User Interface Façades: Towards Fully Adaptable User Interfaces

Wolfgang Stuerzlinger
Interactive Systems Research Group
Dept. of Comp. Science, York University
Toronto, Canada
http://www.cs.yorku.ca/~wolfgang

Olivier Chapuis, Nicolas Roussel
Projet In Situ
CNRS, Université Paris-Sud & INRIA
Orsay, France
http://insitu.lri.fr/{~chapuis,~roussel}

ABSTRACT
User interfaces are getting more and more complex, and adaptable and adaptive interfaces have been proposed to address this issue. Previous studies have shown that users prefer interfaces that they can simply adapt themselves to self-adjusting ones. However, most existing user interface toolkits provide very little support for creating adaptable interfaces. As a consequence, interface customization techniques are still very primitive and usually constricted to particular applications.

In this paper, we present *User Interface Façades*, a system that provides end-users with simple ways to adapt and recombine existing graphical interfaces, through the use of drag-and-drop paradigm. User Interface Façades employs a more appropriate level of granularity for adaptation compared to previous work and also allows end-users to adapt the interaction of arbitrary applications. Finally, we show several examples to demonstrate the power of the new technique.

ACM Classification: H5.2 [Information interfaces and presentation]: User Interfaces. - Graphical user interfaces.

General terms: Design, Human Factors

Keywords: Adaptable user interfaces

INTRODUCTION
User interfaces are becoming more and more complex as the underlying applications add more and more features. Although most people use only a small subset of the functionalities of a given program at a given time [McGrenere2002], most software make all commands available all the time, which significantly increases the amount of screen space dedicated to interface components such as menus, toolbars and palettes. This quickly becomes a problem, as users often want to maximize the space available for the artifacts they are working on (e.g. an image or a text document). One reason for this problem might be that most user interfaces are still designed by software programmers today, a fact that is only slowly changing. However, even trained interface designers cannot always foresee how a software package is going to be used in practice, especially if the package is used by a large amount of users. This makes creating flexible user interfaces a major challenge.

Consider GIMP as an example. This image manipulation program opens automatically many toolboxes in separate windows. This in turn increases the window management overhead and increases the distance to the drawing tools and functions from the drawing area. Users adapt with varying strategies, such as having all toolboxes on a secondary monitor, overlapping the drawing with the toolboxes, etc. On the other hand, some applications use an all-in-one window logic, which provides less flexibility in terms of user interface layout.

Screen space is today mainly limited by the size of monitors, and this factor increases only slowly. Efficient display space management remains a problem for people using multiple monitors, since the distances the mouse cursor has to travel across the screens can get fairly large [Baudisch et al., 2003]. Interacting with several applications on several screens can quickly turn into a frustrating and painful experience as it forces users to move their hand and head constantly.

One way of dealing with the growing number of application features and the desire to optimize screen space is to allow users or applications to customize the interfaces. These two concepts have been studied for some time by the HCI community (see [Kantorowitz 1989] or [Kühme 1993], for example). Today, they are most often referred to as (user-) *adaptable* and *adaptive* (or self-adapting) interfaces [McGrenere 2002]. Adaptive interfaces change their appearance based on some algorithm, such as a least-recently used criterion. One recent example is the menus of the Microsoft Office suite. Adaptable interfaces, on the other hand, can be configured by the user to suit his or her own

criteria. As an example, many toolbars used in Mac OS X applications can be interactively customized with simple drag-and-drop operations.

Adaptive interfaces can have some unpleasant side effects, such as to surprise the user by moving or removing menu entries. Adaptable interfaces suffer from the problem that new interface elements and interaction techniques must be added to the software to support the customization of its "primary" interface. A comparison of static, adaptive, and adaptable menus showed that users could optimize their performance if they knew about the possibility of adapting and were able to adapt their menus with a simple interface [Findlater 2004]. Another interesting finding is that the adaptable user interface did not perform worse than the other two alternatives. Furthermore, participants greatly preferred the adaptable interface to the two other alternatives, a fact that we see as strong motivation for additional research in this area.

RELATED WORK

One of the simplest forms of user interface customization is the notion of *skins* and *themes* available in some applications and user interface toolkits. The term *skin* comes from video games such as Quake that allowed players to alter the appearance of their character by providing a set of images. Soon after its introduction into video games, a number of media players adopted this customization mechanism. Today, user interface toolkits such as GTK+ or QT include a styling engine that makes it possible for several applications to share a common visual style — or *theme* — specified at runtime by the user. A skin (or a theme) can simply consist in a set of colors or textures used by some existing drawing code. It can also partially or completely replace that drawing code, possibly adding complex output modifications [Edwards1997]. In addition to the visual style of the interface elements, skins can also specify the layout and behavior of these elements. Recent work by [Fogarty et al. 2002], [Chatty et al. 2004] and others has extended this approach to bridge the gap between appearance and semantic meaning. However, although they allow visual designers to customize interfaces using off-the-shelf drawing tools such as Adobe Photoshop or Illustrator, these systems remain out of reach for end-users who can simply choose between pre-defined skins or themes.

Bentley *et al.* distinguished between surface and deep customizations [Bentley & Dourish, 1995]. An example for a surface customization is the option provided by some applications to reconfigure their menus, toolbars or palettes. In this case, the user is constrained to using the functions the system developer has provided, and cannot customize the actual behavior of the system. An example of deep customization is the ability to integrate a word processor with a foreign language translation program, when this operation was not pre-programmed by the system developer. Overall, Bentley characterized the customization gulf by two interrelated problems. The first one is the level of customization possible, and with most systems this lies above the functionality of the application, rather than within it. The second problem is the language of customization, and traditional systems provide limited facilities to express customization requirements using the skills users already have, requiring the learning of new languages to describe new system behaviors. Both of these problems combined make it hard for or prevent the user from reaching into the system and customizing the way in which functionality (and not simply the interface to the functionality) relates to their accomplishment of work.

One of the biggest obstacles for adaptable interfaces is that it requires a fairly substantial programming effort to add this capability to a software package. Most user interface toolkits offer no support for implementing adaptable interfaces. This factor has certainly hindered the adoption of the idea of adaptable interfaces. As a notable exception, Apple's Cocoa toolkit provides developers with a toolbar widget that users can customize at runtime using drag and drop operations. However, the customization interface it provides is far from optimal (Figure 1), as it doesn't provide for undoing changes, reverting to previous versions, and employs a fixed window. Microsoft Office applications also allow users to customize their various toolbars and menus. Again, the customization interface has again some serious flaws (Figure 2), which are similar to the Apple example. In both cases, what we have is some permanent surface customization (if not permanent, at least hard to get rid of or to revert).

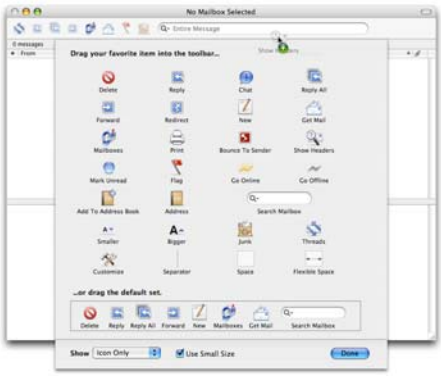

Figure 1. Apple Mail's interface for customizing th toolbar. Individual icons and the standard icon set can be dragged to/from the toolbar, but these operations cannot be undone and there's no Cancel button. The interface is about 600x600 pixels and is attached to the main window (it cannot be moved elsewhere). Icons already in the toolbar still appear in the customization interface.

Tan et al. introduced the idea of WinCuts [Tan2004]. This prototype allows the user to select arbitrary rectangular regions from a window, and presents each such region in a separate window. The authors present this as a collaboration tool and also as a way to reconfigure user interfaces. However, the inability of WinCuts to merge regions into new windows is clearly a limiting factor here. Another problem is that WinCuts does not allow for full interactivity, due to focus management problems and does not allow the user to change the interaction itself.

Hutchings and Stasko recently presented several papers in this area. They first proposed "copying a part of a window" [Hutchings and Stasko03] and then developed the idea of replicating dialogs boxes on multiple monitors until the user interacts with one of the copies [Hutchings and Stasko05]. Also, they presented the idea of removing (unused) parts of a window to keep only the part that is relevant to the user [Hutchings and Stasko04].

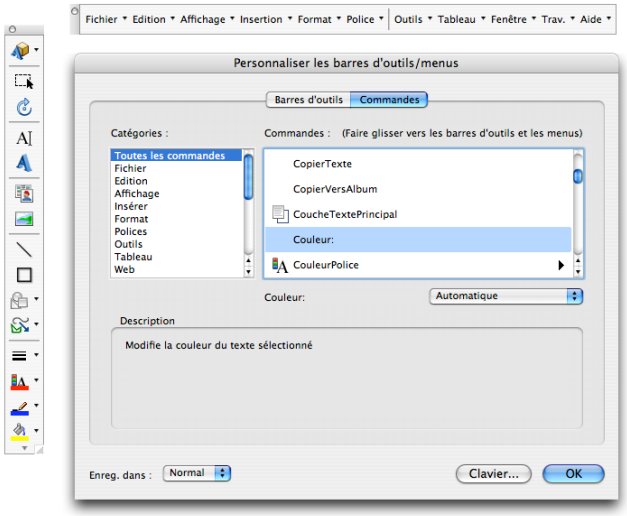

Figure 2. Microsoft Word's interface for customizing menus and toolbars. The list on the left side contains 22 categories (including *All the commands*). The list on the right side shows the commands relevant to the selected category. More than 1100 commands are available through this interface. Command icons and labels can be dragged to/from menus and toolbars, but these operations cannot be undone and there's no *Cancel* button. The interface is about 600x500 pixels, it can be moved but not resized. Icons already in the toolbar still appear in the customization interface.

The idea of Web Clipping was also introduced to synthesize new overviews based on information available on several different WWW pages. Here new pages are generated based on individual elements selected from other pages. Recently, this idea was generalized to a more interactive version by Fujima et al., who added Excel-like computational facilities to this concept [Fujima2004]. However, this approach is limited to WWW technologies and does not allow the user to change the interaction behavior of the UI.

Given that most of the mentioned factors are not going to change in the short term, we see a strong need to come up with new ideas for adaptable UIs. While the idea of adding this functionality to software toolkits seems attractive at first glance, it has the drawback that it will make the already very complex API (application programming interfaces) for user interface toolkits even more complex, requiring yet more code. This is clearly not a positive thing and would not speed adoption of the fundamental paradigm of adaptable interfaces.

USER INTERFACE (UI) FAÇADES

This work focuses on direct-manipulation applications, where the focus of work is usually a single (or a best a few) document(s) and we will limit our discussion in this paper to this class of applications. In such applications the screen is dominated by a large work area, with UI elements clustered around. Examples for this are drawing packages, text processors, spreadsheets, etc. This is in contrast to dialog-based applications, which rely on a query-answer paradigm with popup boxes – such as the Windows configuration dialogs or many WWW-based banking applications.

When reflecting on new ways to allow end users to redesign the interface for arbitrary applications, we realized that one of the simplest paradigms for handling arbitrary regions of screen space is the "cut, copy & paste" paradigm, as this is a form of functionality that practically all users are very familiar with.

Based on the above discussion, we formulated the following criteria for adaptable user interface:

1) *Granularity*: Redesigning interfaces necessitates that individual elements can be moved around. This is very similar to the way a window manager permits the user to move individual windows around. However, a much more appropriate level of adaptability for interfaces is that each widget (and potentially even parts of widgets) can be selected and manipulated.
2) *Level of control*: As discussed above, it is beneficial if users can reconfigure the interface themselves. The alternative of changing the user interface via software is clearly inferior from the end-user's viewpoint.
3) *Modify interaction*: While many user interfaces allow for changes in the visual appearance (e.g. themes), adaptability should also extend to the level of the interaction with the user interface elements.

In this section, we present our new User Interface (UI) Façades. UI Façades supports the rearrangement of UI elements, the modification of the interaction, as well as several other capabilities. Each of these will be discussed in turn in the next sections.

Initial concept: cutting, copying and pasting arbitrary interactive screen regions

A basic functionality of UI Façades is the ability to rearrange UI elements. For this, we employ the paradigm of cut, copy & paste. More precisely, the user can select arbitrary rectangular regions in a special mode, which is activated via a modifier key or by selecting a special option in a window manager menu. The user can then copy (i.e. duplicate) or cut (i.e. remove) these selected UI elements and paste them into a UI Façade window. UI elements can come from arbitrary windows and can be pasted into UI Façade windows. New UI Façade windows are created simply by pasting a UI element onto the desktop (background). Alternatively, a left click on a Façade selection pops up a menu which proposes different actions on the

selection (the system does not send mouse events under a selection in this mode).

Figure 3 shows a user constructing a façade from four different regions (three on the left image, one on the right one, which the user has selected one after the other). Each region appears as an overlaid transparent gray rectangle.

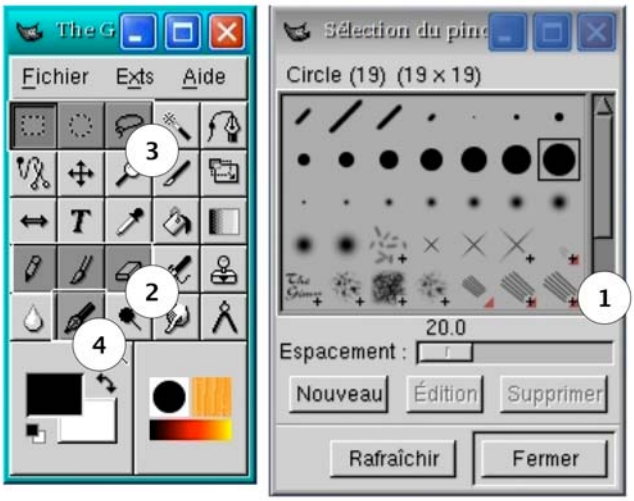

Figure 3. Constructing a façade from screen regions, 3 regions have been selected from the left dialog, one from the right.

Figure 4 shows the façade created from the four regions selected in Figure 3. A new toolbox is build from two toolboxes: some tools were selected from the main GIMP toolbox, as well as the most interesting parts of the pencil dialog. Once the new toolbox is created, the two other windows can be hidden to save screen space.

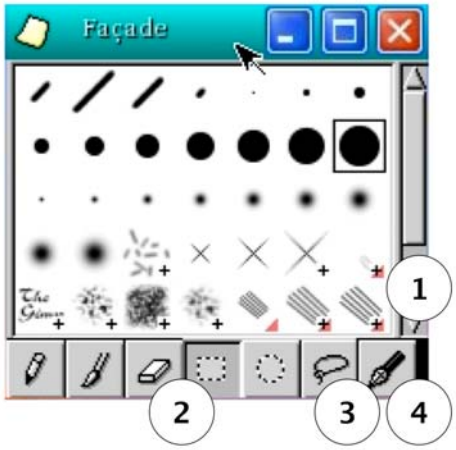

Figure 4. UI Façade constructed from the elements marked in Figure 3

After the UI Façade has been created, the user can hide (or iconify) the original windows and the system transparently passes mouse movements and clicks to the underlying application. Updates to the window content, by the application are duplicated into the UI Façade. Similarly overlay window such as menus appear in the Façade. The system also transparently manages the focus according to standard window manager rules. In effect, the behavior of an UI Façade is indistinguishable from the underlying application. A screen region can be used in as several UI Façades in parallel and a UI Façade can contain an arbitrary number of regions. In this instantiation of UI Façades the Façade cannot be resized, as this would require modifications to the underlying GUI toolkits, but an enhanced version described below solves this problem.

Additional UI elements can be added to an UI Façade using drag-and-drop. If there is a free space in the UI Façade window, an element is simply added (enlarging the window if necessary). If there is no free space, the new element is added to the side of the window where the cursor was when the drop occurred. I.e. if the cursor was near the top of the UI Facade window, the window is enlarged appropriately and the UI element is added at the top of the window. See Figure 5 for an example.

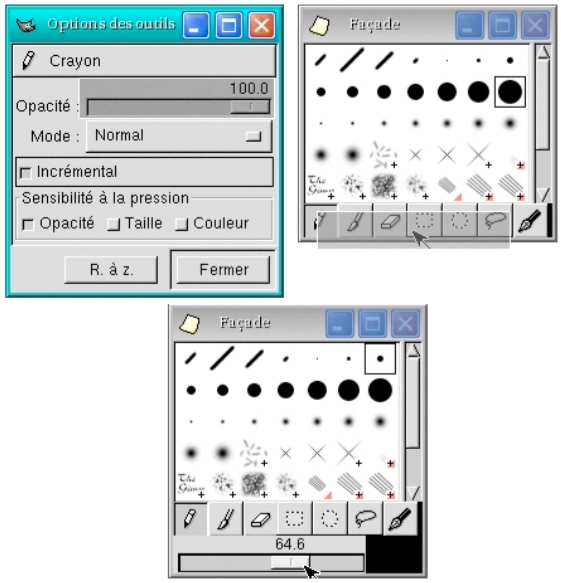

Figure 5. Selecting another screen region (the opacity slider, top left) and then dropping it into an existing UI Façade (top right) enlarges the Façade (bottom image)– see text for details.

If there is little or no desktop space available, UI Façades can also be created by first selecting several regions and then instantiating a new UI Façade window that contains all active selections with a special key combination.

To enable quick recall of a UI Façade, the user can save a UI Façade and give a name to it (via a Save Façade menu item of the window menu of the Façade). The system uses the geometry, class, and resource names of the windows used by the Façade as an identifier for a UI. An entry in the desktop menu (available with a right-click on the background) gives the user the ability to recreate the UI Façade.

Implementation details

We implemented UI Façades based on Metisse [Chapuis,

Roussel04]. Metisse is a fully functional X Window system designed to easily prototype new rendering techniques as well as novel window management techniques. This system contains an X server (called the Metisse server) that can render application windows off-screen as well as a slightly modified version of a standard X window manager, FVWM. Finally, the system also contains the UI compositor also called FvwmCompositor, which communicates with both the Metisse server and FVWM. The UI compositor renders the contents of the windows of the Metisse server in a full-screen window of the native windowing system to simulate a full window system. All these components usually run on a single machine.

The Metisse server and the UI compositor communicate with a protocol similar to VNC, but at a per-window level. In particular, each time an application updates its window, the server sends the corresponding region to the UI compositor, which uses OpenGL to render the desktop using textures. This way, the visual appearance of the final screen layout can be very different from the layout on the Metisse server.

The UI compositor receives all use input. While keyboard events are simply forwarded, mouse input is transformed into the original position of each source region of the Metisse server desktop.

Besides rendering the desktop, the UI compositor also enhances the capabilities of FVWM. For example, FVWM can ask the UI compositor to perform some non standard operation, such as scaling a window. Selecting a screen region for UI Façades is implemented as a local operation in FVWM, i.e. the related events are not forwarded to the server. Also, FVWM can request that the UI compositor create a new UI Façade. Technically speaking, each UI Façade is an independent instance of a simple program which creates a new X window to hold the UI elements of the Façade and stores all associated information such as size and the source region of all its UI elements. Based on this information, the UI compositor redirects any content changes of the UI elements to the UI Façade. The UI compositor then handles any necessary focus changes as the mouse is warped to various source application windows. Special care is taken to render transient overlay windows (popup-menus, tooltips, …) in the right place. Closing a UI Façade simply destroys all these associations (and potentially closes all source applications, depending on an user-selectable option).

Interaction "Skins"

One idea that has not appeared frequently in the discussion about adaptable UIs in the literature is that one can not only adapt the visual appearance of the UI, but also the interaction part of the UI. Consider e.g. the replacement of a long dropdown list widget containing all countries of the world with a map widget showing the world or alternatively a small set of radio buttons for the short list of countries that the user needs frequently in his work.

To achieve this, the system only needs information about widgets and can then map each user action on a widget in an UI Façade to one or more actions on the widget(s) of the original application. This results in a very powerful way of modifying the behavior of applications. Other benefits of this approach is that the selection of screen regions can snap to the boundaries of widgets and it is possible to have UI Façades that resize automatically when the original window is resized.

For this part of UI Façades, information about the size, type current state, and parameters of each widget is needed. One way of getting access to this information is to use modified version of user interface toolkits such as GTK+, Xaw, Qt, Swing, etc. that reports this information to UI Façades. Once this info is available, it is fairly straightforward to write code that can activate the different functionalities of each widget (such as selecting the n^{th} entry in a list, clicking on a spinner, entering text, etc.) as dictated by the mapping of the actions on the UI Façade widget to the original, underlying widget(s). Other mappings require several actions on the underlying widgets, e.g. if a radio button Façade selects a specific entry from a long drop-down list widget it may be necessary to first "drop" the list "down" with a system generated mouse click in the right place, then scroll to the top with more simulated clicks and then scroll to and finally select the desired entry with yet more simulated clicks.

We have implemented some of the described mapping via additional code. However, it should be fairly straightforward to create a tool similar to a GUI builder that lets end-users specify new mappings by selecting pre-defined or pre-programmed actions and compositing these appropriately.

Given that the above described approach gets fairly complex if multiple toolkits and many widgets are involved, we are currently investigating a better way based on the features of the accessibility API that some of these toolkits include as these APIs include functionality to query and interact with widgets.

WORKING EXAMPLES

This section introduces several applications of and variations on the basic idea of UI Façades.

Combining UI elements

We already presented an example of the power of combining UI elements based on GIMP above. Another example is the creation of a notification UI Façade from different applications. Most e-mail programs display the "inbox" as a list of one-line items containing information on the sender, subject, etc. Selecting (part of) this list and the two last lines of an instant messaging (IM) application allows the user to compose a novel "contact" UI Façade notificator application. The advantage of such a notification application compared to the usual small notificators in the taskbar is that it gives simultaneously information on new mails and new IM messages (especially the sender name). The users can use this information to decide whether to switch from their current work to answer a message. Moreover, the

user can even answer an e-mail message at once without switching to the full mail reader window as he/she can interact with the mail application by right-clicking on an e-mail's header line. One disadvantage of such notification window is that it uses more screen space than taskbar notificators, which use very little screen space. However, Metisse has the ability to scale windows. Hence, such windows can be also scaled (e.g., by reducing the size of 30% to reduce the screen usage while maintaining readability).

Duplicating interface elements

Another interesting application of UI Façades is to change the UI of a software package designed for right-handed people into a left-handed version, e.g. by moving the scrollbar from the right to the left-hand side. Another interesting idea is to duplicate a toolbar on both sides of the work area (or even on all four sides), which has the potential to significantly decrease average tool selection time. The next figure (Figure 6) shows a file browser (Konqueror) with an additional toolbar on the bottom.

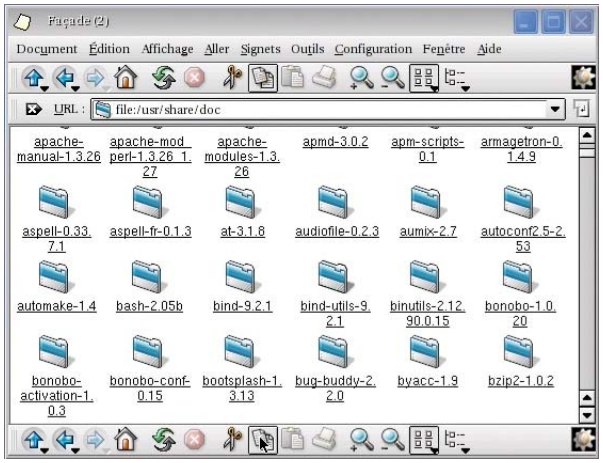

Figure 6. Filebrowser with duplicated toolbar (at the bottom).

UI Façades also support the full duplication of whole windows, similar to the work presented in [Hutchings and Stasko03, Hutchings and Stasko05]. This functionality is activated via a menu available by clicking on the title-bar of a window and short-cuts. Duplication can be extremely useful in a multiple monitors setting as it allows the user e.g. to duplicate the task bar or a panel with launch buttons on every monitor (with changes visible everywhere simultaneously). Another application of this idea is best illustrated with an example: Alice has two monitors on her desk, a laptop monitor and an external monitor, which can be turned in any direction. Paul arrives in Alice's office and sits down on the other side of the desk. Alice turns the external monitor so that it faces Paul and duplicates her web browser onto the external monitor. Alice can then freely show her work while Paul is able to observe the demonstration.

Another interesting example is the duplication of the GIMP toolbox window: one toolbox window can be created for each drawing window. We can even have two toolbox windows on each side of a drawing window. Figure 7 illustrates such a layout.

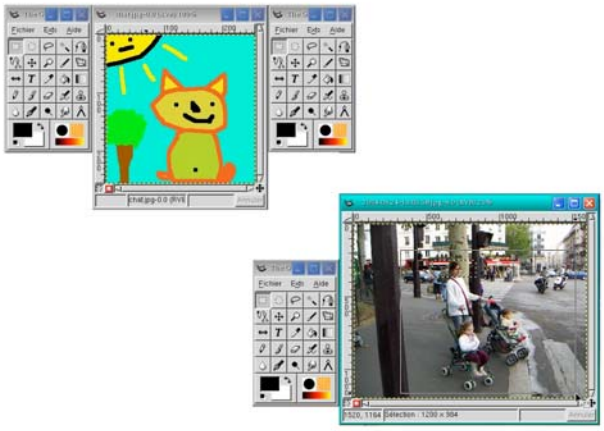

Figure 7. Screen layout with duplicated toolboxes.

Many enhanced window managers support a large virtual desktop and/or multiple independent desktops. This lead to the introduction of a "sticky" window: a window which appear on all desktops (typically a taskbar or a application panel). Some window managers allow this for arbitrary applications. One classical use of this concept is a user wanting to work with two windows which "reside" on different desktops. By "sticking" one of them the user can simply move to the desktop of the other window and have both available. After having finished working with these two windows, the user can simply "unstick" the window to restore the original desktop layout. The above interaction has a few problems. It is necessary to unstick the second window as its presence on other desktops can be annoying. Moreover, when moving to the "working" desktop it may happen that the two windows overlap and one window must be moved. This may breaks the position of the sticky window relative to its original desktop, which can be problematic. Window duplication provides a better solution. Instead of "sticking" the window the user duplicates it and moves it to the desktop of the others window. The duplicate window can then be moved without disturbing the position of its original and do not appear on others desktops. The duplicate window can also simply be iconified or closed in stead of "unsticking".

Yet another application of UI Façades is the control of applications on secondary display devices. The main issue here is the reduction of mouse travel across large distances. We describe a two monitor scenario that significantly extends an example from a technical report of Hutchings and Stasko [Hutchings and Stasko03]. Paul is a web developer and he edits a web page on his main monitor. On his secondary monitor he runs two different web browsers to test the result of his work in real time. For this Paul first creates an UI Façade consisting of the two reload buttons and the two vertical scrollbars of the browsers. Then he places this UI Façade in his main monitor just to the right of the web editor. This allows Paul to quickly test his design, while his

mouse never needs to leave the main monitor. Figure 8 illustrates this.

Figure 8. Example with 2 monitors, page source code and a UI Façade containing duplicated reload buttons and scrollbars on left image, the two target WWW browsers on the right image. The user can control everything from the left monitor.

Integrating Elements from One Application into Another and Holes

Another application of UI Façades is to duplicate useful notification areas into the area of an arbitrary window. As an example consider duplicating the clock from the taskbar into the title bar or another unused area of a window. This is clearly interesting for multi-monitor setups and for full-screen applications. Figure 9 shows an example of this.

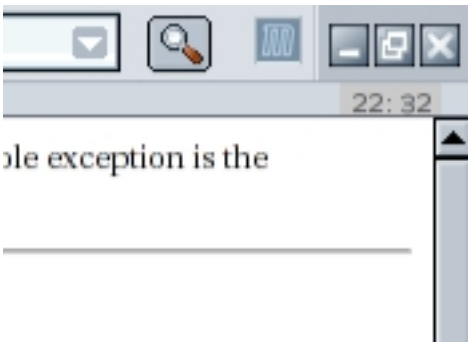

Figure 9. Duplication of taskbar clock into an unused area of Mozilla (near top right).

UI Façades provides also the ability to let the user cut a 'hole' into a window to expose underlying content. The user can do this by selecting a screen region and cutting the region. This idea was first proposed by Hutchings [Hutchings and Stasko04] and is also implicitly mentioned in the description of Ametista [Roussel03]. Hutchings suggested that holes can be also used to hide part of a window from the user's view. Another way to use holes is to access a utility application beneath another, e.g. the input/output field of a calculator beneath an unused region of a text editor (e.g. an empty part of a menu- or toolbar). The advantage of this scheme is that the calculator can be accessed via keyboard interaction while the mouse is over its area without visually blocking any part of the text (using X windows focus semantics). This is not possible with normal window managers as the calculator application is much larger than the input/output field and will obscure other things. See Figure 10 for an example image.

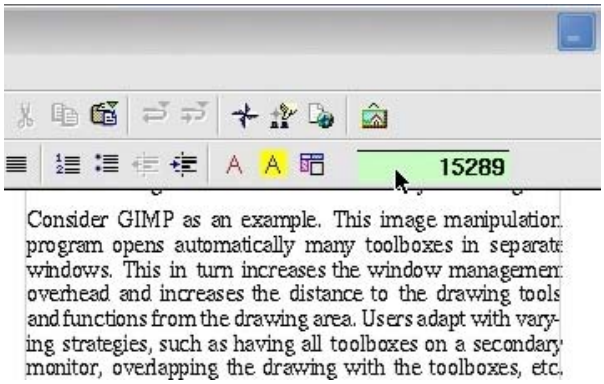

Figure 10. UI Façade simulating a text editor with a "built-in" calculator.

This is especially interesting if the text editor is run in a maximized/full-screen mode. Furthermore, it is possible to raise the complete calculator window by clicking in the hole. As usual, a click on the word processor will restore the previous state, i.e. the state depicted in Figure 8. Note that this does not necessitate switching the word processor out of its maximized view! In UI Façades holes can be deleted via a title-bar menu or a keyboard short-cut.

ENVISIONED EXAMPLES

The examples described below in this section are not yet supported by our implementation of UI Façades. However, our current explorations with the accessibility API of GTK+ and Apple's Cocoa convinced us that the following examples are relatively simple to implement, given enough time.

Transforming palettes into Toolglasses

Some previous research has shown that toolglasses can improve user performance [Kabbash 94]. Toolglasses are transparent UI elements, whose position is controlled by the non-dominant hand. The user then "clicks-through" the desired mode-icon of the toolglass with the dominant hand to activate the function at the current cursor location.

This is realized in UI Façades when the user first selects a screen region and then activates a special key sequence to designate the region as a toolglass. Then, whenever the user clicks the screen with the input device in the non-dominant hand, the toolglass is blended on top of the application and clicks with the dominant hand in the region of the toolglass are both forwarded to the palette (to activate the correct mode) as well as the underlying drawing/text-editing area. One of the attractive features of UI Façades is that no change to the underlying application is necessary to fundamentally improve the user interface.

Follow-me guys!

As an alternative to toolglasses we envision another way to have mode-switching widgets within easy reach. For this, the user first selects several regions corresponding to en-

tries in a palette or toolbar. After the use of another special key combination, the system will then keep the selected widgets near the cursor (more precisely not closer than a preset distance), similar to fans following a star or flocking behavior in animals. Quick mouse movements then temporarily "scatter" the widgets, which allow seeing hidden content, while slower movements allow the user to interact with the mode-switching widgets. Such behavior is easily implemented with a simple flocking algorithm, but the constants need careful tuning to keep this technique unobtrusive.

Replacing min/max sliders with an alpha slider

Tanin *et al*. [Tanin96] introduced the concept of range sliders, which is similar to a normal slider, but where the width of the slider itself signifies a minimum/maximum range. With the mechanism of interaction skins, UI Façades can implement this by mapping interactions with a range slider widget to the appropriate actions on a pair of numeric entry fields. The only enhancement that is needed relative to previously described things is the ability of mapping clicks on various regions of the range slider to numeric values.

GUI Tool for Modifying the Interaction

One important motivation for the work on UI Façades is to provide end-users with the ability to adapt the interaction of an application, i.e. modify its behavior. As discussed above, a GUI tool to specify the mapping from user actions on widgets in an UI Façade to actions on the application's widgets would bring us fairly close to the overall goal of easily adaptable UIs. We already discussed most technical issues for this in the section on interaction skins. One issue that remains to be explored in future work is the creation of a simple-to-understand description of widget actions – at a level that even end-users can understand easily.

CONCLUSION AND FUTURE WORK

We have presented a new implementation of adaptable user interfaces. Our UI Façades allow end-users to change the UI of any application as they desire via the cut, copy and paste paradigm, combined with the ability to drag screen regions into a Façade. Furthermore, our approach also allows the user to adapt the interaction behavior of arbitrary applications, something that no previous work has allowed. We presented several examples that demonstrate and extend the basic concept in several interesting directions (window management, multiple monitors, etc.). Furthermore, we discussed several future extensions, which we believe to be relatively easy to implement.

From a global perspective, we believe that UI Façades offer a good complement to direct programming of UIs. From the user's view UI Façades greatly increase the flexibility of an application. From the programmers view, UI Façades are transparent as no programming is required to give the user the ability to change the user interface. In the future, appropriate APIs to UI Façades may even enhance the UI programmer's/designer's ability to create good user interfaces.

For future work, we plan to investigate the integration of many of the presented techniques into UI toolkits and/or window managers, as this is the logical next step. In this context it is interesting to realize that UI Façades extends Apple's vision of the "Window system as a digital image compositor" [Graffagnino02]. More precisely, we can define that the addition of UI Façades to the standard window management/UI paradigm allows us to talk about the vision of the "window system as an interactive graphical component compositor".

The generalization from rectangular regions to more arbitrary regions is fairly simple from a high-level point of view and may increase the utility of UI Façades even further.

REFERENCES

1. Baudisch, P., Cutrell, E. and Robertson, G. High-Density Cursor: A Visualization Technique that Helps Users Keep Track of Fast-Moving Mouse Cursors. Proceedings of Interact 2003, pp. 236-243.

2. Bentley, R. and Dourish, P. Medium versus Mechanism: Supporting Collaboration through Customisation. Proceedings of ECSCW'95, pp. 133-148. Kluwer Academic.

3. Chapuis, O. and Roussel, N. Metisse is not a 3D Desktop!. LRI research report, Université Paris-Sud. 2004, submitted for publication.

4. Chatty, S., Sire, S., Vinot, J-L., Lecoanet, P., Lemort, A. and Mertz, C. Revisiting visual interface programming: creating GUI tools for designers and programmers. Proceedings of UIST 2004, pp. 267-276. ACM Press.

5. Edwards, W., Hudson, S., Rodenstein, R., Smith, I. and Rodrigues, T. Systematic Output Modification in a 2D UI Toolkit. Proceedings of UIST 1997, pp. 151-158. ACM Press.

6. Findlater, L. and McGrenere, J. A comparison of static, adaptive, and adaptable menus. Proceedings of CHI 2004, pp. 89-96. ACM Press.

7. Fogarty, J., Forlizzi, J. and Hudson, S. Specifying Behavior and Semantic Meaning in an Unmodified Layered Drawing Package. Proceedings of UIST 2002, pp. 61-70. ACM Press.

8. Fujima, J., Lunzer, A., Hornbæk, K. and Tanaka, Y. Clip, Connect, Clone: Combining Application Elements to Build Custom Interfaces for Information Access. Proceedings of UIST 2004, pp. 175-184. ACM Press.

9. Graffagnino, P. Apple OpenGL and Quartz Extreme. Presentation at SIGGRAPH OpenGL BOF, 2002.

10. Hutchings, D. and Stasko, J. Revisiting Display Space Management: Understanding Current Practice to Inform Next-generation Design. Proceedings of Graphics Interface 2004, pp. 127-134. Canadian Human-Computer Communications Society.

11. Hutchings, D. and Stasko, J. An Interview-based Study of Display Space Management. GVU Technical Report GIT-GVU-03-17, May 2003.

12. Hutchings, D. and Stasko, J. Shrinking Window Operations for Expanding Display Space. Proceedings of Advanced Visual Interfaces 2004, pp. 350-353. ACM Press.

13. Hutchings, D. and Stasko, J. mudibo: Multiple Dialog Boxes for Multiple Monitors. To appear in CHI 2005 Extended Abstracts. ACM Press.

14. P. Kabbash, W. Buxton & A. Sellen. Two-Handed Input in a Compound Task. Proceedings of CHI '94, 417-423.

15. Kantorowitz, E. and Sudarsky, O. The Adaptable User Interface. Communication of the ACM, 32(11), pp. 1352-1358, November 1989. ACM Press.

16. Kühme, T. A User-Centered Approach to Adaptive Interfaces. Proceedings of the 1993 International Workshop on Intelligent User Interfaces, pp. 243-245.

17. McGrenere, J., Baecker, R.M. and Booth, K.S. An evaluation of a multiple interface design solution for bloated software. Proceedings of CHI 2002, pp. 163-170. ACM Press.

18. N. Roussel. Ametista: a mini-toolkit for exploring new window management techniques. In Proceedings of CLIHC 2003, pages 117 124. ACM Press, August 2003.

19. Tan D.S., Meyers B. and Czerwinski M. WinCuts: Manipulating Arbitrary Window Regions for More Effective Use of Screen Space. CHI 2004 Extended Abstracts, pp. 1525-1528. ACM Press.

20. Tanin E., Beigel R. and Shneiderman B. Incremental data structures and algorithms for dynamic query interfaces. SIGMOD Rec. 25 (4), pp. 21-24, 1996.