Designing user interfaces for problem solving, with application to hypertext and creative writing

Short title: Problem solving in hypertext and creative writing.

Harold Thimbleby
Department of Psychology, Stirling University, Stirling, Scotland, FK9 4LA

Email: hwt@compsci.stirling.ac.uk.
FAX: +44 786 467641.
Phone: +44 786 467640.

Thursday, October 1, 1998

Designing user interfaces for problem solving, with application to hypertext and creative writing

Abstract: Interactive computer systems can support their users in problem solving, both in performing their work tasks and in using the systems themselves. Not only is direct support for heuristics beneficial, but to do so modifies the form of computer support provided. This paper defines and explores the use of problem solving heuristics in user interface design.

A demonstration hypertext system, Hyperwriter, was developed as the outcome of considering general problem-solving heuristics as goals of interactive systems design. Whereas hypertext joins and relates ideas, Hyperwriter additionally, and as a direct outcome of the design approach, has ways of separating, ordering and polishing ideas. As an application for creative writing, Hyperwriter supports effective thinking over a period of time. (This paper was drafted, organised and re-organised using the system itself.) Some issues arising in implementing Hyperwriter are also discussed.

Key words: problem solving heuristics, hypertext, user interface design

Short running title: Problem solving in hypertext and creative writing

Introduction

Heylighen (1991) claims, “in order to efficiently tackle complex problems, user and support system should intimately interact, complementing each other’s weaknesses.” He then rationalises the development of a particular hypertext system. Our motivation in this paper is to take problem solving heuristics, claimed to be successful for human problem solving, and to explore their reification in interactive computer systems design. That is, if certain heuristics have aided people solve or avoid problems, then there may be advantages in designing systems so that, in some fashion, the application of those heuristics are supported and naturally encouraged by the system.

There are two complementary aspects to this approach: the heuristics may be applied *implicitly*, in a deep sense, guiding the designer but perhaps not explicitly available to the user; secondly, the heuristics may suggest specific features or functionality, and are therefore directly available *explicitly* to the user as commands. In either case, the heuristics

may support problem solving in the application domain, or problem solving in applying the interactive system effectively to the tasks in hand. In this paper we need not take a rigid view on whether the system itself should implement the heuristics (which results in 'intelligent interfaces'), whether the human should, or whether the human-computer as a pair should do so. Nor need we take a rigid view on whether the heuristics (or in some cases, algorithms) are simple or complex, formal or aesthetic. Thus, at its most general, we may expect to find useful ideas anywhere from the recommendations of Aristotle to the heuristics of modern AI. The intention is to improve human-computer interaction, with a view to solving users' problems.

A possible trap of the proposed approach is that the designer may be tempted to provide facilities to help solve users' problems that are themselves caused by inappropriate design! Of course, in this case, the designer has failed to apply the well known problem solving heuristic, "avoid problems rather than solve them." Although heuristics make a contribution to design, there is more to user interface design than just solving the user's problems. Aesthetics (such as colour schemes, graphical user interfaces) are as much a contribution to effectiveness even as complex, formal problem solving (such as logical inference). They are also a major contribution to marketing a system.

The present paper, then, develops the idea of explicitly supporting problem solving to improve human-computer interaction. To the extent that a *particular* system, and indeed a particular system development exercise, can support any *general* claim, we have succeeded. Moreover, we certainly show that to pursue this approach results in a creative widening of perspectives, which in itself may be beneficial in a design exercise regardless of whether one might wish to adopt those perspectives within a particular context.

A case study

Hyperwriter is the name given to the system developed to explore the ideas summarised above. Hyperwriter is an attempt to explore using problem solving heuristics in the design process *and* in the choice of design facilities. Also, Hyperwriter is an application program to support the task of writing articles: problem solving heuristics also apply to the task domain itself.

It would of course have been interesting to report on specific studies with Hyperwriter, its impact on creative writing and on real-life problems, both in and out of the classroom environment. In a sense this would have taken us too deep into the particular domain of creative writing, rather than the general area of user interface design and problem solving. Instead, Hyperwriter is available from the author,[1] and interested parties may proceed in any direction they wish — Hyperwriter is in source form and may be modified easily. For this reason we will discuss at some length below Hyperwriter's development background and

various ideas to overcome some of its present limitations. Moreover, we believe it is an important scientific statement that the program *exactly as discussed in this paper* exists and that its limitations should be admitted, whether or not one is actually interested in the implementation!

In the case explored in this paper, we start with certain of Edward de Bono's heuristics (e.g., de Bono, 1992) and apply them in an appropriate application area, which is hypertext authoring.

de Bono's problem solving heuristics

Briefly, Edward de Bono's ideas may be summarised as follows. He considers thinking is a skill; he compares it to manual skills, to make the point that people can be trained to think more effectively, and that they can use tools to help them think. De Bono's tools correspond to our problem solving heuristics. He is particularly concerned with mnemonics, and his tools often have catchy names. Part of the purpose of this is so that the choice, manner and order of application of thinking tools can be explicitly thought about (as well as brought to mind as needed), and in groups, their use can be negotiated. Secondly, de Bono makes a distinction between reductive thinking and perceptual organisation. Reductive thinking is how we apply logic to move from premises to conclusions; perceptual organisation is how we choose those premises to be concerned with, and how we influence our inferences by our value systems. In everyday thinking, perceptual organisation (and conflicts with others' different perceptual assumptions) is crucial. Thirdly, de Bono is concerned with enabling people to think effectively in the social setting, for example to cope with negative criticism from others and to avoid it in themselves.

De Bono's ideas are popular and academically controversial, though very little is evaluated empirically in the accessible literature (Gilhooly, 1988). A critique of his ideas, however, is not the purpose of this paper. For our purposes, de Bono's tools for thought are an *example*.

Related literature

The reader of this paper need not accept de Bono's notions as such, but he or she is invited to consider the constructive application to user interface design of other thinking and problem-solving principles. Classic examples are Descartes (1968), Hadamard (1945) and Polya (1981). Gilhooly (1988) has a psychologically-informed position on some of these and other methods. Thimbleby (1987) uses a 'make principles' principle of Descartes for a software system design; Runciman and Thimbleby (1986) explore Polya's 'thinking backwards' heuristic to a range of interactive systems; Thimbleby (1988) discusses 'delaying committment' as a heuristic applicable to a wide range of software design issues. These approaches to problem solving are essentially informal; it is also possible to apply well-defined, formal AI and other algorithmic methods of problem solving to user interface design (e.g., Thimbleby,

1992b). Thimbleby (1990) argues that conventional mathematical problem solving techniques (e.g., algebraic simplification) can be used widely in user interfaces. Funke (1991) reviews human aspects of complex problem solving where the technical problem (e.g., linear optimisation) is well understood.

It is essential to distinguish between the problems a computer is solving (or intended to solve) and the problems a user faces. Very many computer applications solve well-defined problems, such circuit optimisation. To the extent that an application solves the problem, the problem solving is not a user interface design issue, whether or not it is a substantial AI problem in its own right; that is, the problem *for the user* is not *what* the computer solves. In other applications, the computer helps the user form a solution to an underspecified problem, and the computer and user then cooperate in searching for one, for example, the computer may *critique* the user's progress (Fischer, Lemke, Mastaglio & Morch, 1991).

Since we chose hypertext as the application area, it is easy to make the distinction between the form and content of the user interface; simply, the computer does not interpret the content (being unconstrained natural language). We are therefore able to focus easily on applying the given heuristics to the form of the design. But it is also possible to work in reverse: starting from an application area, then to search for applicable heuristics. De Bono's thinking tools have been used in other domains of system design: for example, Portmann and Easterbrook (1992) discuss his tools in knowledge elicitation.

We can contrast *generative principles* (in this case: use problem solving heuristics in design) versus *interpretative attitudes*. Laurel (1991), Suchman (1987) and Winograd and Flores (1986) are substantial examples of the interpretative attitudes: at least to this author, it is not obvious how to use their ideas constructively in the design process, as opposed merely to improve understanding of design issues. Certainly the value of these authors is to provide a new perceptual framework.

There is a considerable and dynamic literature in both creativity and, more specifically, writing with computers, which we cannot begin to do justice to here. This journal, *AI & Society*, **7**(1) is a special issue on creativity, as is the *AISB Quarterly*, (85), Autumn 1993. For writing, see Costanzo (1989) or the *Computers and Writing* series of conferences (e.g., Holt & Williams, 1992).

Hyperwriter

The hypertext application Hyperwriter is a writer's tool for supporting creative thinking, following the suggestion that writing itself is deferred thinking (Bolter, 1991).

Hyperwriter does not distinguish between ‘authors’ and ‘users.’ All Hyperwriter users are assumed to be creative writers, and hence wanting as much flexibility as possible.

Hyperwriter is implemented in HyperCard (Coulouris & Thimbleby, 1992). It supports HyperCard’s standard operations, and as a hypertext system supports the usual hypertext operations.

Hyperwriter’s design applies heuristics both implicitly and explicitly in the user interface:

Implicit heuristics: More than linking

By likening the skill of thinking to the skill of carpentry, de Bono suggests that ideas can be cut up, stuck together and shaped up.

Most hypertext systems, however, only let users (primarily hypertext authors) stick concepts together, using methods such as maps, links or hot text. It is arguable that the inevitable result of too much sticking is a sticky mess: indeed conventional hypertext sticking is routinely criticised for being incomprehensible. Users get lost, partly because hypertext authors have to try to anticipate all reasonable ways of sticking concepts together, and this generality, stickiness if you like, is overkill for the scale of any *specific* task a user embarks on.

In contrast, Hyperwriter lets the user cut, stick and shape ideas easily. The way Hyperwriter does this is innovative (though not definitive) and suggests possible extensions to other forms of hypertext.

Cutting is the converse of sticking. But Hyperwriter also provides a tool for sorting ideas and moving them around; this is an aspect of de Bono’s idea shaping. With a lot of shaping, it is important that Hyperwriter keeps track of what parts the user is shaping. Thus, with a single menu selection Hyperwriter finds an idea that has yet not been shaped, or one that has been shaped but has since been cut up or stuck to another idea.

The usual advantages of hypertext are also available in Hyperwriter: the user can navigate through the text, and sections of text (hypertext nodes) can be related to each other. Furthermore, the structure of the hypertext is easily rearranged: sections can be easily reordered and related to each other in new ways. This is a large-scale form of shaping.

Sticking ideas

Ideas can be joined to other ideas expressed earlier or later. Hyperwriter can find ideas that are immediately adjacent, or written using the same thinking tool, and can join them together. The feature enables similar — either created with the same tool, or adjacent in the document — ideas to be combined.

(Sticking ideas is the weakest part of Hyperwriter as implemented: but this is hardly an issue, since linking is so well-supported in conventional hypertext that it needs no exhaustive reworking here.)

Sorting ideas

A table of contents is used as an interactive map of the hypertext. The table of contents is in view at all times, and is shown next to the current section the user is working on.

The table of contents can be used to rearrange and to re-link. The table of contents is similar to an outliner for a word processor: sections can be picked up and dragged around to change their position within the hypertext. The nesting level of a section can be changed by dragging a selection horizontally. By selecting *parts* of sections, these too can be moved by clicking in the table of contents to indicate their intended location.

Three commands **Reverse**, **Rotate down** and **Rotate up** change the order of sections selected from the table of contents. A process can be described in the order in which it occurs, but it may be better explained or thought about in reverse, from effects to causes or *vice versa*. Likewise it may be more productive to think backwards from a solution to the problem, but easier to understand when the problem is given first. **Reverse** reverses the order of sections and is used for such reasons. **Rotate down** puts the last section (of a selection) first, and moves the other sections down, which can be useful if a summarising section turns out to be more appropriate as an introduction. Conversely, **Rotate up** makes the first section of a selection the concluding; it can also undo an unwanted **Rotate down**. With these three tools, the nesting level of sections is preserved. An example of rotation is shown in figure 1.

Cutting ideas

Any section can be cut up. One can either make a new section out of part of a section, by splitting it, or you can move part of a section into another section, to extend it. The purpose is to help focus thought, by extracting features of ideas, to analyse and break down concepts into simpler thoughts. As will be described below, each section can be labelled with the thinking tool that was used while it was being written: when a section is split or joined, Hyperwriter keeps track of the relevant tools in use. This can be useful at a later stage, particularly as Hyperwriter can review the tool usage, so the author can consider whether the particular use (e.g., the order in which the writing was created by different tools) is appropriate for the purpose.

Finding and shaping ideas

After moving, splitting and joining ideas, it is useful to further shape and polish them. But where are the ideas you have been working on? Hyperwriter keeps track of which sections have been worked on and can find those that are not yet polished. Importantly, ideas can be polished systematically, even though the user may lose track of them.

Each section has a date and time associated with it, which is updated automatically whenever the section is changed (except when it is merely moved). Each section also has a flag that is used to indicate whether the user has 'finished' polishing it. Hyperwriter can then find the most recent unpolished ideas. Hyperwriter also says how long ago each section was worked on, in appropriate units (months, days, hours, minutes, seconds). The time sequencing makes it possible to maintain a thread of thought, since it is a replay of the history and can easily be reviewed.

The purpose of shaping ideas is to help make judgements, comparisons, hypotheses and to stand back, for instance to review the work. Of course, the shaping mechanism can also be used to simplify and improve style rather than content. (Hyperwriter could clearly be extended to deal with many different sorts of shaping, and could be provided with many tools to help the user locate the most productive texts to shape up.)

The opposite operation to polishing might be called scratching. (In Hyperwriter this is achieved by unchecking a 'polish box.') Typically, the user will polish ideas until Hyperwriter says there are no unpolished ideas left, then the user unpolishes everything (as a single command) and starts over, refining their text. It might be desirable to have finer control on polishing, but the current approach has the advantage of simplicity and lower probability of user error than a more sophisticated approach.

Thus it is apparent that what polishing supports is the separation, or independence, of micro-level and structural-level writing. In a conventional text editor, the user would have to keep track mentally of the other levels, thus interfering with efficient work and risking failing to make appropriate changes at the other levels. That Hyperwriter provides many tools to make very quick — but possibly major changes (editing, cutting and sticking) — redoubles the importance of providing polishing support.

Finally, the user can choose what section they want explicitly. This is normal browsing; Hyperwriter allows users to move from place to place in a variety of ways. Clicking on a line in the table of contents takes the user to the corresponding section; this is the simplest form of browsing, using the table of contents as an active map. Various buttons are provided for different sorts of search, for example to move to the start or end of the current text's enclosing section.

Explicit heuristics: Thinking tools

Hyperwriter is a particular hypertext application and provides other features to support creative thinking. These features are independent of Hyperwriter's hypertext features, described above, except that they are similarly based on de Bono's approach.

de Bono suggests 'six thinking hats,' each of a different colour. The white hat indicates 'information gathering' type thinking; the black hat indicates 'judgement' oriented thinking; and so on. The virtue of the hats is that they enable conscious decision making about the appropriate style

of thinking, or in Hyperwriter, the appropriate style of writing. For example, before a red hat (emotional) section, we should generally make sure we have sorted out the relevant information (white hat). In writing a document, one further has the choice whether the style of thinking is *disposable*: whether it is merely an aid to clear thinking on the part of the author, but not necessarily helpful for the reader. (This would typically be the case for red hat thinking in an academic document.)

In Hyperwriter, any section can be labelled with a hat colour to mark the style of thinking used in that section of the hypertext. If the user forgets what a colour means, an option says what the hat means rather than attaching the colour to the current section. A summary of the sequence of thinking hat colours used in the entire hypertext is then available. Different sorts of problem and different sorts of task benefit from different orders of using thinking hats: Hyperwriter both lets the user see the current sequence and to change it if desired.

Sections of the same colour can be stuck together directly: Hyperwriter can locate same-coloured sections earlier or later and join them together. This is useful if the user wishes to join together and review all thinking under the same coloured hat. Conversely, it is easy to split sections apart again so that they can be moved independently.

Sequence provides a summary of the sequence of colours that have been used in the hypertext (figure 2). **Review** provides a similar summary, but ordered by time rather than position. **Review** also provides the dates and times of using colours, as this can be useful if it is a month since any blue hat thinking has been performed! Unfortunately these summaries are not active in the same way that the table of contents is active. It would be possible to make direct manipulation actions on the colour summary perform similar actions as actions on the table of contents.

As well as Thinking Hats, other of de Bono's tools for thinking are available. As with the hat colours, no special (e.g., linguistic) support is provided, though Hyperwriter remembers when and where a user has deployed a tool. As with the coloured hats, if the user forgets what a tool is, holding an option provides a brief summary.

PMI is an example of a tool that interacts with other tools.[2] Using PMI, a thinker considers the plus points of an idea (P), then the minus points (M), then the interesting points (I). This is sequential development of a line of thought; indeed, one purpose of the tool is that it is an encouragement to consider different aspects of an idea rather than jump to a quick judgement on it. It is exactly at the times when an idea seems obvious, or is obviously simple, that a tool such as PMI increases creativity. It is likely that the user will then wish to split up the thinking PMI has prompted and reorder it for exposition: thus Hyperwriter

provides the other tools, such as **Split** and the direct manipulation features of the table of contents.

A simple (but effective) random word tool 'works' — it generates a random noun, and the word remains displayed until the user dismisses it. The use of this tool is remembered by Hyperwriter, like any other tool. Its use, similarly, can be reviewed.

The user can add or remove tools according to their needs, as well as define help text for them or redefine help for existing tools. For example, a 'same as' tool, a tool which is normally used for *destroying* creativity with the put-down, "that idea is the same as something we already know," might be added to help the user remember to consider and write defences against such negative criticism.

A **Save** function is provided, and allows the user to save the document as a text file that may then be opened by a conventional word processor. It can also save in a format suitable for LaTeX. (This is how this paper was originated: it was drafted in Hyperwriter and then imported into Microsoft Word.)

The development process & limitations

Hyperwriter started out as an evening's work of a few hours duration inspired by reading Edward de Bono. Hyperwriter was then in a stage that could be talked about 'hands on' with colleagues, however we had to get it to work more realistically and that took about two days of intermittent tinkering, finally with the results exactly as described in this paper. Hyperwriter shows how effective hyperprogramming systems like HyperCard are (Coulouris & Thimbleby, 1992), particularly for exploratory design.

This section on the development and limitations of Hyperwriter skirts around the tension that a development system like HyperCard raises. On the one hand, there are very many design possibilities and there is very much scope for development: an unfinished outcome consistent with the claims of this paper, that the direct application of problem solving heuristics in user interface design is productive. On the other hand, HyperCard as an implementation base is quite pushed to its limits with Hyperwriter: HyperCard is excellent for exploring concepts, but not for converting them to practicality.

Possible developments

Text can be split and joined. It would be in keeping with the spirit of hypertext if text could also be referred to, by cross references. Naturally the cross references would be updated as the sections referenced changed or were moved. Clicking on the cross references would take the user to the referenced section.

Cross references can be considered a partial order on a hypertext (e.g., place definitions before mentions): there is no reason why a hypertext system should not provide topological sorting as one of its interface tools (Thimbleby, 1992a). Since there are generally many total orderings in which a particular cross reference ordering may be embedded, a tool for creative writing would be able to generate all such orderings — some might be surprising, and suggest either improved thought or improved exposition to the user.

The sorting features (reverse and rotate) could be extended to provide numerical or topological sorting based on the user's valuations of sections: at present, the user sorts sections based on what they remember of their content and the purpose of the document as a whole. Had Hyperwriter supported annotating sections (such as, "this is a section rating 5 in importance"; or, "this section should follow this one defining the concept it uses"), then these annotations could have been used to help sort sections with no burden on the user's memory.

The table of contents is not equal opportunity (Runciman and Thimbleby, 1986; Thimbleby, 1990), and this is a noticeable limitation. Users can change section titles only in the text region; the table of contents itself cannot be typed into. Since the table of contents provides a concise overview without the clutter of section text, it would be desirable to permit the user the freedom to type in either place. Many other operations, such as cut, copy and paste on the table of contents would be useful: the user could construct some text from, say, an encyclopaedia, copy all sections they have worked on, and then work, in a far more focused way, on those sections pasted into a new and much simpler hypertext, without the confusion of the rest of the entire encyclopaedia.

Hyperwriter does not provide structures for organising long sequences of thought, say, as might be appropriate for book-length writing projects. A menu might easily be added to allow the user to introduce structures, such as 'experiment report' which would be a series of appropriately nested sections: Title, Abstract, Introduction, Method, Results, Discussion, Conclusion, References. de Bono himself suggests several such structures specifically for directing creative thought. Other possibilities include the stages of 'structured design methods' and structures used for literary and dramatic composition.

Hyperwriter shows the user all sections of a hypertext document; they all have the same visible status. It would be useful to be able to hide and reveal selected sections depending on the purpose of the user (and whether as writer or reader). Some sections could be written to help the user get their ideas sorted out but might not be for other people to read. Some hidden sections could contain draft ideas for future development but yet allow a 'finished' document be shown to other people at any time. Some sections could be 'thinking aloud' between several authors about the text rather than about the topic of the text. And so on: this idea is

consistent with de Bono's notion of sorting ideas, in the sense of 'sorting out' the ideas one wants to work with and ignoring others. It is also one way of implementing the idea mentioned above, for reducing clutter and enabling users to work with parts of larger hypertexts more conveniently. Hyperwriter would keep track of the currently hidden ideas.

Finally, Hyperwriter provides no features such as spell checking, grammar checking or layout. Although authors may miss such features, they would probably interfere with creativity (at least at the level we have been discussing it), and they are anyway more properly dealt with in a word processor where page layout and presentation can more naturally be attended to. Removing stylistic faults from a document is only part of the process of writing, and perhaps the easiest part. (Which is another way of saying that implementing a decent DTP package is a *lot* of work, and was not our present business. It would be much more reasonable to embed Hyperwriter-like features within an existing word processor if one was available. Perhaps future interapplication scripting languages will make such integration feasible.)

Hypertext or outliner?

A hypertext is usually a directed graph, and navigation is greatly simplified by displaying a map, where the user can point-and-click to browse (move to sections) or use direct manipulation to rearrange them. To make Hyperwriter simpler (indeed, to make its implementation feasible in HyperCard), we chose the map to be a spanning tree of the graph. This is a natural choice for the application, since it can be shown easily and unambiguously as an indented 'table of contents.' Browsing is by point-and-click operations, and direct manipulation on the table of contents supports general rearrangement. The tree map is a powerful outliner.

The tree map has advantages:

- There is a unique and obvious conventional linear text based on the tree.
- Sections can be numbered in a simple and obvious way.
- Trees do not have cycles, and there is a unique path from any section to any other. Users are far less likely to experience 'getting lost in hyperspace.'

de Young (1990) suggests other principled approaches to simplified hypertext linking. Rada (1992) describes how he converted conventional texts (in mark-up) to hypertext.

The implementation

Hyperwriter was implemented in HyperCard, in about 1200 lines of script. HyperCard is impressive as a rapid development tool, but is sadly lacking in support for complex programming (Thimbleby, Jones & Cockburn, 1992). There are no data types, abstract or otherwise. There is

no modularisation. The language (HyperTalk) has severe design flaws. The implementation of HyperCard itself has bugs (e.g., certain events get lost). Hyperwriter therefore has bugs.

A profound problem is HyperCard's notion of undo, which undoes primitive HyperCard actions (such as undoing text deletion). In a programmed system such as Hyperwriter, the user will wish to undo operations that have been implemented as several primitive HyperCard operations (such as undoing a section rearrangement). Although we can intercept the user's request to perform an undo, there is no way that Hyperwriter can determine what the user's last operation was that should be undone.[3]

Some of these bugs are irritating; some can be concealed by careful programming; some seem unavoidable. As bugs are fixed, the size of the program increases, then hitting 30k byte limitations in the size of HyperCard programs! (As a compromise, Hyperwriter has a reset button that can be pressed when things go awry.) Briefly, HyperCard permits 90% of a system to be implemented almost immediately, but cannot implement the remaining 10%. In contrast, a conventional programming system could implement all of a system, but it would take much longer.

It may be said that HyperCard makes a radical distinction between *prototype* and *deliverable* system. The bugs in a HyperCard prototype of a system would force designers to proceed to a new implementation. Overall this second implementation will be more reliable that a conventional prototype that was forced to become the production product. From this point of view, HyperCard's rapidity and inadequacies are a sophisticated combination!

Even if the bugs could be fixed, a HyperCard implementation of Hyperwriter would still be too slow for serious use with large hypertext documents. Indeed, fixing some bugs requires redundancy, and would further reduce speed. This is why we have not paid too much attention to polishing Hyperwriter itself — its role can only be to illustrate an approach to the design of interactive systems.

Literate programming

As we implemented Hyperwriter we tried to document it clearly. Thus as Hyperwriter was being implemented, a natural hypertext document to work on was its own description, indeed the text that resulted in this paper. We found that explaining Hyperwritier while it was being developed is itself a very beneficial activity. The implementation of the system was immediately available and could be edited and tested concurrently with writing the documentation.

Had the explanation of Hyperwriter been written conventionally, in a text file isolated from the implementation of Hyperwriter itself, the effort of switching from 'explaining' to 'programming' would have inhibited this natural process of improvement. An example of this concerns

HyperCard's **First**, **Prev**, **Next** and **Last** commands: it was easier to re-implement them correctly than explain why they didn't work properly! Thus our experience strongly supports Knuth's literate programming views (Knuth, 1992). That many improvements were encouraged by literate programming reinforces the importance of making the transition from explaining to programming as unobtrusive as possible.

Wider issues

Isn't the approach trivial?

Isn't it trivial to apply problem solving heuristics to problems? It is, of course.

Problem solving heuristics are only of significant interest when they are applied to provocative problems. As soon as questions can be solved they do not call for creative problem solving techniques. Questions that can only be solved by people who know the standard answer are merely irritating, and are just party tricks.

A lot of confusion over the application of heuristics stems from the fact that when they are explained by simple examples, of course, any results are so obvious that no heuristic seems required! However designers who have tried being creative will know that it is not as easy as hindsight. Hindsight seems easy because you already know what to look at; part of the problem of design is to focus on the pertinent issues. (The reason for iterative design is that a designer does not know all the pertinent issues until after a system is designed.) Similarly for users, solving a problem by pressing the twiddle-alt-control key is only a simple fact once they know it. The issue, in design or use, is to move from the profoundly unknown unknown to the trivial — the easy to use.

Why should Hyperwriter improve creativity?

Why should Hyperwriter (or any other similar tool) enhance creativity? If a tool is properly designed, it will reduce the user's overheads, the distraction caused by doing operations. In a badly designed system (or for an unskilled user), whatever task it is supporting, the 'housekeeping' of using the tool itself interferes with the task-oriented goals of the user.

Whatever creativity is, it certainly involves operating with ideas, and we follow de Bono in supposing that intentional control of creativity is important: creativity need not be an uncontrolled or unconscious process. Thus a tool supporting creative processes could enable the user to postpone and interleave types of mental operations without having to keep track of them or their priorities. For example, one heuristic is to work backwards from a solution to the problem; if the computer supports the development of ideas in any order, then the user need not explicitly keep track of this subsidiary process.

Though word processors sacrifice certain very human skills (such as calligraphy), even they can facilitate creativity (so long as the creativity is not in details of typography or other aspects of design whose overheads the word processor has assumed from the user). In turn, an outliner further increases creative expression. Hyperwriter is both a word processor and outliner, and it also adds heuristic operations and structures claimed to support creativity. Hence a weak answer to the question would be that Hyperwriter enhances creativity because even word processors do; a stronger answer is that Hyperwriter also provides features specifically for creative purposes, and it supplements this with *discretionary* features to enhance creativity. Where these features are useful, obviously they will succeed, when they are not useful, they need not interfere.

Hyperwriter, by its specific tools and general design, serves to remind the user that there are alternatives. There are alternative perceptions on whatever the user is working on. There are alternative expository orders. There are different styles and attitudes of thinking. Hyperwriter enhances creativity because it reminds the user that there are more possibilities, and it makes exploring them and how they affect the current project very easy.

Ultimately, whether Hyperwriter improves creativity is both an empirical and a conceptual question. Any empirical answer would in turn beg the question of what creativity is, and what 'good' creativity is.

Social aspects of problem solving

Computers are increasingly used on a social scale; the discussion above was concerned with the single-user single-computer system. Clearly, just as heuristics can be applied in user interfaces design to solve an individual's problems (both task-related problems and problems of the computer use itself), groups of users and systems designed for them might be approached from the same point of view.

One direct suggestion for applying single user heuristics has been made (Thimbleby, Anderson & Witten, 1990): an individual's problems over time can be transformed into problems over space — suggesting that one consider designing systems that treat users in different places analogously to treating the same user at different times.

With an emphasis on explicit computer-based problem solving support, Fischer and Reeves (1992), amongst others, show how some of these ideas can be realised in 'co-operative problem solving' systems. Nardi and Miller (1991) discuss cooperative work using conventional spreadsheets: they claim that groups of people organise themselves so that problems can be solved collectively.

More specifically, if the task is creative writing, as it is in Hyperwriter, how might the approach be generalised from single user designs up to a social scale? When a single user is creative, there are few ideas to consider; but when many users cooperate on some creative enterprise it

becomes useful to evaluate and compare multiple ideas, lest one merely wants as many ideas as users. Some users' ideas will be better than others (however 'better' is measured). The best ideas, then, should then be 'taken forward' in some sense. Of course, this is just an informal distributed problem solving scenario, and is therefore amenable to the appropriate heuristics, including genetic algorithms. Johnson-Laird (1993) has suggested that genetic algorithms are anyway closely connected with creativity.

When more than one user is involved, access control becomes an issue. At its simplest, some or all of the text could be fixed. Fixed text, then, would be supplied by 'authors' and other text (including that resulting from high level operations operations) would be for the 'users.' The distinction between authors and users would be especially useful where proprietary information was involved: a Hyperwriter approach would let users construct information *out of* the initial hypertext and yet would not be able to corrupt the original material. This is analogous to Hyperwriter's present approach of permitting users to construct new ideas (or new presentations of ideas) out of and with their existing material. Nelson (1990) shows how such ideas might be implemented on a large scale and support social conventions, such as royalties.

Conclusions

This paper describes an approach to interactive systems design, a particular instance of that approach, and a particular system resulting from employing that approach. The paper as a whole claims that the success of Hyperwriter as an interesting and particular application is an outcome of the approach rather than the only such plausible outcome.

Hyperwriter was designed for creative writing with hypertext, and was inspired directly by Edward de Bono's thinking tools. Hypertext conventionally links ideas. Hyperwriter demonstrates that hypertext can be more than linking. Hyperwriter allows ideas to be separated, reordered, joined and systematically shaped as well as linked. Independently of this, it supports several tools for creative thinking.

The work described in this paper supports two main claims:

- Heuristics for problem solving can be applied profitably to the problems users face when interacting with computers, as well as to the problems in their task domain.
- As an instance of the first claim: hypertext systems can be more flexible and can better support creative use. Linking is not sufficient. (The approach suggested by de Bono's thinking tools requires more than 'linking' concepts, the forte of conventional hypertext. In Hyperwriter, the user can cut, rearrange, stick and shape ideas.)

Also of interest are the following two points:

- The development of explanatory material (documentation) is usefully interleaved with design and implementation. This is a design heuristic for the designer, rather than the user, though the user benefits with improved manuals.
- Ironically, limitations in prototyping tools can be advantageous for reliable production systems.

That Hyperwriter, as an outcome of a particular design process, is of interest in itself lends further support to these claims. Beyond the scope of the case study of this paper, there are many other sources of problem solving heuristics: it will therefore be exciting to see a wide range of heuristics being applied systematically in interactive systems design to empower users.

References

E. de Bono (1992), *Teach Your Child to Think*, Viking.

J. D. Bolter (1991), *Writing Space*, Lawrence Erlbaum Associates.

G. F. Coulouris & H. Thimbleby (1992), *HyperProgramming*, Addison-Wesley.

W. V. Costanzo (1989), *The Electronic Text: Learning to Write, Read, and Reason with Computers*, Educational Technology Publications.

R. Descartes, translated by F. Sutcliffe (1968), *Discourse on Method and The Meditations*, Penguin.

G. Fischer, A. C. Lemke, T. Mastaglio & A. I. Morch (1991), "The role of critiquing in cooperative problem solving," *ACM Transactions on Information Systems*, **9**(3), pp123–151.

G. Fischer & B. Reeves (1992), "Beyond intelligent interfaces: Exploring, analysing, and creating success models of cooperative problem solving," *Journal of Applied Intelligence*, **1**(4), pp311–332.

J. Funke (1991), "Solving Complex Problems: Exploration and Control of Complex Systems," in *Complex Problem Solving: Principles and Mechanisms*, edited by R. J. Sternberg & P. A. Frensch, pp185–222, Lawrence Erlbaum.

K. J. Gilhooly (1988), *Thinking: Directed, Undirected and Creative*, 2nd. ed., Academic Press.

J. Hadamard (1945), *The Psychology of Invention in the Mathematical Field*, Princeton University Press.

F. Heylighen (1991), "Design of a hypermedia interface translating between associative and formal representations," *International Journal of Man-Machine Studies*, **35**(4), pp491–515.

P. O'B. Holt & N. Williams, eds. (1992), *Computers and Writing: State of the Art*, Kluwer Academic Publishers.

P. N. Johnson-Laird (1993), *Human and Machine Thinking*, Lawrence Erlbaum Associates.

D. E. Knuth (1992), «*Literate Programming*», Center for the Study of Language and Information, Stanford University.

B. Laurel (1991), *Computers As Theatre*, Addison-Wesley.

B. A. Nardi & J. R. Miller (1991), "Twinkling lights and nested loops: distributed problem solving and spreadsheet development," *Computer-supported Cooperative Work and Groupware*, edited by S. Greenberg, pp29–52, Academic Press.

T. H. Nelson (1990), *Literary Machines 90.1*, Mindful Press.

G. Polya (1981), *Mathematical Discovery*, Combined Edition, John Wiley & Sons.

M–M. Portmann & S. M. Easterbrook (1992), "PMI: Knowledge Elicitation and De Bono's Thinking Tools," in *Current Developments in Knowledge Acquisition—EKAW92*, edited by Th. Wetter, K–D. Althoff, J. Boose, B. R. Gaines, M. Linster & F. Schmalhofer, pp264–282, Springer Verlag.

R. Rada (1992), "Converting A Textbook To Hypertext," *ACM Transactions on Office Information Systems*, **10**(3), pp294–315.

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

L. A. Suchman (1987), *Plans And Situated Actions*, Cambridge University Press.

H. Thimbleby (1987), "The Design of a Terminal Independent Package," *Software—Practice and Experience*, **17**(15), pp351–367.

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

H. Thimbleby (1990), *User Interface Design*, Addison-Wesley.

H. Thimbleby (1991), "Can Humans Think?" *Ergonomics*, **34**(10), pp1269–1287.

H. Thimbleby (1992a), "An Author's Cross-referencer," in *Computers and Writing: State of the Art*, edited by P. O'B. Holt & N. Williams, pp90–108, Kluwer Academic Publishers.

H. Thimbleby (1992b), "Heuristics for Cognitive Tools," in NATO ASI Series F, *Proceedings NATO Advanced Research Workshop on Mindtools and Cognitive Modelling*, Cognitive Tools for Learning, P. A. M. Kommers, D. H. Jonassen & J. T. Mayes, editors, pp161–168, Springer Verlag.

H. Thimbleby, S. O. Anderson & I. H. Witten (1990), "Reflexive CSCW: Supporting Cooperative Long–Term Personal Work," *Interacting with Computers*, **2**(3), pp330–336.

H. Thimbleby, S. Jones & A. J. G. Cockburn (1992), “HyperCard: An Object Oriented Disappointment,” *Building Interactive Systems: Architectures and Tools*, P. D. Gray & R. Took, editors, pp35–55, Springer-Verlag.

L. de Young (1990), “Linking Considered Harmful,” in *Hypertext: Concepts, Systems and Applications*, edited by A. Rizk, N. Streitz & J. André, pp238–249, Cambridge University Press.

T. Winograd & F. Flores (1986), *Understanding Computers And Cognition,* Ablex.

Acknowledgements

Colin Beardon, Andrew Cockburn, Peter Ladkin and Mike Sharples have made very useful and constructive suggestions that I have been able to incorporate into this paper.

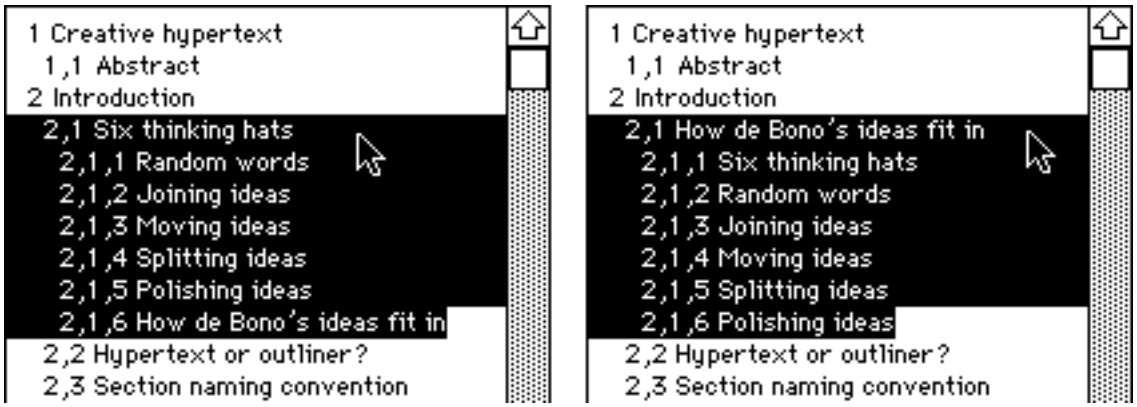

Figure 1. Before and after rotating sections

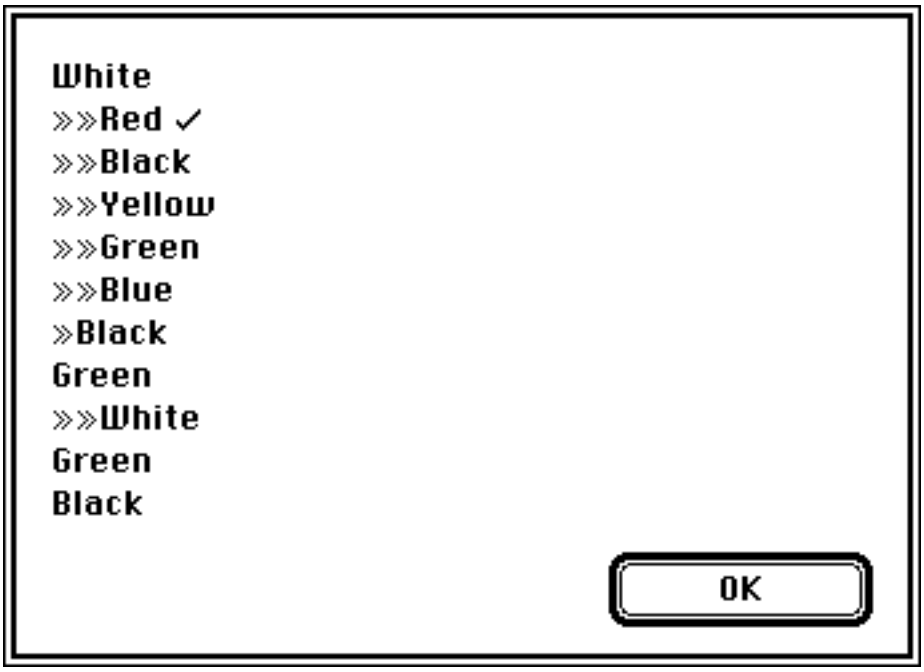

Figure 2. A nested tool-use report, here of use of the Thinking Hat colours

Notes

1 Hyperwriter is implemented in HyperCard, currently at version 2.1. It will run on any Apple Macintosh computer capable of running HyperCard. To obtain Hyperwriter, in the first instance please email the author for details.

2 PMI was used by Portmann and Easterbrook (*op. cit.*).

3 We can't ask the user to confirm that the last operation was implemented by Hyperwriter (rather than being a basic HyperCard operation, such as a text edit) — even supposing they could tell — because an avalanche of errors would ensue if the user incorrectly affirmed it was.