CHAPTER 12

The Structure of Users' Activities

ALLEN CYPHER

The exasperated shout, "I can only do one thing at a time!" usually comes from people attempting to do six things at once, at a time when they are only capable of doing five. One way we do five things at once is by interleaving the activities. We dice the chicken while waiting for the water to boil, turn down the heat on the rice while frying the onions, but wait a moment to take out the bread because the white sauce is just starting to thicken.

Whether we are preparing dinner, debugging a program, or getting married, a good part of our mental energies are spent *linearizing*. The many parallel tracks of activities must be organized into a single linear stream of actions to be performed. [1] In a sense we are very skilled at scheduling multiple activities: It is a fundamental process which is a constant part of our mental life. But at the same time, we make a lot of scheduling errors: We let the bread burn while stirring the sauce, or let the sauce get lumpy while removing the bread from the oven.

The field of ergonomics attempts to design objects that take into account the realities of the human body—if a keyboard is too high, the typist's wrists will have to bend and typing will be uncomfortable. In

[1] For a discussion of some of the complexities of multiple activities, see Chapter 13 by Miyata and Norman, especially the section entitled *Multiple Activities: Current and Suspended.*

an analogous manner the field of human–computer interaction attempts to design interfaces that take into account the realities of the human mind. In this chapter, I deal with the reality that people do not simply perform one activity at a time. Program designers put a great deal of effort into allowing users to perform single activities well, but considerably less effort goes into allowing users to arrange those activities. If computer systems are designed so that they actively support and facilitate multiple activities, they will be more comfortable for the user.

CHARTING THE FLOW OF ACTIVITIES

Our computer system provides support for multiple, interleaved activities.[2] In order to get an idea of how frequently users actually do interleave activities, and in order to get a sense of the types of interleavings that occur, I modified our local computer programs so that they would keep a record of each command typed by the user. Whether the commands are to the top-level interpreter or to the text editor or the mail program, they are all collected in the order in which they are typed. Therefore, interleaved activities show up as interleaved sequences of commands. Since it is sometimes difficult to associate commands with the activities which they serve, users may also insert comments explaining their current activities. The Appendix describes the history-collection program in more detail.

Figure 12.1 diagrams the results of one of these historical records. It charts the flow of my activities during one morning of computer use. In the figure, each activity is shown as a box, with interruptions shown as gaps in the shading. Important subactivities are shown as sub-boxes.

It can be seen from the figure that there are numerous cases of interleaved activities. Let's examine some of these cases in detail.

> *EXAMPLE 1. Three interleaved activities:* **read mail → reposition window → msg conversation**
>
> By starting at the lower left hand corner of Figure 12.1 and following the arrows, you can see how I began my day with three interleaved activities.

[2] The system includes a VAX computer and a network of 15 Sun workstations. All use the Berkeley UNIX operating system and the Sun supports multiple windows.

I start off the session by **reading my mail** [1]. While I am waiting for that program to start up (it takes about 10 seconds), I type "toolplaces" in a second window — I was annoyed at the default location of the "history" window, and "toolplaces" will help me **reposition the window** [2]. "Toolplaces" takes about 30 seconds to run, so I start up an electronic **msg conversation** [3] by sending off a message to a couple of friends. I then resume the **reposition window** [2] activity: the "toolplaces" program has finished running, and it tells me that the history window is already in its correct location. But that can't be right.... I am confused so I return to **reading my mail** [1].

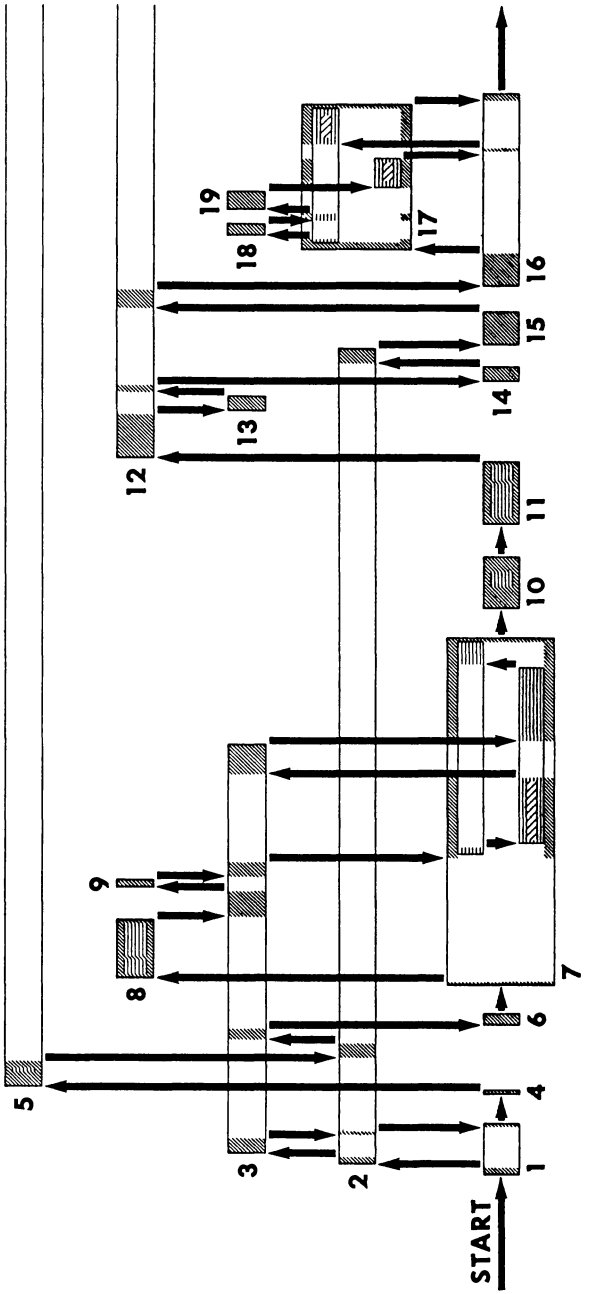
I temporarily abandoned both the mail program and "toolplaces" because they are slow to start. In the mornings I am impatient with even short delays, but later in the day I still will be impatient when the computer spends several minutes compiling a program. "Waiting for the computer" is a common impetus for interleaving activities.

EXAMPLE 2. External Interruptions: **helping A**

A short time later, as I am about to start the activity **respond to P's message** [7], I am interrupted by a phone call [8]. User A wants some help with the "fmt" command. I consult my personal database to get the information and then log in to her machine to try it out.

Just as I finish, the message "You have new mail" flashes across my screen. I invoke the mail program and read the message, which continues the **msg conversation** [3] I started earlier. After reading the mail, I have trouble remembering what I intended to do before the interruptions. I eventually recall that it was **respond to P's message** [7].

In this case, two interleavings are prompted by events external to the user. External interruptions can be particularly disruptive since they need not occur at a natural transition point for the user. Computer systems can be helpful in these cases if they are able to provide reorienting information when the user attempts to return to the interrupted activity.


START
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19

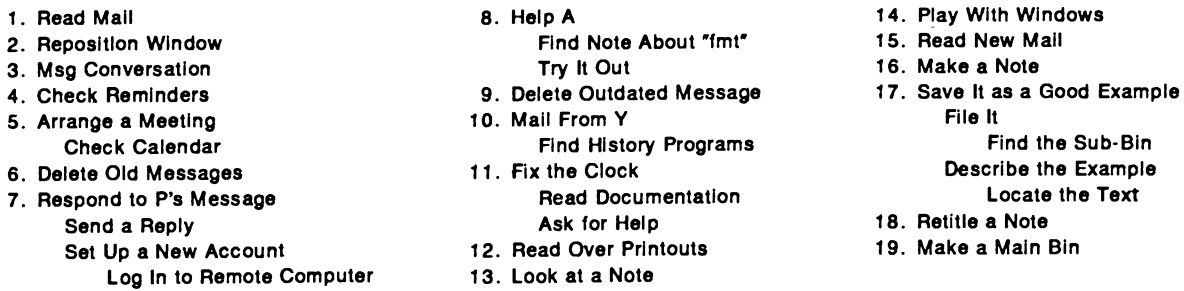


FIGURE 12.1. This figure charts the flow of activities during one morning of computer use. There are 19 activities, shown as boxes. Shaded areas show when an activity is actually being performed, and arrows show when the user switches between activities. This figure only covers part of one day. Activities {5} and {12} were not completed until later in the day.

The sequence of events during the day can be found by following the arrows. The session starts with **read mail** [1], switches to **reposition window** [2], switches to **msg conversation** [3], returns to **reposition window** [2], and so on.

Subactivities are shown as sub-boxes. The box for **help A** [8] contains two sub-boxes, since this activity involved first "finding a note about the 'fmt' command," and then "trying out the command." Activity [7], **respond to P's message** is more complicated because the subactivities are interleaved. Shortly after the first subactivity of "sending a reply" was started, it was interrupted by the second subactivity of "setting up a new account."

The horizontal axis is scaled by number of commands.

Note how different this case is from the previous one. In example 1, I went out of my way to engage in several activities at once, since I was bored doing one thing at a time. I was willing and able to accept a greater mental load. In the second example, activities are being thrust upon me, obliging me to abandon a simple one-at-a-time sequence. The real-time demands of the tasks at hand impose a mental burden that pushes the limits of my abilities. In this chapter I am not too concerned with users' motivations for interleaving activities; I concentrate instead on ways to support the resulting multiple activities. Chapter 13 by Miyata and Norman focuses on what is going on inside the user's head, and they present a detailed analysis of users' motivations.

EXAMPLE 3. Subactivities: **arranging a meeting**

I want to **arrange a meeting** [5] with R, so I decide to send her a message. I type:

```
snd                    ("snd" is our program for sending electronic mail)
 To: R
 Subject: Meeting
 Let's get together to talk about the project.
 I'm free at

[remind tomorrow]      ("remind" is our calendar program)

          1:00 tomorrow.
 Is that a good time for you?
 -Allen
```

The "remind" command was typed in a different window. It displayed my schedule for tomorrow so that I could see when I was free.

EXAMPLE 4. Subactivities: **responding to P's message[7]**

I am responding to a message asking me to set up a new account on a remote computer. I start typing a return message saying that I will do it, but instead I just set up the account and report that it is done. This takes longer than I expect because I have trouble linking to the remote computer.

Here the Main Activity "respond to the message" contains two Subactivities: the Subactivity "send a reply," and the Subactivity "set up the account," which contains a Sub-subactivity "link to the remote computer." You can see in the flowchart that "send a reply" is temporarily interrupted by "set up the account." Incidentally, this entire activity is interrupted by new mail which continues the **msg conversation** [3].

User Activities

When we design programs, we think about the sorts of things that users will want to do and then we build tools for carrying out those tasks. Implicit in this approach is the belief that there will be a fairly good match between *computer programs* and *user activities*. But how well does this assumption hold up in practice? Among the activities discussed so far are "reading the morning mail," "arranging a meeting," "carrying on an electronic conversation," and "helping another user with a command." The programs used by these activities include the "mail program," the "calendar program," and the "personal database" program. So in this list, the only case where an activity and a program are matched is in reading mail.

Single Activities and Multiple Programs

One way for a mismatch to occur between activities and programs is for a single activity to call upon more than one program. The activity of **arranging a meeting**, for example, involved two programs: mail and calendar. The ramifications of this simple fact are far-reaching and have a dramatic effect on how a user interacts with a computer system.

A system which is oblivious to the use of several programs for a single activity places all of the burden of program management on the user: in the case of **arranging a meeting**, I would have had to abort my message when I realized that I did not know when I was free tomorrow. Then, after consulting the calendar program, I would have then had to call the mail program anew and retype the message.

Saving context. At issue here is the fact that when one engages in an activity, one builds up a *context*. My context in "arranging a meeting" included a partially typed message. When you are engrossed in any challenging activity, you build up a rich mental environment that is populated with current problems, attendant confusions, and potential

solutions. Even a momentary interruption will cause this elaborate *mental context* to collapse. Complex activities on a computer can lead to equally elaborate *computer contexts*. A half hour with an interactive debugger, painstakingly stepping through deeply nested code, can produce a context which could be reconstructed only by repeating the half-hour procedure. With proper design, these elaborate computer contexts can be interrupted without collapse.

Freezing programs. Computer systems can provide support for interruptions by saving the complete state of a program. The Berkeley UNIX "Stopped Jobs" facility is an example of this approach. It allows any process to be frozen at any time, thereby freeing the computer for some other process. When a process is frozen, its entire state, or context, is remembered so that it can be resumed precisely where it was left off. Most window systems (but not, for instance, the Macintosh) allow for multiple interactive processes for a single user. It is often said that windows support multiple activities, but in fact it is the underlying "multiple user processes" that enable multiple activities. Windows are a particularly good representation for the user.

In the "three interleaved activities" example, I interrupted the activity of reading my mail. Part of the state or context of the mail program is a variable which specifies the "current message," the message that I am currently reading. This context was saved when I interrupted this activity. When I resumed reading my mail, I typed the "next message" command, and the mail program knew where to continue.

To be precise, a "stopped jobs" facility supports multiple *programs*, not multiple *activities*. This distinction is significant in those cases where activities do not correspond perfectly with programs. Suppose in the **arrange a meeting** example that I suddenly want to interrupt this entire activity while I am consulting the calendar. I could freeze the calendar program, but there is no sense in which the system is aware that my mail program is also associated with this activity. What I want to do is to freeze my *activity*, not just one of the programs associated with that activity. But an "activity" is not a part of the computer system's vocabulary—it only knows about programs. When faced with the fact that the system does not think in our terms, we just do our best to align our activities with the units that are available. So we tend to identify an activity with the program that it is currently using. As a consequence, if I digress from the **arrange a meeting** activity for a long time, I may return to the calendar program and forget that it is associated with the previous mail program. And if I attempt to hide this activity by closing the calendar's window, the window for the mail

program will remain open. In the context of activity management, then, the idea behind "user centered system design" is to allow users to treat their activities as manipulable units.

Windows. Windows divide one screen into multiple virtual screens, each behaving like a complete screen. Windows are a further step in saving a suspended context. Not only are all internal variables saved, but the screen image that is presented to the user is saved as well. One of the considerable advantages of windows over standard terminals is that this saved image can be present on the screen even while the user is engaged in an interrupting activity. It is hard to imagine a more effective reminder of the interrupted task than its image on the screen.

Windows improve the user's vocabulary by adding a concept that is one level above that of programs. By keeping all of the programs for a single activity inside a single window, the user can indeed treat those programs as a single unit. In practice, though, windows are often used to make the multiple programs of a single activity available simultaneously. This is common, because when two programs are being used within the same activity, it is likely that the user will want to pass data between them, see what they are both displaying, and interleave commands to them. In fact, in the **arrange a meeting** example, I used a second window to run the calendar program. This example suggests that window systems can better support activity management if they allow the user to link related windows and treat them as a manipulable unit. So, for instance, the "close" and "open" functions would close and open all of the windows together. This is discussed further in Chapter 14 by Reichman.

We can go one step beyond the use of multiple windows in support of an activity and consider using multiple screens. By a "screen" I mean a collection of windows that fill the computer's screen. The Symbolics Lisp Machine demonstrates the potential of this approach. This machine is organized around a few general-purpose tools which are integrated so that they can all be used in concert to perform an activity (typically the development of a program). In the process of debugging, the user may have one screen which contains a window for source code, a window for giving commands (the "lisp listener" and "debugger"), and a window for the running program's interface (the "application"). A second screen may contain the file system editor, a third screen the "flavor examiner," and a fourth screen the data-structure "inspector." When one screen is active, there is no visible trace of the windows associated with the other screens. In normal use, then, the user will hop amongst screens in the course of performing a single activity.

Multiple Activities and Single Programs

The other way for a mismatch to occur between activities and programs is for more than one activity to call upon a single program. This lack of correspondence between activities and programs also leads to activity-management problems. The potential difficulty with sharing a program is that the two activities will each want to establish their own contexts, but since the program only has one set of context variables, the contexts will clash. The one activity's setting of a context variable will get clobbered each time the user switches to the other activity.

Context Clashes

In the "three interleaved activities" example, the first activity (**read mail** [1]) and the third activity (**msg conversation** [3]) used the same program (the "mail" program), so there was the potential for a clash. The main context variable in the mail program is the "current message" variable. Commands to "display message X" or "display next message" make use of this variable. In the "read mail" activity, I read messages 1 and 2 by using the "display message #1" and "display next message" commands. This set the "current message" variable to 1 and then to 2. Next, I **repositioned the history window** [2] and then went to the **msg conversation** activity. Suppose that in that activity I had chosen to "read the message from J," which happens to be message number 8. This would have set the "current message" variable to 8. Had I done this, when I returned to the original **read mail** activity and asked to "display next message," I would have been shown message number 9 instead of the desired message number 3. I was lucky in the real example because I chose to "send a message to J" rather than "read the message from J," and sending does not affect the "current message" variable.

As another example of context clashes, consider multiple-buffer editors such as Emacs. A "multiple-buffer" editor allows the user to edit several different files from within a single instantiation of the editor program. The editor handles this by maintaining a full complement of context variables for each of the files in question. This feature can be used effectively for single activities involving multiple files. As such, it is a good example of a design which anticipates the needs of the user. If you decide to change the word "brother" to the word "sibling" throughout a set of six files, you can do this in a single operation. However, this feature can also be used to perform multiple activities

within a single editor. This opens up the possibility of the word "sibling" cropping up unexpectedly in files belonging to an unrelated activity that just happened to be in the way.

While-I'm-At-It Activities

A third example of context clashes occurs with "While-I'm-At-It" activities. A common example of "While-I'm-At-It" activities occurs when you are looking at a listing of files and happen to notice an outdated one—it is convenient to just delete the file right away and then continue on with whatever you were doing. Technically, you have interleaved an activity ("delete file" in this example) that is completely unrelated to your primary activity except for the fact that it occurs within the same context or environment as the primary activity. That is, the two activities share exactly the same context but their goals have nothing in common: I refer to such activities as "While-I'm-At-It" activities. In Figure 12.1 there is an example of a "While-I'm-At-It" activity (**delete outdated message** [9]) which occurs as a quick digression from **msg conversation** [3].

Handling context clashes. It is valuable for program designers to try to anticipate the typical "While-I'm-At-It" activities for their application. For these activities, it is possible to resolve the potential context clash by ensuring that the "While-I'm-At-It" activity does not change the setting of any context variables. For instance, consider the "current message" variable in the mail program. In order to read my first three messages, I would give the commands "display message #1," "display next message," and "display next message." These commands set the "current message" variable to 1, 2, and 3, respectively. But suppose I interrupt my reading after the second message, and delete message number 8. Now what will happen when I give the "display next message" command—will I see the desired message number 3, or will I instead see message number 9? It would have been natural and consistent to make the "delete message" command change the value of the "current message" variable to the message after the deleted one. Fortunately, the designer was more clever: The "delete message" command leaves the "current message" variable unchanged, so in fact I will see the desired message number 3.

The success of this design approach—of allowing for simple operations which will not alter the user's overall context—is dependent upon the fact that "While-I'm-At-It" activities tend to be short and self-contained. When unanticipated difficulties arise and the digression

becomes more protracted, the user may regret not having performed it as a separate activity.

But what about the more complicated types of context clashes? How can we handle the case of **msg conversation** clashing with **read mail**? A partial solution is to design modularity into the program. The program is segmented into several subparts, each with its own context variables. The idea is to group together commands whose use typically coincides with a common user subactivity. Then if multiple activities call upon different subparts of the program, they will be able to use the same program without clashing. For instance, the mail program has a subpart for sending mail which is distinct from the subpart for reading mail. This accounts for the fact that **msg conversation** and **read mail** shared the same program without clashing. However, it is inevitable that two activities will eventually want to access the same subpart of a program, so segmenting cannot be considered a complete solution to the problem. A more involved **msg conversation** would eventually have caused the value of the "current message" variable to be changed.

The most direct solution to the problem of clashing contexts is to call up separate instantiations of the program for the different activities. This way, the user has a complete, independent set of context variables for each activity. This solution is commonly used with text editors. Which file is being edited is such an important context variable that a user who wants to edit two unrelated files will generally choose to invoke two separate instances of the editor.

One drawback to separate instantiations is that there is a time penalty for the user in starting up a new instance of the program. A more interesting drawback is that the two instances will have *no* context in common. This hardly seems a problem, since I have been assiduously attempting to avoid shared context. But in the next section on "Related Activities" I discuss situations where some sharing is important.

A fourth solution is to design activity management explicitly into the program. In this approach, the program eliminates clashes by keeping separate copies of its context variables for each different activity. When some shared context is desired, some of the variables can be shared, and separate copies can be made for the others.

I wrote a program called Notepad which adopts this approach. It has special "Interrupt" and "Resume" commands for switching amongst activities. Because it is so easy for users to engage in multiple activities within Notepad, the history flowcharts show considerably more interleaving when it is used. An example can be seen in the latter part of Figure 12.1 (activities [16]–[19]). Notepad falls short of the goal of true activity management because it only remembers a single note for

each frozen activity. This is quite analogous to the example where UNIX would freeze the calendar program but be unaware that the mail program was part of the same activity. An improved Notepad would allow groups of notes to be manipulated as a single unit. Another way of saying this is that there needs to be a context variable called "current activity."

When activity management is explicitly designed into a program, it yields the added benefit that its features are available for managing the subactivities of any activity which uses that program. Consider a complaint that I have heard about our mail system: In the middle of composing a message, the user realizes that a copy of the message should be sent to a third party. The mail program has a "compose" command (it is called "input" mode) for composing messages, and a "cc:" command for sending copies. But the "cc:" command cannot be used while in "input" mode—that is, the "composition" subactivity cannot be interrupted by the "cc:" subactivity. So the user waits until the text is complete, intending then to add the cc line. But the user forgets and sends the message, forgetting to use the "cc:" command. A similar frustration occurs when a user is composing an answer to a message and finds it impossible to view the message that is being answered.

Explicit activity management within the program could remedy these situations by allowing the subprograms to be individually interruptible. The program would manage its *subprograms* in the same way an operating system manages its *programs*.

Related Activities

My focus so far has been on interleaving activities that are not related to each other. For instance, the activity of **arrange a meeting** [3] had nothing to do with the activity of **read mail** [1]. In situations of this sort it is perfectly reasonable to assume that the user is unconcerned with the one activity while busy with the other. If the previous activity remains on the screen, it is only as a reminder of its existence; it has no bearing on the execution of the latter activity. In contrast, there are numerous important situations where the interleaved activities are related to each other. The most common source of related activities is *subactivities*. Consider the situation where you are answering a message and you need to take a look at the message that is being answered. The subactivity of "looking at the message" is related to the "answering the message" activity: You are looking at the message in order to help you carry out your answering activity.

Simultaneous Interaction

In cases of *related* activities, the user (a) wants both of the activities to be visible simultaneously; (b) wants to be able to interleave commands to the activities; and (c) wants to be able to pass data back and forth between the activities. We can summarize this by saying that users desire "simultaneous interaction" with related activities.

How can systems accomplish this simultaneous interaction? Making both activities visible and receptive to commands can be achieved by using windows, and a limited degree of data-passing can be achieved through "cut and paste" facilities. But full data-passing, which honors the contrasting data types of different programs, is a complex problem requiring a sophisticated programming effort (See Chapter 11 by Mark). The **arrange a meeting** [5] activity illustrates the need for data-passing. In that example, I consulted a calendar as a subactivity and used a second window so that the calendar would be visible while I continued composing my message in the mail program. If the information displayed by the calendar program had been complex, I might have wanted to copy its display into my message.

Shared Context

In the **arrange a meeting** example, it was necessary for the system to provide special support in order to facilitate "simultaneous interaction." The reason for this is that the subactivity used a different program from the main activity. In cases where subactivities use the same program, there is no difficulty in viewing both activities, giving them commands, and passing data between them. In fact, since there are no such problems, it is unlikely that the user will even be aware that multiple activities are present. However, related activities which share the same program have their own special problem: The issue of "context clashes" which I discussed earlier is now complicated by the need for a certain amount of "shared context." This problem is best explained by an example. Suppose I am busily fixing a bug in a program, using the text editor to modify the code. I run into a problem so I send a message to a colleague for help. While waiting for the reply, I decide to work on another bug in the same program. This is a situation where two subactivities are interleaved. The special twist presented by situations of this sort is that the subactivities have some context which is shared and some which is unshared. The shared context is that they are both modifying the same file. The unshared part is that each subactivity will want to establish its own markers into the text, its own set of strings that are saved in buffers, and its own independent command

history. If I just continue to use the same editor, I will mix the markers, buffers, and history of the former subactivity with those of the current one. On the other hand, if I start up a new instantiation of the editor, the two editors will develop clashing versions of the shared file. What I really want to do is to start up a new editor which inherits the shared context of the two subactivities and which creates its own context for the unshared part.

The Emacs editor has a facility which approximates this: it is called a "recursive editing level," and it has the effect of invoking a new instance of the editor inside of the original instance. The new instance shares some context, such as the file and the location pointers, but it maintains its own command history. Deciding which parts of the context to share and which to separate is a complicated question which must be tailored to the expected needs of the application. The recursive editing level in Emacs is mainly used within long, complicated commands, so that the user can make a few changes without having to abort the complicated command.

The principle of recursive instantiations forms the basis for another highly successful program, the "break" package, which is found in many LISP systems. The "break" package allows a user to nest subactivities (and sub-subactivities) within other activities in an attempt to resolve subproblems that crop up while working on other, larger, problems. Each subactivity inherits the complete context of its parent activity.

REMINDING

Users engaged in many different activities can appreciate some assistance in keeping track of those activities. If some of the cognitive burden of keeping track of activities is removed, users can concentrate more on actually performing the tasks, instead of spending their energies on remembering and scheduling those tasks. Chapter 13 by Miyata and Norman discusses ways for assisting users in remembering their activities.

Windows Are Not the Whole Solution

Windows and icons provide one way to remind users of interrupted activities. They can be quite effective—up to the point where the activities become so numerous that they clutter the screen. At this point, it may prove more effective to replace the jumble of overlapping windows with a more concise list of titles or descriptors of the activities.

Another way (besides clutter) that visual reminders lose their effectiveness is when they are not easily associated with their activities. In

the same way that activities need not correspond one-to-one with programs, it is also true that activities need not correspond perfectly with windows. When there is no visible unit on the screen that is uniquely associated with an activity, there will be no direct way to be reminded of that activity. For instance, the display of a calendar program may not serve as a useful reminder for an activity that the user conceives of primarily as sending a message. Likewise, if a window is being used by three different activities at once, it cannot effectively remind the user of all of those activities.

Given the shortcomings of windows as reminders, it is interesting to note how far the Symbolics Lisp Machine has moved away from the simple use of windows as reminders. One reason for this is that it makes such extensive use of windows within a single activity that most of its window operations (moving, reshaping, etc.) are commonly enlisted to serve the current activity. In order to compensate for this, there are special predefined keys for retrieving particular windows (e.g., the "Lisp Listener" window or the "Flavor Examiner" window) or the "previous" window. This obviates the need for a window to be visible in order to select it.

Reorienting

Once you have successfully located the activity you want to resume, you may still be in need of some assistance from the computer: If you are returning to an activity after a lengthy digression, you may have difficulty remembering where you left off. It is generally the case that you need more information to resume an activity than you need when you are engrossed in performing that activity. For this reason, it is valuable if systems can provide additional visual and contextual cues to reorient the user who is trying to resume an activity. Text editors, for instance, maintain a considerable amount of hidden information: There are buffers containing text to be copied or moved, and pointers to important locations in the file. On returning to an editing session, then, it may be very helpful if the editor lets you display the contents of these buffers and pointers. Also, a "movie" of earlier screen images and a listing of the commands given prior to the digression may help you to remember where you left off.

User-Centered Activity Management

I have emphasized the distinction between a user's *activities* and a computer's *programs*. The interfaces of traditional operating systems take the computer's programs to be central and leave it to the users to

manage their activities. What if the interface took the users' activities to be central? This means that the top-level units that mediate the users' interaction with the computer would be activities rather than programs. Of course, the users would still have to deal with computer programs, but the programs would now occupy the second tier of conceptual units. The interaction would be "user-centered."

Let me describe a session on a hypothetical system with this sort of "activity manager." The interface will maintain an "activity list" of all of the activities which I have started but not completed. In addition, it will have available a list of recent activities which have been completed. The reason for including a list of completed activities is that the history lists which I have examined contain a surprising number of cases where an activity that was presumed to be complete is "resurrected" some time later. A glance at Figure 12.1 will convince you that **reposition window[2]** is an example of a resurrected activity, and that **arrange a meeting** [5] was resurrected later that day. This list of completed activities will serve a function similar to an "undelete" command.

An activity manager that keeps a list of all interrupted activities opens up the possibility of maintaining activities across sessions with the computer (Bannon, Cypher, Greenspan, & Monty, 1983). The activity of writing a program, or a paper, may span several weeks, and it would be convenient for the system to keep track of your progress on such an activity and the associated computer context. There are also activities which one wants to perform once a day, or once a week. An activity manager could insert reminders for these tasks into the activity list. Furthermore, such an activity list could be conveniently used as a "todo" list—a place to jot reminders about tasks that need to be done in the near future. The "displacement activities", which are discussed in Chapter 17 by Owen, also have a place in the activity list. These activities are special, in that the user ordinarily wants them hidden from sight. But at those times when the user has nothing special to do—or actively wants to avoid some particular chore—this list of diversions could be consulted.

When I log on to my computer at the start of the day, it displays the screen just as I left it when I logged off. The activity list reminds me of things I have to do, and the windows for my current activity are already open. I choose to read my new mail first, so I temporarily hide away the current activity. Since all of its windows are linked, they all disappear at once, leaving me with an uncluttered screen. I read my mail and move two of the messages to my activity list. They are automatically assigned their own instantiations of the mail program which will handle any future correspondence on these topics. Finished with my mail, I push the "Resume" key. I need not recall what the

original activity of the day was, as the activity manager has remembered this for me.

A few minutes later, the "debug program W" item in my activity list begins to flash. I select this activity and see that my colleague has answered my query. She has figured out how to fix this troublesome bug, so I make the change and recompile the program. Without waiting for it to finish, I return to my original activity and complete it shortly thereafter. I then decide to make a modification to a paper I am writing, but suddenly the phone rings, so I jot "clarify X" on my activity list and grab the phone. The caller informs me of a terrible bug in program Y, so I turn to my table of "standard layouts" and select a "debugger."

Because certain common activities (like debugging) always require a particular set of programs, and because users become accustomed to certain ways of laying these programs out across various windows and screens, the activity manager maintains a table of a user's "standard layouts." So when I select the "debugger" layout, a collection of useful tools arrange themselves on my screen.

I soon tire of the debugging chore and I look for a change of pace. I see several choices in my activity list, but I am unable to recall what they refer to. So I ask for more information, and I am treated to miniature displays of the various activities. For some activities I am shown a miniature screen, for others, a particularly representative window. For yet another activity, a short descriptive paragraph that I wrote months ago appears. Chapter 16 by Draper suggests various ways for allowing the user to choose the amount and type of information that is displayed for each activity.

The Notepad Program

The Notepad program implements some of these ideas about activity management. Like the ThinkTank program for personal computers, it is designed as a tool for "thought-dumping"—the process of quickly jotting down a flood of fleeting ideas. Thought-dumping places a premium on the ability to record an idea with the minimum of interference. In Notepad, notes are organized by assigning titles to them and filing them in bins. These operations can be postponed arbitrarily so that they will not interfere with the process of jotting down ideas. In addition, the process of writing a note can be interrupted at any time by the Title and File commands.

Notepad has explicit activity management commands. If the current idea is suddenly interrupted by a new idea, the user gives the "Interrupt" command. This puts the current idea on an "Interrupted Notes" stack and supplies a blank page for the new note. The stack is

constantly displayed to the user and serves as a reminder of interrupted ideas. When the new note is complete, the user gives the "Resume" command to resume the interrupted idea.

The use of a stack means that Notepad embeds activities; resuming activities in an arbitrary order is not supported as well. Notepad supports postponing with a special "Jot" command, which allows the user to jot down a short reminder about a new idea without having to leave the current idea. This gets the idea out on the table and allows Notepad to remind the user to pursue it.

Problems With Activity Management

An activity manager can take over some of the user's burden of managing activities, but it is restricted by what it knows about the activities. Many user activities are performed only partly on the computer. Various subactivities and meta-activities may be performed by talking to a colleague or by simply remembering them. This can cause significant problems for an activity manager because there are arbitrary holes in its picture of the activity. If the user consults an on-line manual, the computer can remember this activity when it is interrupted. But if the user consults a printed manual, or a local expert, the manager will be unaware of the activity. Similarly, a user who interrupts the activity of debugging a program may be unable to remember where to pick up the task once it is resumed, because the interrupted activity was some complicated reasoning process that was totally within the user's head.

ACKNOWLEDGMENTS

I would like to thank Autumn Chapman for designing the original activity flowcharts, and Steve Draper and Dave Owen for providing many helpful comments. I would also like to thank Don Betts for designing and drawing Figure 12.1. The impetus for the study of activity structures came from the Activity Structures working group, consisting of Lissa Monty, Liam Bannon, Steve Greenspan, and myself. This early work was published as "Evaluation and Analysis of Users' Activity Organization" (Bannon, Cypher, Greenspan, & Monty, 1983). This group developed many new ideas about creating a complete working environment for each user activity. I am particularly indebted to Lissa Monty for her ideas about gathering history list data, and to Liam Bannon and Steve Greenspan for their ideas about workspaces.

APPENDIX: COLLECTING HISTORY DATA

Activity flowcharts are prepared from an "annotated history list"—a record of all of the commands that the user types during the computer session. The history list records the linear stream of commands that the user performs to carry out his or her activities. If activities are performed one at a time, the commands will cluster into neat, separate packets, whereas an interleaved activity will have its commands sprinkled throughout the historical record. I refer to these lists as "annotated" because users can add explanations of what they are doing, as they are doing it.

The history list used in this chapter was recorded from a Sun Workstation, that has a bit-mapped display and a window system. It uses Berkeley UNIX. Because this system uses both windows and the Stopped Jobs facility, it provides considerable support for multiple user activities. Had these data been collected from a personal computer, I would expect the results to be quite different. The software currently available on personal computers is generally quite unsupportive of multiple activities.

To record these data, the major interactive programs on the system—the top-level (shell) interface, the text editor, the mail program, and the note program—were modified to send a parsed copy of every command to a central collection program. The collector receives commands from all of the windows in whatever order they are performed, so that embedded activities show up as embedded commands. A segment of a history list is shown in Figure 12.2.

This collection technique has an advantage over the standard technique of recording keystrokes in that it lets each program parse the input string into a meaningful command. However, it is important to note the limitations of this history-list data. Only the user's commands are recorded; there is no record of the computer's responses or of computer initiatives (such as "you have new mail"). This unfortunately reinforces the stereotypical view that "the user commands, the computer performs." It makes it difficult to appreciate situations where the user and the computer are working together, like co-routines, and situations where the computer is initiating activities.

Another problem is that the history list includes only those parts of the activity which actually take place on the computer. For instance, consulting another person is an unrecorded activity, so social interactions of the sort discussed in Bannon's Chapter 21 are missing from the record. The user can mitigate this problem somewhat by making annotations about relevant unrecorded activities.

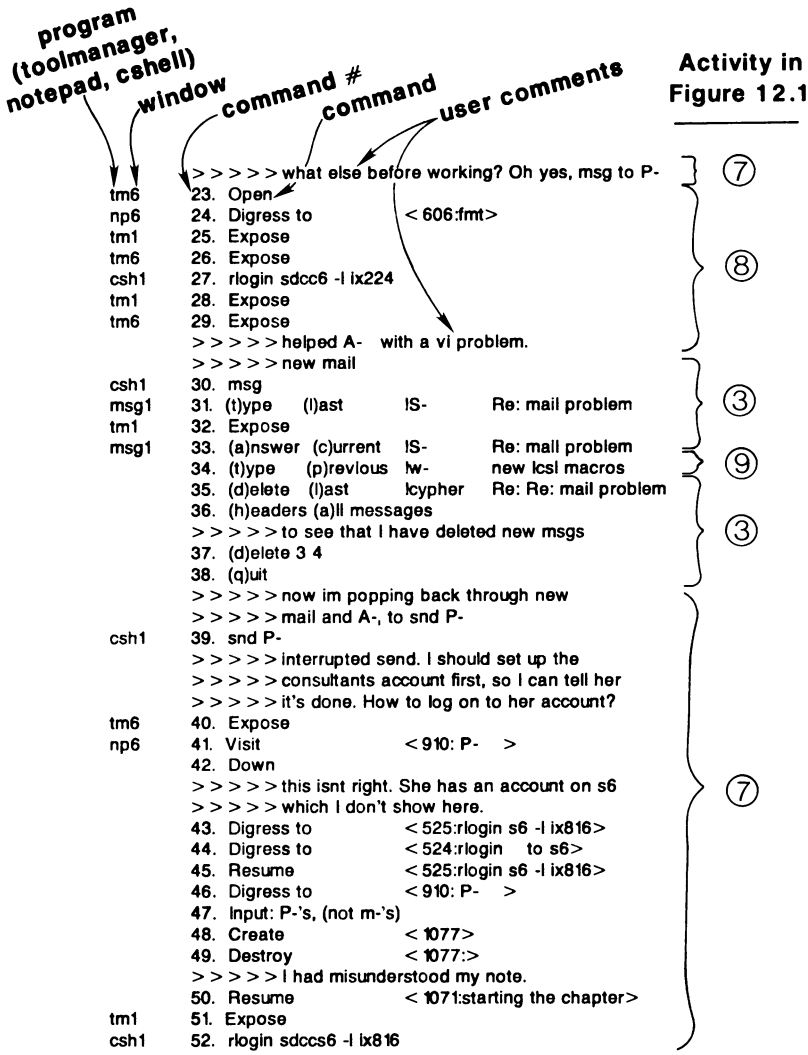


FIGURE 12.2. A segment from the annotated history list for the session shown in Figure 12.1.