CHAPTER 4

Sharing and interference in window managers

4.1 Introduction

Multi-windowed systems allow the user to compare several related displays on the same task, or to intersperse activities on more than one task. Both add to the complexity of the tasks that can be performed and to the complexity of the interface. It is the latter on which we will particularly focus in this chapter. In particular, we will be interested in the way that activities in different windows may interfere with one another and hence how the tasks associated with these windows may interfere.

First of all, to put our discussion in context, we will consider different types of window manager and their purposes. We will compare the MAC-type window manager with the sorts of windows typical in programming environments, and for the latter discuss the problems they are designed to solve. Section §4.3 will then discuss informal concepts associated with sharing, concepts such as level, granularity and perspective. We will reject models based around the computer's data model and instead opt for a behavioural view of sharing.

We will then get on to define a model of window managers suitable for expressing properties of dynamism and sharing, the *handle space*. Previously this has been called a view space (Dix and Harrison 1986) but the less evocative name is used here to avoid confusion with the "views" of Chapter 9. In section §4.5, we will use this model to give definitions of two types of sharing: *result independence* and *display independence*. In addition we will discuss general properties of these and other dependency relations. A formal definition of independence could be used during design to stop a windowed system having any sharing at all; however, in practice this will not be a totally desirable goal (we want some sharing). Because of this, the next section goes on to discuss the

various ways the dependency structure of the windows can be used to improve the user interface and reduce the disorienting effect of unexpected sharing. If this sharing information is to be used, there must be effective procedures for detecting it. Section §4.7 discusses the various ways this can be achieved for different types of sharing. It also mentions the far more difficult issue of detecting interference induced by the user's own model.

4.2 Windowed systems

4.2.1 Windowing styles

There are two major genera of windowed systems, the first aimed primarily at office automation, Xerox Star, Apple Lisa and the ubiquitous Macintosh. The other major strand comes from programming environments, principally Smalltalk, Cedar and InterLisp. Most traditional Unix-based windowed systems lean towards the latter style, but a few, such as Torch Opentop and X-Desktop, are aimed towards the former (reflecting the expected markets). Comparing the MAC-style interface with these UNIX-based systems uncovers some important distinctions in the purpose and paradigm of mult-window environments.

MAC style

The metaphor that the MAC (and its predecessors) follow is physical, the desktop. Windows and icons are related to manipulable objects. You select objects and you do things to *them*. That is, the screen model reflects the *computer data* model. The purpose of the interface is to open up the guts of the computer and remove as much hidden functionality as possible.

A consequence of espousing a physical metaphor is that users will infer that it obeys physical laws. This is exploited when, for instance, selecting an object and dragging it about is very similar to the corresponding physical action. This physical metaphor does have important repercussions for sharing. In the physical world, when you do something to one object, you do not expect other objects to be affected unless they are physically connected: action at a distance, magnetism or radio controls have an almost magical air to them and are "not really" physical. This would cause problems if several windows were allowed onto a single database. If you were allowed to "peel off" several database windows, then these physically separate windows would be secretly linked. Alternatively, if you opened the database as single window, and within this application had several subwindows, these subwindows would have to be considered as different in kind from the main windows (even if they share the same physical appearance and controls). Whichever way the problem is tackled the user is involved in a context switch when changing from normal physical

windowed work to the database.

UNIX style

At the simplest level this can be regarded as the multi-tty style: the windows are virtual terminals, some graphic, some character-based and these are multiplexed onto the physical screen. At this level the window manager is a *screen resource manager.* Each window corresponds to an application or a standard terminal session, and looking at it this way we see that the screen model most closely reflects the *computer process* model.

One reason for having several parallel terminal sessions, is in order to engage in several tasks and considered thus the screen model to some extent reflects the *user task* model. Viewed as such we could say that the window manager multiplexes the user, or is a *user resource manager.* We will examine this aspect and why it is necessary in the next section.

4.2.2 Handling multiple user tasks

To see why multiple windows are necessary for handling multiple user tasks, we consider a typical terminal session:

```
% type old_prog.c
    ....
% edit new_prog.c
    ....
    ....
% compile -Debug new_prog.c &
% edit letter
    ....
    ....
% typeset letter
% run new_prog
    ....
    ....
```

First of all the user looks at an old program; she wants to write a new but similar program. Having examined the old program, she then proceeds to create the new program using the editor. When she has finished, she compiles her program as a background process (indicated by the &). To fill in the time while it is compiling she types in a letter and then gets it typeset ready for mailing. By this time the program has compiled and she runs it to see if it has worked as expected.

Looking at this example we can see first of all that there are two main tasks, writing the new program and writing the letter. The user will experience several problems when trying to perform these two tasks:

- *Granularity of interleaving* – The user switches between the tasks but the granularity of interleaving is gross, at the level of complete invocations of applications. Whereas the computer when it is compiling concurrently with running the editor will interleave these at the level of arbitrary groups of machine instructions, the user cannot easily switch back and forth between editing the program and the letter.
- *Loss of context between tasks* – When the user compiled the program she specified a debug option. By the time she comes to run the program, the editor will have cleared the screen and the compile command will no longer be visible for her to refer back to.
- *Loss of context within a task* – Even when looking just at the programming task we see that the listing of the old program which is being copied from will be cleared from the screen by the editor, making it difficult to refer back. Many editors allow the user to execute other commands "shell out" while in the middle of an edit, partly relieving this sort of situation, but of course this is still a very gross and unnatural level of interaction.

Programming environments' window systems are an attempt to deal with these problems. Separate tasks are run in separate windows, so the interleaving can be as fine as the user wants and is limited only by quirks in the system (like dialogue boxes or menus that take precedence over normal windowing) and the cost of popping the required windows (Card *et al*. 1984). Because the tasks are in different windows, when a task is resumed its window is as it was when it was left, thus dealing with loss of context between tasks. In a similar way applications that use up a lot of screen space, such as editors, are usually run in separate windows to avoid context loss within tasks.

This example also gives us examples of two of the ways sharing can arise in windowed (and non-windowed) environments:

(i) We said that the user would know that the compilation had finished. Quite likely she would know because it would have displayed diagnostics all over the screen while she was in the middle of editing her letter! The two tasks seemed totally independent with no shared data, but they interfere with one another because they share the screen as a common resource. Having the two tasks running in different windows would not remove this problem completely, as the user might well have started preparing a data file for the test run while the program compiled. This is often particularly disconcerting, as having entered the command invoking the background process, the user will reach closure and forget about it.

(ii) The second example is due to shared data. If several windows belong to one task, as would happen if editing the program in the example produced a separate window, then the two windows would both be acting on `new_prog.c` and the user may accidentally invoke the compiler without

writing the edited file. Again this is a closure problem, as the user reaches closure when the new program is completed, but before it is written. The non-windowed version did not have this problem, as it was impossible there to start the compilation until the edit was complete; however, it would be possible to edit the program while the compilation was proceeding in the background, a common error. On balance though, these closure errors do seem more prevalent when the user has several concurrent activities.

4.2.3 Importance of simplicity in windowing systems

It is especially important that users can use a windowed system without regard to the complexity of the windows themselves, as they are likely to be using the windows to offset mental load.

Two analogies of user–computer interaction can help us to understand this process.

(i) Card *et al*. (1984) suggest that we can draw an analogy between the user's use of windows and a computer's use of virtual memory. Essentially the screen corresponds to the primary storage, which is not big enough for all the memory requirements (the windows) and therefore some information is consigned to secondary storage (hidden windows).

(ii) Young *et al*. (1989) and Runciman and Hammond (1986) suggest the use of programmable user models to estimate the cognitive load of a system. This essentially draws the analogy between the user's process of using the system and a program to use it. Limitations to the ability to perform a task would include its time/space complexity, especially as regards short-term memory. If using a system exceeds these limits the user's performance will either degrade or they will have to off-load some of the memory demands, perhaps using windows to store information that would otherwise have to be remembered. We can liken this very small short-term memory to the registers in a CPU, and we get the following combined model:

user's short-term memory	—	visible windows	—	hidden windows
	(i)		(ii)	
CPU's registers	—	primary storage	—	secondary storage

Typically a user will gather information onto the visible windows to perform some task; this process may possibly use information from hidden windows. Because people tend to drive their abilities to the limit, and because of the cost of transferring information between memory media, the user will probably only use each method of off-loading information when they are close to their cognitive

capacity. Therefore any additional load caused by the volatility of the storage media (lack of display independence), can be disastrous in terms of lost efficiency. In particular, events that require movement between visible and hidden windows will cause the user to forget information relevant to the present task in this "system" task of paging.

In an attempt to reduce the cost and complexity, Card and Henderson (1987) designed a windowed system around the idea of rooms, wherein one performs specific tasks, and in which are collected the various windows connected with a particular task. Additional methods to reduce the need for paging might include mechanisms by which the user can view critical parts of hidden windows on the visible screen; these include the use of extensive folding to minimise window size, or active icons to display status information such as compilation progress.

As well as the direct cost of paging it is necessary to consider the complexity of the interaction between different windows: any unexpected interaction will add mental load to a system which should be reducing it.

4.2.4 Summary

There are two major genera of window manager, one shaped around the computer's own data model, and the other around the support of multiple user tasks. Problems of interference between tasks are very important, especially if the windows are used to hold contextual information when task switching. This chapter is aimed mainly at the task-based view of window managers, although much of its analysis will be applicable to the MAC-style as well.

4.3 Sharing, informal concepts

4.3.1 Levels of sharing – actors

As we have seen, problems of sharing can occur in both windowed and non-windowed systems; however, the separateness of windows tends to emphasise independence and thus interference can be more disconcerting. We can classify the types of sharing that can occur at three levels:

- *Several independent actors* – This is what is normally understood as sharing, in database applications for instance, where several different users may access the same piece of data or possibly a user may access data at the same time as a system process.
- *One controller – several actors* – This is where the user sets off one or more background processes and these interact with each other or with the user's foreground activities. As all the processes are under the user's control she should in principle be able to predict the sharing. However,

especially if one of the processes is invoked long before another, she may not. Further, the form of interference may not be obvious, as an application with a window of its own may still print certain error messages in its parent window, which the user may no longer associate with it. As we have seen, this is not a problem merely of windowed systems.

- *One actor – several personae* – This is a particular problem of windowed systems: two windows may be accessing shared information and, for instance when examining data from several perspectives, perhaps necessarily so. As the user is consciously acting on each window, one might argue that she had little excuse for errors arising from this type of sharing; however, if the user is context switching between several tasks she is in effect acting as several personae and should not be expected to carry over information from one to another.

When thinking of the particular problems of windowed systems it is this last type of sharing which we will have predominantly in view. It is not confined solely to windowed systems: for instance, a clerk may be entering sales figures into a database but, when he has spare time, trying to produce sales totals, and the two tasks may easily be performed on a traditional terminal system in the same way as the two tasks in the programming example. The reason for saying that this is particularly important in windowed environments is the frequency, pace and granularity of context switching which is rarely encountered elsewhere. Having said that it is this last type of sharing that is of interest, most of what follows could be applied to all of them.

4.3.2 Granularity of sharing

Although we have been talking about sharing, it is not obvious what precisely is meant: the three levels given apply to who or what experiences the sharing, but there are also obviously different types of sharing, for instance sharing of physical resources such as the screen, or information resources such as databases. If we are to understand sharing and perhaps detect and warn the user of it we must define it more precisely.

The most obvious option is to look at some sort of data model such as files, relations or records, then say that two windows or processes interfere if they have access to a common resource. This approach has two major drawbacks:

- *Granularity* – What level of resource granularity ought we to use? Should we say two windows share if they have access to the same file system, file, record or field? If we opt for the smallest level possible, then we might say that if two windows are editing opposite ends of the same document they are not sharing, is this acceptable?

- *Choice of perspective* – From what perspective ought we build our data model? Do we say windows share if they have access to the same record or the same disk block? At a very concrete level, if a process uses up all the free space on a disk, any other processes accessing that disk will be interfered with. Logically equivalent database designs may have very different expressions in terms of file and record structure, giving rise to different definitions of sharing for the same logical design.

When considering these points we realise that a definition based around a specific data model is not sufficiently abstract; further, it may be both too strong and too weak in its definition. It may be too strong in that it gives rise to spurious sharing, for instance if two processes access the same record but are using logically distinct parts of it. It may be too weak in that it may ignore less obvious paths of interference. The most obvious case concerns consistency relations on databases, such as not being able to delete a non-empty directory. Further, by concentrating on the data model one ignores other links in the system, such as interprocess communication. Lastly and most difficult to detect are indirect links, where one process does something which causes an intermediate process to behave differently, thus affecting a third process. We cannot deal with this last problem simply by taking the transitive closure of our dependency relation as this would almost invariably be far too strong: in most operating systems, all processes interact with the kernel and would hence all be said to be interdependent.

We can avoid these problems by taking a *behavioural* or *implicit* definition of sharing as opposed to the constructive or explicit definition implied by a data model. This is (of course!) also more in keeping with our surface philosophy for describing interface issues, and we will define sharing in terms of the observed effect at the interface.

4.4 Modelling windowed systems

4.4.1 Fundamental features of windows

In order to build a model within which we can give an implicit definition of sharing, we will have to decide what are the features of windowed systems that are relevant and that should be included in the model. The first thing that we will abstract away from is spatial positioning of windows and the method of selecting them. These features are relevant for the user, but in this chapter we consider only the computer's implicit data model. In essence, we assume that we are looking beyond the physical screen and have unrestricted access to the entire array of virtual displays, in the same way as we might abstract beyond the physical display of a word processor and consider the document being edited.

Having so abstracted there seem to be two remaining essential features:

- *Content* – This concerns what is actually displayed at the window, whether graphical or textual.
- *Identity* – We assume that the window can be regarded as an object in its own right, and can be identified and manipulated as such. The identity of windows is important when considering change in time: as a window's content change it still remains the same window.

Some user actions may be addressed to a single window: for instance, we might set compiler options by opening up the compiler icon then typing them into slots in a form. Other actions may require two or more windows: for instance, we might compile a file by dropping the file icon onto the compiler icon. Some actions may require no windows at all: for instance, a global undo as a menu option. All these cases can be managed quite easily, but to avoid complexity we will confine ourselves to unary commands, limiting consideration to UNIX-style rather than MAC-style windows. The extension to the n-ary case is straightforward but increases the combinatorics with little gain in understanding.

4.4.2 Aliasing

It was said above that we were going to ignore presentation issues. However, there is one important issue that we should discuss before continuing. Several authors have produced specifications of window and graphical presentation managers. (cd) These are at the level of specific windowing policies and are aimed at defining the graphical nature of these systems. It is also useful to tie in the underlying applications to the presentation component at a more generic level. Previously, (ce) I have experimented with models of windowing based on collections of PIEs (as the underlying applications). The presentation component was left rather vague: the model assumed only that there was a selection method to choose the current window for input and that the currently selected window was always visible. This allows many different kinds of windowing policy, tiled or overlapping, and different kinds of selection mechanism. The model was also very uncommitted with regard to the applications, as the PIE model covers nearly all simple interactive systems. However, the very fact that it was a collection of different PIEs meant that the windows were necessarily independent and without sharing.

The intention was to prove that if the component systems had various predictability and reachability properties, then so would the collected system. This sort of modular proof is of course very powerful in building systems, especially because of the wide applicability of the models concerned. In fact, the experiment was largely a success, limited only by the simplicity of the data model (which is addressed in this chapter). The proofs all fell out as expected.

In order to prove predictability properties (not surprisingly), I had to assume that there was some way of telling from the display not only *what* was in the current window, but also *which* was the selected window. Now by this I mean not only which of several displayed windows was the selected one (although this is part of the problem), but which underlying application was associated with the current window. To paraphrase in the terms introduced a short while ago, we need to know not only the *content* of the currently selected window, but also its *identity*.

What does this "proof problem" mean in practice? The necessity in the proof arises because in some circumstances two applications may be different but may temporarily have identical displays. Thus knowing what is displayed in the current window does not tell us which application is selected. We recall that a similar problem arose when we considered a simple model to express WYSIWYG (§1.5). There we could not tell where in a document a part of it belonged because different sections could look the same. So this is another instance of *aliasing* as applied to window managers.

Now the language we have just developed for windowing applies reasonably well for editing systems. We need to know not only the content of the visible part of the document (or diagram) but also where in the whole it is (the identity). We could therefore define the problem of aliasing as: *content does not determine identity.*

When we discussed aliasing before, we discussed solutions such as scroll bars. We can look for similar potential solutions for windowed systems. Most window managers allow title bars naming the windows. Assuming the naming is unique, this would (at least at a formal level) solve the aliasing problem. Of course, people may not notice the banners, so this may not be sufficient. Also in many systems the name bars are heavily under-used (most of the windows on my screen are called "shell" or "spy"). Screen positioning is also a very powerful cue to users, as are window shapes, colours, etc. Most probably, window appearance would be determined by the function being performed (editing, compiling, mail, etc.), but it is most likely that aliasing problems would arise between different instantiations of the same function. Perhaps appearance and naming should be associated more with topic than function?

So we have seen aliasing raise its head again. For the rest of this chapter we will assume that aliasing has been dealt with at the presentation level, and that we can look beneath this. That is, we assume that there is a collection of windows and that it is possible to associate the content of windows with their identity, and to address commands to any window (by identity) at any time.

4.4.3 A model of windowed systems – handle spaces

We now need to flesh out the assumptions and frameworks of the previous sections at a formal level.

We will call the set of machine states E and the set of commands C, to emphasise similarity to the red-PIE. We also have a result space (R) and a result map:

$$result : \; E \; \rightarrow \; R$$

The difference here is the set of *handles* (Λ) used to embody the idea of identity. These handles are not meant to represent anything meaningful to the user and are purely a device to represent the fact that the user can identify the display of a particular window and address commands to it. At any time only a subset of handles will be meaningful; we will call this set the *valid handles* and we will represent it by a map:

$$valid\_handles : \; E \; \rightarrow \; \mathbb{P}\Lambda$$

The result is defined without respect to a particular handle, and would probably be the file system or something similar. The display, on the other hand, depends on the window, and this is represented by including a handle as parameter:

$$display : \; E \times \Lambda \; \rightarrow \; D$$

This is a partial map, however, only defined for valid handles:

$$e, \lambda \in \mathbf{dom}\ display \quad \Leftrightarrow \quad \lambda \in valid\_handles(\ e\ )$$

In terms of windows, we cannot ask for the display of a window that doesn't currently exist! In a similar way, commands are directed to a particular window, and hence the state transition function *doit* also includes a handle as a parameter and a similar condition:

$$doit : \; E \times C \times \Lambda \; \rightarrow \; E$$

$$e, c, \lambda \in \mathbf{dom}\ doit \quad \Leftrightarrow \quad \lambda \in valid\_handles(\ e\ )$$

As we've said, we will restrict ourselves to analysing the case of unary commands, and the model has therefore exactly one handle parameter for update. For different purposes it is easy to develop variants allowing allowing n-ary and 0-ary handle parameters.

Many different kinds of system as well as window managers can be described using this model. For instance, it may also refer to the internal process model (assuming this is not in one-to-one correspondence with the windows) where each handle is a process, or to the file system where each handle is a file name. In UNIX files are referred to at a deeper level by numbers called i-nodes and

these are handles at a different level of abstraction, so both the file system with file names and i-nodes and the windows and processes can be regarded as handle spaces at different levels of abstraction. With the file system, we get at least two different handle spaces depending on whether we take the commands available to the programmer, or the commands available to the user. With the latter view, we would say that typing into a file was a unary operation, copying from one file to another was binary (requiring the more complex handle space model), and an editor which can work simultaneously on several files would be n-ary. A 0-ary operation might be file creation. The file system with file names is probably easiest to conceptualise but is not perfect as a handle space, as the presumption is that the handles themselves have no meaning whereas this is not usually true. However, it applies at a certain level of abstraction.†

In each of these example we can identify phases that are *static* or *dynamic* in terms of the handles available for operations. In the file system, editing a file is static in that the set of available files remains constant, whereas file creation and deletion are dynamic. Similarly, window creation and deletion are dynamic events.

The same command may be static or dynamic in different circumstances. For instance, looking at the process-level abstraction, if an editor is in insert mode "x" may type as itself, whereas if it were in command mode it might exit and hence destroy the process. We must therefore define whether a command is static in relation to a particular handle in a particular state:

$$static(\ e, c, \lambda\ ) \quad \equiv \quad valid\_handles(\ doit(\ e, c, \lambda\ )\ ) \ = \ valid\_handles(\ e\ )$$

Having said all that, however, in most systems many commands are always static, so we can define a predicate to apply to commands:

$$static(\ c\ ) \quad \equiv \quad \forall\ e, c, \lambda\ \in \textbf{dom}\ doit \qquad static(\ e, c, \lambda\ )$$

For most of this chapter, we will restrict ourselves to looking at the sharing properties of static commands, to avoid dealing with two types of complexity at once! We will return to dynamic commands in §4.8, when we will consider the possible principles for them.

---

† This is similar to the problem in denotational semantics where the denotation of a fragment of program will abstract away from the textual form and hence lose connotation:
`year_average = year_total/12` and `week_average = week_total/12`
are not equally valid.

4.5 Definitions of sharing

As we've said earlier, we are looking for some sort of implicit or behavioural definition of sharing. We will consider two ways of measuring independence of commands issued to windows: using display or using result. By comparing their properties we are led to consider several attributes that will distinguish general dependency structures.

4.5.1 Result independence

The result map is defined to be the final result of the system as a whole and is thus, in a sense, more important than the ephemeral displays. In the simple model of a window manager where each process had its own private display, result and state, it was easy to see that each process was completely independent, because they didn't change each other's internal state. However, even when processes use shared resources, file systems, databases, etc. they may still be accessing "independent" parts of the resource and at least over a subset of their available commands be "independent" themselves. We need to characterise this independence externally to a detailed breakdown of particular shared states which may hide some interdependencies and imply others which don't exist. We can do this using the handle space model and the notion of commutativity. When we are interested only in the final state of something this is the natural method of definition and is used, for instance, in deciding what database updates can be executed, locked and backtracked over independently. Essentially, two actions A and B commute if the combined effect of A followed by B is the same as that of B followed by A. So we say that two static commands c_1 and c_2 issued to handles λ_1 and λ_2 in state e are *result independent* if:

$$result(\ doit(\ doit(\ e, c_1, \lambda_1\ ), c_2, \lambda_2\ )) = result(\ doit(\ doit(\ e, c_2, \lambda_2\ ), c_1, \lambda_1\ ))$$

Essentially, this is drawing an analogy between the user with multiple windows and a scheduler with several processes. The scheduler has an easier job if there are no hazards caused by interference, and by analogy if those hazards are not present or minimised between windows the user's job will be eased.

We may want to strengthen this definition so that not only is the current result the same but all future results are the same no matter what *doit*s follow. To do this we need to define result congruence ($\equiv_r$) on the state space E. This is defined inductively by:

$$
\begin{aligned}
e \equiv_r e' \quad \equiv \quad & result(\ e\ ) = result(\ e'\ ) \\
& \textbf{and } valid\_handles(\ e\ ) = valid\_handles(\ e'\ ) \\
& \textbf{and } \forall c \in C, \lambda \in valid\_handles(\ e\ ) \\
& \qquad doit(\ e, c, \lambda\ ) \equiv_r doit(\ e', c, \lambda\ )
\end{aligned}
$$

We can then say that c_1 and c_2 issued to λ_1 and λ_2 in state e are *strong result independent* if:

$$doit(\ doit(\ e, c_1, \lambda_1\ ), c_2, \lambda_2\ ) \equiv_r doit(\ doit(\ e, c_2, \lambda_2\ ), c_1, \lambda_1\ )$$

We may also want to say that two commands are independent only if they commute in all contexts. However, because the same command given to different handles may have completely different interpretations, we will almost certainly either have to limit ourselves to a specific pair of handles or introduce a type structure for handles. If we take the first case we need to restrict ourselves to the set of contexts that preserve the nominated handles (λ_1, λ_2, say). That is, we would demand the commands to commute when issued to λ_1, λ_2 in any of a set of states (E') defined recursively by:

$$
\begin{aligned}
& \forall e \in E', c \in C, \lambda \in valid\_handles(e) \\
& \qquad \lambda_1, \lambda_2 \in valid\_handles(\ e'\ ) \implies e' \in E' \\
& \text{where} \\
& \qquad e' = doit(\ e, c, \lambda\ )
\end{aligned}
$$

The second option, of typing the handles with a type set T and a typing function $type : E \times \Lambda \rightarrow T$, would lead to a definition of *typed result independence* for commands c_1, c_2 issued to handles of type t_1, t_2:

$$
\begin{aligned}
& \forall e \in E, \lambda_1, \lambda_2 \in valid\_handles(\ e\ ) \\
& \quad type(\ \lambda_1\ ) = t_1 \ \textbf{and} \ type(\ \lambda_2\ ) = t_2 \\
& \qquad \implies doit(\ doit(\ e, c_1, \lambda_1\ ), c_2, \lambda_2\ ) \equiv_r doit(\ doit(\ e, c_2, \lambda_2\ ), c_1, \lambda_1\ )
\end{aligned}
$$

4.5.2 Display independence

When discussing result independence we said that the result was what was really important. However, if we take human factors into account we must modify this position somewhat, firstly because we are interested not only in the functionality of the system but also in its impact on users, and secondly because what the user *does* depends on what he *sees* and thus the result is affected indirectly by changes in the display. For instance, if a message is sent to a window in which there is a screen editor, if the user does not notice the message arriving then he may enter erroneous commands based on the corrupted screen.

We can state a simple form of display independence easily by saying command c issued to λ is display independent of λ' in state e if:

$$display(\ e, \lambda'\ ) = display(\ doit(\ e, c, \lambda\ ), \lambda'\ )$$

That is, the command issued via window λ has no effect on the display in window λ'. We can extend this definition in similar ways to that for result independence. There are, however, some major differences between the two:

- Result independence (and its negation, result dependency) is symmetric. Display independence, on the other hand, is directed: we can talk of one window/process affecting another without the opposite being true. This is especially true where the second window is a read-only view of a data structure being manipulated in the first: for instance, a formatted view of a text.
- If a set of commands and handles are pairwise (strong) result independent, then they may may be composed in any order without affecting the result. Pairwise display independence, on the other hand, does not in itself imply any group property.
- As formulated, display dependency is an event at particular instant: this command to this handle changes that display *now*. Result dependence is harder to pin down, as it involves two commands, and we need to ask which command actually causes the sharing.
- If the handle space corresponds to an underlying data model then display dependency can be inferred "from the outside". Thus a window manager would be able to detect and report such dependency. Result independence is much more difficult to infer "on the fly", and probably requires some extra information to be passed from the underlying data to the window manager.

4.5.3 Observability

We recall the example given above of the message arriving on a screen editor window and causing command errors. It has its impact because the user was expecting a correspondence between the display and the object being edited, a correspondence that was disrupted. Thus the lack of display independence led to a lack of ability to observe the object and predict the outcome of actions. This effect is exacerbated if the individual processes are well behaved, being themselves observable and predictable. (Dix and Runciman 1985) In this case the user may have inferred these principles and will rely on them. The scenario presented is not unlikely, as, in many window managers, utilities frequently report errors back to the parent window, a relationship that may well be forgotten by the time an error occurs.

One reason for the problem is that many utilities are defined in terms of the changes they make to the screen, because this is how they are implemented, and the window manager's job is to ensure that those changes are carried out. If the

contents of the window are not as expected then the changes may be meaningless. If instead the window manager were defined in terms of preserving a correspondence between the screen contents and some description of it in the individual process, then this would not occur so easily. (Note that this does not mean that the implementation cannot make use of change information supplied by the process but merely that the *definition* has this property; the separation of definition and implementation implied by formal specification techniques facilitates this). Gosling and Rosenthal (1985) noted that when the style of the window manager implementation requires the individual processes to be able to regenerate the screen contents, this imposes a certain beneficial discipline on the programmer.

Even if the user is aware of the change to the current display, there may still be a significant increase in the cognitive load as the user tries to reconstruct the desired display. This load is especially onerous as the user is likely to be using the window manager precisely to off-load such effort.

4.5.4 General dependency

The above definitions are not the only structure on the processes presented by a window manager. As we have noted before, there will be user model structure and explicit data model structure as well. The importance of the above definitions in comparison to the explicit data model is that they refer to *implicit* dependency that might otherwise be missed. This is especially important as we would expect the user to be more aware of the explicit structure anyway and thus find non-explicit interference more difficult to assimilate. This could be an argument for demanding of the designer of the underlying data abstraction, that all implicit interdependence be contained within the explicit model: this has been a major concern of database theorists in the definition of the relational model and onwards. If the window manager could be assured of this then its job would be much easier. The relation to the user model is more complex and is discussed later in this chapter.

As we have noted these definitions are by no means the last word, either in implicit or explicit structure; however, in comparing them we noted some important differences and these can be used to classify dependency structures whether arising from implicit or explicit data structures, or from the user model:

- *Symmetry* – Is the relation between objects symmetric or is there a particular direction of dependence? This will appear in the explicit data structure, as for instance the difference between the "sibling" and "parent" relations for families, or between the "in the same module as" and "compiled version of" for program sources and objects. It also may appear in the user model: for instance, windows displaying documentation, running a spreadsheet and running a mail utility may all be in the symmetric relation of

being part of the task of discovering and reporting a discrepancy between the documented and actual behaviour of the spreadsheet. On the other hand, if I am reading the documentation in order to run a program then I could also regard the information flow as being more one way. As this last example demonstrates, the distinction is by no means as clean as it might be.

- *Group properties* – Can group properties be inferred from pairwise properties? This is especially important, since properties of the form "A doesn't affect C and B doesn't affect C, but A and B together do" are especially difficult to assimilate.
- *State and event properties* – Is the property to do with a pair or group of handles in general, or can the dependency be isolated to a particular event? Often a definition that is framed in one context can be moved to the other. For instance, as we noted, display dependence is essentially an event; however, we could change it to a relation between handles by saying that λ_1 affects the display of λ_2 if there is any command that can be issued to λ_1 that is not display independent of λ_2. Similarly, we could take the much more diffuse property of result independence and take a subset of commands for two handles that are all result independent and say that the event of result dependency occurs on the first command not in one of these sets. Alternatively, we may want to partition the result space by projections for each handle signifying the part they are "interested" in, and assign the event of result dependence in an identical manner to display dependence.
- *Inferring structure* – Can the property be inferred by the window manager, or does it have to ask someone (the data model or the user)? This is most important for the efficiency of the window manager. If the dependency relation is difficult for the window manager to detect, it probably means that it is difficult for the user to assimilate also and thus an efficiency issue becomes a cognitive human factors issue.

4.6 Using dependency information

Assuming there is some sort of interdependency between windows, and that it has been detected by the window manager, how are we to use this information? We consider two situations. First, we consider ways that the computer can make the dependency information available to the user. Then we look at the way that sharing properties influence our choice of undo strategy.

4.6.1 System response to sharing

The appropriate response to make in order to inform or warn the user of sharing will depend on the type of sharing that has occurred. Also we have earlier intimated that it depends on the user's task model.

Display dependence, we recall, is an event. Such event-based dependencies lend themselves well to some sort of *post hoc* warning, perhaps a bell or a flashing window border. As the focus of the user's attention is not on the changed window, one could argue for an audible signal in order to make the change obvious but, on the other hand, the problems arising from display dependence are to do with the user feedback loop and thus the signal could wait until the user actually focuses on the changed window. The latter argument would push towards a strategy such as altering the border of the changed window, possibly supplemented by a more explicit warning when it is next selected. A further advantage of this approach is that it does not interrupt the flow of work on the selected window. By the same token, however, by the time the changed window is selected the context in which the interference took place may be forgotten. A window manager could address this latter problem by allowing the user to examine the previous window contents. In a system with an existing history mechanism the user could even be allowed to find out which window caused the changes, browsing the context in which it occurred. Almost certainly the closer two windows are in the user's task space, the more subtle the form of signalling that is required. In the extreme, where two windows are just different views of the same object, no signalling is required at all.

Result dependence is a relation between windows. The obvious way to make it apparent to the user is by some correspondence with the spatial and visual relationship of the windows. If the relation is explicit in the data model then this will also be a correspondence between the data model and the screen model in the tenor of the MAC-style interface. Specific options might include always displaying all interdependent windows together, having interdependent windows all subwindows of some grouping window, or linking together interdependent windows (with rope?). A more fundamental way of dealing with the problem is to structure the command set so that most of the commands never cause sharing, the remaining commands requiring more conscious and deliberate use in order to remind the user of the impact. The further from the user's focus of attention (as measured by the user's task) the sharing is, the more difficult and conscious the use should be.

4.6.2 Undo strategies

When we look at the problem of undo in a multi-window environment, we are instantly posed the question of whether we mean a global undo (rolling the clock back for all windows), or a local undo (undo for a single window). The former method has least semantic holes and is most easy to understand; however, it has the unfortunate consequence that by the time I discover my mistake in one window I may have engaged in some independent activity in another window, which will then have to be undone in order to rectify my mistake. This problem is not just one for window managers, as the same sequence of events could happen, for instance, in a word processor where edits to one paragraph are found to be erroneous after another has been edited: however, it will tend to be more recurrent in windowing environments because of the greater tendency to switch contexts, and also more major as the units of interaction may have greater effect.

Only allowing global undo would be unacceptable because of the implied independence of windows. We can thus look to the dependency structure for help in understanding the semantics of local undo. Of particular value is the notion of commutativity as used in the definition of result independence. If actions A and B are independent, in the sense that doing A followed by B is the same as B followed by A, then we can undo A even if it occurred before B because it *could* have been issued after it. That is, the local undo is equivalent to a global undo on a *possible* derivation. In terms of the dependency structure we can undo any process until we get to an action that affected another process.

The situation is not quite as simple as this, however, as in practice we may only have the implicit and explicit data structure, and not the user semantics of the operations. For instance, if information has been written to, and then read from a common cut/paste buffer, and we wish to undo beyond the write, is this allowable? If the semantics of the operations were "I want this text in the buffer, and then I want the text from the buffer copied out", then we are all right in doing the undo; if, however, the semantics are "I want the same text in document B as in document A and I will achieve this aim using cut/paste", then clearly the aim fails if thereafter the contents of that text in A are undone and those of B left alone. Propagating such undos, even if semantically feasible, would probably be almost impossible to predict by the user, so we would probably have two classes of dependency; one where the user absolutely cannot undo beyond without major effort, such as sending mail, and the other as in the above where undo is possible but the user is warned that corrective action may be necessary, and is made aware of the scope of the broken dependency.

When we consider user semantics, life will not even be as rosy as painted above. For instance, if I use information in one window to guide my decisions about actions in another window then I will set up a dependency that is not captured in the computer: therefore there will be no way I can be warned of such

broken dependencies. We at least have the consolation here that we can expect the user to be aware of such explicit dependency when performing the undo – or can we?

4.7 Detection of sharing

We can use the dependency structure only if the window manager can detect it. Explicit data dependency such as sharing files is fairly easy to detect, and is well understood. We examine below the additional methods used to detect the implicit dependency obtained through the result and display, and also the even more difficult issue of detecting the user's own model of the system, and the interdependency that it implies.

4.7.1 Detecting result and display dependency

In order to make use of the definitions of sharing, we will need to decide when to detect it. If we are interested in forbiding sharing altogether, and we are designing a complete system, we can apply the relevant independence principle to the specification and use it to constrain design. More likely the situation will be more complex:

- The restrictions on sharing may not be absolute. Perhaps certain windows are expected to share and we want to signal this to the user.
- The entire system is not under our control. We are merely designing the windowing and system architecture around applications that already exist and for new applications written by third parties.

These considerations mean that in practice we are more likely to be designing a system that seeks to detect sharing at run time rather than prevent it all together. Depending on which type of sharing we are interested in, different parts of the system will be more appropriate to detect it.

Display dependence

This is simple for the window manager to detect. It merely has to signal any output to windows other than the one to which commands are addressed. It becomes more complex if we allow the possibility of background processes which may direct output to the current window without responding directly to commands from it.

Result dependence

The window manager clearly cannot perform experiments to determine commutativity, so detection of result dependence must rely on the underlying database manager and process architecture. This is not back-pedalling on our decision to opt away from data models, as any algorithm for the detection of sharing will be validated by the implicit definitions developed independently of the particular model. In other words, an explicit data model is fine for dealing with the specification and implementation of a specific system, but is dangerous for the initial definition of our terms and principles.

4.7.2 Capturing the user model

We have talked about the user's task model and using that to guide the use of the sharing information. We have also alluded to the way that the user may cause unseen sharing and interdependence. To see this we consider several updates to a database from two sources A and B:

Scenario 1

A reads $x = 10$, $y = 5$
B reads $x = 10$, $y = 5$
A writes $x \rightarrow 15$
B writes $y \rightarrow 10$

operations commute?

Scenario 2

A reads $x = 10$, $y = 5$
B reads $x = 10$, $y = 5$
A writes $x \rightarrow 15$
B writes $y \rightarrow x$

operations do not commute

If we just consider the updates then the operations in scenario 1 commute with one another whereas those in scenario 2 do not. However, if we consider the intentions and the information available the situation is different. From B's point of view, both scenarios could be attempting to set y to the value 10, in which case scenario 1 is acceptable and B just chose a poor way of expressing that intention in the second case. However, if B's intention is that y should have the value of x then this time scenario 1 is the bad choice of operation as it hides the interference due to the user's intention. If A and B were different users with different displays then they could be warned when the other user performed an update which invalidated the information shown on their screen. However, if A and B are the same user operating via different windows then the information

available is not limited to the selected window. Further, the interdependence may be much more subtle in practice.

It is clear that we can never entirely capture this user model, and it is thus of prime importance that we make the implications of the user's actions as obvious as possible. However, while not neglecting that it is sensible to capture as much as possible in order to aid the user:

- *Automatic capture* – One option is to try to infer the user's model from behaviour, or to force the user to work in such a way that it becomes obvious. Direct manipulation systems already go a long way to providing this. Users of such systems often prefer to copy from window to window, even quite small pieces of text, rather than retype. This makes the user's cross-referencing obvious to the system. For instance, in the example above if b were to construct the $y \rightarrow 10$ request by copying from the display $x = 10$, the system could infer that the update was dependent on the value of x and warn the user of possible sharing in scenario 1. This is far from perfect, of course: the user may still choose to re-enter rather than copy, and it would miss more complex relationships that cannot be achieved by copying (although the calculator could be included). On the other hand, it might also produce spurious relationships where copying was for lexical rather than semantic reasons ($x = 10$ was just a good place to pick up a 10).

  Automatic capture could be a lot more hi-tech, perhaps attempting an on-the-fly working set analysis to determine interdependent windows.

- *Explicit capture* – A more conservative option is to encourage users *to explicitly tell the system* their task model. This explicit capture can be either in the data model or in the screen model (if these differ significantly). An example of the former would be a user having several roles: each window would be logged into a specific role and protection schemes would ensure that there was little sharing between roles. A similar, but more dynamic, example of the latter would be project workspaces, where the user would create projects and subproject windows on the fly within which the other tools windows would reside. If the separation between database and window manager is strong then one would imagine the more long-term relationships being logged in the former and the shorter-term subtasks in the latter, as the examples suggest.

  It is interesting to note that the "rooms" system of Card *et al.* (1987) would be suitable for these purposes. It was created in order to aid the user's context switching when working with several tasks each consisting of several windows. Although it has a different aim from the worries here about sharing, it has the same goal, namely to capture a certain amount of the user's task model.

4.8 Dynamic commands

All the discussion so far has been in the context of *static* commands, where no new windows are created or existing ones destroyed. These events are obviously very important, but are significantly more complex. We will just take an overview of some of the properties that we might want to consider in relation to dynamic commands without going into great detail. I have given this issue a slightly more formal treatment elsewhere (cg) but even there the complexity of the issues means that a very limited treatment was given.

We have already given a formal distinction between *dynamic* and *static* commands. An obvious distinctions to make within the dynamic commands is between those that create and those that destroy windows. One requirement that we might wish to make about a system is that there should be a simple distinction for the user between these classes of commands. Thus the user can know whether a command is static but may change things, may create windows but do nothing else, or may destroy windows. This is unlikely to be true at the physical level as many different operations may be invoked by a mouse click, but if we look at a more abstract, logical level of commands this may be an appropriate distinction.

Whether such a split is desirable also depends on what we take as windows. If we include dialogue boxes then a large range of commands may be static in some circumstance and have an effect, and be dynamic in others when the dialogue box is required. Similarly, when the dialogue box is responded to, the command is likely to be both dynamic (because the dialogue box disappears) and have an effect (such as writing a file). This emphasises that the appropriate principles to apply to a system depend crucially on the level of abstraction we choose to adopt.

Another set of issues for dynamic commands are to do with setting limits on their dynamism. This is important since the appearance or disappearance of large numbers of windows is likely to be confusing. On the one side we may ask that windows are destroyed only when commands are addressed directly to them. On the other we may define the *fecundity* of a command to be the number of new windows created, and set strict limits on the acceptable fecundity.

We also need to extend the concepts of independence introduced earlier to the dynamic case. A natural extension would be to say that windows have some sort of dynamic independence if the way they create or destroy windows is independent of the order in which the commands are used. There is a conflict, however, between this sort of independence and the forms of static independence described previously. For instance, if we are opening windows onto a database the desire for static independence will require some form of locking to prevent the windows from opening onto the same part of the database. This locking will

inevitably mean that the events of opening the two windows will interfere.

4.9 Discussion

We have produced a simple model of windowed systems, the *handle space*, suitable for the formal discussion of principles of sharing and creation and destruction of windows. Over this model we have identified two specific sharing properties, *result independence* and *display independence*. These can be used to warn the user of possible interference between windows, and to inform undo strategies of allowable rearrangements of the user's event history.

Display dependence can be detected by the window manager but result dependence requires aid from the database and operating system, the exact information that these supply being constrained by the *implicit* definitions given. Detecting dependency induced by the user's own model is more difficult; some suggestions have been given, but they leave many open questions. In particular, the information gleaned by implicit capture will tend to be partial, and it is not necessarily the case that some is better than none in this context. For this reason the explicit capture methods are probably to be preferred at the moment (but aren't as much fun.)

Dynamic properties have only been dealt with only briefly, but are clearly complex and need to be examined closely for any particular multi-windowed application.