creative mental act of problem solving (i.e., creating new interaction designs) and the physical act of capturing (i.e., documenting) a representation of the design.

Some existing behavioral techniques that are user-task-oriented include the Goals, Operators, Methods, and Selection (GOMS) model (Card, Moran, & Newell, 1983), the Command Language Grammar (CLG) (Moran, 1981), the keystroke-level model (Card & Moran, 1980), the Task Action Grammar (TAG) (Payne & Green, 1986), and the work by Reisner (1981), and Kieras and Polson (1985).

In practice, most of these techniques can be used to support synthesis, as well as analysis, but typically, they cannot represent the direct association of feedback and state with user actions. Also, many of these models—the GOMS, CLG, and keystroke, in particular—are models of expert error-free task performance in contiguous time (without interruption, interleaving of tasks, and without considering interrelationships of concurrent tasks); these are not suitable assumptions for the synthesis-oriented aspects of interaction design.

On the other hand, the GOMS model is very important to task analysis for interaction design. However, although the amount of detail generated in a GOMS description of interaction allows for thorough analysis, it can be an enormous undertaking to produce. GOMS and the UAN have similarities, especially at higher levels of abstraction, where tasks are described in terms of subtasks. Given this brief description of several other behavioral representation techniques and their uses, the rest of this chapter and the next chapter discusses the UAN in detail.

6.4 INTRODUCING THE USER ACTION NOTATION (UAN)

The UAN was created within the Human–Computer Interaction Project at Virginia Tech (Hartson, Siochi, & Hix, 1990). Our friend and colleague, Antonio Siochi, is the originator of the UAN. Our project had a contract to develop a software tool, and Anton, the interaction designer for the tool, grew tired of struggling with the imprecision and verbosity of prose descriptions of how the interface should behave. In addition, our implementers grew tired of reading and trying to understand his prose descriptions. Out of this need arose the behavioral representation technique we called the User Action Notation, or UAN. Since its creation, many people, both at Virginia Tech and elsewhere, have contributed to extending and formalizing the UAN. So far, the UAN has been used by more than 50 interaction developers and researchers outside Virginia Tech.

6.4.1 Why Use UAN?

Before describing details of the UAN, it is important to explain why on earth such a level of detail is needed for an interaction design. Without such an explanation, it may not seem worth all the effort it will take to produce UAN descriptions. Much earlier, in Chapter 1 we espoused different roles for different activities in the user interface development process. We especially emphasized the different skills, attitudes, and perspectives needed to do design as compared to those needed to do implementation of a user interface. *The UAN is intended to be written primarily by someone designing the interaction component of an interface, and to be read by all developers, particularly those designing and implementing the user interface software.*

The UAN reader is, therefore, usually a software engineer or programmer. As discussed in Chapter 1, programmers probably should not be doing interaction design unless they are working closely with people who are trained specifically in interaction design; if so, then they can use the UAN to record their joint decisions. In most cases, however, by the time a programmer writes code, design decisions for the interaction component should already be made, in detail, and the programmer should simply implement that design. If details are omitted from a design, a programmer will have to make some guess about what the designer intended or will have to find the designer and ask, neither of which is a desirable situation. A technique such as the UAN is an appropriate mechanism for recording that design precisely, concisely, unambiguously, and in detail, so that decisions about interaction design are not left up to a programmer.

Another reason is sometimes given as to why the level of detail offered by the UAN isn't needed. In particular, people may say, "We've all worked together for

so long, we don't need to record this kind of detail. We just know what every one else is thinking." Members of a user interface development team shouldn't believe in mind reading. Moreover, what happens when a new person joins the team? It normally takes someone from three months to a year to become acclimatized to a new environment, to pick up the vibes from everyone else, and to know what others are thinking. What happens to that new person (and other team members with whom the new person has to communicate) in the meantime? User interaction designers should not rely on corporate culture and philosophy as a communication technique for interaction designs. A structured approach such as the UAN can maximize the communication bandwidth among development team members as an interaction design evolves and can minimize misunderstandings and confusion about that design.

6.4.2 Overview of the UAN

The UAN is a user- and task-oriented notation that describes physical (and other) behavior of the user and interface as they perform a task together. The primary abstraction of the UAN is a *user task*. An interface is represented as a quasi-hierarchical structure of asynchronous tasks, the sequencing within each task being independent of that in the others. User actions, corresponding interface feedback, and state change information are represented at the lowest level. Levels of abstraction hide these details and are needed to build the task structure. At all levels, user actions and tasks are combined with temporal relations such as sequencing, interleaving, and concurrency, to describe allowable time-related user behavior.

The UAN is supplemented with scenarios, showing sequencing among screen pictures. The need for such detailed scenarios and task descriptions is articulated by Gould and Lewis (1985): "detailed scenarios [show] exactly how key tasks would be performed with the new system. It is extremely difficult for anybody, even its own designers, to understand an interface proposal, without this level of description." The UAN is also supplemented with state transition diagrams showing interface state changes resulting from user actions, and with discussion notes documenting the history and rationale of design decisions so that developers are not later doomed to repeat those decision processes. In sum, this whole set of representation techniques—the UAN task descriptions, scenarios, state transition diagrams, and discussion notes—is needed; a single one is not sufficient.

In teaching use of the UAN to interaction designers, we have found that teaching "by example" is an effective approach. Therefore, we begin presentation of the UAN with a simple example, at the lowest level of abstraction, the level of most detail in the UAN task descriptions. The discussion stays at this low level for

the rest of Chapter 6, to establish notation for describing basic physical user actions. Then, Chapter 7 shows how you build up levels of abstraction to complete the whole task structure that comprises an interaction design.

6.5 INTRODUCING A SIMPLE EXAMPLE

Most of the following examples use a setting that is familiar to many of you: the desktop metaphor used in the Apple Macintosh™, Microsoft Windows™, and many workstations with a graphical direct manipulation interaction style. *The discussion and the notation are not limited to this interaction style,* but given its popularity and familiarity, it effectively illustrates the concepts herein.

Imagine a description in a user manual for the *task of selecting a file,* described in prose as

Step 1. Move the cursor to the file icon.
Step 2. Depress and immediately release the mouse button.

The UAN description corresponding to these two steps is as follows:

Step 1. `~[file icon]`
Step 2. `Mv^`

Note that a `typewriter-style` font is used to denote UAN expressions, including task names.

In step 1, the ~ denotes moving the cursor. The destination of the cursor movement is the context of some object represented in the `[]`. In this case, the object is a file icon. So we read step 1 of the UAN as "move the cursor to the context of the file icon." Step 2 represents depressing (v) and releasing (^) the mouse button (M). So we read step 2 of the UAN as "depress and release the mouse button," or simply "click the mouse button."

This task serves as an example for the next few sections, to explain, in detail, various aspects of the UAN. Our goal is to represent this, and other more complex user tasks, in a notation that is easy to read and write, but one that is more formal, clear, precise, and unambiguous than English prose.

6.5.1 Create Your Own Symbols

We tried to make the symbols in the UAN representation technique compact, to limit the amount of work designers have to do to record large amounts of notation.

6.5
6.6

Terse symbols are more difficult to remember, though, so the symbols must also be mnemonically suggestive of their function, a trait we call "visually onomatopoetic." For example, the tilde (~) represents "move the cursor to," because the curvy tilde gives the impression of motion. Similarly, v suggests a down arrow and ^ an up arrow.

If you don't like the symbols presented here for the UAN, you are welcome to create your own for any part of it. For example, you might prefer to use `MOVE TO` instead of the ~. However, anyone using the UAN for a serious design will write these expressions often and will probably be glad for the more terse symbols. The specific syntax of the UAN is not the most important concept here, but rather the expressive power, the levels of abstraction, and the design detail that can be obtained using the UAN. In fact, the UAN is an *open notation* in the sense that it can be adapted to the individual needs, standards, and styles of each development team. Symbols can be substituted, and as new devices requiring new kinds of user actions appear, appropriate new symbols should be added to the UAN by a development team using it.

6.5.2 Primitive User Actions

Because the task of selecting a file is directly made up of physical user actions on hardware devices (e.g., moving the cursor and clicking the mouse button), the UAN description of this task will reflect these low-level physical actions. Task descriptions can also include user actions other than physical ones—for example, memory, cognitive, perceptual, and decision-making actions (Sharratt, 1990). However, they are not discussed here because they are not (yet) formally included in the UAN.

The physical user actions are termed *primitive* because they are not decomposed into further details. We could go into detail about how cursor movement is accomplished, for example, by involving kinesthetics and eye–hand coordination. These pragmatic issues of devices involve important human factors issues. However, by being abstracted out of the notation, these issues can be settled independently and incorporated into the design. This means that, for the present purposes, these details can be ignored. Other task descriptions are built on top of these primitives, using levels of increasing abstraction that hide individual physical actions. In order to identify each of the primitives, the developer must first identify each device and the physical user actions that apply.

6.6 DEVICES AND PRIMITIVE USER ACTIONS

The UAN symbols discussed here are used to represent devices and user actions involving devices commonly found in many interaction designs.

6.6.1 Cursor Movement

In Step 1 of the example task of selecting a file, the first device mentioned is one for moving the cursor. Such a device is mentioned only indirectly, not indicating exactly how the cursor is to be moved. A cursor can be moved by a user in many different ways, such as by moving a mouse, a trackball, a joystick, the cross hairs on a digitizer, arrow keys, locations on a touch panel, or an eye tracker.

As in the preceding section, some levels of abstraction here may help to keep the concept of moving the cursor separate from the definition of how a user does it. This separation of definition from use is important because it allows a user interaction designer to change the definition of how a user is to move the cursor without having to find and change all the places in the design where the cursor can be moved. This is sometimes termed *locality of definition*, because the definition is in just one place. The notation for representing a user moving the cursor, then, must be independent of the way in which it is done.

6.6.2 Switchlike Devices

Step 2 in the prose task description of the example task involves another device—namely, the mouse button; the actions are depressing and releasing. This kind of pushing down and letting up action is used to operate many kinds of switchlike devices, such as a mouse button (there might be more than one on a mouse), a keyboard key, a special key (e.g., a "Ctrl", "Esc", or "Shift" key), a function key, a knee switch, a foot pedal, or a "puff-and-sip" tube. We usually use the letter `M` to represent the mouse button device, and you can use your imagination for others. Pushing down is mnemonically and visually represented with a small arrowhead pointing down (`v`), and letting up is represented with an upward arrowhead (`^`). Thus, depressing the mouse button is `Mv`, and releasing the mouse button is `M^`. Clicking the mouse button is `Mv M^`. Because clicking the mouse button is used quite often as a single action in itself, a click can be represented with this shorter idiomatic expression: `Mv^`.

6.6.3 Character String Generators

Another class of devices contains the ones from which user actions result in character strings. Examples are keyboards and voice-recognition devices. Although keyboards are comprised of keys, individual keys are abstracted out of the description because the significant feature is the character string that results from their use. For simplicity, `K` represents the keyboard as a device. In task descriptions, there are two kinds of strings, literal and variable.

When using a copy command, for example, it is possible that a user is required to type the *literal string* "copy." This is represented as

```
K"copy"
```

On the other hand, some strings typed by users are variable. For example, a user may be asked to enter his or her name, a string that varies with each user. This entry of a *variable string* is represented as

```
K(name)
```

If desired, a regular expression can be used additionally inside the parentheses to specify the lexical definition of the variable to be entered by a user, for example,

```
K(user id = [A-Z][A-Z 0-9]+)
```

The `[A-Z]` means the first character must be alphabetic, and the `[A-Z 0-9]+` means there must be one or more alphanumeric characters to follow. Again, you may use your own favorite variation of regular expressions (including prose).

6.7 OBJECTS AND THEIR CONTEXTS

So far, the example of selecting a file has covered moving the cursor and depressing and releasing the mouse button. What about the part of the Step 1 prose description that says "to the file icon?" Well, when a user moves the cursor to an object, it is often for the purpose of manipulating that object in some way—for example, to click on it to select it, or it may be to grab it and move it or change its size or shape. The *context of an object* is that by which you manipulate the object. However, because a user can grab and manipulate an interaction object in different ways for different tasks, the meaning of the context of an object can be task dependent. In these cases, the difference must be made clear, often through use of a figure.

For example, to select a file on a desktop, a user uses the mouse to move the cursor to the file icon itself. The user of a graphical drawing application can use the mouse either to move a line segment or to change its length and/or orientation; the line serves as its own context for moving. However, for stretching or reorienting in some drawing applications, a user must move the cursor to one of the little square grab handles at an endpoint of the line. In this case, if the description of this task (either in prose or any other representation) simply says to move the cursor to the line, it would not be precise enough.

In the case of stretching or reorienting, the handles aid the user in discerning their function by being indicative of the task. However, this is a human factors issue, and it would be nice to abstract this issue out of the notation. To this end, the UAN includes the task-dependent notion of the *context of an object*, denoted by putting square brackets around the name of the object, like this:

```
[object]
```

visually indicating a kind of box, to represent context, around the object.

So,

```
[file icon]
```

denotes the context of `file icon`, which might be the icon itself. On the other hand,

```
[line]
```

represents the context of a line in a line drawing for the task of changing its length, and it might be a grab handle at one of the line endpoints. The distinction could be made more clear by using a subscript or some other explicit notation, to refer specifically to this kind of context:

$$[\texttt{line}]_{\texttt{endpoint}}$$

This little bit of abstraction, to keep the concept of grabbing an object separate from the definition of how a user does it, uses locality of definition, just as was done for moving the cursor at the beginning of Section 6.5. This allows a user interaction designer to change the definition of how a user grabs an object for a given task without having to find and change all the places in a design where that object can be grabbed.

This completes the detailed description of all the UAN needed to represent the user actions of the example file selection task, repeating as a summary:

Step 1. `~[file icon]`
Step 2. `Mv^`

Even this simple example illustrates the brevity, readability, and precision of the UAN.

6.8 INTERFACE FEEDBACK

The preceding UAN captured the physical user actions in the example task, but what about feedback from the system in response to these user actions? Feedback is certainly part of interaction design; including it in these representations allows a more complete description of user and interface behavior as they interact. *Feedback* is displayed by the system, but *is included in our behavioral description* because it is so directly tied to the *user actions* and because it represents *perceptual actions* on the part of the user. Thus, it is an important factor in usability of the system and therefore is important in the behavioral design domain.

There are various kinds of feedback. In the example task of selecting a file, users generally expect (because this is the way it is done in most designs today) the icon of the selected file to be highlighted, indicating that the user has selected the file. Highlighting is the system's way of getting the user's attention, exclaiming a state change in response to the user's (or maybe the system's) action. It seemed natural to use an exclamation mark as a visual and mnemonic symbol for highlighting. Highlighting of a file icon is represented in UAN simply as

```
file icon!
```

Similarly, highlighting of any interaction object would be represented as

```
object name!
```

Notice that this does not say how highlighting is accomplished or how it looks or sounds (or feels?). Just as was done in representing both cursor movement and the context of an object, abstraction and locality of definition are used for separating the definition of the ! symbol from its use. In its definition, highlighting might be specified to mean reverse video (common for icons of selected objects), but it can also mean, for example, blinking, a change of color, putting a box around the object, and/or an audible beep.

Given the symbols to represent highlighting of the file icon in the file selection example, where should it be written? It would be nice to have each indication of feedback side-by-side with the user action that caused it, so the two can be associated. But what user action caused the file icon to highlight? Such details are not always apparent unless we especially look for them. You have surely seen the following kind of description in the manual for your computer, which is a variation on the previous description of selecting a file:

1. Move the cursor to the icon.
2. Click the mouse button, and the icon will be highlighted.

Technically, this seems to say that highlighting occurs after the button is clicked (i.e., depressed and released). That may be an accurate description for some computers, but on many computers, highlighting occurs when the mouse button is depressed. You can see how a small difference such as this can be the source of imprecision in documentation. When this documentation is used to tell an implementer, for the first time, how a particular interaction feature works, this imprecision can cause confusion and misinterpretation.

It is, of course, possible to be quite precise with prose descriptions, but people rarely take the time to write (or to read) lengthy, very precise descriptions. The UAN has the power to be both precise and concise. In order to associate highlighting feedback with depression of the mouse button, simply break up the `Mv^` that represents a mouse click. The resulting task description, with feedback, looks like this:

TASK: **select file**	
USER ACTIONS	**INTERFACE FEEDBACK**
~[file icon'] Mv	file icon'!
M^	

Note that the UAN task descriptions have been made a bit more structured, by adding a *task name*, and headings for *User Actions* and *Interface Feedback* columns. Also, because the name `file icon` is like a general variable in mathematics and can therefore refer to any file icon, a prime has been added to the name, `file icon'`, to indicate that *the name refers to some specific file icon*, thereby distinguishing this instance of the variable from other general references to the variable `file icon`.

As with most Western notations, the UAN is read top to bottom and left to right within a line. In the first line of this more structured format, highlighting of the file icon is associated (by being on the same line) with depressing of the mouse, `Mv`, with the cursor in the context of the file icon.

The way in which UAN represents that the file icon in the Interface Feedback column is the same one represented in the User Actions column is because exactly the same name is used. In mathematics, the name `file icon` is termed a *bound variable*, which means that all occurrences of that variable, within the scope of binding (here within the whole task description), refer to the same thing. The `file icon` to be highlighted is the same `file icon` over which the mouse button is clicked.

There is a little more to know about highlighting. First, the use of highlighting suggests the need for a way to represent unhighlighting. Removing something,

such as highlighting, can be thought of as similar to subtracting, so the subtraction sign is the visually mnemonic UAN representation of unhighlighting:

```
file icon-!
```

Second, in a single interaction design there might be several different kinds of highlighting. There might even be more than one kind of highlighting within a single task description. Suppose a designer wants to design an interface so that file icons have a different kind of highlighting than, say, command icons. The designer can use

```
file icon!
```

for file icons and

```
cmd icon!!
```

for command icons. Then, ! and !! are defined elsewhere, again using the concept of locality of definition. Similarly, the meaning of ! can be specified as different for each kind of object.

The more experienced you are with interaction design, the more you become aware of all the details and hidden complexity. This awareness will lead you to more complete design descriptions. For example, feedback in the preceding task description is still not complete. In the desktop metaphor on some computers, ordinary selection of files (and, therefore, highlighting of file icons) is mutually exclusive. Unless a designer does something specifically to allow multiple selection (discussed in a later example), a user can select only one file at a time. Maybe this example task technically should be named, `select one icon and deselect all others`. To refer to "all others" in UAN, use the universal quantifier, ∀, from predicate calculus; literally it means "for all." For example, the expression

```
∀(file icon)
```

might be used to refer to all file icons. To refer to "all others," when "others" means all file icons other than the one that is being selected, is denoted by the UAN expression

```
∀file icon ≠ file icon':
```

which means "for all file icons except (not equal to) the specific one being selected."

Next, the designer has to say what should happen with all these other file icons when the specific one is selected by a user. In this case the designer wants them to unhighlight. This is represented in UAN as follows:

```
∀file icon ≠ file icon': file icon-!
```

Literally, this means "for all file icons except the selected one, unhighlight them."

When this is added into the task description, it yields a more precise representation of feedback behavior.

TASK: **select file**	
USER ACTIONS	**INTERFACE FEEDBACK**
~[file icon'] Mv	file icon'! ∀file icon ≠ file icon': file icon-!
M^	

This description of feedback in the UAN is more detailed, complete, and precise than the prose description, which said only, "click the mouse button, and the icon will be highlighted."

There is an alternative approach, however, to representing this mutually exclusive behavior—the unhighlighting of file icons other than the one selected—that you should consider for your interaction design representations. Some developers feel that this added detail (e.g., for unhighlighting all icons except the selected one) is not properly part of the mainline user task behavior. Thus, when it is included directly in a task description, it can detract from clarity and readability. The case could be made that this mutual exclusion under single icon selection is more properly expressed as a property of the whole set of file icons. In this case, it is better represented with a separate technique, perhaps as a declaration of an attribute of the set of objects that are file icons. It is possible to augment the UAN with declarative definitions of interaction objects and types, such as these:

```
file icon is an element of a set of type FI.
FI is mutually exclusive under single selection.
```

Another alternative, which works best with computer-based tool support, is to have two or more different views of task descriptions—one for mainline descriptions and one with all the ancillary details. Before pursuing this approach of including ancillary detail in task descriptions, you should realize that it is possible

to get trapped into including details that really have little to do with the user task being described. For example, suppose that the containing window in which a file icon is being selected is not active. In some interaction designs, depressing the mouse button over the icon will activate the containing window, and a second mouse click is required to select the icon. In other designs, a single click will both activate the containing window *and* select the icon. In this situation, these extraneous concerns can be sorted out by, for example, requiring the containing window to be active as a condition of viability (discussed in Section 6.10) for the selection task. This has the effect of constraining a user to activate an inactive window before selecting a file icon in it. This constraint may or may not be appropriate for a particular design.

For the sake of clarity and simplicity, this discussion leaves out ancillary details from task descriptions in cases where they don't add any useful information concerning the design. When writing out a design (e.g., in UAN), you should use your own judgment to determine the appropriate level of detail, to give implementers sufficient unambiguous information without burying them with extraneous description.

That's enough for now about highlighting. There are other kinds of feedback, though. For example, how do you represent moving a box on the screen, as it is dragged by a user? For this, the UAN symbol is an arrow that is intended to show that the box follows the cursor as it moves:

```
box > ~
```

Don't mistake this > for the mathematical "greater than" symbol; when combined with the ~ symbol, it means "follows the cursor."

Also consider the case of "rubber-banding," where an object is stretched out by grabbing part of it and pulling it with the cursor. This might be used, for example, to change the size or shape of a box on the screen. After a user has moved to a corner of, say, a box and grabbed it with the cursor, the UAN represents the rubber-banding action with two arrowlike symbols written close together, as the box corner follows the cursor:

```
box corner >> ~
```

Even more detailed symbols can be used as an interaction design calls for them. For example, suppose that as feedback for the task of creating a new file icon, the system is to display that file icon at some specific location, `(x',y')`, on the screen. This could be represented in a UAN task description as

```
@(x',y') display(new file icon)
```