User-Tailorable Systems: Pressing the Issues with Buttons

Allan MacLean, Kathleen Carter, Lennart Lövstrand and Thomas Moran

Rank Xerox EuroPARC,
61 Regent Street, Cambridge CB2 1AB, England

ABSTRACT
It is impossible to design systems which are appropriate for all users and all situations. We believe that a useful technique is to have end users tailor their systems to match their personal work practices. This requires not only systems which *can* be tailored, but a culture within which users feel in control of the system and in which tailoring is the norm. In a two-pronged research project we have worked closely with a group of users to develop a system to support tailoring and to help the users evolve a "tailoring culture". This has resulted in a flexible system based around the use of distributed on-screen Buttons to support a range of tailoring techniques.

KEYWORDS: Tailorability; Modifiability; Customization; User Interface Design; Office Systems; Design Process.

TAILORABLE SYSTEMS

User-tailorable computer systems have been a goal in some form for a number of years, but achievements to date have not lived up to the promise. Progress has been made by producing higher level languages which have increased the productivity of programmers and like-minded users [21], and an increasing number of applications are available which have their own languages built in (e.g. Framework in the PC world [1]; Apple's HyperCard [12]). Alternatively, some systems provide their user with a range of predefined parameters which allow limited control. EMACS is probably one of the most successful examples of a tailorable system ("extensible and customizable", as its author describes it [23]). EMACS is tailorable both by programming and by setting parameters, and there are many people who would use no other editor. However, there are also many people who are inhibited from using EMACS, far less tailoring it, because of its complexity [22].

Significant effort is required to acquire the skills necessary to use these tailoring mechanisms. Figure 1 characterises the relationship between the amount of skill required and tailoring power using a mountain climbing analogy. In a system like EMACS with over a hundred parameters a steep incline has to be climbed to begin to understand how or if a desired change can be made and what kinds of changes are possible. After this, there is a more gentle slope where the effects of different parameters can be explored. To make more extensive changes beyond what is possible by changing parameters, a programming language has to be used. A vertical cliff represents the barrier to understanding and using that language.

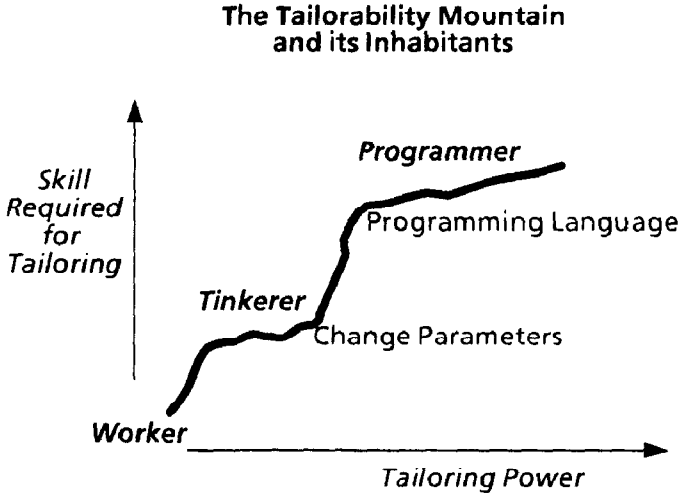

Figure 1. People with different levels of tailoring skills, and changes in skill required for increasing tailoring power using two common tailoring mechanisms. Steep slopes are barriers to skill acquisition.

The different terraces on this mountain lead to different cultures with people of different skill levels inhabiting them. At the risk of minor caricature we can characterise these cultures as follows:

- **Worker**: *Lives on the plains*. No interest in the computer system *per se*. Just wants to get work

done. No expectation of being able to tailor the system.

- **Tinkerer**: *Lives on the foothills*. A worker who enjoys exploring the computer system, but may not fully understand it.

- **Programmer**: *Lives on the peaks*. A guru who understands the system inside out. Has formal training or extensive experience in computing. The programmer may have an application support role, but more often is not accessible to ordinary workers.

The "worker" is a particular challenge for building a tailoring culture. If workers have no expectation of controlling changes to the system, they are not in a good position to understand what changes might be possible let alone make them happen themselves. In such a position, communicating ideas or requirements to tinkerers or programmers is also likely to be problematic. Our goal is to give the worker a feeling of ownership of the system, to feel in control of changing the system and to understand what *can* be changed. From such a position and with an appropriate system, we would expect workers to be able to carry out a considerable amount of change for themselves, to know who in the community to ask for help, what to ask for and how to interpret offerings from others.

The foregoing discussion highlights two routes to make systems more tailorable for the worker. First, we can aim to make tailoring mechanisms accessible. The major problem suggested by figure 1 is that it is all too easy to spend considerable effort in attempting to learn new skills with no reward in increased power over what changes can be made. One goal is therefore to remove the cliffs and reduce the slope of the inclines so that more continuous progress can be made as effort is expended. As a second route, we believe that tailoring should be a community effort. Programmers, tinkerers and workers each have important and different roles. We want to form a single culture within which we can take full advantage of these different skills for the benefit of the community as a whole. Again, "smoothing off" the mountain should allow people with different skills to intermingle more and so communicate better with each other. (It is worth emphasising that we are interested here in skills required for *tailoring*. We do not intend to suggest that a "programmer" is more skilled than a "worker" in any absolute sense. If we were to focus on skills in the work domain, we would find that the worker was most skilled and the programmer least skilled.)

This paper describes a project within which we have implemented a software system designed to make tailoring a reality for the "worker". We have made the design process explicitly include user participation, and one of the goals was to bring together the cultures above to grow a "tailoring culture". The design process and the use of the system as it developed were themselves objects of study.

Our "workers" were four members of the EuroPARC administrative staff. None of them were "tinkerers", far less programmers, and when the project started they had no experience of tailoring their system either for themselves or by asking for system facilities they felt would be helpful in their work. Although working within our research community, all have well defined roles to play within the organisation and it was important that our research not impede their day to day work in any way.

The two main activities by the project research team were to build a flexible architecture to make tailoring accessible and to help the workers gain control over changing their system to their own requirements. To facilitate the latter, a new role emerged for one member of our project team, to add the previous ones [6]:

- **Handyman**: *Lives in the foothills* and *the peaks*. Sure-footed individual. Bridges between workers and computer professionals. Works alongside office staff. Responds to their immediate needs. Also able to communicate user needs to programmers for longer term or more complex development.

BUTTONS: TAILORABLE INTERFACE OBJECTS

We have been exploring the issues of end-user tailoring by developing a system based around on-screen buttons. On their own, buttons are not particularly novel interface objects. However, we can claim novelty for the integration we achieve with buttons as *tailorable* interface objects, in a community, and in a user-participative design process.

User Interface to Tailorable Objects

Our Buttons are screen objects in Xerox Lisp which look "pressable" and when pressed (by clicking with the mouse) carry out an action. Buttons can be used without any understanding of the details of the encapsulated action, and thus are a convenient way to tailor the Xerox Lisp environment for individual user needs. The Buttons architecture allows users with little or no programming experience to modify various aspects of buttons for themselves (the labels, the graphical image, aspects of the actions). At the same time, the architecture is sufficiently powerful and flexible to allow users with programming skills to create buttons for novel applications.

A number of modern user interfaces use the concept of a button (or related concepts) as important

components. For example, icons can be thought of as button-like in many respects, but they are typically no more than a mechanism for starting up an application, or referring to a file or process. They are not tailorable.

The Xerox ViewPoint office system has an option for a "Customer Programming Language" (CUSP) which allows desktop activities to be automated and accessed via buttons [25]. However, CUSP buttons can only exist inside documents, not on the desktop; they rely on the CUSP language so there is a barrier for many end users in changing them; and the language used is different from the ViewPoint implementation language, so CUSP cannot access other ViewPoint applications in a flexible way, and cannot be used to extend the environment.

In the Apple world, HyperCard [12] makes extensive use of buttons, again with an English-like programming language (HyperTalk) behind them. In contrast to CUSP, HyperCard is itself just one of many Apple applications - or perhaps more accurately, an application builder. So, although HyperCard contains a number of features which give accessibility for non-programming users, it is aimed at building structures of information. It has no way of accessing internal parts of other applications to tailor their behaviour.

Buttons were originally built into Xerox Lisp as part of the Rooms system [14]. The architecture we are currently using is a considerable extension of and is upwards compatible with Rooms buttons. Buttons can exist on the desktop or in documents. Since all applications such as text processors, mail, etc. in the Xerox Lisp environment also have Lisp as the underlying language, details of the behaviour of these applications can be modified using Buttons. Architecturally, the Buttons system is based on a simple object-oriented framework with an Active Property Engine (in the spirit of Loops' Active Values [3]). Properties may have agents (procedures) which are activated when certain operations are performed on the property, such as setting or getting its value, editing or copying it, etc. This mechanism is able to handle computed values, side effects, dependencies, indirection, lazy evaluation, and more object-oriented paradigms such as class abstraction and delegation. The Buttons architecture will be described more fully elsewhere [16], but note here that the architecture provides a great deal of flexibility for programmers.

The Tailoring Culture

We stress the importance of building a community with a culture of changing the workstation environment. This is more difficult than it may at first seem. Tailorability shifts some of the system design problems to end users, a role for which they are ill-equipped. For example, Grudin and Barnard [13] have shown that when users are asked to carry out what might seem a simple interface design task - designing a set of abbreviations for a given command set - they do a very poor job. Users typically have more problems with the abbreviations they produce themselves than they do with a set which has been designed to conform to a simple abbreviation rule structure. Given such observations, we should be cautious in expecting users to optimise a system for themselves through tailoring. Additional support will be necessary. One approach to help users better understand the possibilities for tailoring would be to design a system so that the range of variations and their consequences were a salient part of the design – MacLean, Young and Moran [19] suggest the possibility of including a rationale with a system as a way of achieving this. Fischer and Lemke [10] advocate a combination of "construction kits and design environments" – basically a collection of possible options and design information to help the user with ways of combining options. However, we have already observed that the type of non-programming user we are most interested in supporting in our project currently lives in a culture whose members have no expectation of being able to change their computing environments. It is therefore unlikely that simply presenting such users with even a helpful and intelligent system will change their expectations.

The approach we employed here was to have a member of our design team (the "handyman") working closely with the target users [6],[7]. This arrangement provided a mechanism for the designers to take careful account of the users' real requirements and for the users to gain a better understanding of how their working environment could be different by helping design it themselves. This latter point is critical in helping to develop a "tailoring culture", as an attitude which has "design" as a component is also one which understands change. Certain Scandinavian approaches to system design argue that this kind of participation of users is essential [2]; Ehn and Kyng argue that system design "should be done *with* users, neither *for* nor *by* them" [9]. Our focus on tailorability extends the value of this approach. We not only build a mutual understanding which helps users to influence the design of the system - we also help them to adopt an attitude which will help them make better *use* of the system.

TAILORING TECHNIQUES

As a first step in growing a tailoring culture, we claim that it should be as easy to change the environment as it is to use it (clearly *all* changes one might want to make will not be so easy - but it is important that *some* should be). Since our buttons are independent objects which can be easily moved around the screen, they provide an excellent mechanism for helping users to evolve their own

personalised environment. One role of the handyman was to seed the environments of users with buttons appropriate for their own personal day to day activities. For example, one of our administrative staff had the task of sending the weekly EuroPARC calendar out by email to a number of different people, to the nearest printer for some other people, and distributing hard-copies to yet other people. A button was produced to carry out most of these tasks with a single mouse click. Other buttons support a small community of users rather than a single individual. For instance, all members of the administrative staff have a button which allows them to add an item to an agenda for weekly meetings. These cases exemplify some of the ways in which buttons directly help users to work more efficiently: by making functionality accessible (the button is visible on the screen at all times), as an accelerator for regularly performed sequences, and as a memory aid for complicated operations. So a remarkable amount of tailoring can be done simply by "begging, stealing or borrowing" appropriate buttons and placing them in strategic places on the screen.

Situated Creation

The next step beyond simply placing buttons around the screen is to give users mechanisms to create new buttons for themselves. *Programming by example* is one possible technique which could be used to minimise overheads for the user by recording the sequences of actions to carry out tasks (see Myers for a review [20]). However, on a multi-process mouse-driven workstation it is very difficult to determine the intent of an action (e.g. is a mouse click referring to a location on the screen; a relative location within a window; a specific object within a window...).

We take an alternative approach which we call "situated creation". It relies on capturing relevant aspects of the system state into a button for later re-use. The idea is that the user carries out some task using normal manual methods, and is then able to encapsulate relevant parts into a button without doing anything which looks like programming. Unlike programming by example, this approach allows the computer to regenerate the state in the most efficient way (if the user got there through a long-winded route, that route is not preserved in the button). For example, a user may be writing a report which requires the same long phrase to be repeated several times. We provide a mechanism which creates a button "containing" the relevant phrase. When this button is pressed, the phrase is entered into the text. In this case the user may only keep the button for a few hours until the report is finished. In other cases, some buttons may become a relatively permanent part of the user's environment. As another example of situated creation, we have modified the window system to include a "buttonize" option. When this is selected, a button is automatically produced with properties relevant for the type of window. For example, the button produced from a text window will allow the user to recreate that window without worrying about the precise location of the underlying file in the filing system; a button produced from a filebrowser will allow a directory to be re-examined at a later time without worrying about re-entering the file pattern and viewing parameters.

Copying and Specialising Buttons

Tailoring can be seen as a process of users evolving the system gradually along with their own changing skills and requirements. So they may have a button of their own, or one provided by a colleague, which does almost what they now want, "except for...". This situation closely relates to the object-oriented programming concept of specialisation, where all the behaviour to remain unchanged is inherited from a suitable object and only the novel behaviour has to be explicitly specified. However, it is difficult for non-programming users to think in terms of abstract object-oriented concepts. Borning and O'Shea [5] have shown that even experienced programmers can have enormous difficulty with some aspects of class inheritance in Smalltalk [11]. Although the Buttons architecture supports a form of inheritance, we currently use it in a more limited way than a traditional object oriented approach. Our Buttons user who wants to create a variant of an existing button would typically copy the entire button and then change a few details as necessary. Although this approach means more duplication of code, it has several advantages. By making individual buttons independent objects they are conceptually simpler for the user to understand. In addition, if a user wants to send a button to someone else by email, it does not require the recipient's environment to already contain a complex hierarchy of classes on which the button relies. In this respect Buttons are for most purposes more akin to what has become known as the "prototype" approach to object-oriented systems [4, 15].

Tailoring-Oriented Attributes

Although object-oriented concepts hold promise for handling certain aspects of tailorability, it is clearly not sufficient simply to provide users with an object-oriented environment and expect them to be able to tailor their system. The arguments already presented about the overheads of a programming language for tailoring apply. One way of alleviating such problems is to identify in advance the kinds of things that the user is most likely to want to be able to change and make sure that they are easily accessible and easily understandable. This strategy has been successfully used for rapid prototyping the structure and content of menu driven dialogues in an environment for creating presentation graphics material [17, 18]. In the design of Boxer, DiSessa uses a concept, which he calls *shallow structuring*, meaning that "...anything the novice is likely to

need to use or modify must be near the surface of the environment" [8].

We have taken this approach in Buttons both globally and locally. From the global perspective (i.e. common to all buttons), we give users direct access to attributes of the appearance and to a text label. These are both very visible parts of the button and are good attributes for users to change as it gets them used to thinking in terms of change and seeing the consequences of their modifications, so helping develop the tailoring culture.

When it comes to making modifications to the behaviour of a button a global approach is clearly impossible since the range of actions a button might carry out is open-ended. Our solution is to provide direct editing access on all buttons to "parameters". Each individual button has its own specification of what attributes should be presented as parameters. So the precise details of what the user has easy access to can be varied from button to button, depending on the attributes the user is likely to want to change. As a simple example, if the user wants to add a facility for checking a regularly used file directory, we have already mentioned that a button can be created directly from a file browser window. Such a button is automatically created with separate parameters that refer to the file pattern, the number of subdirectories to check, and the information to show. Once the user has one such button on the desktop, that button can be copied and have its parameters changed to create a new button giving a fast way of accessing different file information. To take a concrete example, let's say I have a button which will generate a file browser on all files and subdirectories under my personal "text" directory, showing the creation date and author of each file. I can copy that button; change the file pattern parameter to search for all files with the substring "CHI" in them (using a string editor); change the search depth to "1" (using a numeric keypad) to only search one directory level; and choose to show the size of each file (selecting from a menu). So each parameter can be changed using an editor most appropriate for that parameter. The structure of the button is such that I am only presented with parameters which might be especially relevant for the sort of action that particular button carries out (i.e. file browsing options in this case). The type of editor which is brought up implicitly helps understand the type of value that is appropriate for the parameter.

Modifying Program Code

Buttons support the user with some experience of programming, but who is by no means expert in Lisp (i.e. the "tinkerer"). Since the relevant piece of Lisp is encapsulated within the button, someone who has some feel for programming and who wants to carry out minor modifications to the button is faced with a relatively small piece of code and so can be quite happy working out which part of the Lisp expression to change. A number of our research staff have started to use buttons in this way. A specific example was one in which one of our researchers who is not a Lisp programmer observed some of us exploring new buttons which allowed us to open two-way audio-visual connections between members of EuroPARC staff. Despite warnings that some of the software on which these buttons relied was unstable and would be superceded in an incompatible way, he persuaded us to email the buttons to him. Within a short time he had modified some of the internal Lisp code to make connections relevant for his own use. He gave these buttons to a few other people as well, allowing them to explore the use of our A/V infrastructure sooner than would otherwise have been possible. We were particularly impressed by this experience as it was one we had not engineered in any way - indeed we had tried to discourage it if anything. It helps to demonstrate that the approach we have taken with Buttons is useful for a wide range of users with very different expertise.

Building Blocks

An experienced Lisp programmer can place any arbitrary piece of Lisp inside a Button, and so the range of things which can be done with Buttons is incredibly flexible. In practice, however, we need to encourage a constrained approach to creating new buttons. One obvious example is that we want a consistent interface style for users to interact with buttons. We provide a set of user-interface building blocks that can be used when the button action causes some interaction with the user. For example, they provide information to the user, ask for yes/no responses, ask for string input and so on. Domain-dependent building blocks provide high level functionality to support common applications. For example, there are text editing functions which give direct control over various text window properties such as labels, location on the screen etc. Other building blocks extend the range of possible applications within the Lisp environment as a whole, such as the functions to communicate with our A/V server, giving control over A/V connections from buttons. These building blocks are similar in concept to the construction kit approach advocated by Fischer and Lemke [10], or to the "programmer's interface" which provides high-level access for the programmer wishing to tailor NoteCards [24].

THE BUTTONS USER'S VIEW

Part of the handyman's role was to observe how the use of Buttons evolved, and to interview the users about their perceptions of and reactions to Buttons. Output from this has already been used to illustrate some of the specific tailoring techniques. The aim of this section is to give a more general overview of how Buttons were perceived and used. One particularly striking observation was a change in

attitude towards Buttons. Early on, users talked about Buttons as being "not my personal buttons" or being "sewn to the screen" (i.e. not under personal control). Later, we started getting quotes such as "I don't know what I'd do without my Buttons" or "Buttons are my friends, always there...". Note the use of "my" in these quotes. Buttons became perceived to be very personal - indeed when one user's system was accidently modified by a workstation "gremlin", she compared it to her house being ransacked by burglars. The tailorability aspects of Buttons were also appreciated by users. For example, during an interview one user reflected that she "used to think you had to be a programmer to make buttons". Buttons seemed to serve a stronger role than we expected in communicating between people with different programming skills. They are regularly distributed by email, but perhaps more interesting, it is not uncommon for someone to request "a button to do X", where "X" may be something for which a button is quite definitely not the solution. We suspect that some of these requests would not be made at all if it were not possible to articulate a putative solution in terms of something concrete and comprehensible such as a button.

Different people adopt different strategies for organising their buttons on the screen. Figure 2 shows a typical screen from one of our users. This user is particularly tidy in the way she sets up her screen. Note how she has adopted both graphic image and spatial location to identify different types of buttons which are used in different contexts. Such strategies have functional as well as aesthetic roles - another user commented "I like my buttons to look different so I don't have to read them". One comment which people who are not familiar with buttons often make is that the screen can become very cluttered. So far this has caused remarkably few problems in practice. Users tend to regard their Buttons as "screen furnishings" or "wallpaper" - very much part of their environment. The user of the screen in figure 2 has the screen laid out so that for most tasks she would only use one group of buttons. She simply organises her working windows over groups not currently required, leaving easy access to the buttons she needs for her current task.

REALISING THE TAILORABILITY PROMISE

A major reason for our success in enabling non-programming users to tailor their own workstation environment is that we have produced

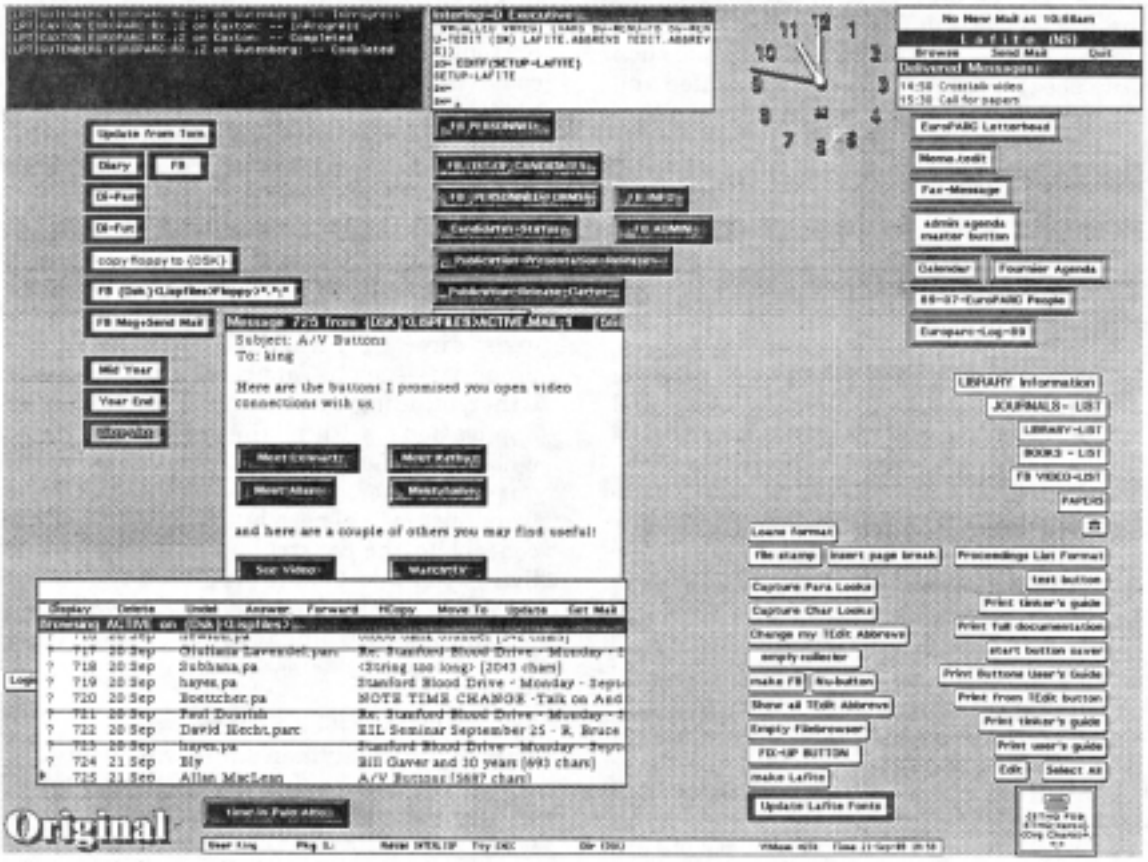

Figure 2. A Buttons user's screen, showing a variety of buttons. Buttons can exist either on the desktop or in documents. This snapshot shows the user reading an electronic mail message with buttons in it. If she wants to keep them available they can be easily transferred from the message to the desktop.

an architecture which supports a large number of tailoring techniques. Figure 3 relates these techniques to the skill required to use them. This characterisation suggests that we have succeeded in producing a much less rugged landscape than the mountainous one with which we started. Some tailoring techniques are therefore accessible to workers of any level of expertise. As they learn new techniques, the increment in skill to learn yet another one is always relatively small, so there is little barrier to learning new mechanisms.

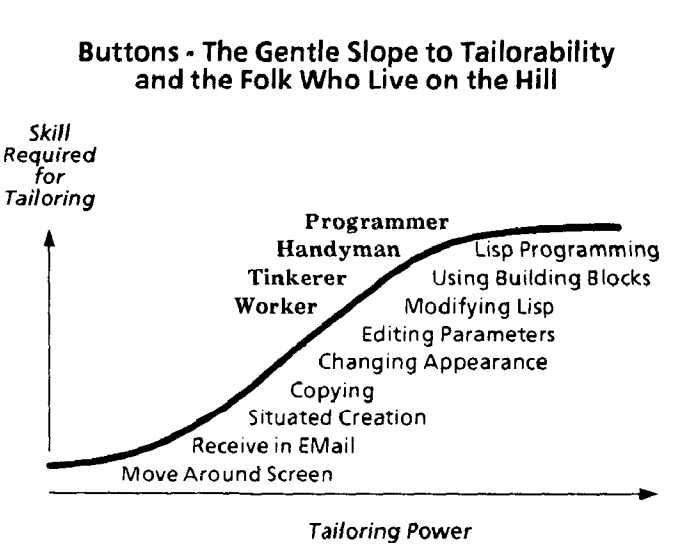

Figure 3. Tailoring techniques in Buttons. Roughly plotted as a function of skill required and tailoring power. There is not a simple monotonic relation between the different techniques, but some idea of the relationship between them is illustrated. Each category of "tailor" has access to techniques lower down the hill, but has to put in some effort to learn new techniques and climb the hill.

Starting at the bottom of the hill, the least skillful techniques rely on users simply having the control to *move* buttons where they want them on the screen. A remarkable amount of tailoring can be done by relying on buttons produced by other people. Buttons can be kept in documents and can be easily passed around by *email*, thus they are a tool for the user community to augment the Xerox Lisp environment by combining individual innovation and by sharing improvements with others. The fact that small-grain improvements can easily "diffuse" throughout the user community is a powerful principle for supporting user-driven evolution of systems. *Situated creation* allows the user to create a button by capitalising on a system state which has been created by more tedious methods. Buttons can be easily *copied*. It may be useful to have multiple copies of a given button available for easy access from different parts of the screen, or for use in different Rooms [14]. More interesting though, is specialising a copy to change it in some way. This permits new buttons to be created without having to worry about all the details required to produce a button from scratch. There are various ways in which a button can be modified, some very lightweight. Each button contains menu options to change a number of attributes of its *appearance*. Many buttons have *specialised parameters* which allow important attributes of their behaviour to be changed. It is possible to get direct access to the Lisp code inside a button if one desires. Minor modifications are easily done by someone with limited programming skills, either by *modifying the existing code*, or by combining high-level *building blocks*. More experienced programmers can insert any *arbitrary Lisp* into a button.

With this range of techniques, we now have our worker off of the plains, living half way up the hill, able to tailor by a good number of methods lower down the hill. There are still inevitable differences in what people with different skill levels can do, but there are no longer insurmountable barriers between these people. Most importantly, since the workers have now become familiar with the kinds of changes which can be made, they are in a much better position to envisage changes and communicate their ideas to tinkerers, handymen or programmers when they are not able to make the changes themselves. This is important in building a single culture within which we can take full advantage of different people's skills for the benefit of the community as a whole.

In summary, we must take a broad perspective if we are to achieve the promise of genuinely user-tailorable systems. A range of techniques is required to make as much tailoring as possible feel no different from making use of the system, and to allow migration between different techniques. We must develop a tailoring culture which encourages individuals to think in terms of improving their computational environment by tailoring it, and encourages members of user communities to help each other by sharing insights and expertise. We must grow a design culture which has tailorability as a major goal of system design, and which works closely with users to achieve it. Helping users to "think design" to contribute to this process is also an important component of the mindset for users to tailor their own systems, so supporting the tailoring culture. The Buttons project described here has helped us to better understand these issues and ways of tackling them. Some of the success of the project can be measured by the fact that our administrative staff are now regular and committed users of buttons, as are many of our research staff. Our experience with Buttons suggests that we *can* realise the promise of user tailorable systems, and our Buttons system itself shows some techniques which can be used to achieve this.

ACKNOWLEDGEMENTS: Many of the insights reported in this paper would not have been possible without the willing participation of the EuroPARC admin staff (Kathleen Kavanagh, Christine King, Sian Wicklow, Penny Wisdom). Austin Henderson wrote the original version of Buttons software and helped us understand the potential which led to the project reported here. Thomas Green and Moira Minoughan gave valuable contributions to many discussions of using buttons for tailorability. Richard Southall assisted with the design of the Buttons graphics. Bob Anderson, Alan Borning, Bill Gaver, Jonathan Grudin and Richard Young provided comments on previous versions of this paper which helped improve the final version.

REFERENCES

1. Ashton-Tate. Framework II Reference Manual. 1986.

2. Bødker, S., Ehn, P., Kammersgaard, J., Kyng, M. and Sundblad, Yngve. A UTOPIAN experience: On the design of powerful computer based tools for skilled graphic workers. In Bjerknes, G., Ehn, P. and Kyng, M. (Eds) *Computers and Democracy - A Scandinavian Challenge*, Avebury, Aldershot, England, 1987.

3. Bobrow, D.G. and Stefik, M. The LOOPS Manual. Tech Rep. KB-VLSI-81-13. Xerox Palo Alto Research Center, 1981.

4. Borning, A. Classes versus prototypes in object-oriented languages. In *Proc ACM/IEEE Fall Joint Computer Conference*, (Dallas, Nov 1986), 36-40, 1986.

5. Borning, A. and O'Shea, T. An empirically and aesthetically motivated simplification of Smalltalk-80. Proceedings of the European Conference on Object-Oriented Programming, (Paris, June 1987), 155-165, 1987.

6. Carter, K. Two Conceptions of Designing. *IRIS Conference on "Creativity in System Development"*. 1989.

7. Carter, K. Methods for designing with users. *PICT workshop on Social perspectives on Software*. UMIST, Manchester, July 19-20 1989.

8. DiSessa, A. A principled design for an integrated computational environment. *Human-Computer Interaction*, 1, 1-47, 1985.

9. Ehn, P. and Kyng, M. The collective resource approach to systems design. In Bjerknes, G., Ehn, P. and Kyng, M. (Eds) *Computers and Democracy - A Scandinavian Challenge*, Avebury, Aldershot, England, 1987.

10. Fischer, G. and Lemke, A. Construction kits and design environments: Steps toward human problem-domain communication. *Human-Computer Interaction*, 3, 179-222, 1988.

11. Goldberg, A. and Robson, D. *Smalltalk-80, the language and its implementation*. Addison-Wesley Publishing Co., 1983.

12. Goodman, D. *The complete HyperCard handbook*. Bantam Books, New York. 1987.

13. Grudin, J. and Barnard, P. When does an abbreviation become a word? And related questions. In *Proc CHI'85* (San Francisco), ACM, New York, 121-125, 1985.

14. Henderson, A. and Card, S. Rooms: The use of multiple virtual workspaces to reduce space contention in a window based graphical user interface. ACM Transactions on Graphics.

15. Lieberman, H. Using prototypical objects to implement shared behavior in object-oriented systems. In *Proceedings of OOPSLA'87*, ACM Press New York, 214-223, 1987.

16. Lövstrand, L. *Buttons: An object-oriented architecture to support tailorability*. In preparation.

17. MacLean, A. Human factors and the design of user interface management systems: EASIE as a case study. *Information and Software Technology*, 29, 192-201, 1987.

18. MacLean, A., Barnard, P. and Wilson, M. Rapid prototyping of dialogue for human factors research: The EASIE approach. In Harrison, M and Monk, A. (Eds.) *People and Computers: Designing for Usability*. CUP, Cambridge, 180-195, 1986.

19. MacLean, A., Young, R. and Moran, T. Design Rationale: The argument behind the artifact. In *Proc. CHI'89*, Austin, Texas, April 30-May 4, ACM, New York, 247-252, 1989.

20. Myers, B. Visual programming, programming by example, and program visualisation; a taxonomy. In *Proc CHI'86* (Boston, MA. April 13-16), 59-66, 1986.

21. Rich, C. and Waters, R. Automatic programming: Myths and Prospects. *IEEE Computer*, August, 42-51, 1988.

22. Ritchie, R. and Weir, G. Menu-based extensions to GNU Emacs. In Sutcliffe, A. and Macaulay, L. (Eds.) *People and Computers V*. CUP, Cambridge, 245-260, 1989.

23. Stallman, R. EMACS, the extensible, customizable, self-documenting display editor. *Proc ACM SIGPLAN SIGOA Symposoium on Text Manipulation*. Portland, Oregon, June, 1981.

24. Trigg, R, Moran, T. and Halasz, F. Adaptability and Tailorability in NoteCards. In Bullinger, H.-J. and Shackel, B. (Eds.) *Proceedings of INTERACT'87*, London, 723-728, 1987.

25. Xerox Corporation. *ViewPoint CUSP Button Reference*. ViewPoint Series Reference Library. 1988.