USER-CENTERED DESIGN OF INTERFACE TOOLKITS

Mark Roseman and Saul Greenberg
Department of Computer Science, University of Calgary
Calgary, Alberta, Canada T2N 1N4
Phone: +1 403 220-6087
E-mail: roseman,saul@cpsc.ucalgary.ca

ABSTRACT
Current user interface toolkits unnecessarily complicate the process of creating user interfaces. This paper suggests that the design of these toolkits must be rethought. We advocate a user-centered approach to the design of these development tools, considering the application developer as user and applying principles from interface design to design of interface toolkits. The process of user-centered toolkit design is described, making use of design affordances to influence the developers use of the toolkit. User-centered design is shown as a useful framework for understanding a number of toolkit issues and their solutions.

KEYWORDS: graphical user interfaces, toolkit design, user-centered design, development methodologies

INTRODUCTION

Toolkits are one of the most commonly used tools for developing graphical user interfaces [7], and most environments have one. There is Motif and Xview for X, NextStep for NeXT machines, MacApp for the Apple Macintosh, and Windows for the IBM PC. However, a survey of programmers suggested that developers using toolkits devote a larger percentage of time and program code to the interface than developers using a UIMS or interface builder [7]. More surprisingly, development using a toolkit fared worse than using no interface tools at all! This supports a great deal of anecdotal evidence that current toolkits are very difficult to use.

We suggest that many of the principles used to guide the design of interfaces may be successfully applied to the design of tools that better support the interface developer. We will advocate a *user-centered* toolkit design, taking the needs of the developer-as-user as a starting point. We will also introduce the notion of a *design affordance*, where the properties of the toolkit objects suggests how application developers can produce better programs for end users.

There are several benefits that can potentially be realized by adopting a user-centered approach to toolkit design. The first is better tools, constructed from a better understanding of the needs of the developer and the uses the toolkit will be put towards. This in turn can result in higher quality applications that are developed more quickly using the toolkit. Finally, user-centered design can serve as a useful framework for thinking about the problems found in toolkits, helping to reason about their probable causes and possible solutions.

The next section develops the concept of user-centered toolkit design, taking the perspective of the application developers themselves as end users of programming toolkits. This is followed by an introduction to design affordances as a useful concept toolkit builders can use to influence better design of application programs. We then outline the steps to building a toolkit through a user-centered approach. Next, a number of issues arising in current toolkits are discussed and interpreted from this perspective. Finally, we illustrate how we are applying these concepts to the design of GROUPKIT, a prototype toolkit for constructing real-time groupware applications.

TOWARDS USER-CENTERED TOOLKIT DESIGN

A goal of toolkits is to ease developers' chores when building specific portions of a program, such as the human-computer interface. In order to succeed, the toolkit must provide the necessary components and functionality required by the developer. Yet how can the developer's needs be determined?

Consider traditional interface practice. When designing interfaces for end users, developers are taught to observe the users of their system, to focus on the type of work the users need to perform, and to take into account their needs and preferences. Such user-centered design has proven to be reasonably successful at determining the requirements for new systems.

We would argue that a user-centered approach is equally important for building other sorts of software, including software for developing user interfaces. The users here are the application developers who construct interface software. These developers have particular needs in building their applications that should be carefully considered by toolkit builders.

One way to motivate improved toolkit support is to study the problems developers encounter while building software. In fact, difficult problems such as high learning curves and portability are now being addressed specifically because developers need solutions to these problems. We consider such work to be examples of a user-centered and iterative toolkit design, comparable to the user-centered and iterative design of interfaces.

An important consequence of user-centered toolkit design is that the best way of determining the components that should be included in a toolkit is to look at the types of applications that developers are building. What features are necessary, which are important, and which will never be needed? These important questions can only be answered by considering the needs of the developer, many of which are embodied in the applications being designed.

DESIGN AFFORDANCES

In the previous section, we advocated the idea that toolkits should provide the components developers need as building blocks to good programs. But merely providing these components is not sufficient. Developers must be encouraged to actually *understand* and *use* the appropriate components when the situation demands it. With careful design, toolkits can actually encourage the creation of better programs. Here we develop the idea of a design affordance to address this issue.

Affordance theory has been applied to the design of interfaces by several people [2, 8], where an affordance is defined as the properties of objects that suggest particular uses to users. Gaver [2] believes the theory can guide us in designing artifacts which emphasize desired affordances and deemphasize undesired ones.

We define a *design affordance* as a property of a toolkit object that suggests how it can be used. The design objects are at the level of underlying software tools, not end-user applications. Design affordances therefore involve interactions between such tools and program developers, not end users. In this sense a toolkit could have affordances, suggesting appropriate uses of the toolkit to the software developer.

It is important to distinguish between design affordances and "features." Features can be hidden all too easily within a toolkit, be difficult to use, and can be hard to comprehend when or why they would be used. In contrast, design affordances are situated in the overall toolkit so as to present themselves to developers when needed. Design affordances clearly suggest to the developer *how* a particular feature can be used and *why* it should be used.

Design affordances can be used by toolkit builders to aid developers in building better programs. Most toolkits, for example, provide "control panels" that collect buttons, valuators, menus, and so on. As an affordance, this suggests a particular interaction style to the developer, and indeed control panels are prevalent in many of today's applications. Other common toolkit constructs-as-affordances are graphical canvases, text editors, and terminal windows.

Similarly, a toolkit supplying consistent looking components will encourage building consistent looking interfaces. Supporting keyboard accelerators by default at the toolkit level will encourage their adoption. Programs built using a toolkit with an embeddable control language (e.g. TCL [10]) will themselves be structured to support this language. Nothing forces developers to incorporate these features in their applications, but because of the toolkit's design affordances it is simply easy and natural to do so.

DESIGN PROCESS

The previous two sections looked at the general principles behind user-centered design of toolkits, as well as introducing design affordances that suggests to the designer how to use toolkit features. This section outlines the steps toolkit builders can take to follow a user-centered design approach. The steps reflect those found in more typical user-centered design: observe users, design and prototype, then repeat.

Identify Toolkit Domain

The first step is to identify the domain the toolkit will be used in. What problem is the toolkit meant to solve? A toolkit meant for building menu and form filling interfaces will be quite different from one supporting the creation of direct manipulation graphical editors. Toolkits for applications with a prescribed look and feel (e.g., Motif) will be different from those supporting a very configurable look and feel. Special needs application domains such as hypertext systems or groupware can benefit from toolkit support quite unnecessary for most applications.

It is important to take this first step, so that it can be determined what features will and will not be included in the toolkit. As with end-user applications, the more general the domain the more difficult it will be to construct a good system which meets the needs of its users.

Identify Developers

Having identified the target domain, it is then important to identify the developers who will use the toolkit. Depending on organizational considerations, this step will often precede the first. Knowing the types of users will have important consequences for how the toolkit will be designed, the level of functionality provided, and how that functionality will be delivered to the developer. A number of important considerations can arise.

Programming Languages. What programming languages do toolkit users already know? If possible, toolkits should allow use of known languages, or at least have bindings for familiar languages. If designing a toolkit in an unusual but perhaps more elegant language would require a large amount of learning, this might outweigh the advantages of using the toolkit in the first place.

Previous Toolkit Experience. What other toolkits have developers worked with? Wherever possible, a toolkit should exploit already familiar concepts, which may include things like main event loops and callbacks. Terminology is also important, and familiar concepts should be given familiar names.

Learning Time. How much time is available for users to learn and then use the toolkit? Will the toolkit be used for short-term or long-term projects? For example, a longer learning curve may be appropriate only when development will take place over long periods of time, and when sophisticated tools are required.

Identify Use of Toolkit

Toolkits can either provide building blocks for making domain-specific objects, or the domain-specific objects themselves. In the first case, the developers must act as widget builders, creating high-level components that are then used to construct the end applications. In the second case, the high-level components in the toolkit are used directly in the end applications.

As one example, a toolkit could provide all the buttons, scrollers, etc. required to construct a file chooser dialog, or could provide just the file chooser itself. Similarly, toolkits could provide very general components, which developers need to customize to use. Alternately, finished complete components with particular behaviors could be included.

The choice will depend on the ways developers need to use the toolkit. If only standard interface components are required, the toolkit need stock only a fixed set of widgets. Toolkits for larger problem domains may benefit from a layered approach, where core functionality is provided but in an extensible way. Developers extend the functionality to create their own widgets. This approach might also be beneficial for newer or more experimental domains.

Consider Target Applications

The toolkit features required by developers depend heavily on the applications they will be constructing. Studying these applications can suggest common needs that can be translated into functionality and features that are best incorporated into the toolkit. Existing applications in the selected domain are the easiest to consider, but considering possible future applications can be valuable as well. This step is useful to decide what features are important to include and which are unnecessary, refining decisions made as a result of identifying the toolkit domain. Members of a UIST Panel on X toolkits [13] argued that it is invaluable to design toolkits in conjunction with a number of typical target applications.

Design for Proper Use

The previous steps generated a set of necessary components and features of the toolkit, raised issues concerning flexibility of the components, and suggested information about the implementation language, and other important concepts. But combining and structuring this information in a useful way is important as well.

A toolkit provides more than an alphabetic list of routines in a library available to the developer. A toolkit should contain a philosophy of how applications should be developed using the toolkit, and should encourage developers to construct good programs properly.

This is where design affordances can be used. Make it obvious to the developer, through the structure of the toolkit, how applications should be built. If features deemed important or essential are hidden deep within the toolkit or are difficult to use, they will be ignored. As described earlier, the toolkit developer can greatly influence the way the toolkit is used. Done properly, this can result in developers building better programs.

There are other ways in which the toolkit developer can aid the application developer in creating better programs. Good documentation describing the toolkit can highlight not only how to use particular widgets, but more importantly when to use them. Example programs included with the toolkit are extremely valuable resources, and probably more likely to be used than even the best documentation. Good examples can highlight instances of good interface design, which are sure to be remembered when similar new projects arise. Interface style guides (e.g. Motif Style Guide [9]), and tools to help the developer follow them (e.g. KRI/AG [5]) can all assist in the process of creating better interfaces.

Iterate Design

Finally, it is essential to iterate the design, using preliminary versions of the toolkit to design standard applications, identifying problems, redesigning, and trying again. The toolkit should be used by the application developers, who will identify problems that would not be found by the toolkit builders. As with applications, the chance of getting the design right the first time are extremely small. Iterating the design will help refine both the set of features and the ways of structuring those features.

ADDRESSING TOOLKIT ISSUES

The user-centered approach can be used as a framework for considering problems and issues that arise in toolkit design and use. This section looks at a number of these issues and some of the proposed solutions from a user-centered perspective.

Learning Curve

One of the most common complaints against user interface toolkits has been the incredible learning curve often associated with them. Toolkits can require learning hundreds of new library calls, or in the case of object-oriented toolkits, dozens of new object classes, each with potentially large numbers of access methods. Achieving proficiency in most modern toolkits is a long and painful process. Even worse are layered toolkits that require knowledge of *underlying* libraries. For example, proficient use of MacApp [15] requires an intimate understanding of the Macintosh ToolBox. Similarly, most higher level X Windows toolkits demand knowledge of lower level Xlib calls.

From a user-centered perspective, this sort of design is appropriate only when developers already have the required background knowledge or have compelling reasons to learn it. When that knowledge is not present, higher level libraries should be designed to encapsulate, and not just extend lower level libraries. Implementation details should be hidden, at least at first, requiring the minimum amount of learning to accomplish basic tasks. As more involved tasks are attempted, underlying details may be exposed. The amount of learning should be proportional to the complexity of the task. In other words, simple things should be simple, and hard things possible.

There are some systems that address this high learning curve by trying to match its complexity with the needs of the developer. The Simple User Interface Toolkit (SUIT) [11] provides basic graphical interface components such as buttons, sliders, text widgets, yet boasts a learning curve measured in hours, not weeks. SUIT was specifically designed for quick learning, in particular for use by student during a single course. InterViews [4] is one of several X toolkits that requires no knowledge of underlying X internals, catering to those with no need or desire to learn

them. Other learning aids exist outside the toolkits themselves, ranging from books and sample programs, to systems such as the View Matcher and others developed at IBM [14] to aid developers in coping with the large SmallTalk class library.

Integrating Application Code

Toolkits are typically characterized by the separation of interface and application code. While this separation is generally seen as useful, one of the greatest problems with toolkits is integrating application code. The most common mechanism for this is the callback. Unfortunately, as the number of interface components grows, so does the number of callbacks, resulting in large amounts of application code closely tied to the toolkit. Some interesting work was done by Myers [6], who studied how developers used callbacks, and from that identified common tasks they were used for. This knowledge was then used to redesign a toolkit which reduced the number of callbacks the developer needed.

Portability

Another general issue is that of portability, or allowing applications developed with the toolkit to be used on different physical platforms. While this is a generally desirable feature for a toolkit, it will depend heavily on the needs of the application developers, whether they must support applications across several platforms. A number of toolkits are available supporting some degree of portability across platforms (e.g. XVT, SmallTalk/V, SUIT, X).

Another issue is supporting different look and feels from the same code (e.g. an application's appearance can be toggled to Motif or Open Look). This is becoming increasingly necessary for application developers due to the wide variety of look and feels and the high cost of building systems. InterViews, for example, provides different "widget factories" for different look and feels, each with the same programming interface. The end user can call up the look and feel of choice when the application is invoked.

Extensibility and Abstraction

Different toolkits provide vastly different levels of support to the developer. At the bottom level, simple graphics primitives (e.g., drawing points, lines, circles and bitmaps) are technically sufficient for designing any interface, albeit with incredible effort on the part of the developer. At a much higher level, complete dialog boxes can be provided by the toolkit, saving countless hours of programming time, but severely restricting the flexibility of the resulting interface. Extending the interface is possible with low-level primitives available, yet higher level toolkits can make extensions almost impossible.

This is inherently a trade-off, between the time saved by high-level toolkits, the configurability provided by the low-level toolkits, and the degree of standardization required. Yet this is again best understood in terms of the developers needs. Time will only be saved by high-level components if the components needed by the developer are provided. Configurability may not be as important if the proper components are available, if the task domain requires little variation, or if the developers are not themselves skilled at widget design.

An interesting approach to this problem is taken in InterViews. Many low-level components are provided in the toolkit, e.g. allowing the creation of buttons with almost arbitrary appearance and behavior. This provides a high level of functionality, but at the price of great complexity. However, InterViews also provides a WidgetKit which provides a simplified front end to the underlying functionality. The WidgetKit allows developers to very simply instantiate commonly used types of buttons, such as push buttons and radio buttons. By taking this approach, InterViews caters well to developers needing standard components as well as those with a need to design custom components. Simple things are simple, while harder things are reasonable.

Internal Structure

Borenstein [1] has noted that many user interfaces are among the worlds most horrible programs in terms of their internal structure and maintainability. Much of this difficulty is due to the inherently iterative design required in developing interface software.

However, much of the difficulty can also arise from a toolkit that does not adequately fit the needs of the application developer. It is a rare project indeed that can handle the entire interface in a completely standard way, and that is completely supported by the toolkit. New interaction techniques in particular can cause endless grief to the developer, who must spend considerable time hacking around the framework provided by the toolkit.

As a particularly pathological example, we developed an object-oriented group drawing editor called GROUPDRAW [3], which included large multiple cursors for all group members that are always displayed on all workstations. This technique has proved useful for allowing participants to gesture around the drawing artifacts. However, implementing multiple cursors with the toolkit we used involved a great deal of effort, for the toolkit (as with most window systems) supplies only a single cursor. As a consequence, various pieces of the cursor handling code found themselves into completely unrelated corners of the application code. Here, the underlying toolkit had not provided the primitives we required to implement this feature, resulting in a confusing and difficult implementation.

GROUPKIT: A CASE STUDY

We are currently developing a groupware toolkit called GROUPKIT, described in detail elsewhere [12]. GROUPKIT arose primarily out of our frustrations in building groupware applications with conventional single user interface toolkits [3]. As such, we were concerned with building an infrastructure that would meet our needs. This section reflects on our ongoing experience in working with the GROUPKIT prototype, to help illustrate some of the concepts of user-centered toolkit design.

From the beginning, we constrained the toolkit domain to consider only real-time groupware applications for geographically separated participants. These include applications such as shared drawing and text editors, analogies to face to face meeting tools, but excluding asynchronous groupware such as electronic mail. The needs of the two types of groupware are different enough that supporting both is non-trivial.

Because groupware is a relatively new and very diverse application domain, we knew the toolkit would have to be designed in a very open-ended and extensible way. This suggested the use of an object-oriented language. Since most groupware is being developed in research labs, we felt comfortable in assuming fairly technical developers, who expect to build their own groupware-widgets from primitives. The need for robust communications suggested a Unix platform. Because of its flexibility and easy composition of small interface components into larger ones, we selected InterViews as the platform for building GROUPKIT. While InterViews does have a reasonably high learning curve, the basic mechanisms required by GROUPKIT are relatively quickly absorbed.

In designing GROUPKIT, we began by selecting existing groupware applications which we wanted to be able to replicate using the toolkit. We also began to speculate on the type of applications we might want to build in the future. From looking at these various sorts of applications, we distilled a preliminary set of requirements, which are detailed in Table 1. We identified two types of requirements. *Human-centered* requirements emphasize groupware features that benefit end users — including them in the toolkit encourages developers to include them in applications, resulting in better programs. *Programmer-centered* requirements emphasize the required technical features hidden from the user — including them in the toolkit aids the developer and results in quicker development time and better structured programs.

We created a design affordance in GROUPKIT to meet the first requirement, supporting actions (in this case gesturing) over a work surface. Gesturing has been shown to be an important activity in the shared workspaces that many groupware programs strive to emulate [16]. Despite this evidence, many shared workspace programs do not support gesturing, likely because of the great implementation difficulty [3]. GROUPKIT provides for a gesturing overlay that supports gesturing over any workspace by adding a single line of code to the application. Because the overlay model is easily conceptualized by designers, and because gestures are so easy to incorporate, it serves as a design affordance, encouraging developers to think about gesturing and to include it in their programs where appropriate, thereby producing better groupware programs.

As another example, we designed a feature called open protocols as part of our registration system. Because different groups have different work practices, it is necessary to accommodate different working styles in the software. Open protocols allow different registration modules to be easily added to groupware applications, where each module

Requirements	Rationale	Examples
Human-centered requirements		
Supporting multi-user actions over a visual work surface	Human factors work indicates people often gesture over and annotate diverse artifacts	• provide support for gesturing • provide support for graphical annotation
Structuring group processes during a meeting	Many conferences need to be structured, but different groups require different sorts of structure in order to accept software	• provide various floor control policies • support different registration methods • support latecomers to the conference
Integration with conventional ways of doing work	People are comfortable with using other media, e.g. telephone and other programs	• integrate other forms of communication • allow use of single-user applications
Programmer-centered requirements		
Technical support of multiple and distributed processes	Programmer must, for any conference, manage a set of multiple, distributed processes, and connections between them, including starting up connections, keeping them active, and tearing them down.	• provide processes for basic conference management • provide a robust communications infrastructure • provide support for persistent sessions
Technical support of a graphics model	Many applications can be seen as shared visual work surfaces, which will require textual and graphical primitives which can cope with issues such as concurrency and WYSIWIS view sharing	• provide primitives to a shared graphics library • provide object concurrency control • separate the view of an object from its underlying representation

Table 1. Human-centered and programmer-centered design requirements in GROUPKIT, showing the requirements, the rationale behind the requirements, and several examples of specific guidelines which could be used to develop the toolkit.

can implement a particular registration style best suited to the needs of the group. Open protocols serve as a design affordance by facilitating this flexible and open-ended design that is needed to accommodate group differences.

The toolkits development was accompanied by the development of several applications we deemed typical for the application domain. These included a group sketchpad, a group drawing program, a shared terminal, a brainstorming tool, and a voting tool. This concurrent development highlighted a number of problems in early designs, resulting in numerous changes and one complete reorganization of the toolkit structure. As we continue to work with GROUPKIT, we expect further changes and additions to be made, both in the toolkit and our design requirements.

SUMMARY

This paper has applied many of the user-centered design concepts traditionally used for designing user interfaces to the design of interface toolkits. We believe it is important to view toolkits not as entities in themselves, but tools which are used by developers to accomplish a task, that of building application programs. Only by analyzing the work and needs of the developer can we adequately support that task.

The notion of design affordances reinforces the relationship between the toolkit and the developer, emphasizing the need for making important features and their uses visible and obvious. Design affordances can reinforce an underlying toolkit philosophy, and if used correctly, assist developers in creating applications that better suit the needs of their users.

Finally, we feel the user-centered approach is a valuable framework for considering issues related to toolkit design. When problems are encountered by developers in building applications, the language of user-centered design can help explain and perhaps solve these problems.

REFERENCES

1. Borenstein, N., *Programming as if People Mattered.* 1991, Princeton University Press.
2. Gaver, W. Technology Affordances. in *ACM SIGCHI Conference on Human Factors in Computing Systems.* 1991.
3. Greenberg, S., Roseman, M., Webster, D., and Bohnet, R., Human and Technical Factors of Distributed Group Drawing Tools. *Interacting with Computers*, in press, late 1992.
4. Linton, M.A., Vlissides, J.M., and Calder, P.R., Composing User Interfaces with InterViews. *IEEE Computer, 1989.* **22**(2).
5. Löwgren, J. and Nordqvist, T. Knowledge-Based Evaluation as Design Support for Graphical User Interfaces. in *ACM SIGCHI Conference on Human Factors in Computing Systems.* 1992.
6. Myers, B. Separating Application Code from Toolkits: Eliminating the Spaghetti of Call-backs. in *ACM Symposium on User Interface Software and Technology (UIST '91).* 1991.
7. Myers, B. and Rosson, M. Survey on User Interface Programming. in *ACM SIGCHI Conference on Human Factors in Computing Systems.* 1992.
8. Norman, D., *The Psychology of Everyday Things.* 1988, Basic Books, Inc.
9. Open Software Foundation, *OSF/Motif Style Guide.* 1988.
10. Ousterhout, J.K. Tcl: An Embeddable Command Language. in *Proceedings of the 1990 Winter USENIX Conference.* 1990.
11. Pausch, R., Young, N., and DeLine, R. SUIT: The Pascal of User Interface Toolkits. in *ACM Symposium on User Interface Software and Technology (UIST '91).* 1991.
12. Roseman, M. and Greenberg, S. GroupKit: A Groupware Toolkit for Building Real-Time Conferencing Applications. in *ACM Conference on Computer-Supported Cooperative Work (CSCW '92).* 1992.
13. Rosenberg, J., Asente, P., Linton, M., and Palay, A. X Toolkits: the Lessons Learned (Panel Session). in *ACM Symposium on User Interface Software and Technology (UIST '90).* 1990.
14. Rosson, M., Carroll, J., and Sweeney, C. A View Matcher for Reusing Smalltalk Classes. in *ACM SIGCHI Conference on Human Factors in Computing Systems.* 1991.
15. Schmucker, K., MacApp: An Application Framework, in *Byte.* 1986, p. 189-193.
16. Tang, J.C., Findings from observational studies of collaborative work. *International Journal of Man Machine Studies*, 1991. **34**(2): p. 143-160.