7

More on Using the User Action Notation

7.1 INTRODUCING TEMPORAL RELATIONS

The UAN was introduced in the previous chapter and now continues, with more examples and more about how to use the UAN to represent a user interaction design. In particular, this chapter discusses *temporal relations* that describe how user tasks and user actions are related over time. This chapter gives quite a different flavor to the UAN than was seen in Chapter 6. There, the UAN was used to describe low-level details of a design; in this chapter the UAN is used to present task descriptions at higher levels of abstraction, with emphasis on temporal relations. We will continue with use of the Calendar Management System example to illustrate application of the concepts.

Task descriptions cannot be represented in the UAN without the use of temporal relations. At a minimum, two simple user actions within a task are temporally related by being in a *sequence*—one occurs after the other in time. Chapter 6 included several examples of user actions in a sequence, as well as some examples of *iteration*, another kind of temporal relation. This chapter explores some addi-

tional temporal relations for use in UAN task descriptions. Formal definitions of the temporal relations are given in Hartson and Gray (1992). Table 7.1 shows a summary list of the temporal relations discussed in this chapter. Later we describe how each of these is used in interaction designs.

The question of temporal aspects enters into the user interaction design process when the relative timing of tasks is considered. The easiest case for an interaction designer is often the most constraining for a user. For example, a designer can easily specify a sequence, but in doing so, the designer is strictly requiring a user to complete one task before beginning another.

For example, in the Calendar Management System, to add a new appointment, a user must first access the proper day and time slot on the calendar, then type in the appointment. The two tasks of accessing and typing are related temporally by being in a *sequence*; they cannot both be active at the same time.

However, users often wish to interrupt a task and, while they are thinking of it, perform another task, later resuming the original one. For example, entering a new appointment may cause the user to remember a problem with an existing appointment; perhaps there is some missing information for an existing appointment that the user had intended to enter. In order not to forget it again, the user wishes immediately to interrupt the entry of the new appointment and go to the existing one to add the desired information. Then the user wishes to return to the new appointment without losing its context. This is a good example of a user need that might not be anticipated during task analysis unless it is supported by observation of users performing representative tasks.

A major purpose of asynchronous direct manipulation interaction styles is to support this kind of interleaved user task behavior. It follows that there is a need for a behavioral way to represent the designer's intention to allow interleaving of tasks by a user. This need is met by the *interleavability* temporal relation, which is used to connect these kinds of tasks in UAN task descriptions.

Most design representations and almost all task analysis techniques leave this

TABLE 7.1. TEMPORAL RELATIONS IN THE UAN

Sequence
Iteration
Optionality
Repeating choice
Order independence
Interruptibility
Interleavability
Concurrency
Waiting

question of intertask temporal relationships implicit, if not ambiguous or undefined. Such specifications often lead to arbitrary design on the part of the interface software designer or implementer. For example, in designing for the task of adding a new appointment to the calendar, a designer may look to an interface toolkit for an appropriate widget. It could be reasonable to the designer to use a preemptive style (modal) dialogue box, requiring the user to enter information for the current appointment before performing a different task. This, of course, does not support the needs of a user to go off and work on an existing appointment while in the midst of creating a new one, as was discussed earlier.

This illustrates the *danger of doing user interaction design in the constructional domain, letting the design be driven by available widgets rather than by users' task needs.* Unfortunately, much interaction design these days does, in fact, begin with, "What widget should we use?" A better question might be, "What widget do I need to use to support the behavioral design?" Unfortunately, the choice of available widgets is limited in many interface toolkits, and coding new ones can be very expensive in a commercial environment.

If developers do observe users during task analysis, though, they may also see a user who wishes to create two or more related appointments at once, or to set an alarm while still creating an appointment. If a user does somehow end up with a dialogue box, say, for typing in appointment information, good interaction design suggests that the user should be allowed to close the dialogue box without completing the associated data entry task for the current appointment, and any information entered so far should be retained. However, the user would still be left with the responsibility for temporarily ending (suspending) one task and starting the other, and would have to use human working memory to carry information from one task context to another. In such a case, it would be useful to provide copy and paste operations for the user to move information from one related appointment to the other. From the user view, there is a task interruption, but the design still does not really support it well. Proper evaluation and design iteration can lead to an even better design, one that allows fully interleaved task performance.

These examples should motivate the usefulness of temporal relations in behavioral interaction design representations. First, temporal considerations might be in a design but cannot be explicit in its representation without temporal relations. The UAN temporal operators allow an interaction designer to represent explicit temporal relationships among and within tasks. Second, treatment of temporal aspects in the previous designs was informal, whereas temporal relations in the UAN help a designer to think a priori about temporally related issues as part of the design process.

7.2 SEQUENCE

Perhaps the simplest temporal relationship between two (or more) tasks is expressed by a *sequence*; one task is performed immediately and entirely after the other. This idea of sequence does not allow any intervening action between two actions in sequence, an observation that is important later in this chapter, when we examine the temporal relations of interleaving and interruption.

In the UAN, a temporal sequence is represented by writing the actions as a spatial sequence horizontally (left to right on the same line) or vertically (top to bottom, from line to line). Chapter 6 showed examples of both. For example, when writing

```
~[file icon] Mv
```

on a line, it is a sequence meaning first to move the cursor to the context of a file icon and then to depress the mouse button.

7.3 COMBINING SMALLER TASKS INTO LARGER ONES

A temporal relation combines pairs, and, more generally, groups of user actions or tasks into a larger single task. This means that whenever two user actions or tasks are connected with a temporal relation, a third task is created. For example,

```
~[file icon]
```

and

```
Mv
```

are each single user actions, but when combined in a sequence,

```
~[file icon] Mv
```

they form a third, new task—in this case, the sequence itself. Any task or grouping of tasks can be enclosed within parentheses. The result is a new task that is composed of the tasks and temporal relations contained within the parentheses. For example, if the preceding sequence

```
~[file icon] Mv
```

were enclosed in parentheses, it would create the new task:

```
(~[file icon] Mv)
```

This task, in turn, can be connected to another task via another temporal relation. The parentheses are generally used for clarity; they do not change the meaning of a task description.

The following definitions summarize the concept of *tasks and how tasks can be combined in the UAN*. Several of these terms and symbols are so far mentioned only in the list of temporal relations in Section 7.1. Each is explained later in this chapter.

- The primitive physical actions on devices described so far are tasks. Examples include all actions, such as `~[icon]`, `Mv^`, and so on.
- If `A` is a task, so are `(A)`, `A*`, `A+`, `<A>`, and `{A}`.
- If `A` and `B` are tasks, so are `A B`, `A (t > n) B`, `A|B`, `(A|B)*`, `A&B`, `A⇒B`, `A⇔B`, and `A||B`.

By combining smaller tasks into larger ones, a designer can represent the complete task structure for an entire interaction design. The articulatory detail of the design appears only at the bottom-most levels.

7.4 TASK NAMES, MACROS, AND LEVELS OF ABSTRACTION

By the time you read through all the detail for describing simple tasks in Chapter 6, you may have been wondering, "If every task takes this much effort and detail, how will I ever get through a whole design?" The answer is that in Chapter 6, you were operating only at the level of abstraction with the most detail. This lowest level, where all the physical actions appear, is the *articulatory level*. The effort a designer puts into task descriptions at this level is an investment that can be reused. Most of a design represented in UAN is at higher levels of abstraction, where this detail is neither necessary nor visible.

A *task description* written in the UAN is a *set of actions, possibly grouped with parentheses and interspersed with temporal operators*, according to the rules for their use (defined later in this chapter). Each task thus formed can then be named and the name used as a reference to that task. A task name used as a reference to a task is termed a *macro task* because the name is used as a substitute for the whole task description. (Don't allow the analogy to programming macros cause you to forget that these procedures are *performed by the user* in the behavioral domain.) The effect is similar to combining tasks and grouping them with parentheses, as described in the previous section, except that now the resulting task is named. This *name ref-*

erence can be used as a user action in another higher level, or containing, task. The containing task is said to be at a higher level of abstraction because the macro task is only named there; the details of its description are hidden by abstraction at the higher level. A good example of macros, showing these levels of abstraction, is given in the next section, for the task of `delete multiple files`, which follows introduction of the concept of choice, or disjunction.

During a user's performance of a task, a task name in the User Actions column is an invocation of a user-performed procedure, serving two purposes (just as invocations do for procedures in programming). The first purpose is for *abstraction*, to hide details of the procedure; the second is for *instantiation*, to create a task instance. This abstraction operation can be applied repeatedly to build levels of abstraction, allowing the entire interaction design to be organized into a quasi-hierarchical user task structure. Just as in the case of program code, levels of abstraction are necessary in the UAN, for controlling complexity and to promote understanding by its readers and writers.

Because a physical user action is a (primitive) task, and a macro task name can be used as a user action, we use the terms *task* and *user action* interchangeably. They have essentially the same properties within a UAN task structure, except that primitives are not decomposed into more detailed user actions.

Getting back to macros, what else is involved in turning a task description into a macro? To gain the most benefit from packaging these task descriptions into macros, they must be general enough to be *reusable*. For example, the task description for the `select file` task from Chapter 6 can be generalized so that it can be used any time the design calls for a user to select a single object from a set of objects having the same mutually exclusive behavior for ordinary selection. That task description can also be extended to represent selection of lots of different interaction objects—any file icon, a specific file icon, a command icon, a button, and so on. This means that the object must be given parameters; that is, `file` can be made more general by making it a parameter of the task and calling it `object` (or `object'`, so that it is clear that it refers to a specific object, as explained in Chapter 6). Here is what the UAN description for a generalized selection task might look like:

TASK: select(object')	
USER ACTIONS	**INTERFACE FEEDBACK**
~[object'] Mv	object! Vobject !≠object': object -!
M^	

As an aside, the second line in the Feedback column describes the mutually exclusive property of objects that can be selected using this generic selection task. It says that when one object is highlighted (single selection) via this task, other selected objects (in this class of objects) become unhighlighted. By omitting this line, a different behavior can be specified—for example, one that just keeps adding to the set of highlighted objects as the user clicks on more icons.

When the designer needs to include in a task description the selection of something, such as an "open" command (using an "Open" button), the macro is invoked like this:

```
select(Open button)
```

This binds `Open button` as the `object` during instantiation of the `select` task.

Note that the Interface State column is not included here. In its more general form, the `select` task does not always cause a change in interface state. For example, selection of a button for a command (e.g., the "open" command) merely causes the command to be executed. There is no state change associated directly with this selection, as there is with the selection of a file.

This variability of the Interface State column information means that it must be factored out of the general form of the `select` task. Although the Interface State column was not needed for this task description, it would reappear, as needed, in association with an invocation of the `select` task as a step in another task, such as the following extract of an `open file` task:

TASK: **open file**		
USER ACTIONS	**INTERFACE FEEDBACK**	**INTERFACE STATE**
`select (file icon')`		`selected = file`
.		
.		
.		

One final issue arising in macro tasks is the treatment of feedback. There are no strict rules here; designers are advised to do what is most effective in communicating the design. If all else is equal, articulatory feedback is usually limited to the lowest level and is not included at the macro level. In the preceding example, the `open file` task is at a higher level of abstraction than that of the `select` task. It therefore does not show highlighting behavior of the file icon, which was already defined within the `select` task description. However, in general, it is acceptable

to repeat the highlighting behavior in higher-level task descriptions if that will help to clarify the design.

Now that you know about macro task descriptions, you have a very important tool for building task structures. For example, you now can take advantage of a characteristic of top-down development, referring to something you want to have happen in your design before you have worked out the details of how it will happen. For example, you can define a high-level task for adding a new appointment to the Calendar Management System as a sequence of two other tasks, tasks that themselves have not yet been defined:

TASK: **add appointment**
access appointment
edit appointment

To add an appointment to the calendar, the user must first access the appointment (i.e., find day and time slot) and then edit (type, make corrections to) the appointment. Details of the `access appointment` and `edit appointment` tasks, developed later, will tell precisely how these tasks are performed by a user.

7.5 CHOICE

The *choice* relation is a disjunction, or logical "OR." It is used to describe a set of alternative ways to perform a task, or part of a task, from which a user chooses exactly one each time the task is performed. Because just one choice is made from the set of possibilities, the choice relation really corresponds to an exclusive "OR." The choice symbol | represents only the choices offered to the user, not anything about how the user might make the choice. Extensions to the UAN that include cognitive user actions, however, are being considered to describe, and help support in the design, how users make choices and other decisions.

Before introducing the choice symbol to the UAN, let's review the example task to `delete multiple files` from a desktop style interface, given at the end of Chapter 6. This example of choice also shows how task names as abstractions offer modularity, consistency, and reusability. To refresh your memory, here is the `delete multiple files` task description without use of abstraction:

TASK: **delete multiple files**			
USER ACTIONS	**INTERFACE FEEDBACK**	**INTERFACE STATE**	**CONNECTION TO COMPUTATION**
(Sv (~[file icon'] Mv M^)+ S^)+	 file icon'-!: file icon'! file icon'!: file icon'-!	 selected = selected U file selected = selected - file	
~[file icon''!] Mv			
~[trash icon]	outline(∀file icon'!) > ~ trash icon!		
M^	erase(∀file icon'!) erase(outline(∀file icon'!)) trash icon!!	selected = null	mark selected files for deletion

The double line shows where this task of `delete multiple files` can be broken to decompose it into two tasks:

1. Select files (the top block in the preceding task description)
2. Delete selected files (the bottom three blocks in the task description)

Both these subtasks may be performed often and might appear as part of other tasks as well. Thus, they are good candidates for making into macros, called, respectively, `select multiple files` and `delete multiple files`.

Here is how the overall task description is stated in terms of names for these lower-level tasks, which are then defined, exactly as shown, elsewhere:

TASK: **delete multiple files**
select multiple files
delete selected files

As an aside, when the name of a task appears as a user action in the description

of a given task definition, the named task is sometimes termed a *subtask* of the given task. A subtask is just a task; the term is used only when emphasizing the hierarchical relationship between two tasks, in cases where one task is defined in terms of the other. In the preceding example, `select multiple files` and `delete selected files` are subtasks of `delete multiple files`.

Chapter 6 described the task of selecting multiple files using the "Shift" key, as shown in the top block of the preceding detailed task description. That method could be called the `shift multiple select` task. Multiple file selection can also be accomplished in (at least) one other way. A user can drag out, on the desktop, a selection rectangle to enclose or intersect the desired file icons. This method could be called the `drag box multiple select` task. Now we can describe the task of multiple file selection as a choice between two alternative subtasks. This choice is represented in the UAN by the word OR or by a vertical bar (|):

TASK: **select multiple files**
shift multiple select
| drag box multiple select

This example is further expanded and completed in Section 7.14, when we discuss representation techniques to complement the UAN. By now, you can see the advantages of using task macros like this. They make higher-level task descriptions shorter and simpler, hiding details at the lower levels of abstraction. They also promote and support reuse of task descriptions and consistency across an interaction design.

7.6 REPEATING CHOICE

Combining the OR or choice, described in the previous section with the UAN iteration symbols * and + yields a combination used often enough to have its own name, the *repeating choice*. To start with,

```
(A|B|C)
```

denotes a single task that is a three-way choice among the user actions or tasks A, B, and C. To perform this choice task, a user chooses and performs one of the three tasks, and that is all that happens. Suppose, however, that the designer wants a

user to be able to repeat this choice task—that is, to be able to choose a task from A, B, or C, perform it, choose one again, and so on. This could be the situation at a high level of design if, for example, A, B, and C are choices among major system functions. To represent this in UAN, the designer uses a repeating choice:

```
(A|B|C)*
```

The repeating choice is interpreted in the following way. First, the ordinary choice inside the parentheses means that tasks A, B, and C are simultaneously and equally available. The * means, as usual, that the whole phrase inside the parentheses (the choice itself) is performed zero or more times. Once a task from the choice is begun, it is performed to completion (a kind of modality), at which time the three tasks are equally available again. The cycle continues for as long as one of the three tasks is selected by a user and performed to completion.

To see how this notation can be used as a compact high-level description for a choice among major system functions, try it out on the Calendar Management System. Recalling, from Chapter 5, the five basic functions of the Calendar Management System, the highest-level task in this system can be defined as a repeating choice among tasks corresponding to these main user operations:

TASK: **manage calendar**
(access appointment
| add appointment
| update appointment
| delete appointment
| establish alarm)+

The use of + here, as opposed to *, illustrates a fine point—to rule out the case of a user performing the contents of the parentheses zero times, because that is the same as not performing the `manage calendar` task at all.

This kind of repeating choice can be implemented as a pull-down menu, for example. When the system in launched, the user is initially faced with a choice among these basic functions. When the user makes a choice, that function is carried to completion, and the user returns to the main menu, where the same choice for all functions is available again.

A repeating choice is also used in the `access appointment` task definition, as we begin to decompose this subtask for the `manage calendar` task:

TASK: access appointment	
USER ACTIONS	INTERFACE STATE
(search	
| access month	
| access week	
| access day)*	
access time slot	view level = time slot

This example shows how easy it is to use temporal relations to represent very specific combinations of tasks. The `access appointment` task is a sequence of two parts: the first part of the sequence is a repeating choice (in the parentheses), and the second-part is the `access time slot` task. If a system log were to record observable user behavior in performing the `access appointment` task it would record a series of zero or more instances from among its four subtasks of `search`, `access month`, `access week`, and `access day`, ending with a single instance of the `access time slot` subtask. This leaves the state of the view level at the time slot level.

Note that because these are higher-level tasks, interface feedback is not described here. Rather, it is detailed at lower levels, and so the Interface Feedback column has been omitted. You could have left that column in the task description and simply left it empty, if you wished.

7.7 ORDER INDEPENDENCE

In the use of interactive computer systems, as in the world in general, it is not uncommon to find situations where a number of tasks are to be performed, but the order of their performance is immaterial. Such tasks are represented in the UAN as an *order-independent* group. In this situation, a user must perform all the actions of the group, and each one must be completed before another is begun. There is no constraint on specific ordering among the actions. The UAN symbol to connect order-independent tasks or user actions is the `&` (ampersand). An example of order independence at the articulatory level is seen in the task of entering a command-X to delete some selected object(s) in many Macintosh™ applications. This is a combination of the ⌘ and X keys. because the ⌘ key must be depressed before the X key, but the order of their release does not matter, the task is defined in UAN as:

TASK: **command-X**
⌘v Xv (⌘^ & X^)

The `edit appointment` task provides an example of order independence in the Calendar Management System. Suppose that an appointment object has text fields for name of person, description of appointment, and location. The task of editing an appointment breaks down into the set of tasks for editing these smaller objects, but the order in which they are edited does not matter:

TASK: **edit appointment**
view level = time slot:
(edit person
& edit description
& edit location)

7.8 INTERRUPTION

There are several ways in which tasks can be interrupted by other tasks, as the following subsections illustrate.

7.8.1 One Task Interrupting Another

Once a user begins a task, that task does not necessarily remain active until it is completed. One way that a task can become inactive is due to interruption by another task. An interruption occurs when the user and system activity of one task are suspended before that task is completed, and activity of another task is begun in its place. Task interruption often occurs due to actions initiated by the user, but it can also be the result of system-initiated actions (e.g., to update a clock or to announce the arrival of electronic mail).

7.8.2 Interruptibility

Because a design representation is intensional—that is, it describes what can happen rather that what does happen in any specific instance, there is no symbol in the UAN for *is interrupted by*. Rather, there is a temporal operator to denote cases

of *interruptibility*, situations where interruption *can* occur. While order independence relaxes the sequentiality constraint, *interruptibility* goes even further by removing the constraint of one task having to be performed to completion before beginning another task. Interruptibility allows one task to be interrupted by another and then returned to later for completion.

In UAN, the symbol → denotes that one task can interrupt another. For example, the expression

```
help → edit document
```

represents the ability of the user to interrupt the document-editing task with a help task; literally read from the UAN, `help` can interrupt `edit document`. In practical terms, this means that a user can invoke the help task at any time during editing, but closure of the help task is required before editing can continue. This ability to be interrupted naturally suggests the need to develop the notation used to express exceptions to interruptibility.

7.8.3 Uninterruptible Tasks

Sometimes a user interaction designer needs to define exceptions within some scope of interruptibility. That is, although a particular task is generally interruptible, there may be a couple of cases when it is important to prevent interruption. One kind of exception occurs when the user action in question is a primitive. For example, a designer should not have to think about another task interrupting the primitive user action of pushing down the mouse button; either the button was pushed or it was not. Thus, primitive user actions are generally defined as not interruptible.

A second situation where a task instance must be specified as uninterruptible occurs in preemptive (modal) interface features (Thimbleby, 1990). Certain kinds of dialogue boxes provide a good example. For example, while using a dialogue box in task A, a user cannot click in the window of task B to change tasks until the dialogue box is exited. While in the dialogue box, a user can still interleave tasks, but only among tasks available from within the dialogue box. In the UAN, angle brackets, < and >, are used to enclose those parts of a task description that are *uninterruptible* by other user actions at any level. For example, `<use dialogue box>` denotes that this task cannot be interrupted by any other user task.

7.9 INTERLEAVABILITY

Two tasks are *interleavable* if and only if they can interrupt each other. In practical terms, this means that a user can do part of one task, skip to another task and do

part of it, then return to do more of the first task, and so on. Interleavability is represented in UAN with the ⇔ symbol, as in the following example from a desktop publishing application:

```
edit document text ⇔ create graphics
```

As an example from the Calendar Management System, consider the five main user operations discussed earlier as subtasks of the main task, `manage calendar`. As a reminder, those subtasks are `access appointment`, `add appointment`, `update appointment`, `delete appointment`, and `establish alarm`. In Section 7.6, these were represented as a repeating choice. A more asynchronous design would allow an instance of each subtask to be created in its own window. A user could go back and forth, interleaving the activity among instances of the subtasks by activating one window after another, say, by clicking in each window. The UAN task description for this interleaved design is:

TASK: **manage calendar**
(access appointment
⇔ add appointment
⇔ update appointment
⇔ delete appointment
⇔ establish alarm)$^+$

Note once again use of the $^+$ to indicate that the entire construct can be repeated, allowing multiple instances of the same task and/or different tasks at once.

7.10 CONCURRENCY

With interleavability, user actions can be alternated among tasks; with *concurrency*, user actions for two or more tasks can occur simultaneously; that is, their physical actions overlap in time. Concurrency is a temporal relation that has not been greatly exploited in user interfaces. Nevertheless, there are cases in which it is possible and, indeed, preferable, for a user to carry out more than one task at the same time. For example, a user can be perceptually responsive to information on the display while typing or manipulating the mouse. Buxton has described input techniques that rely on concurrent use of both hands (Buxton, 1983). Such situa-

tions require the full representational power of the concurrency relation as just described.

Another kind of concurrency is seen in the actions of two or more users doing computer-supported cooperative work, or CSCW. These users, using different workstations, may be able to perform actions simultaneously on shared instances of application objects, possibly operating through different views.

The UAN symbol to represent a design that allows users to perform tasks concurrently is a symbol that implies parallelism—namely, | |. For a representation of the Calendar Management System in the case where periods of activity among tasks can overlap, the UAN task description becomes:

TASK: **manage calendar**
(access appointment
|| add appointment
|| update appointment
|| delete appointment
|| establish alarm)$^+$

7.11 INTERVALS AND WAITING

Time intervals are also important in task descriptions. For example, the prose description for a double click of a mouse button might tell a user to click the mouse button and immediately click it again. If the designer really wished to be precise, it requires the expression of a constraint on the time interval between clicks. In such cases where the time between user actions (e.g., clicks) is significant in a task description, the timing interval acts as a temporal relation between the actions, constraining the temporal distance between actions in a sequence. This task can be represented precisely in the UAN. The complete UAN task description for double clicking is

```
Mv^  (t<n)  Mv^
```

where t is the time between mouse clicks, and n is a numeric value in units of time (e.g., seconds). Often the value of n can be controlled by the user via a control panel setting, and an appropriate default setting can be empirically determined by developers.

Another way a time interval can be used in a UAN description as a temporal

relation between two tasks is to indicate a minimum wait time to cause some kind of timeout by the system. A specific instance might be written as

```
Task1 (t > 5 seconds) Task2
```

7.12 SUMMARY OF UAN SYMBOLS

Tables 7.2 and 7.3 summarize the UAN symbols presented in Chapter 6 and this chapter. Table 7.2 summarizes all the temporal relations among user tasks discussed thus far, as well as all other UAN symbols currently in use in the UAN. The symbols in Table 7.2 are used in the User Actions column. The symbols for the basic physical user actions are at the lowest level of abstraction. Table 7.3 summarizes the symbols used in the Interface Feedback column. All symbols are suggested, in the sense that the UAN is, as discussed in Chapter 6, an open notation, often adapted and extended by interaction designers.

7.13 EXERCISES ON USING UAN

Now that you have been reading about the UAN, as well as seeing examples of it and doing small exercises for almost two chapters, it's time for you to try out the UAN yourself on your own Calendar Management System design. In the next several exercises, you will get practice in using the UAN for interaction representations in the design developed for the Calendar Management System in Chapter 5. At the end of this chapter, you will then write UAN descriptions for your own Calendar Management System design. Feel free to look back through this chapter and Chapter 6 to review the UAN, as needed, while doing the exercises. Most people learn the UAN pretty fast, with a little practice.

Much of the UAN representation of our Calendar Management System design has already appeared a bit at a time, as examples illustrating various aspects of the UAN. This set of exercises will pull it together into one place.

Exercise—High-Level Design Representation Using UAN

GOAL: To use the UAN to begin developing a behavioral representation of user tasks in the Calendar Management System's interaction design produced in Chapter 5, at the high levels of design.

MINIMUM TIME: About 10 minutes.