Permissive user interfaces

Harold Thimbleby

January 2, 2001

Abstract

User interfaces often only support one way to do a task when the physical interface or the requirements of the task would permit other ways. In contrast, a user interface that supports multiple approaches is *permissive*. This paper argues that permissive user interfaces are easier to use — and even when they are not applicable for particular applications, considering permissiveness is a productive design heuristic.

Many user interfaces are difficult to use yet very easily demonstrated or explained by experts — with the result that users become frustrated because hindsight makes usability problems look like the users' own fault. The lack of permissiveness in such user interfaces explains this paradox.

Keywords: Permissiveness; User interface design; Design principles; Usability; User interface programming.

To appear in *International Journal of Human-Computer Studies*, **54**, 2001.

1 Introduction

Good user interfaces — whether for computers or for consumer devices — are flexible, and empower users, giving them confidence and mastery. Bad user interfaces, in contrast, have mysterious features that users cannot guess, and in consequence make them feel helpless. When the befuddled user asks for help, an expert who knows *the* trick will get the device to work, and will make the user feel like an idiot for not knowing something soooo simple.

When a user interface is designed, the designers know how it can be used, so they may underestimate the importance of alternative ways of the user achieving goals. When they demonstrate a prototype user interface, it may seem to work well enough to be put into production. Yet when the users get the device — without the benefit of the designers' inside knowledge — working out how to use it for some tasks may be practically impossible. Guessing how to use the interface may result in it doing unplanned things — which merely increases the number of mysterious problems the user has to cope with!

When devices are designed, we tend to describe and discuss them in scenarios [5], narratives highlighting illustrative ways of interacting with a device. We are so used to narrative that we imagine all sorts of possibility from the least interactive of outlines: consider how engrossed we can get in a film, which on reflection is just a trivial reel of plastic with a linear, non-interactive storyline.

Imagine a film involving a user interface (such as one in a science fiction movie): in our imagination it will appear to work brilliantly — yet we know that in the reality of the film set, the user interface is a fake, and only works *exactly* as filmed, without any choices for the users who have to follow a script. So when it comes to designing interactive devices, often the possibilities tacitly imagined do not get implemented. Literally, the users who come later to use the device live in a different, and more restricted, world than the designers' imagination.

The 'two worlds' problem (the user's world is not the idyllic designer's world) becomes clear when a device is used for explicit communication. Suppose a device (say, a WAP phone) sends

email. Of course, the person or system getting email never gets requests from users who, for any reason, cannot send messages. The failures in the design are not visible, and the design will seem successful if enough users are successful regardless of the problems.

⋆

How can we design user interfaces to empower users?

This paper suggests that poor design results in part from unnecessarily restricting what the user can do, so that there is only one or a very few 'right' ways of working. It is hard to spot restrictive user interface designs, because anyone who understands the design well enough to be able to see the problem must also know enough not to find the restrictions a problem! When explaining or thinking about a design, it is easy to be trapped into linear thinking because it works — yet a user given a task to perform is unlikely to follow the best laid plans of the designers. Users do not always know the 'right' way of using a system, so they may be forever stuck if there is only one right way of working. In the communication example, even if some users are stuck the system is still *apparently* working as designed.

We therefore propose some simple terminology to focus on the issue: this paper proposes the term *permissive* for interfaces that permit alternative ways of working.

Curiously, complex user interfaces almost always provide many ways of achieving the same results, and the simpler interfaces become, the more restrictive they become. For example, on most word processors, there are many ways to move the cursor from *here* to *there*, say, a point two lines up and two character positions left: for instance, typing [↑][↑][←][←] or typing [←][←][↑][↑] both should work equally well. Even for this simple operation, there are 6 different ways of achieving the end result, not counting innumerable variations such as $\cdots[\leftarrow]\cdots[\rightarrow]\cdots$ that can be interleaved into the sequence and have no overall effect! Yet for a comparatively simple device like a video recorder there is essentially only one way to record a tape. One consequence of the inflexibilities of a VCR is that when a user makes errors (as they do) it is much harder to recover — in contrast, consider that for the word processor cursor movement it does not matter whether the user presses [←] or [↑] first, or indeed whether they do something completely different (there is always the undo key).

Devices of intermediate complexity, between word processors and video recorders, such as mobile phones, are often permissive in limited ways, as we shall see.

2 Definitions and introductory examples

A *permissive* interface allows users to activate functions in more than one way. Many successful user interfaces use permissiveness to provide short cuts: thus allowing users to progress from naïve use to expert use with ease. Examples are the Apple Macintosh menu command-key short cuts, and the Nokia 5110 mobile phone's menu numeric short cuts. In both cases, a user can browse the user interface and find explained commands, and when they want to move on to accessing commands more efficiently, the menus say what the short cut to use is.

A *non-permissive* (or *impermissive*) user interface has only one (or a very few) ways of using it correctly. Often an impermissive user interface will have disabled features that in various modes do nothing.

Suppose one is recording a live TV broadcast using the JVC HR–D540EK video recorder, and presses [▮▮], to make the recording pause (e.g., during adverts). A further press of [▮▮] does nothing when this VCR is paused. Apart from switching the VCR off, about the only thing that can sensibly be done to a paused VCR is to get it out of the pause mode, so one wonders why the pause key is dead in pause mode... Somehow the user has to know the one and only key that gets this VCR out of pause and will return to recording is to press [▷] (i.e., the *play* key).

The terms permissive and impermissive can be used to describe specific features of a user interface, or to describe the overall approach taken. For example, the Macintosh menu interface is permissive, but most application menus have impermissive features — typically because providing

more than twenty six command short cuts would get confusing. (One way around this is to allow users to define their own short cuts.)

This paper will use the terms *permissive* and *impermissive* freely, but when working with clients it may be more helpful to use self-explanatory design slogans that the clients can understand intuitively, just as when writing user manuals it may be better to explain design features in lay terms. The user interface design principles of the EPOC user interface include the slogan [7]:

⋆

Don't force just one input method.

⋆

As they put it, "each time the user's perfectly reasonable action is rejected because it's not the designer's or programmer's preferred way of doing things, they're one step closer to giving up and throwing the product away."

2.1 Searching for functions

The main user interface design problem that permissiveness tackles is searching for functions when the user does not know to start with where those functions are. If a user is not likely to know how to find a feature, pemissiveness suggests proving several routes to the feature. If the device is simple, like a VCR, the functions probably correspond directly to button presses; if the device is complex — certainly if it has more functions than buttons — the functions are probably organised in a menu, and are accessed by pressing a sequence of buttons (e.g., to navigate the menu).

English (paper) dictionaries provide a familiar example of searching for features, in this case English word definitions.[1]

To find a word in a conventional dictionary, the user has to know how to spell the word. Suppose we are looking up *bearhug*. The user first selects *b* (or somewhere near *b* in the dictionary), then *be* (or somewhere near *be*), and so on until they get to a short list of headwords, which they read sequentially. Unfortunately, as looking up *bearhug* in almost any dictionary will show, this method fails![2] The user is supposed to know, somehow, that *bearhug* is classified under the headword *bear* — so *bearhug* comes in most dictionaries before *bearable*. Thus the dictionary illustrates the problem faced by a user interface designer: features must be placed in a tree (here, an alphabetic tree), and if they only appear in one place, then there is a problem how to organise them. If a user relies totally on spelling, then *bearhug* must come after *bearable*. But if a user knows roughly what a word means, then the dictionary can be organised partly by meaning: thus, *bearhug* comes as a subsidiary definition inside the entry for *bear*. Whichever strategy the dictionary follows, there will come a user of the other sort. The problem is that *bearhug* only appears once in the dictionary: why not have it both under *bear* as well as (again) after *bearable*?

For most dictionaries, this is only the start of their problems. Where would a word such as *sea-bear* go? (The shorter OED [9] repeats *sea-bear* under both *bear* and *sea*, but not in the long alphabetical list following *sea* which has numerous words like *sea-bean* and *sea-cow*.) How should variant spellings be treated? Such classification problems for lexicographers are similar to the problems faced by user interface designers!

Conventionally, dictionaries place each word in just one place. This has several advantages for the lexicographer (i.e., the designer): there is only one definition to maintain as new meanings for the word appear, and since they are experts using the dictionary, they can find where to put the word easily. Typically users of dictionaries persevere: the word *must* be in the OED, so they try searching somewhere else.

[1] Dictionaries provide far more words, in the hundreds of thousands, than even the most complex user interfaces: yet they are not too difficult to use — suggesting that mere number of features is not the usability problem so much as poor structure.

[2] This example was suggested by [11], which *is* in strict alphabetical order.

	call divert
FAX or data	*call*
recent	*calls*
FAX or	*data call*
call	*divert*
	FAX or data call
	messages
security	*options*
	personal reminders
	phone settings
	recent calls
personal	*reminders*
	security options
phone	*settings*

Figure 1: KWIC index for some mobile phone functions — for illustrative purposes only a few functions are shown (typically there would be about 100). Every keyword (just to the right of the vertical line) is in alphabetical order "in context" of its function name. Thus a user can easily find a phrase knowing only one word of it.

Clearly, if a dictionary was computerised, the problem practically disappears. Words can be put everywhere that they can appropriately be placed. There does not need to be any slavish adherance to single appearances of headwords.

KWIC (keyword in context) indices are dictionaries for phrases, which *do* repeat words to make phrases easier to find. (KWIC is often used in concordances and dictionaries of quotations: looking up any significant word will find the whole phrase.) By way of example, suppose the phrases are *recent calls*, *messages*, *FAX or data call*, *call divert*, *phone settings*, *security options*, and *personal reminders*; Figure 1 shows how they would be presented in a small KWIC index. A user can now find any function knowing only one word from its phrase. For example, if the user wants to do something with reminders, they can find this function under "reminders" and access it — they will also find out that the manufacturer's preferred name for the function is "personal reminders."

KWIC has been used for mobile phone user interfaces, combined with using the alphanumeric keys for searching, and has been shown to be empirically faster than the conventional tree-based approach [10].

2.2 Relationships to Cognitive Dimensions

Cognitive Dimensions [6] provide a useful vocabulary for discussing cognitive problems in notations and user interfaces. Thus *premature commitment* is a problem users may have if they are too-soon forced to make decisions, perhaps before adequate information is available to make informed decisions. Our notion of permissivness is, in contrast, a feature of the user interface which may reduce the incidence of premature commitment. Permissiveness, then, is a design feature (say, like direct manipulation), which under different contexts may or may not raise interesting cognitive issues. In particular, permissiveness is intended, like direct manipulation, to be an idea that is implementable by system designers. It is not so much intended to critique existing systems — although it can be used in this way — but as a possible, and potentially beneficial, design option.

At the other extreme, are implementation principles, such as *delaying commitment* [16], which in themselves have no inevitable impact on the user interface *per se* except in that they reduce the barriers to modifying software. However, in Section 3.4 we briefly mention a user interface where delaying commitment (in this case, a specific technique, *equal opportunity*) is made visible in the user interface.

2.3 Relationships to formal methods

If cognitive dimensions provide a useful cognitive vocabulary, generative user engineering principles [14, 17] are user engineering design principles that are formalisable. Permissiveness is easily formalised (in several ways), but what is not easily formalised is the near-permissiveness of many actual systems.

One of the main reasons why programs have bugs and behave incorrectly is that they are developed informally, with no engineering rigour. A formal way to view permissiveness is in terms of user interface specifications. User interfaces viewed formally should be sound: they should disallow invalid actions. However, user interfaces may not be complete: there may be valid actions they do not permit. Incomplete user interfaces are an inadequate way of satisfying requirements, but precisely because of this they are easier to implement. Unfortunately their soundness ensures they are sufficient to demonstrate (i.e., they can be shown to accomplish all user tasks): anyone demonstrating them can think of actions that are correctly implemented, and these are the ones to demonstrate. Indeed, one of the main reasons for undertaking user testing is that users will think of ways of using systems that would not occur to the designers — that is, user testing takes testing outside of the known sound system to expose the possibly incomplete implementation.

3 Further examples

3.1 The Nokia 5110 mobile phone

At the time of writing the Nokia 5110 mobile phone is the world-wide best-selling GSM handset. As a market leader, we shall use it several times in this paper to make user interface design points: we are not picking on it (or Nokia), but the handset is likely to be familiar to readers, or easily accessible, and the user interface features it has are broadly representative of a wide range of devices.

The Nokia 5110 mobile phone has a keypad lock, so that it can be left in standby without it accidentally activating functions or making expensive phone calls. Deactivating the keypad lock is easy since when any button is pressed, the mobile says how to deactivate the lock. However, setting the lock is difficult for anyone who does not know how to use it: the mobile has a menu system that does not include the keypad lock.

The keypad lock has to be set in a special 'efficient' manner that is faster than accessing any menu function. *If* a user knows how to set the keypad lock, setting it is indeed easy and efficient. If a user does not know how to set the lock, there is no way to find out how to do it.

We say that the Nokia is (in this respect) impermissive.

A permissive approach to the keypad lock would have been to provide a menu function ***`keypad lock`*** *as well* as the existing approach. A naïve user would then be able to find the keypad lock in a conventional place (perhaps after a search). As with all other functions in the mobile's menu, the keypad lock would have a short cut — which could be Nokia's original method of activating the lock.

Thus, the keypad lock could be activated in two ways. One way would be the conventional (but potentially awkward) way; the other way would be via the menu hierarchy. For an expert, this change to the user interface would make no difference; for a new user, it would have a major benefit. A previously obscure and hidden part of the user interface (which could originally only be used after reading the manual or being told by an expert) could be activated without additional help.

Now, with the new design, imagine a user who has had a mobile phone for a few weeks — they long since finished reading the manual — and they have just taken the handset out of their pocket and, since it has started to dial something, they will perhaps notice the advantage of having a keypad lock (perhaps they read about this function ages ago and did not understand why it might be useful). With the permissive user interface, the user has a good chance of being able to find out how to lock the keypad.

The Nokia mobile phone design is discussed again in Section 4.1.

3.2 Vending machines

Hamilton Airport, New Zealand, 27 May 1997. My then nine year old daughter Jemima wanted to buy some sweets from a vending machine. She decoded the symbols 110E5 underneath her choice as meaning that choice E5 cost $1.10, so she keyed in [E] [5] on the keyboard. The display showed first `READY`, then as she pressed the buttons, it showed first `E*`, then `E5`, and finally `$1.10`. She counted out $1.10 in New Zealand coins and put them in the coin slot. Nothing happened. She pressed the big button, but this was the coin return. She tried again. I suggested she read the instructions (at the top of the machine — she wasn't tall enough to have noticed them). The instructions said,

Insert coins only
▽
Make your selection

By then she had got the money in already, so she keyed [E] [5] and collected the packet of sweets ...

What is happening here? This is an interface that *in principle* could be used in at least two ways: insert money then make selection; or make selection then insert money (or any interleaving of these, possibly with other users, and guarded with timeouts). Yet it only supported one of these ways, and in this case, not the one the user chose. Moreover, the display screen apparently gave feedback that pressing keys in the wrong order was in fact succeeding (i.e., pressing the [E] before inserting any coins changed the display to `E*`, not to a message saying something like "please insert coins first"). The designers have only allowed one of the possible ways, and moreover, the way that is less forgiving. If you put your money in, then made the wrong selection (which is easy because of the complex item coding), you would get the wrong stuff out of the machine and lose your money. The other way around, you can make a selection, change your mind, make another selection — and so on — then enter your money. In this case, the machine could also prompt you how much more money (if several coins were required) should be inserted. This would be useful if you had so-much cash and wanted to find out the best product to buy with what you could afford. The actual machine cannot do this: it can only tell you how much you've put in — for it doesn't yet know what you want to choose and how much it costs.

Interestingly, although only one order is permitted for the two main steps of buying from the machine, the code for the chosen item (such as E5) could be entered in either order, as [E] [5] or [5] [E]. It is unlikely that a user would enter a code displayed as E5 as [5] [E], but there is no harm — and potential benefit from time to time (especially for confused users!) — for this permissive code entry that the vending machine did support.

User studies might show that Jemima is unusual; or they might show that the design was wrong — that her behaviour was typical; or perhaps they would show that different people use the interface differently. Whatever, the design potentially could support many different orders but the designer has only provided one, and arguably a bad one.

Vending machines are an example of a large class of walk up and use interfaces, which also includes cash machines (ATMs) and ticket machines.

On the London Underground,[3] some automatic ticket machines have a large array of buttons for every destination station. To the left of this array are a few buttons that select ticket type: single, child, return and so on. The ticket machine requires the ticket type to be selected, then the destination, then the cash inserted (to at least the cost). In principle, the conditions for determining the ticket can be met in any order. Although the order the London Underground requires is sufficient to work, it is not permissive. Somehow users have to know, out of all the possibilities, exactly what the designers required. The user interface design issues are thus very similar to standard vending machines, though complicated by providing more options. (Further issues are mentioned in Section 8.)

[3] Last checked in 2000.

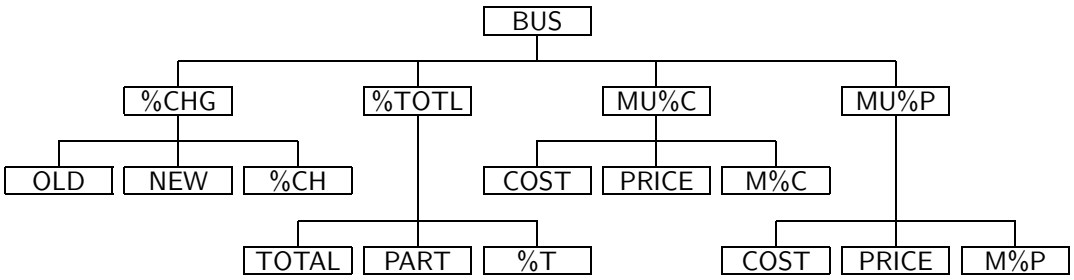

Figure 2: Part of the menu structure for the Hewlett-Packard 17BII handheld financial calculator. Note that COST and PRICE are both shared under two different headings. The terms are HP's and are exactly as they appear on the calculator display: BUS means *business*, %CHG means *percent change*, MU%C means *markup as percentage of cost*, etc.

3.3 An expert's view

In his book *Usability Engineering*, Jakob Nielsen reports that of 24 users asked about a graphical user interface to a calculator, 11 thought it could be used by mouse or keyboard and 13 thought it could only be used by the mouse [12, p61]. Nielsen says that these 13 have an "erroneous model" of the user interface.

Surely it would have been more positive to say that these 13 users have a sound model which is incomplete. The right view is that the calculator is permissive (it permits two ways to operate it) and it is sufficient to know either way to be able to use it.

In this case we *do* have a permissive user interface design, but Nielsen is suggesting that users not knowing all alternatives is erroneous. In our view, a permissive interface allows users to work in any — or many — of several ways, none of which are erroneous. Users may have incomplete models but that often does not matter. Only knowing part of a permissive user interface is not erroneous.

3.4 Calculators

All handheld calculators require the user to enter calculations in a specific order. We are so accustomed to thinking "this is how calculators work" that it is perhaps hard to see the intrinsic impermissiveness.

Consider converting 98.6 Fahrenheit into Celsius: we may be able to remember the formula $32 + 9/5 \times c = f$, but we now have to do some algebra to convert it to $5/9 \times (f - 32) = c$, and on simple calculators without brackets we have to convert it further to $f - 32 \times 5/9 = c$. In contrast, the calculator discussed in [18, 19] allows the user to enter $98.6 = 32 + 9/5\times$ *in that order* (or indeed in any other order) and it would calculate the correct value (37, in this case). The point is that there is not a 'right' order. Mathematically (i.e., from the point of view of the user's task) there is no requirement to impose any specific order, so why impose a restrictive order from the user interface?

The power and ease of use of permissiveness for this application is hardly shown in such a simple example (without a lot more explanation), but more details are provided in [18, 19, 21].

Figure 2 shows an extract from the structure of the Hewlett-Packard 17BII handheld financial/business calculator. The structure looks like a tree, but some items are actually shared between branches — the routes to such items are permissive. The reason this is a good idea is because the functions are related; given a cost and a mark up percentage, we can use the entries under MU%P to calculate the price; then, changing over to MU%C we get the percentage change without having to enter any new information. Thus the equations are (permissively) used in different 'directions': numeric values *in the same fields* are used for inputting data and for getting answers – an — *equal opportunity* again [15].

Trees are a concept from graph theory, and strictly trees do not share items as the HP 17BII does (we discuss trees further below, in Section 5). Thus the tree structure in Figure 2 is somewhat

misleading, since it looks like a tree only because the repetition of COST and PRICE is not visually obvious. Drawing the structure 'pedantically' so that each item had exactly one box would in general make lines cross over, and the diagram would become cluttered and unhelpful. In other words (although it has not happened with the 17BII) merely drawing trees as a visual aid in the design process tends to lead designers into making impermissive user interfaces — great care must be taken that repeated entries are the same entry, rather than ending up being implemented separately.

3.5 Screen-based user interfaces

Permissiveness is a concept that can have a large impact on hand held and push button style user interfaces, but it is also applicable to desktop computer (large keyboard/large screen) style user interfaces.

Graphical form-filling user interfaces are naturally permissive. Typically, the user can fill in data fields in any order. When all fields are filled in, the user can submit the form. Forms are familiar, and have been discussed at length elsewhere; we do not need to discuss them further here — except to highlight the fact that they are such a pervasive example that it is easy to overlook them as a positive example of permissiveness.

4 Tradeoffs

4.1 More efficiency with more permissiveness

The Nokia 5110 has functions accessible through a menu style user interface. The menu is structured like a tree — as it were, pressing a button moves up the tree, presenting the user with a choice of branches to follow; and the mobile's functions are leaves at the ends of branches, where there are no further choices. Thus to select any function takes many button presses, and which buttons should be pressed depends on where in the tree the user has got to. Using the tree therefore requires the user to pay close attention to the mobile's display, to see what choices it is offering (of course, some of the very easiest functions may be memorised by the user).

It is likely that experienced users will find this approach frustrating, so Nokia have provided short cuts for all functions. When a user selects a function (by any means) the mobile's display shows the name of the function as well as a numeric short cut. For example, *write messages* can be accessed either as a sequence of conventional choices: [−][∨][∨][−][∨][∨][∨][−] (i.e., pressing the Navi key, then Down twice, then the Navi key again, etc.), or as a short cut [−][2][3]. The short cut code is briefer and slightly more mnemonic (a two digit number is easier to remember than seven arbitrary key presses, and it might also be pressed without referring to the LCD screen for intermediate feedback). Nokia's explicit menu navigation has an expected cost of 8.83 to reach any function (i.e., key press counts averaged over all functions), in comparison to using the faster short cuts which are have expected cost of 3.64 [20].

(Of course, just because the short cuts are faster to access individually does not mean that short cuts are always better, regardless of the user's task. The short cuts are very difficult to browse: if the user doesn't know what function they want, using the [∨] etc. will get around the list of functions, taking on average just over one key press per function, but using short cuts requires 3–6 key presses per function, including a long-hold of the [C] key! Again, this is an argument to provide both, permissively, since the designer does not know *a priori* what the user will want to do.)

Nokia's short cut codes closely reflect the menu structure (e.g., two down, then three down) and are not as efficient as they might be — for instance, many short codes are never used. If we permissively *also* allowed the shorter codes that Nokia have not already allocated, then the user could select the more efficient short cuts if they wanted to. For example, *vibrating alert* has a Nokia short cut of [−][9][7], but since [−][0] (for instance) is not allocated to a function, it could be used, so saving one key press. This approach would give some functions two short cuts: we would retain Nokia's original short cuts (on the assumption that they are intrinsically useful

— perhaps they work uniformly across models), and provide more efficient short cuts for users who want them as even faster alternatives.The expected cost of this permissive approach is 2.87 (compare with Nokia's short cut cost of 3.64).

Since there are not enough short cuts to make every function faster, a good design would probably allow users to allocate their own short cuts. This would increase the user's involvement with the device and increase their satisfaction of having power over it (literally empowering them). A full discussion of short cuts and costs for the Nokia 5110 can be found in [20].

4.2 Searching versus browsing

From an abstract point of view, when a user operates a device, they search for a function which they then activate. To be more general, it is possible that no function is found, that the function has no effect, or that the user does not activate the function, or indeed that an inappropriate function is activated, and the user subsequently embarks on error-repair (or on making things worse...). From the user's point of view, they will lie approximately along a spectrum from "I know what I want, and I can achieve it by doing the following operations" to "I do not know exactly what this device can do, nor what it calls its functions, so I shall explore it; I may then activate a function I want." In the former case, the goal is to *search* for a function, making use of the known structure of the device; in the latter case, the initial goal is to *browse* the set of functions. Of course, things are more complex because the user may want to search but have an incorrect knowledge of the function names, or they may wish to browse and have an inefficient method of browsing. From a computer science point of view, the former case calls for a user interface that codes the functions efficiently, and the second case calls for a user interface that permits a linear search. There are four important design tradeoffs:

1. Efficient codes (e.g., Huffman codes [20]) are difficult to remember. The user interface should be structured less efficiently to be easier to use for less-skilled users.

2. Users do not necessarily know the specific structure chosen. Permissiveness suggests providing more than one structure, but providing more structures makes optimal use less efficient.

3. Users' tasks are not just searching but may include browsing and other activities. These are more efficiently undertaken with other structures (e.g., efficient browsing is linear search). Again, supporting more tasks makes any specific task harder to do.

4. Device operation is not error free, and users may change their tasks mid-way, so features may be required for correction, undo and other forms of error recovery.

When a user searches for a function, they know what they are looking for (whether they are right or wrong), and permissiveness suggests that the system provide alternatives so that the user does not need to look for exactly the 'right' thing (or to provide the instructions in exactly the right order, etc). Unfortunately, as more and more equivalent functions are added — which increases the probability that the user has an exact match between what they try to do and what the system permits — the longer it would take to do a systematic search. Thus, if a user does not know what they want to do, but instead wishes to browse the user interface to see what features it provides, permissiveness can increases the task time.

There are two obvious design choices: first, the user interface can provide two styles of interaction (say, some features for users who want quick access, and some features for systematic search); secondly, empirical user studies may be used to establish what sorts of tasks users will likely engage in. For example, a ticket machine is likely to be used by people who know where they want to go — but do not necessarily know what the ticket machine calls the destination — which suggests permissiveness. Within the same task, users may not know what sort of ticket they require, so the ticket type selection might be better supported by browsing rather than by searching. Again, users may not want to decide the ticket type until they have selected the destination, and they may want to browse choices (for example, if they have £10 available, and want to get as close to home as possible); or they may have a voucher that entitles them to certain benefits for certain ticket types.

How one balances support for a wide variety of possible tasks, against imposing complexity on all users however routine their tasks is again a question best answered empirically. Nevertheless, by exploring the design possibilites more creatively, new and simpler interfaces may be considered.

A real example of this tradeoff occurred in an experimental evaluation of an alternative user interface for a mobile phone handset [10]. Users were expected to use the new efficient search feature, but they chose to take longer and use a browse (scrolling) feature that was also available; overall their performance with the new user interface was comparable with the old (we had expected a significant improvement), but they found it much more satisfying.

4.3 Monotony and anti-permissive interfaces

We have used the term *impermissive* to suggest an avoidable restriction in a user interface. However, sometimes secret features are constructive! For example, making a password check permissive would be counter-productive to security — illustrating the standard tension that ease of use means easy to use for anyone, whereas security means ease of use for just one person (who, for instance, knows the password). Thus, we suggest the term *anti-permissive* to mean deliberate impermissiveness, as opposed to mere oversight or poor design.

Another example of anti-permissive design are lock outs and other safety features. For example, a covered switch can only be used after the cover is raised. This makes operating the switch more likely to be deliberate, rather than an accident. Had the design of the switch been permissive, the cover might have been raised before or after the switch was pressed — thus raising the design question why the cover was necessary.

⋆

User interfaces need only be anti-permissive when there is no undo.

⋆

(One would assume that a security breach has no obvious undo.)

Sometimes interfaces are so badly designed that a system that looks permissive in fact behaves like a security system. The Matsui VX1107A VCR is a case in point: although it has ten digit keys, to set a time, the up and down arrows (also labelled with forward and stop) must be pressed. The interface could have been permissive (allowing digits or arrow keys to set the time), but instead it *appears* to be permissive but is not — rather like a secure system that allows the user to press 'anything' but only pressing the right things will access the system.

Jef Raskin, in his excellent *Humane Interface* [13], uses the word *monotonous* to describe a user interface that has only one way of doing things. His intention is to contrast a monotonous interface with a mode-ridden interface. He makes many relevant points — for example that making user interfaces backwards compatible results in unnecessary permissiveness. Often systems are not truely backwards compatible, and the short-comings of implementation lead to their own problems. Sometimes permissiveness is justified because some users prefer one method, and other users another. Again, Raskin dismisses this as a justification for permitting many ways of using a system — as we would. Our purpose in promoting premissiveness is not to allow lots of users to operate a device in their own idiosyncratic ways, but to facilitate individual users mastering the interface. Indeed, when many users can each have their own ways of using an interface, they are effectively thwarted from helping each other — one user's solutions may not help another.

Clearly, permissiveness is a user interface feature that can be abused. Successful design is not merely a case of following recipes (whether ours or Raskin's). Raskin's term *monotony* is useful to bring to designers' attention the deeper questions of design.

5 A city is not a tree

Trees as a way of structuring user interfaces were described briefly in Section 3.4, by way of explaining how the HP 17BII's functions are organised. Trees are in fact a very common structuring

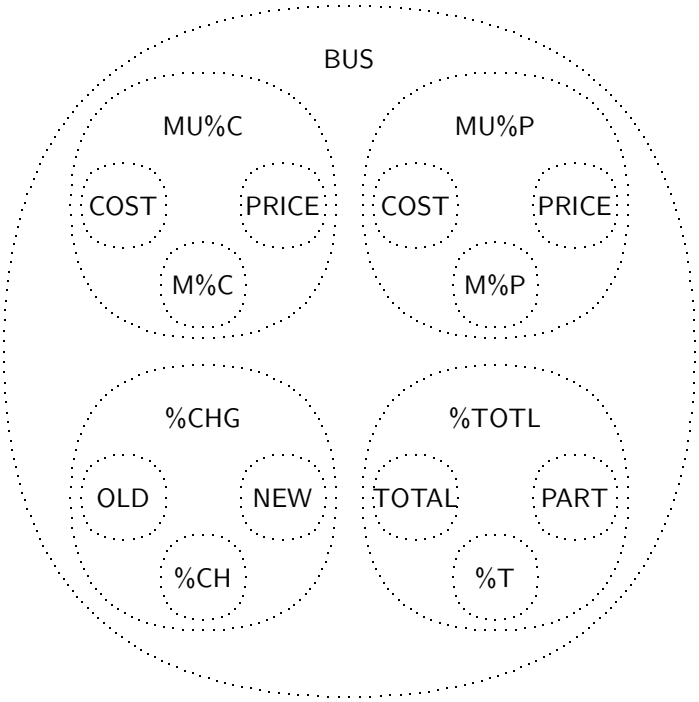

Figure 3: Another way to view trees, as nested non-overlapping circles (as Venn diagrams). Compare with Figure 2. In actual fact, the 17BII is not strictly structured as a tree, and the sets MU%C and MU%P *should* overlap, with their intersection containing COST and PRICE. Thus: drawing simple diagrams tends to make user interfaces impermissive, or encourages design errors.

method, because they are "so obvious" and easy to draw, as we mentioned above. In fact, as we now argue, an easy way to slide into impermissiveness is to use a tree to structure the user interface.

Chris Alexander, in his classic discussion of the design of cities [3], argues that designers too easily lay out a city as a tree. In a tree-structured design, the work area is over here, the housing is over here, and the shopping is over here. A tree-structured design makes it conceptually very easy for the designers, but harder for the users who have to live in the city. In contrast, a well designed town would mix the housing, shopping and work places (otherwise, for example, the shopping area will be dead at night, and the housing will be dead during the day). Mixing means, essentially, that shops do not appear in one place. A city is not a tree, because it is permissive — most things can be found in several places in several different ways. Indeed, even in a highly regimented city it is hard to imagine that there would be only one place where a user could find a newspaper — people find duplication of resources (whether newspapers or user interface features) helpful.

Figure 3 shows an alternative view of the tree drawn in Figure 2. The new Figure can be imagined as viewing a tree seeing the menu choices as enveloping circles: each menu encloses all of its submenus. More formally, the alternative view is of a tree as a Venn diagram, as a collection of sets. Each set is a menu, and its elements are the submenus (or functions) of the menu. Notable about a tree is that the sets do not overlap — which is equivalent to saying that in the conventional drawing of a tree that the branching lines do not join up again.

Different devices use different techniques for selecting branches: the Nokia 5110 mobile phone uses up and down keys (labelled [∧] and [∨]) to move; the HP 17BII calculator (illustrated in Figure 2) uses soft keys, so a branch can be directly selected. In general, the user is faced with the problem of getting to a function (call it x), and there are basically three approaches open to them (apart from giving up):

- The user knows and understands the design's structure. x is under y, and y is under z (for example, PART is under %TOTL which is under BUS), so the first thing to do is select z, then y, and then x. In the set view, if the user knows that $z \supset y \supset x$, then they can find x easily.

- The user does not know where things are exactly, but they do understand the design's structure (e.g., how [∧] and [∨] work). Knowing this, the user searches systematically for x, y or z... indeed anything related to x. On finding any one of them, they select it, and continue searching. The user is guaranteed to get closer and closer to their goal (provided they *do* understand the structure and can recognise x, y and z).

- The user has no idea where x is nor what the structure is, so they must systematically search the tree as best they can, going up and down branches till they hit on x (or possibly a y that sounds like it leads to an x). The user has no sense of progress and may go around in circles — clearly, in the worst case, the user must remember everywhere they have been so that they do not repeatedly get lost. To make *any* progress the user needs to assume that [∧] and [∨], and [-] and [C] are mutual inverses (e.g., that [∧] [∨] does nothing, so an erroneous press of, say, [∧] can be corrected).

For an expert user, clearly it does not matter much exactly where functions are — and designers are expert users *par excellence*. Experts can always find functions efficiently, because they know where they are. However, as the cases above make clear, the less a user knows the harder functions are to find.

⋆

If functions appear more than once
(particularly in different forms or aliases)
they are easier to find,
even though this gives users more choices.

⋆

For a user who does not know how a complex structure works, or what classification scheme a tree structure has used, a linear list would be far better. There is no wasted time in learning how the list is organised, and a simple scroll (e.g., repeatedly pressing [∨]) is guaranteed to get through the entire list very simply, and will therefore inevitably find whatever the user wants (provided it is available at all).

6 Permissiveness as a design heuristic

Permissiveness is sometimes hard to support, and sometimes it is hard to support in its full generality.

In the introductory example of cursor movement in a word processor, it was claimed that 6 different ways of moving the cursor would all have the same effect. But in many word processors such freedom is rare. Consider moving the cursor up from a longer line to a shorter line. The cursor may end up too far to the left, since the upwards motion takes the cursor leftwards to the end of the shorter line. It means that a sequence of keystrokes like [→] [↑] [←] [↓] only sometimes leaves the cursor in the same position as it started.

A designer has several choices:

- Users will certainly get used to this sort of quirk, so don't worry.

- Permissiveness is paramount, and the word processor's design must be changed so that cursor keys always work predictably. See [4] for a full discussion of a word processor with this uniformity.

- Permissiveness is a trade-off. Where permissiveness helps lead to a better design, exploit it; where it is counter-productive or impossible, ignore it. But just because a design cannot be everywhere permissive, don't lose its advantages for the places where it can be.

7 Perceived complexity: design versus interaction

Designers, no doubt, wish to make 'simple' systems — certainly, they have to design systems that they understand. To them, it is likely that permissiveness introduces complexity. Certainly a permissive system has more routes through its program, and when designed as a conventional imperative step-by-step program, this permissiveness will look like unnecessary complexity.

⋆

If anything should be simple,
it is not "the user interface" but a user's interaction with the system.

⋆

The user only experiences one possible interaction with a system each time it is used; that there may be other ways of using the system does not, in itself, make *the interaction* more complex. Indeed, as we argued, the absence of expected interaction paths makes a system worse.

Permissiveness, then, is a "user-centred design slogan" to encourage the designer to consider the complexity of *each* possible — and impossible — interaction, not to consider the complexity of the system *per se*, seen from the system's point of view of *all* possible interactions.

A major advantage of building simple systems, particularly ones with clear hierarchies with no overlapping functionality between parts of the design, is that there are few unforeseen interactions between parts of the design. Feature interaction is a serious problem in many systems, even in apparently simple interactive devices [21]; it either has to be avoided by building tree-structured systems (i.e., minimising overlap) or by using sophisticated design techniques that can handle the complexity. Ironically — despite his views about the design of cities (Section 5) — Alexander's *Notes on the synthesis of form* [2] promotes an isolating tree-based approach to design, perhaps because his notes are concerned with making *design* easier rather than *use* easier.

8 Further notes

When raising user interface design issues, since every issue is relevant in some context, it is tempting to cover many issues. For example, this paper has not covered contextual issues, permissiveness *versus* affordance, and a host of other relevant issues that, in any real design process, must also be considered (but which need not be a topic for this paper). However, there are some important, but not so obvious issues:

Access. Visually impaired users and others with specific needs are empowered by permissive user interfaces.

Not all options are equally good. There are many occasions when supporting many uses, in particular many orders of interaction, will be counter-productive. Closure errors (also called post-completion errors) are cases where the user completes *their* task, but they have not completed the interaction. The standard example is the user at an ATM: the user's salient task is to get some cash, but the full task also involves retrieving the cash card. Therefore a safer interface design requires the user to withdraw the cash card *before* they have completed their task.[4]

[4] All ATMs tested in South Africa (Standard Bank, ABSA Bank, etc., July 2000) provide cash and, when that is taken (i.e., accidentally completing the user's task), it still remains for the user to extract their cash card. Interestingly, some ATMs had the same external hardware, and therefore probably the same internal hardware, as UK machines that use the other order.

Programming problems. Writing programs is hard enough without also creating good user interfaces! Unfortunately it is much easier to write single-thread serialised programs, rather than ones that can handle more than one thing at a time. It is *much* easier to write a permissive program and then constrain it (for instance because of task-related problems) rather than to write a serial program and then generalise it. Using programming languages or notations that naturally support permissiveness is likely to bring many benefits. For example, in CSP [8] a serial program may be written $A; B$ (say, requiring the user to do A then B) and a parallel version as $A||B$ (allowing the user to do A or B first, or even interleaving them) — in Java or C, such a change, trivial in CSP, would be too hard for a programmer to seriously contemplate.

Although finite state machines are sometimes clumsy for large interactive systems, note that non-deterministic finite state machines [1] can handle permissive designs concisely.

9 Conclusions

We proposed a user interface design principle, *permissiveness*, that is easy to understand, and which suggests or encourages improvements and possible design changes to user interfaces. We showed that sophisticated user interfaces (e.g., word processors) tend to be very permissive, and that unfortunately — and often unnecessarily — permissiveness drops off in simpler devices such as calculators and mobile phones.

Designers see the world in a different way from users, and the widespread lack of permissiveness gives a persuasive illustration of this gulf: while an impermissive user interface can be difficult to use, deceptively it is easy to demonstrate with panache. For example, devices like video recorders are notorious for being difficult to use, because their impermissiveness simultaneously makes them mysteriousness (for someone who does not know the one 'right way' of using them) yet allows them to be demonstrated *as if* they were very easy! Thus while the ideas behind permissiveness are not raised or thought through in the design process, we will continue to have inferior user interfaces, ironically thought highly of by their proponents.

With very few exceptions (which will include safety critical tasks where the user should be highly trained) permissive interfaces are useful, especially for casual user interfaces, for so-called walk up and use interfaces, and for untrained users. Really, permissive interfaces represent an attitude change for designers: allow users alternatives, and do not view any user's choices as 'wrong.' Generally, the specification of tasks is much more general than simple implementations that, for instance, over-serialise and support only a few ways of undertaking tasks. Since user testing exposes such limitations (which then have to be fixed), exploiting permissiveness may be a way of accelerating user interface development (since there will be fewer design restrictions to uncover).

Acknowledgements

EPOC is a trademark of Symbian Ltd. Navi is a trademark of Nokia Mobile Phones. Alan Blackwell, Ann Blandford, Paul Curzon and Gary Marsden made many helpful contributions. This paper was completed during a Royal Society funded visit to Cape Town.

References

[1] A. V. Aho & J. D. Ullman, *Foundations of Computer Science*, Computer Science Press, 1992.

[2] C. Alexander, *Notes on the Synthesis of Form*, Harvard University Press, 1964.

[3] C. Alexander, "A City is Not a Tree," *DESIGN*, **206**, pp46–55, 1965.

[4] R. Bornat & H. Thimbleby, "The Life and Times of Ded, Display Editor," in *Cognitive Ergonomics and Human Computer Interaction*, pp225–255, J. B. Long & A. Whitefield, editors, Cambridge University Press, 1989.

[5] J. M. Carroll, *Scenario Based Design*, John Wiley, 1995.

[6] T. R. G. Green, "Cognitive Dimensions of Notations", in R. Winder & A. Sutcliffe (eds), *BCS Conference on People and Computers*, **V**, Cambridge University Press, 1989.

[7] N. Healey, "The EPOC User Interface in the Psion Series 5," in *Information Appliances and Beyond: Interaction Design for Consumer Products*, ed. E. Bergman, pp131–168, Morgan Kaufmann, 2000.

[8] C. A. R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, 1985.

[9] W. Little, H. W. Fowler & J. Coulson, *The Shorter Oxford English Dictionary*, 3rd. ed., **1**, Oxford University Press, 1973.

[10] G. Marsden, H. Thimbleby, P. Gillary & M. Jones, "Successful User Interface Design from Efficient Computer Algorithms," Proceedings ACM CHI (Extended Abstracts), pp181–182, 2000.

[11] W. T. McLeod & P. Hanks, eds., *The New Collins Concise Dictionary of the English Language*, London & Glasgow: Collins, 1982.

[12] J. Nielsen, *Usability Engineering*, Academic Press, 1993.

[13] J. Raskin, *The Humane Interface: New Directions for Designing Interactive Systems*, Addison-Wesley, 2000.

[14] H. Thimbleby, "Generative User-Engineering Principles for User Interface Design," *Proceedings First IFIP Conference on Human Computer Interaction*, INTERACT'84, B. Shackel, editor, London, **2**, pp102–107, 1984.

[15] H. Thimbleby & C. Runciman, "Equal Opportunity Interactive Systems," *International Journal of Man-Machine Studies*, **25**(4), pp439–451, 1986.

[16] H. Thimbleby, "Delaying Commitment," *IEEE Software*, **5**(3), pp78–86, 1988.

[17] H. Thimbleby, *User Interface Design*, Addison-Wesley, 1990.

[18] H. Thimbleby, "A New Calculator and Why it is Necessary," *Computer Journal*, **38**(6), pp418–433, 1996.

[19] H. Thimbleby, "A True Calculator," *Engineering Science and Education Journal*, **6**(3), pp128–136, 1997.

[20] H. Thimbleby, "Analysis and Simulation of User Interfaces," in S. McDonald, Y. Waern & G. Cockton, eds., *BCS Conference on Human-Computer Interaction*, HCI'2000, pp221–237, **XIV**, Cambridge University Press, 2000.

[21] H. Thimbleby, "Calculators are Needlessly Bad," *International Journal of Human-Computer Studies*, **52**(6), pp1031–1069, 2000.