

# The 100% Solution: Designing for Exceptions

Austin Henderson

Pitney Bowes

35 Waterview Dr.

Shelton, CT, 06484

USA

austin.henderson@pb.com

## ABSTRACT

We deploy our applications for use in worlds that change, and in places not anticipated by their designers. As a result, our applications do not - and, in principle, cannot - anticipate all the circumstances in which they are used. Yet the user must cope with every single circumstance that they confront, and they must find some way to bring the application to bear on those situations. In this paper I explore some thoughts and challenges concerning the resulting inevitable misalignment between the user's needs and the application's capabilities. I explore three kinds of solutions: *fixes* (changing the application), *work-arounds* (going "outside" the application), and *appropriations* (going to school on other cases). The resulting socio-technical systems (humans and applications working together) can address circumstances unanticipated by the applications. I argue that the implication for designers is that we must challenge ourselves to design applications that support, not only the circumstances that we anticipate, but also the systems that people adopt for dealing with circumstances that we have *not* anticipated. I then argue that such safety-net mechanisms for dealing with unanticipated circumstances can also be used to address unusual circumstances - those that, although anticipated, are not worth devoting application development resources to. This could lead to simplifying applications by focusing them on usual cases, leaving everything else to the socio-technical safety net. I close by speculating on reasons why we have not yet focused on this issue or built such systems.

## Keywords

System design, application design, unanticipated circumstances, fixes, work-arounds, appropriations, 100% solution.

## INTRODUCTION

There is a profound structural problem in our practices of design of applications for meeting user needs. Simply put, we cannot possibly anticipate everything the user will need when real applications hit real worlds.

So users will have to do something that goes beyond what we designers have anticipated. Indeed, users of real applications in real-world circumstances become skilled in developing solutions for unanticipated needs arising from unanticipated circumstances.

Such skills in dealing with the unanticipated are essential competencies for users of applications in real world situations. Indeed, as part of routine work, users have to go beyond what designers have anticipated; indeed, going beyond is routine work.

However, I want to argue that we as designers are not addressing this important aspect of routine user work. Our applications do not anticipate that there will be unanticipated circumstances and resulting needs that are not, but must be, met. Our applications do not support the work of going beyond the anticipated.

I present this situation, explore three kinds of solutions that users adopt in order to go beyond what designers have anticipated, argue that designers should be anticipating these user solutions, and suggest that we should be improving our designs to support users in going beyond those designs.

I then argue that such safety-net mechanisms for dealing with unanticipated circumstances can also be used to address circumstances that, although anticipated, are not worth devoting application development resources to. Indeed, it is even possible to simplify applications by focusing them on usual cases, leaving everything else to the socio-technical safety net.

I close with a final question: What systematic perspectives leave unaddressed the need for users to go beyond what designers have anticipated? I do not think any of these solutions have to be very difficult, although I envision advanced solutions that may be very challenging indeed. However, it is surprising to me that designing for going beyond the anticipated is a problem that has not been addressed by those of us participating in developing applications that support user work. Why is this?

## THE NEED: SERVING THE USER IN ALL CASES

### Design: implementation and deployment

We design for particular people doing particular work. I will use the term "designer" for all of this, to indicate that this is activity that happens before the resulting

work is used in the real world. That is, we design, then users use.

In the best of cases, we focus on understanding the users, their work and their needs/wants/desires well. We develop ideas that will support the work and even take it to the next level by rethinking the work and reframing the applications together to create new ways of working. We test these ideas with users through prototyping at many degrees of functionality, completeness, polish, and deployment.

As a result we create offerings that we provide to users to deploy in creating and supporting their work, including tools, suites, and services composed of technological and human resources. I will use the term “application” for all these, as a way of indicating, from the user’s point of view, that these are the things that the users apply (deploy and employ) in their work.

We follow these applications to the field, and engage users as they deploy them and employ them in their work. We learn from what we see and feed it forward into subsequent designs.

### Use

However, despite all this hard work, as designers we cannot generally control the circumstances that users will encounter when using our applications. Our applications may be deployed by users in circumstances that we, the designers, have not anticipated. My Connecticut, USA-based company opens an office in New Jersey, and suddenly the computation of sales tax is different; or in California, and suddenly time zones matter; or in France, and suddenly language and VAT matter; and so on, ad infinitum. Or think of the payroll program designed for engineering in Connecticut that in expanded deployment has to deal with the working practices in sales and finance, or overtime in California, or VAT in Europe.

Many of these are difficulties that we as designers may be able to anticipate, and that prototyping and trialling may uncover. However, some of these happen in the everyday working arrangements with new markets, new customers, and new users. And many of these emerge *in use* - in the midst of working, as part of everyday activity.

Moreover, these unanticipated circumstances are not always the result of the activity of the company or the users of the application. More often, the world changes around the work that the user is doing, and around the deployed applications, producing different work and different needs, and demanding different things of our applications.

For example, every day, some country in the world changes the rules for shipping things: they change how big or small shipments may be; or the rates; or what is considered dangerous, and what you have to do to package things for shipping and to document that packaging; or what is considered “controlled” or

“sensitive” or “strategic” and what you have to do about it; or – well the list is open-ended and mind-boggling. Although some of these realities might be anticipated by carefully thinking ahead, many more are introduced by countries (or companies, or clubs) in the midst of their activity as their worlds and circumstances change. It is a full-time job to keep up with these changes, and the mailrooms of Pitney Bowes Management Systems must do that in order to serve our customers.

### Gap between capabilities and needs

As a result, our applications do not – and, in principle, cannot – anticipate all the needs to which they will be put. As designers, we can’t possibly get it entirely right, if for no other reason than we cannot be there with each user in each situation as the world changes.

As a result our applications can fall far behind the circumstances in which they are being used.

### Handling 100% of the cases

Users face many cases every day. The difficulty that users face is that, in order to stay in business, they have to handle 100% of those cases. Of course, many of them will fit within what has been anticipated by the applications that they are using. Those can be handled directly.

However, *all of the cases* must be handled in some way. This is the central reality of the use of applications – that they are part of a railroad that is running, and the trains must be kept running. It is not acceptable to walk away from the cases presented. As they say at the post, “The mail must go through.”

Possibly a user can reject some of the cases, giving up the opportunity to do the business that those cases represent. However, passing up opportunities is not good for business, and many of the applications are in the hands of people doing the “back office” supporting work, where rejecting cases is not considered good form.

Or maybe users can delay some of the cases until the applications can be changed, provided they can slow down the rest of the systems within which they are embedded.

But whatever is done, *the user has to do something*. Even if you want to admit you cannot respond and therefore forego doing the business, someone will want to know that it happened and why you passed. If you want to delay, means of delaying must be in place. Most probably, all cases must be documented.

This is where the rubber hits the road. Confronted with a case where the circumstances have not been anticipated by the applications in use, *what is a user to do?*

### A few non-solutions

Just so that we are clear about the problem being addressed here, let me dispense with a few well-known solutions to nearby (“strong attractor”) problems that

are often confused with being solutions to this problem.

#### *Better analysis*

Proposal: Designers could do a better job of anticipating everything that may occur across all deployments and all possible futures, including better ways of working with users to understand needs, better prototyping of results with users to discover unarticulated needs, and better documentation to convey the needs to designers.

This is a non-solution because real worlds – those not artificially constrained by rules – are rich, and richer than we can imagine. They will always outrun us. I'm all in favour of doing the best job you can in the area, but don't expect to be completely successful.

#### *Better design*

Proposal: Make a perfect design, so that users will not have to face the difficulty proposed by unanticipated situations.

There is pressure for designers to ignore certain realities about the practicalities of real deployments. By ignoring these realities, we miss the opportunity to design for those realities. As a result, we miss the chance to really serve our users.

For example, consider user interface designers who hope that their perfect user interface will prevent all user errors, and so avoid having to spend any time designing for helping the user when they do make mistakes.

The very idea that you might be able to be completely successful – that you might do a perfect “user needs analysis” – can get in the way of addressing the problem we are talking about here. If you deny the fact that you will get it wrong, you can avoid thinking about and supporting the poor user when they confront something you have not anticipated.

Better design is a non-solution because if there is even the slightest chance that you will fail, you must still have the means in place for the user to deal with the unanticipated.

#### *Better feedback to designers*

Proposal: Provide better ways for users to communicate difficulties back to designers, and better ways for designers to quickly modify the applications.

This is a non-solution because it is not very realistic: in the real world, the designers are usually busy doing other things when the unanticipated case turns up. The fast feedback will go right onto the long list of things to be considered, and only if it is a case that occurs enough to be considered worth the effort, and only when and if it gets to the top of the list, and only resources get assigned to it, it may get done.

### **SOLUTIONS: THREE STORIES**

So what do users actually do when confronted with an unanticipated set of circumstance? Here are three

stories from my experience that have been powerful drivers of my thinking about this question.

#### **Case 1: Managing contracts with negative balances.**

Consider John, a contract manager, at Acme Corp, a manufacturer in Pointe Claire, Quebec, in the summer of 1965, managing the contracts that Acme had with its suppliers. Every day, shipments are received at the Acme loading dock, are recorded, and the records are sent to John. John identifies the relevant contracts, and reduces the amount outstanding accordingly.

Now consider John doing his job with support from a computer-based system. In fact, along came Austin Henderson, representing the IT department (we were called “The Systems Department” then), with the news that IT was building a new computer-based system, and John himself has to consider this impending possibility. Austin wants to know whether the outstanding balance on a contract that John manages is ever likely to “go negative.” That is, will a supplier ever ship more than Acme contracted for? (Let's assume a perfect system, and ignore the possibility that Acme's records might be wrong.) John gives five reasons, including allowing for breakage, and quality control, and a delay in signing a new contract so continuing to (over)ship under the current one. Austin had suspected that such might be the case, and promises John that the new contract management software will be designed to handle all five of these cases. John stops Austin, and requests that the software not handle the cases, but rather *be* built so that John can handle the cases. Instead of the software being aware of breakage or quality control or delay in signing, the program should allow John to make adjustments to bring Acme's records in line with the world: to adjust a contract balance for whatever reason, and to offset balances on one contract against those on another.

John does not see the new contract management system modelling the world; he sees himself using the system to model the world. He does not see the truth as being in the system; rather the truth is in John's understanding of the world. Through John's work, the system is made to reflect as much of the world as is necessary to help John get his job done.

#### **Case 2: Address for a copier on an ocean-going barge.**

Consider Susan, an order-entry clerk, at Xerox Corporation, in the summer of 1978 taking telephone orders for supplies (paper and toner) for copiers. On one call, the customer cannot supply a shipping address, because the copier is on an ocean-going barge and the address is a function of when the shipment will be made. Luckily Susan was working on paper, and resolved the problem by filling in the address field with a telephone number and “Call Bob”.

Now consider Susan doing her job with support from a computer-based system. Such a system might well

reject Susan's instructions in the address field because they are clearly not an address as the designers of the system imagined it. Further there is rarely any way to tell the system that you, the user, know better (because you know more) than the designers. To do so would require that the ramifications of the exceptional case be worked out with human assistance, and modern computer systems aren't designed to ask for help.

### Case 3: Canadian co-ops.

Consider Mark, a manager in research at Pitney Bowes, in the summer of 2004, trying to hire Bill, a co-op student from Canada. For Pitney Bowes to pay Bill, Bill must have an American Social Security number (SSN). To get the SSN, Bill must be in the United States. To enter the United States, Bill must already be on the payroll of an American company. Everyone knows the problem, and has worked out a fix: Mark hires Bill using a dummy SSN (all 9's). Pitney Bowes issues paperwork, Bill travels and begins to work, and Pitney pays Bill against that SSN. When Bill gets the real SSN, all the records are adjusted to reflect the fact.

Now consider Mark doing his job with support from a computer-based system. Such a system to support Human Resources was put in place at Pitney Bowes in 2005. The new HR system would not permit dummy SSNs. Mark spent over 90 hours trying to work around the system, but all workarounds had unacceptable implications, for example, incorrect withholding for taxes. Mark eventually discovered that fixing the HR system to take dummy SSNs had been anticipated by the software supplier, was possible, had been considered when the system was built; but the number of cases was small and did not warrant the cost of building the application that way. And after deployment, it was not attractive to fix it. In the end, the solution involved Mark using his personal funds to pay Bill for a while!

### SOLUTIONS

As intellectuals, we have a responsibility to understand this problem. As researchers and developers, and, indeed, as users, we have a responsibility to suggest and develop solutions. I will discuss three kinds of solutions to this problem. I offer this here, not as a complete or even well thought-out typology, but as the beginning of an exploration which I encourage others to join.

#### Fixes (change the application)

Proposal: Let users make fixes to their applications to fit the needs of the situations in which they find themselves.

The idea is that the user can pause in the case in hand, adjust the way the application works, and proceed with the work (reflection on the activity, and then re-engagement). For example, spreadsheets are designed so that users can adjust them as the world they are addressing, or the analysis they want to do, changes.

In case 1 (above), John the contracts manager could be understood to have been asking for an application that

was a construction set that would allow him to adjust his contract management applications to match his needs in managing his contracts.

These solutions require that we make our applications adjustable in the hands of users. All manner of means for doing this are available: preferences, tailorability, platforms, construction sets, domain-specific languages, end-user programming. There is also finding the person down the hall who knows how to make these changes, or even pushing it back to the supplier or producer. These all have their attendant capabilities and problems; they need not be explored here.

For our purposes here, let us simply make the assumption that the users have the capacity to make such fixes. That leaves the central question of what support we do or do not provide to users when they make fixes. A few might be:

#### *Pause and resume*

When the user shifts into tool-adjusting mode, they need support for pausing in the case in hand, and, when they are finished adjusting the application, resuming the case again *using the changed application*. If the user needs to pause the application and later resume it, there has to be a way to save the state of the application, and then a way to reinstate it when the user is ready to resume. Means of storing, monitoring and managing these pieces of unfinished business are also needed.

#### *Collaboration*

When the user wants to discuss the application with others, they need to be able to share its state with them. Also needed are ways of looking over each other's shoulders, ways of pointing to things, ways of providing pieces of stuff that will help someone else do something you have already done.

#### *Record keeping*

When the user does non-standard things, they need to record what they have done. This will help them remember what happened. What is often more important is that record-keeping can also address the regulatory requirements of our collaborators, our managers, our partners, and our governments.

#### *Ontological drift*

Note that as changes are being made, the conceptual structure of the application may well be changing too. This puts heavy loads on record-keeping systems. They must address the ontological drift of reframing, and they are often not set up to do so. Neither are humans, as an earlier state of understanding is often difficult to remember.

Generally: As designers of applications we should challenge ourselves to create technology that can help people evolve their applications to track the changing world.

**Work-arounds (get beyond the application)**

But even with such "evolvable" applications, I argue that we will inevitably fail to address everything. Indeed we may not even want to. It is often too much work to change the applications, and given a single case, the user may not have any sense of whether this will be a one-shot or a recurring circumstance. Therefore users may not choose to evolve the system in the heat of daily activity.

This leads the user to a second kind of approach: do a "work-around".

Proposal: Leave the gap between what the application can do and what is needed to be *handled by the user outside the application*.

For example, forms have margins. The margin is the institutionalized place for all the stuff that is in the gap, all the stuff that does not fit within the structure of the form.

In case 2 (above), Susan, the order-entry clerk, used the capability of paper to provide material that gave new meaning to Xerox's idea of address. The application "expected" an address. Susan's material went beyond the application, depending on the capabilities of people to get the information they needed when they needed it.

These solutions require that the user and everyone else, understand that the system providing the solution is bigger than the application that supports it. The system is a socio-technical system that uses people as well as machines to run the railroad. Specifically, in order for a name and telephone number to do the job of an address, everyone had to understand that there was an implied expectation that whoever needed the address would call Bob.

Not only do these solutions require looking beyond the application to find a solution in the larger socio-technical system, they require that the socio-technical system be made to work. Moving into the realm of social systems opens a huge range of capabilities and concerns in making systems that work. Among these capabilities are the human capacities for dealing with ambiguity, uncertainty, perspective taking and ontological reframing. They also include the difficulties of human error, human perspective taking, and the challenges of scaling. To take advantage of social solutions, whole new realms of system-building need to be explored and developed.

Here again, let us simply make the assumption that the users have the capacity to invent and build such systems, and to dream up solutions of the kind suggested by the examples. That leaves the central question of what support we do or do not provide to users when they use work-arounds. A few might be:

*Working outside the application*

When the user shifts attention into socio-technical system creation, they need support for materials that are outside the application yet are still regarded by the system as being inside the system. A paper note that

augments a case in an application must somehow be kept associated with that case; ways of pointing/referring from inside the application to the note outside are needed.

Such physical-informational systems are not well supported by current IT thinking or mechanisms. There is much that can be done by marrying the wisdom of older physical practices with modern information-based ones.

*Application carrying beyond-application material*

When the user shifts attention into the creation of socio-technical systems, they need support for adding material to the application that the application carries, but that the application does not try to process. The application must know not to try to find the zip-code in the instructions to call Bob; such material must stand out. The user must be able to augment not only the information in expected fields, but also the structure of the application itself.

In *Control Through Communications*, JoAnne Yates describes the achievement of coherence in management during the later 19<sup>th</sup> century as railroads, conglomerates and multi-location stores invented distributed management. Through many techniques central management imposed regular work processes on distributed teams. One such technique was printed forms, meant to dictate a regularity across locations and cases. Of specific relevance to my argument here is the invention of the *margins on their forms*. The margin was the place to put everything that they anticipated that they would not be able to anticipate, everything that did not fit into that regularity of the form.

As we have moved into the electronic age, we have failed to carry this critical lesson forward. Our electronic forms are surprisingly devoid of margins. It is time we by added something like "margins" to our electronic forms.

*Finding human processors*

When the application needs to do some processing on information that has been augmented with non-application material, it must be able to find a human to help with that work. For the socio-technical system to work, the human processing must be in place. Inevitably though, unlike technical applications, people may well not be expected to be available immediately when that need arises. Therefore cases may have to be paused (see Fixes/pause and resume, above) and queued for people. People need to be able to find these queued cases, resolve the needs, and let the case resume.

*Recognizing regularity*

When people notice a regularity within the exceptions they are handling, they need to be able to capture it as part of the application. This they can achieve either as a fix (see above), or, because the pressure of time is relieved by the capacity of work-arounds, as part of ongoing application maintenance or development.

**APPROPRIATIONS (going to school on other cases)**

Exceptions are rarely understood, nor adjustments made, in isolation.

Proposal: Consider how others have handled the situation.

In school this is called “cheating,” in academia “plagiarism,” but in the real world it is called collaboration. It is standing on the shoulders of giants. The case in hand will remind you of something you’ve done before, or something you heard your colleague talk about at lunch, or of something that we are (or should be) considering for future growth.

For example, you copy and modify completed forms from similar cases, you get John’s code and modify it, you grab HTML from sources of pages that you like, and you use last month’s spreadsheet as this month’s template.

In case 3 (above), Mark hired Canadian co-ops again this year, using last year’s work-around, with certain exceptions suggested (with some force) by the finance department.

These solutions are based on being able to spot the similarity of the case in hand to other cases. This in turn reframes the thinking: instead of just getting this case done (by, and for, itself) we now see this case as part of the whole constellation of cases past, present and yet to come.

Considering the whole constellation of cases takes us beyond the record-keeping associated with either physical records or electronic documents. It requires that we address the changing structure of regularity (pattern) and particularity (exception) as central to the content of the cases. Record-keeping now has to address the fact that the content of the cases is held in human minds as well as in computational memories. Further, present cases are viewed as works in progress, and future cases are only imagined. There is much existing practice in this area that is available for mining.

Here again, let us simply make the assumption that the users have the capacity to store and manage past cases, and to use solutions that the cases suggest. That leaves the central question of what support we do or do not provide to users when they use past and future records. A few might be:

*Working outside the current case*

Under this view, work on a single case requires getting your hands – both physical and informational – around other cases and being able to explore them as an integral part of the current case. What fixes and work-arounds have people implemented around this field (e.g., how has Xerox thought about moving copiers and dynamic addresses)? What work-arounds have been used, and how well did they work?

*Working with, and through, humans*

Finding cases is one thing, finding the people who made those cases work is another. A central idea in

modern knowledge management is that documents take you to people who in turn take you to documents. Neither is effective on its own. The work that people do bridges between people and knowledge. Similarly, it will be necessary to find and engage the people involved in other cases in order to understand those cases and learn from similarities between them.

*Achieving coherence*

If everyone worked the current case blind to everyone else, there would be little likelihood that the system would remain coherent. By watching others, coherence can be aided. But better yet is being able to explore possibilities and impacts. One imagines simulating cases, running them through to completion while at the same time holding off commitment until the larger picture has emerged. Similarly, holding cases until issues have been cleared up would help solutions developed in different circumstances to inform each other. Cases would be gathered, used to collectively inform an encompassing solution, and then individually reworked to reflect this over-arching solution.

*Support for negotiation*

Finally, the reality of systems that permit people to develop solutions on their own is that different people will be driven by different values. Rather than forcing everyone to agree, I believe that the best systems and the best local solutions will result when different perspectives and values can negotiate with each other for the best good of all. Systems need to provide the support for such negotiation, capturing the rationales for positions argued and solutions implemented as part of the cases themselves. These systems need also to provide for re-entering those discussions when similar cases arise in the future, in order to re-assess the arguments in those new circumstances.

Generally: As designers of systems, we should challenge ourselves to address the users’ work of negotiating multiple perspectives, and to create systems that support the production of alignments that achieve not only responsiveness, but also coherence.

**SEEING THE UNUSUAL AS EXCEPTIONAL**

Over time, applications tend to become larger and larger, with ever-increasing functionality, because with time they grow to address more and more cases. But some of these cases may turn out to happen only rarely. Other cases may be things we do not want to generally encourage, while wanting to allow them in special circumstances. Still others may be regularities we do not want to add to the application because implementing it would be beyond the available resources.

Once we have reframed our thinking about how to handle 100% of the cases, and have provided the exception mechanisms that allow humans to handle the unusual and exceptional cases (the “safety nets”), we can consider a further, previously unthinkable, change:

we can make the decision to treat these anticipated, but unusual, cases as exceptions.

Thus, we recast the distinction of what is in the technical domain and what is in the human as, not between anticipated and unanticipated, but rather between usual and unusual, with the unanticipated included in the unusual.

As a result we end up with a system with two interacting complementary channels. Central to this two-channel system is the capacity for the user to choose those cases that we will handle as usual (the regular), and those that we will handle as unusual (the exceptions, the unanticipated, the rare, the difficult, the undesirable, and everything else). There is much more to be said about controlling these choices, and how that control is managed.

#### *Evolving regularities*

Such a two-channel system has a number of advantages when it comes to responding to a changing world.

*Adding new regularities:* as new regularities are encountered in practice (and handled as exceptions), we may choose to add them to the application.

*Changing regularities:* as changed regularities are encountered in practice, they may be added to the application.

*Removing regularities:* if regularities become rare in practice, they may be removed from the application.

#### *Building less, doing more*

Removing regularities presents a further significant advantage: an application can be adjusted – often simplified – to address only the cases that are usual. This holds the possibility that less technical work (including systems analysis and application building) can support a socio-technical system that can address 100% of a larger range of cases.

Building less to support more often requires a rebalancing of the role of design and use. It will require unlearning our IT instincts to do thorough and complete design. We will have to learn to do a much “poorer” quality of design. This will not be easy.

#### *Evolving blank forms*

The extreme case of this shift to simplicity is the idea of “evolving blank forms” (cf., the exploratory programming practice of “debugging an empty program into an application.”): Treat all cases as exceptions, and build a socio-technical system having only the functionality that permits people to handle exceptions. Deploy it, use it, and as regularities emerge (probably the most common ones will emerge first), they can be developed into simple technical systems.

The shape of such an evolving system can be made to reflect the work at hand. Further, and usually unthinkable in the IT mindset, when common things become uncommon, the application can be devolved (un-developed) reducing its complexity in those areas, and enabling complexity to grow more easily in others as needed.

Generally: As designers of applications, knowing that users will have to go beyond the applications we design, we should challenge ourselves to address the larger socio-technical system within which those user-created work-arounds exist, and create socio-technical systems to support them.

### **A PROBLEM WITH PERSPECTIVE**

In this paper, I argue that we need a change of perspective. Instead of our applications defining the solutions, we should let our applications be the scaffolding upon which, and around which, we as users develop solutions. We should provide the means for users to fix those applications, build solutions that go beyond the application but are within the system (regarded as socio-technical inter-working of machines and people), and make full use of other similar cases past, present and future.

It is high time we changed our perspective and built applications to match the evident need. Why is it then that we have not done so sooner? Some possible answers include these:

First, although activity in the world is nuanced, multi-perspective, changing and socially created by the people participating, the computer-based tools from which applications are built are crisp, single-perspective, and statically designed in advance of use. As a result, current computer-based activity-support applications in truth support only simplistic idealizations of human activity.

Second, although some of the mechanisms suggested here would be quite easy to create technically, the advanced capabilities present significant technological challenge.

Third, the thought of users moving freely in whatever directions seem appropriate to them may well scare those who feel need the need for strong control of worker behaviour. The development of the social and organizational aspects of even the simplest of the socio-technical systems suggested here may well offer severe challenges to management and IT academics and professionals.

And finally, and most apropos for designers, we will need to give up on our goals of doing complete and beautiful analyses and designs for applications, leaving the continuing work of meeting users’ needs to the users themselves. Our sense of self-worth will have to move instead to doing adequate designs as starting points that users will evolve in use, and, most importantly, providing the means and mechanisms for support that the users need as that evolution proceeds.

#### ACKNOWLEDGMENTS

To Jed Harris, for years of discussion on pliant computing, and to many friends including Patricia Sachs Chess, Michael Davis-Burchat, Jim Euchner, Judith Gregory, Lynne Henderson, and Ben Singer for more recent discussions.

#### REFERENCES

- Henderson, A. and Harris, J., (1999). A Better Mythology for System Design in *Proceedings of CHI '99* (Pittsburgh PA, May 1999), ACM Press, 138-145.
- Henderson, A. and Harris, J., (2000). Beyond Formalism: The Art and Science of Designing Pliant Systems in *Software Design and Usability*, ed. Klaus Kaasgaard, Copenhagen Business School Press, Copenhagen. (2000).
- Yates, JoAnne, *Control through Communications: The Rise of System in American Management*, The Johns Hopkins University Press, Baltimore, Maryland. (1989)