
Platforms for Interface Evolution

Jeff Dicker

David Cheriton School of Computer Science
University of Waterloo
200 University Avenue West
Waterloo, ON, Canada, N2L 3G1
jadicker@cgl.uwaterloo.ca

Bill Cowan

David Cheriton School of Computer Science
University of Waterloo
200 University Avenue West
Waterloo, ON, Canada, N2L 3G1
wmcowan@cgl.uwaterloo.ca

Abstract

User interface design languages need to support interface evolution: for designers who must provide updated versions, to be sure, and for users who want to adapt interfaces to their work habits. At present, despite several decades of advance in HCI, support for user-controlled interface evolution remains rudimentary, little more than cosmetic. Why? We argue that the answer lies not with design languages themselves, but that they are severely limited in their expressivity by the event-driven architectures they assume. Events must be replaced by a more richly expressive abstraction, a prototype of which is described in this position paper.

Keywords

UIMS, interface evolution, interface platform, adaptation, customization

ACM Classification Keywords

H.5.2. User interface management systems (UIMS).

Introduction

Recent HCI research has produced modern systems with many novel interface techniques, including: pen and sketch-based interfaces, efficient input on handheld devices, multi-touch displays, and gestured input. However, despite recent research that could support customization and adaptation, modern systems make very little and very poor use of these techniques. The facilitation of these two goals will create a paradigm of interface evolution: UI's will be shipped to fit all, but will be

tailored by the user to suit the user's individual preferences and habits. Why, then, do modern interfaces only scrape the surface of features that support evolving interfaces?

Where the Current Paradigm Fails

Allowing a user to customize their desktop environment is something that should make them more comfortable with their computer and more efficient at using it. So, naturally, desktop customizations have existed since the beginning of the desktop paradigm. For example, a user can customize the desktop background, transpose interface elements (including moving menu items to a toolbar), and toggle the availability of features.

Adaptation has a shorter history, but it has been quite well studied thanks in HCI research. Despite this, it is still being applied very poorly (if at all!) to modern systems. One example is in Microsoft's Office XP: menu items that are not frequently used are hidden from the user, causing frustration and less efficient usage [2]. As far back as 1995, Negroponte proposed a "digital butler" as an interface[1]. The digital butler in Office XP manifested itself as a paper clip that asks if you would like tips on writing letters. Why can't the paper clip tell you, "Keep working, I'm pretty certain your 3 new emails are all spam."?

These modern evolution features are merely cosmetic, utilizing little of the power of interface evolution. Imagine being given two common tools in a word processor: cut and paste. A user may find that a common pattern is to cut, then paste immediately to replace the cut text, then paste elsewhere. This is usually known as a copy command, but assume, for the purposes of this example, that no copy command exists in the shipped interface. A truly powerful paradigm of interface evolution should allow "cut, then paste" to be formed into a new copy command. Furthermore, the user should be able to replace the cut button that is in a toolbar in the unmodified version of the word processor with a new button linked to the

copy command and also define a hotkey. This new, evolved version of the word processor's interface now has copy, cut, and paste, instead of just cut and paste.

If current implementations are not the state of the art with respect to HCI research, why are they the state of the art in modern systems?

Why the Current Paradigm Fails

The best modern systems use the event-driven GUI toolkits to satisfy model-view-controller architecture. This platform is not adequate. Using event-driven widget toolkits was a good starting point for separating model from view, but fails to provide an adequate platform for the next generation of user interfaces, because it does not afford enough power to the view layer. The glue between model and view in most modern toolkits is an event system, and event systems are simply callbacks. The reason that it is so difficult to incorporate useful interface evolution in current GUIs can be attributed to this: the functionality of an application has traditionally been exposed only through allowing a "stupid" view layer to make a specified function call per each widget event.

Tying a view directly into an application through callbacks removes the separation between view and model and places the real interface onus entirely onto the application via event handlers. Because of this, the only way that a view can truly evolve is through evolution in the model. This practice of giving the user a view layer with no knowledge of the application leaves the user with no tools to evolve a view. Thus, the poor evolution that does exist in modern systems exists through functionality that has already been finalized when the application was programmed.

Facilitating Interface Evolution

The solution to the problem of the “stupid” view layer is to have a third party manage the calls between view and model. This third party is called the Interface Manager. If an Interface Manager is given a set of commands that it can perform on an application and allows the view layer to make calls to those commands, the problems associated with implementing interface evolution techniques go away.

The Interface Manager presents an intelligent view layer to the user. This view layer does not understand anything about an underlying application, thus keeping model separate from view, but it does understand how to call commands in an application. Past software packages have attempted to create similar models, but they failed to keep the layer between application and Interface Manager thick enough. The application will not be allowed to have event handlers, and neither will the Interface Manager. The event handlers will be commands sent to applications in accordance with default viewing specifications that the user can modify. Conceptually, the Interface Manager would allow a view to evolve separately from any application. With the power of this paradigm, a user can evolve an interface in a way that is entirely independent of applications. This is the advancement that is required to facilitate interface evolution.

Imagine again the cut and paste scenario. The Interface Manager can support the user in specifying a new command that is a combination of two commands the Interface Manager already knows about, because they are in API's exported by the

References

- [1]Negroponte, N. Being Digital. Random House (1995), 149–159.
- [2]McGrenere, J., Baecker, R. M., and Booth, K. S. 2002. An evaluation of a multiple interface design solution for bloated software. In *Proc. CHI '02*, ACM Press (2002), 164–170.

word processor. Combining the commands and changing menu items now becomes a visual programming problem, not an architecture problem.

For the Interface Manager to fully succeed, it needs to be system-wide. For customization, this would allow a user to call, combine and compose commands from any applications on the system. For adaptation, this would allow the Interface Manager to spread an adaptation in one application to others in the interface.

Back to the Future

Readers who have enough grey in their hair will recognize in the Interface Manager an old concept, the User Interface Management System (UIMS). UIMS's died of success, their best features appropriated by UI toolkits, from Visual Basic to QT[4]. But, in addition to its widget set, a UIMS also provided a higher level glue layer than simple events and callbacks. Our proposal enriches the functionality of that layer, giving it abstractions that make it programmable even by the end user. In addition, because a UIMS exports the same concepts to all applications and their interfaces, adaptations can spread from one interface/application to another, even, if desired, without user intervention.

- [3]Gajos, K. Z., Wobbrock, J. O., and Weld, D. S. 2007. Automatically generating user interfaces adapted to users' motor and vision capabilities. In *Proc UIST '07*. ACM Press (2007), 231-240.

- [4]Hudson, S. Personal communication at Graphics Interface 2007.

