Solving scaling problems with the modern GUI (July 2002)

Peter M. BAGNALL

Abstract—**The GUI has changed very little since its invention by Xerox in the mid 70. GUIs are no longer able to present the full range of a typical application's functions in such a way that the user can find and understand them easily. Understanding the objectives a user has in mind while using software can greatly ease this problem, resulting in more appropriate, less complex software with the same power as software we see today. Breaking the application model and using a document or object approach may provide a superior solution.**

Index Terms—**Component Software, Interaction Design, Interface Scalability, Usability**

I. Introduction

Users of modern computers have to face increasingly complex software. Humans are no longer capable of coping effectively with this barrage of technology. The solution does not lie in the engineering domain, but rather in design. This paper therefore does not present any new technology, but rather makes the case for applying some well understood technologies in such a way that software can be designed effectively for all.

II. A Brief History of the GUI

One of the constants of computing over the last twenty or so years has been Moore's Law[1]. The increase of processing power has lead to an increase in the capability of computers that is incredible. But most users of modern computers never use more than a small fraction of these capabilities. There are a number of reasons for this, but one is the failure of the GUI to scale, that is, to present that increased number of capabilities effectively.

In the 70's when Xerox laid down the foundations of the GUI, processing power was, compared to today's standards, trivial. In 1980 a typical processor ran with a clock speed of 1Mhz, now 1Ghz is typical. The amount of storage and memory available to programs was also comparatively tiny. This greatly restricted the complexity of the applications that could be written.

Today, with the help of Moore's law we have massively more powerful systems, but Moore's law is exceeded only by the ability of programmers to consume these resources and create applications of a complexity that bewilder most users.

Meanwhile other parts of the typical computer system have improved much more slowly. Displays have increased in resolution from CGA's 640x200 in 1981[2] up to a typical 1024x768 now, about 6 times the screen area in terms of number of pixels. Even with a resolution of 1280x1024 there is only about a 10 times increase in screen real estate. With WYSIWYG more pixels are required to render a character on screen, so in terms of the quantity of information that can be displayed the difference is even less.

And the most important part of any computer system, the user, has received no upgrades at all!

The fundamental concepts behind the GUI have remained unchanged throughout this time though. The application, the window, the pointer are essentially unchanged. Buttons, drag and drop, and drop down menus are all idioms that have now been with us for over two decades.

The application though is the most important of these. There is a very strong relationship between the application and the process. Processes trace their ancestry back to mini-computer systems doing batch processing. In these early systems it was important that different tasks could not affect each other. Isolation became one of the aims of the process. In Unix, with its protected memory this isolation was well established.

From the vantage of robustness this was a great move forward, but this paper aims to show that in some respects, we now need to move beyond the application to a new paradigm. While robustness is as important as ever, the need for information to be accessed and worked with in ever more fluid ways suggests that this isolation may not be the answer.

III. Limits on the Scale of the Modern GUI

What is it that limits the level of complexity that a GUI can present? There are two main factors.

- Display space
- Users understanding and learning limitations

The display is possibly the most valuable resource an interaction designer needs to consider. It needs to present the information being worked with, and the tools that are being used. Balancing these two needs is vital to good design.

Clearly, the larger a display, the more controls can be made visible, and as displays have become larger, the number of controls has increased. But this is largely out of the control of the designer.

The designer does decide the size of on screen controls

Manuscript received 30 May, 2002.

P. M. Bagnall is an independent software design consultant, specialising in the design of usable systems. Lincolnshire, England. (telephone: +44 (0)7984 168 586, e-mail: pete@surfaceeffect.com).

however. There are two factors that limit the minimum size of a control. It must be large enough to visually represent its function. Icons are popular because they can often express their meanings with far fewer pixels than text. Secondly, they must be large enough to be manipulated by the pointer. Typically users can position a mouse to within a few pixels, but the smaller controls get the longer it takes for users to move the mouse to them. And many older users never acquire the manual dexterity than young mouse users take for granted. For buttons this means about 15x15 pixels is a lower limit.

What really makes complex applications possible though, is UI idioms such as the menu and dialog box. Both of these make functions available with a very low cost in display space. Both do this by hiding the functions behind another control, only presenting the functions temporarily. In theory this means that the number of functions that can be displayed should be, to all intents and purposes, infinite. But as functions become hidden in this way they also become harder to find. This is known as a discoverability problem. A feature is only useful to someone who discovers it.

The clues that are given to help users find functions are critical. Menus force the designer to classify functions into a number of menus. This is very sensitive to the wording of both the menu and the menu items. Getting either the menu name or the menu item name wrong can prevent users from ever finding the function they are looking for. Dialog boxes are even worse at this, simply because dialogs can present a wide range of functions. Finding a good cue, with only an icon or a few words to describe the functions offered is very challenging.

These problems also apply even more to websites, where navigation difficulties account for a huge number of usability problems. Users often fail to find pages, even pages they have seen before. The majority of users have, at some time, had the experience of knowing it's there somewhere, but not being able to find it.

Beyond the presentation of the interface there are deeper problems. Users are primarily interested in achieving their goals. They use computers because they can work more effectively that way. However, they are not interested in learning how the computer works. They therefore learn as little as possible to get the job done, and no more. This is borne out by the fact that users so rarely read the manual, so much so that software houses often no longer provide them!

Users only learn what is of use, and only remember what they use frequently. Any functions that they do not find useful are merely clutter, and make it harder for them to find the functions they find useful. The more complex applications become, and the more functionality they offer the worse this problem becomes. And this is the real failure of the GUI to scale.

IV. Types of Applications

One fundamental feature of modern operating systems is the file system. Applications fall into two rough groupings with respect to how much they rely on the file system to organise a users work. For the purposes of this paper these groups will be called document processors and information repositories.

A. Document Processors

A document processor is any application that relies on the file system to store users work. Word processors, Spreadsheets and so forth all fall into this category. Users create files, save them, close them, open them, work some more, and so on. The file system is ignorant of the internal structure of the files, and so, can offer little help in navigating to the right one.

Documents processors themselves come in different types too. There are the general-purpose editors, like the word processor, and specialised document editors. General-purpose editors are the ones that suffer most severely from scaling issues. This is precisely because they are general purpose and have the capability to do a vast range of different things. At the moment though they make up the bulk of document processors.

Specialised document editors are programs that are aimed at a very specific need. For example, a timesheet program might create files recording the time an employee spent working on a range of projects. The program would record that information, and that alone. This dedication to a single task allows specialised document editors, to have much simpler user interfaces and dodge the scaling issues to a large extent.

B. Information Repositories

These applications are essentially databases. Rather than relying on the file system to organise information, information repositories take this responsibility themselves. A common example is the email client. The email client manages storage of emails itself, and by doing so it allows for much richer interaction. Because it understands the context in which the information is used it can offer functions appropriate to email, which would not be appropriate to other types of data. Information repositories are slowly making some headway. Current examples from Apple include iTunes[3], and iPhoto[4]. The use of information repositories rather than document processors shows a tacit understanding that file systems are not the best organisational system for many types of information.

Information Repositories, because of this tight coupling between the storage/retrieval and the user interface, are almost always specialised. This is largely why they are still relatively uncommon as consumer software. General-purpose software has a wider market appeal, and so has more people to defray the development costs. They are very common however in corporate environments. Because efficient access to information is imperative in modern corporations, it becomes cost effective to develop sophisticated dedicated software. With the rise of the web many businesses saw this as a way of developing information systems that were effective and cheap to deploy. One of the problems with the web though is that it can present anything, and so it is very tempting for IT managers to try to offer all their information in a single place, for all users. This leads to exactly the same loss of focus that

appears in general-purpose software, and has lead to information scaling problems on the web. On the web though this normally goes by the name of navigation.

V. Understanding the User [5]

One way of solving scaling then is to finesse the problem, and simply avoid it. In practise this means moving away from general-purpose applications to more specialised applications. To design specialised applications well it is important to get a good understanding of the people who will use it and the circumstances it will be used in.

As an example, imagine a receptionist in a corporate lobby, who directs visitors as one of her many tasks. We'll call her Susan. Susan works for a large company, at its head office, where they have about 2500 staff. Because of the number of staff she is reliant on the staff phone list to be able to contact people when their visitors arrive. It is important for her to be able to maintain an air of efficiency, and so she needs a system that is going to help her do that.

Clearly she is going to be interrupted very frequently, and so whatever other tasks she may be involved in must be very tolerant of that fact. She must be able to call up the phone listings instantly.

Susan is also somewhat new to the job, and so she is not always very good at guessing the way names are spelt. A system that helps her mask this fact is a great boon.

Now imagine an important visitor arrives, and asks for Marc Smyth. Of course this is pronounced Mark Smith, and Susan has no way of knowing it is not spelt in the obvious way. The system therefore needs to understand that Marc Smyth may be the person she is looking for. Susan is naturally disinclined to ask is "how do you spell Smith?" which has the potential of making her look rather stupid.

This simple example highlights the value of getting a detailed understanding. From the description of Susan's working environment we can immediately see that the phone list needs a pseudo-phonetic system such as Soundex; a simple search may not help her. We can also get some ideas about what her desktop has to be capable of, to deal with interruptions.

Real situations, and real software, are rarely as simple as is suggested by the above example though. To really understand users and their situations it is necessary to go into the field and talk to them. Instead of asking them what features they would like to see in software it is more productive to find out about what they are trying to achieve, and what constraints they have to work under. Users more often suggest new features, or additions to systems, whereas a designer should be looking beyond that, to find an elegant solution to the whole problem.

Observing people in their working environment is also extremely useful; it allows a designer to take note of all the little workarounds, such as the way they use Post-It notes, and probe the reasons why they use them. Observing Susan in her workplace would have revealed the need to be tolerant of interruptions.

While talking to users the designer should be trying to determine the goals they have to achieve. In Susan's case, with respect to the phone listings, they are going to be…

1. Cope well with interruptions
2. Get the number needed quickly
3. Be able to find Marc Smyth without embarrassment

Having understood a user's goals, a designer then has to find a solution, which not only solves the immediate problem, in this case providing a phone listing, but also respects those goals. Much software in existence today, especially operating systems, has not had this level of scrutiny. Only by understanding how software is to be used can designers reliably design highly specialised software that will meet users needs.

VI. Beyond the Application

It is common, in the workplace at the moment, to see general-purpose editors used in place of specialised document editors. Macro languages have made this possible. While this means that very specialised systems can, and are built, it makes the interface problem worse. Users are presented with a system that offers all the functionality for destroying the specialised editor that has been built.

The best example, perhaps, is the Spreadsheet. Because spreadsheets allow for complex calculations based on data entered into them they can be used to calculate mortgages, collect timesheets, process scientific data and so forth. But the spreadsheet it far too prone to damage. Accidentally deleting a calculation cell can make the entire sheet invalid. The problem here might seem to be poor design of the spreadsheet program, but in fact runs much deeper. It is a case of offering functionality in a place where it is inappropriate. In the way these tools are used there is a clear demarcation between constructing the formulae, designing the spreadsheet and then later using the sheet to perform calculations. But the way functions are offered makes no distinction between these types of use, and simply offers all the functions all the time. In this case removing the functions for editing formulae while your mortgage is being calculated would be very beneficial, and would make the chance of error substantially smaller.

The fact that, despite severe problems in the interfaces, people use general-purpose tools to construct specialised tools points very clearly to a demand for highly specialised software tools. The question then becomes, how can the software industry feasibly produce software of this kind.

This is not an especially new problem. OpenDoc, OLE, and Java Beans have all walked along this path before. OpenDoc and OLE however concentrated on making extremely flexible documents, rather than specialised applications. They both assumed that a user would create a document and might decide to add any sort of element into their document at any time. This meant that OpenDoc and OLE both had to have the flexibility to construct these ad-hoc documents. Flexibility of course leads to complexity of interface. OLE, the winner of that race does indeed have a complex and rather confusing UI.

The way OLE is used in applications such as MS Word, and Excel, in effect, leaves design to the end users. It raises the bar for how general software can become, and as such leads

to yet larger UI scaling problems than ever before.

A. Super-Specialised Software

To see how a solution might work consider the Word Processor. In its current form word processors can be used to write virtually anything, from email, articles, faxes, books, websites, letters and so on. Each of these present different problems to their authors. Even writing a book varies greatly, depending on whether it's a work of fiction or non-fiction.

To further make the point consider two extremes, writing a novel, and writing an email. First, the differences. Clearly a novel demands serious planning. The story needs to be planned out, the characters described, and the whole written, and then sent to a publisher who will frequently assign an editor to make recommendations back to the author. The planning aspect is a major part of the whole effort of writing a book.

An email on the other hand is generally brief, and often in reply to another email. It needs to be addressed, and often, by tradition has some form of signature (although whether this is important is debatable).

However, despite the differences there are also a large number of similarities. For both, text must be entered, and a certain basic set of word processing features are useful. Spell checking, cut and paste, text formatting, and so forth. For the common elements there are significant opportunities for sharing software components. The message area of the email editor and the text area for the book might well be the same component. By using the same component consistency would be maintained. Likewise the components for handling keyboard input would also be common, and this would ensure that keyboard shortcuts for basic operations such as cut and paste were consistent, something that has often been a problem under MacOS.

The real benefits arrive in the novel editor though. With this the extra functionality that would be useful to support the author could be presented. For example, there could be tool panes that contain brief biographies of the various characters, which could help an author make sure the story maintained its self-consistency. A way of allowing an editor to make comments directly in the text, and the author then to navigate through those comments, much like Word's reviewing functionality could be presented in another. Allowing the author to set the email address of his publisher to streamline the publishing process might be useful for some authors. Setting up a style for chapter headings, breaks, page numbering would be useful in the context of a book. This functionality is all directly relevant to the objective of the novel author, to write a good book, but not to other users.

With the email client, the editing components would be very much like the novel editor, but the rest would be very different. Email clients are information repositories, not document editors like the novel editor, and so there is the store of messages to present. There should be integration with a contacts database, which should be a component that is used by many different systems. Tools such as anti-flaming warnings as Eudora[6] presents could be added.

Even with the novel editor and email client brimming with useful functions, they would not impact each other. In the monolithic application model to add the features to please the novel author would mean stretching the scalability of the word processor to, and beyond, breaking point. The amount of configuration that would be required to use the word processor in one or other mode would be too much for users. This is precisely why this is a solution to the GUI scaling problem. It allows functionality to only be presented in an environment where it is, in fact, useful. By targeting functionality much more closely it would be possible to increase the functionality effectively offered, while simplifying the UI. The key phrase here is "effectively offered". At the moment, with UI's beyond the scaling limits, functionality might be offered, but since users can't find it is effectively not offered, or indeed, not offered effectively.

B. Functional Templates

While this makes the number of functions offered in any of these super-specialised applications less, and hence scales better, it does however have the potential to vastly increase the number of applications a user uses.

It is questionable whether this is really a problem at all though. Instead it might simply be possible to dispose of the concept of application entirely. Having applications means that the user must decide which of the applications they have is most appropriate for the job they wish to do. However, with the system described above that choice is essentially made for them. If you wish to write a novel, use the novel editor. Thinking of the super-specialised applications as templates rather than applications leads to a very different presentation.

These templates not only describe the style of the document, but also the functionality, and appearance of the software to edit the document, hence the name, "functional templates". With stationary files on the Mac creating a document from a template was as simple as double clicking on the stationary file's icon. A similar approach is expected in this case.

The technology to create functional templates already exists. Java Beans and BML (the bean markup language from IBM alphaworks), are capable of constructing custom applications of this kind already. Some enhancements may be required to allow certain components to work together more effectively. For example, spelling checkers often annotate the text with underlining to show misspelt words. To separate the spelling checker from the text editing component would require a greater level of communication between those two components than is currently normal in GUI component libraries. Systems like Java Swing however could easily be adapted to include these capabilities.

C. Other Advantages

Using functional templates requires an underlying component software framework. Essential features include a mechanism to allow for late binding and a naming service.

Functional templates would have to work together. This means in practise that templates would, in some instances, have to be loosely defined. To achieve the high degrees of

consistency being aimed for a template must not insist on specific components, when equivalent ones are already in use elsewhere in the system. Instead a template might specify that it needs a component to fill a specific role, and make a suggestion about which component would fit. However, if the user has, in other contexts already been using a different component it is highly desirable that it be used instead.

This means that a user can choose those components that match their requirements best, and still have these interoperate with new software they buy later. This relies to some extent on these components being capable of describing themselves to each other. Systems such as Java Beans have precisely this capability. Similar approaches have been used to create late binding systems [7] capable of this kind of adaptation.

This also means, of course, that there must also be a way of changing individual components, and upgrading components in a reasonable manner. This requires no new technology, but it does require a great deal more discipline within the programming and design communities. These problems have, to some extent, been addressed by the Linux community, with tools provided for some distributions to automatically update any outdated packages.

Further, information repositories, being components structures in their own right, and therefore capable of being bound into a functional template also become more useful. Rather than being separated away from other applications they can become embedded in applications where they have relevance. This mixing of information repositories and document processors allows for a more fluid flow of information. It becomes more practical to have functional templates which include data taken from automated information repositories, a feature that some modern applications are just beginning to implement.

VII. Commercial Considerations

The large number of computer users today use general-purpose applications for much of their work. Since the web has become popular even some call centres, which had previously used specially written systems are now uses the general-purpose web-browser as their interface.

For the companies that develop such general-purpose software moving to a component software model is against their interests. By doing so they would have to compete with other vendors for every single feature. As it is they can resist small software vendors who specialise in a specific feature by touting the advantages of a single coherent platform. It is to be expected therefore that this kind of innovation is unlikely to come from the largest of the monolithic software vendors.

The other major commercial threat to this software model is the inconvenience of having to assemble software into functional templates. However, rather than being seen as a threat, this can be viewed as an opportunity for a new industry to emerge. By analogy the Linux community produces vast quantities of small pieces on unconnected software. To make installing a Linux systems practical companies have emerged, such as RedHat[8] and YellowDog[9], that package up useful collections of software and allow users to install them relatively easily. So it would also be possible for functional templates that combine a large number of components from different vendors to be packaged together by companies dedicated to building custom systems for more specific markets. At this point it is easy to imagine companies, who rather than specialising in specific software solutions specialised in certain domains, and packaged software of many different kinds to support users working in these domains.

What makes this approach timely, as has not been the case before, is the relative maturity of the Internet as a deployment medium. Now, it is genuinely possible for a packager to sell a software configuration that may require components from many other vendors. Before the widespread adoption of the Internet, buying software in small units like this would be been prohibitively inconvenient and overwhelming to end-users. Technologies such as Sun Microsystems' Java Web Start[10] clearly demonstrate the potential of the Internet in this respect.

As a simple example, physicists and mathematicians in academia produce papers in which they need to present equations. In most word processors equation editors are separate pieces of software that integrate relatively poorly with the rest of the system. However, a specialised package for mathematicians would clearly have this feature emphasised since it is such a common aspect of their work.

Because this software is aimed at small niches it is easy to imagine that companies would be able to offer excellent software without overbearing competition from current software giants simply because those giants do not have the resources to specialise in everything!

Functional Templates therefore have the potential to be, not only technically feasible, but also commercially viable, and most definitely desirable for users.

VIII. Conclusion

The GUI scaling problem has emerged as the complexity of software has increased over the last five to ten years. Because of fundamental limitations both in computer hardware, but more importantly in human ability, we are at the limit of complexity people can deal with. To solve the problem then we must in fact reduce the amount of functionality we offer people, and to do this we need to specialise our software more to serve their exact requirements.

This requires a more component based software model, and service companies to understand the needs of specific domains and to package software to meet the needs in those domains.

The Internet will be critical as a distribution medium if this model of software is adopted.

Peter M. Bagnall is an independent software design consultant specialising in the design of usable systems. He spent two years working with Cooper Interaction Design in Palo Alto, California, following four years as a software research engineer at British Telecommunications Martlesham Heath Laboratories.

REFERENCES

[1] Cramming more components onto integrated circuits
http://www.intel.com/research/silicon/moorespaper.pdf

[2] The PC Technology Guide – Graphics Cards
http://www.pctechguide.com/05graphics.htm

[3] Apple iTunes
http://www.apple.com/itunes/

[4] Apple iPhoto
http://www.apple.com/iphoto/

[5] The Inmates are Running the Asylum, Alan Cooper
ISBN 0672316498

[6] Eudora Moodwatch
http://www.eudora.com/download/eudora/windows/5.0/Whatsnew.pdf
Using Moodwatch - pp 28

[7] Flexinet Final Report – Chapter 29, Blueprints
http://www.ansa.co.uk/ANSATech/ANSAhtml/98-ansa/flexon.pdf

[8] Redhat Linux
http://www.redhat.com/

[9] YellowDog Linux
http://www.yellowdoglinux.com/.

[10] Java Web Start
http://java.sun.com/products/javawebstart/