User Interface Façades: Towards Fully Adaptable User Interfaces

Wolfgang Stuerzlinger[†], *Olivier Chapuis*[*], *Dusty Phillips*[†] & *Nicolas Roussel*[*]

[†]Interactive Systems Research Group
Comp. Science & Engineering, York University
Toronto, Canada
wolfgang | dustyp @cse.yorku.ca

[*]LRI (Univ. Paris-Sud - CNRS) & INRIA Futurs [1]
Bâtiment 490, Université Paris-Sud
91405 Orsay Cedex, France
chapuis | roussel @lri.fr

ABSTRACT
User interfaces are becoming more and more complex. Adaptable and adaptive interfaces have been proposed to address this issue and previous studies have shown that users prefer interfaces that they can adapt to self-adjusting ones. However, most existing systems provide users with little support for adapting their interfaces. 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 users with simple ways to adapt, reconfigure, and re-combine existing graphical interfaces, through the use of direct manipulation techniques. The paper describes the user's view of the system, provides some technical details, and presents several examples to illustrate its potential.

ACM Classification: H.5.2 [Information interfaces and presentation]: User interfaces - Graphical user interfaces.

General terms: Algorithms, 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 any given time [19], 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 variety of different users. This makes creating flexible user interfaces a major challenge.

Consider GIMP as an example. The latest version of this image manipulation program has 22 persistent *dialogs* for managing brushes, colors, fonts, etc. Although dialogs can be docked together in an arbitrary number of windows, this only increases the window management overhead and increases the average distance to the drawing tools & functions from the drawing area. Users adapt with various strategies, such as having all dialogs on a secondary monitor, or overlapping the drawing area with dialogs. On the other hand, some applications use an all-in-one window logic, which provides less flexibility in terms of user interface layout.

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 user interface. These two concepts have been studied for some time by the community (e.g. [17, 18]). Today, they are most often referred to as *(user-)adaptable* and *adaptive* (or *self-adapting*) interfaces [19]. 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. Many applications, for example, make it possible to interactively customize their toolbars with simple drag-and-drop operations.

Adaptive interfaces can exhibit some unpleasant side effects such as surprising the user by moving or removing menu entries. Previous studies have also shown a desire for the user to be able to control and override the automatic system whenever needed [11]. Adaptable interfaces suffer from the problem that new 'secondary' interfaces and interaction techniques must be added to support the customization of the 'primary' interface. A

UIST'06, October 15–18, 2006, Montreux, Switzerland..

[1]projet In Situ, Pôle Commun de Recherche en Informatique du plateau de Saclay

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 [8]. 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.

While the idea of adding adaptation functionality to user interface toolkits seems attractive at first glance, it has the drawback that it will make the already complex APIs of these toolkits even more complex, requiring yet more code to be written by application programmers. This is clearly not a positive thing and would not speed adoption of the fundamental paradigm of adaptable interfaces. Moreover, modifying the toolkits would leave it to programmers or interface designers to decide what can be configured and how. Yet, again, these professionals cannot necessarily foresee all potential ways of adapting an application. Phrased differently, we believe that users should be in control of the adaptation process, not the original software authors.

In this paper, we present User Interface Façades, a system designed to address this issue. The rest of the paper is organized as follows. In the next section, we present an overview of previous work and motivate our research. After presenting the main ideas of User Interface Façades, we discuss how we implemented them. Then we present several examples to illustrate the concepts, followed by the conclusion.

MOTIVATION

Skins and *themes* are two of the simplest forms of user interface customization. The notion of a *skin* comes from video games such as Quake that allow players to alter the appearance of their character and has been adopted by many media players. *Themes* extend this notion by sharing a common visual style among different applications, as specified by the user at run time. A skin, or a theme, can simply consist of a set of colors or textures used by existing drawing code. It can also partially or completely replace that drawing code, possibly adding complex output modifications [7]. In addition to the visual style of interface elements, skins and themes can also specify the layout and to a lesser degree the behavior of these elements. Recent work has extended this approach to bridge the gap between appearance and semantic meaning [9, 6]. However, although these 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 only choose between predefined theme options.

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 is far from optimal, as it does not allow for undoing changes or reverting to previous versions and employs a fixed window, which is inconvenient in many situations. Microsoft Office applications also allow users to customize their various toolbars and menus. But again, the customization interface has a number of serious flaws (Figure 1).

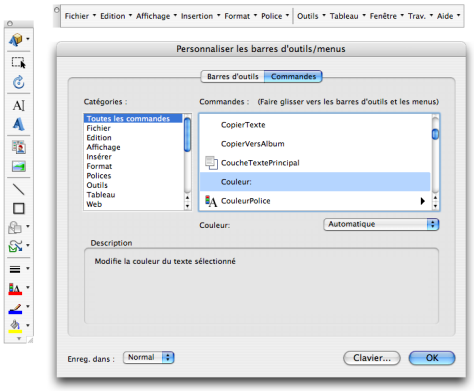

Figure 1: Microsoft Word 2004 interface for customizing menus and toolbars. The left list contains 22 command categories. The right list shows the commands relevant to the selected category. More than 1100 commands are available through this interface. They can be dragged to/from menus and toolbars, but these operations cannot be undone. Commands already in menus or toolbars still appear in the list. The window is about 600x500 pixels, can be moved, but not resized.

Bentley and Dourish [3] introduced an interesting distinction between *surface customization*, which allows users to choose between a predefined set of options, and *deep customization*, which allows them to customize deeper aspects of a system, such as integrating an external translation program with a word processor. They point out two problems that our above examples also illustrate. First, the level of customization provided by most systems lies above the functionality of the application, rather than within it. Second, these systems often require the learning of new languages to describe new behaviors.

Fujima et al. recently proposed the C3W system (Clip, Connect and Clone for the Web) to generate new HTML documents by cloning individual HTML elements from other documents and allowing for computation on these elements using a spreadsheet model [10]. While this approach supports deep customization, C3W is limited to Web technologies and does not allow the user to change or replace widgets nor to add new widgets to existing documents. Hutchings and Stasko proposed the more generic notion of *relevant window regions* and suggested to add the ability to create copies of these regions that could be manipulated as independent windows [14]. Tan et al. implemented this idea in their

WinCuts system [22]. However, this system is unable to merge several regions into a new window, which is clearly a limiting factor. Its implementation also has several problems that make it hardly usable on an everyday basis (e.g. it relies on periodic polling of window content, popup menus and dialog boxes appear on the source window, etc.). Berry et al. introduced a system that can selectively hide content based on the users' privileges via various forms of blurring [4]. Internally, this system works similarly to WinCuts.

Hutchings and Stasko also suggested allowing users to remove irrelevant parts of windows [14]. The same idea was mentioned in [21] and partially implemented (windows could be cropped to a set of pre-defined shapes). Finally, Hutchings and Stasko proposed to replicate dialog boxes on multiple monitor configurations until the user interacts with one of the copies [15]. In this same paper, they concluded that window operations like these should be implemented within the window manager rather than using a separate application.

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

- *Fast, simple, just-in-time customization*: Users should be able to adapt interfaces without advance planning, whenever needed, and should be able to do this in a fast and simple way, e.g. with direct manipulation techniques.
- *Not only global customizations, but also local ones*: Most adaptable interfaces only support global changes, which forces users to undo them at some point. Global/local can be interpreted in different ways (e.g. persistent/temporary, all documents/this document). Users should be able to specify the scope of interface customizations. It should be possible, for example, to customize the toolbars of an application for a specific session only, or even for a specific document.
- *Deep customization*: Users should not be restricted to a set of pre-defined options but should be able to define new ones. Again, 'set of options' can be interpreted in different ways, e.g. a tool set or a set of specific locations where tools can be placed. Users should be able to select anything on the screen, change the way it operates (not only visual appearance), cut it out, duplicate it, or replace it with something else. The latter should be done in a manner that removes the 'old' user interface, or at least makes it invisible.
- *Cross-application customization*: Interface customizations should make it possible to combine or link together different applications.

USER INTERFACE FAÇADES

This work focuses on applications with a graphical user interface, as opposed to command-line systems. We are more specifically interested in applications where the interaction focus is a single or, at best, a few document(s). In such applications a large work area dominates the main window, with user interface elements clustered around. Examples include drawing packages, text processors, spreadsheets, etc.

A user interface *façade* is a user-specified set of graphical interfaces and interaction techniques that can be used to customize the interaction with existing, unmodified applications. This section provides a general overview of how users interact with such façades. Implementation details and more specific usage scenarios follow in the next two sections.

Copying and pasting screen regions

A basic functionality of the Façades system is the ability to copy interface components from one window to another while maintaining a one-to-one functional relationship between the copy and the original. Using the mouse and a specific modifier key the user can select one or more rectangular *source regions*. A drag operation on these regions duplicates them. Dropping the duplicates on the desktop puts them in a new *façade window*. Façade window creation from source regions is also accessible through a menu that pops up when one clicks on one of the regions using the right mouse button. A new command also makes it possible to clone a complete window through its standard window menu.

Dropping duplicated interface components onto the side of a façade window automatically expands the façade to make room for the new duplicate at this side. Dropping components into free space inside a façade window simply adds it in that space. Duplicates can also be dropped on any existing window, and will overlay the dropped component over the existing content. Figure 2 shows a user incrementally constructing a façade window by selecting widgets from three dialogs of the GIMP application. The scenario here is that the user wants to optimize the interface by packaging frequently used tools in an ad-hoc way, rather than using the GIMP developers' pre-packaged toolsets. The upper row of images shows four selected regions in two GIMP dialogs (displayed as semi-transparent rectangles) and the resulting façade window, which contains the duplicated regions. The lower row illustrates the addition of a fifth duplicated component to this window.

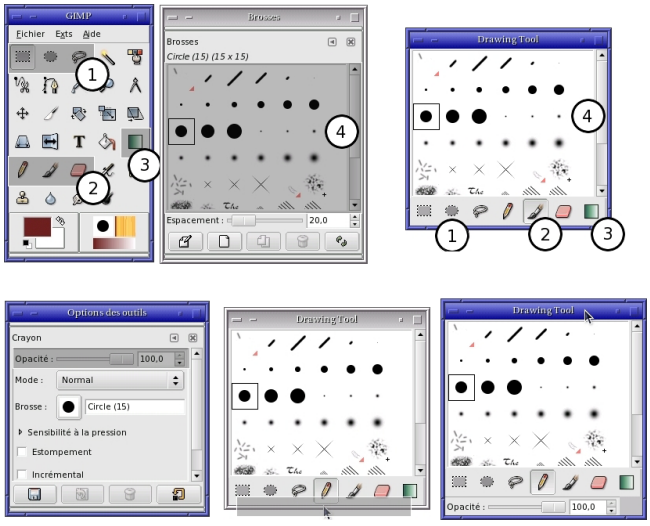

Figure 2: Creating a façade from several GIMP dialogs.

The same source region can be used in several façades (i.e. it can be duplicated several times), and a façade can contain an arbitrary number of duplicates. After a façade has been created, the user typically hides or iconifies the source window(s) and the system transparently passes mouse movements and clicks over the façade to the appropriate source region. Conversely, source region updates are replicated in their corresponding duplicates. Overlay windows such as popup menus are correctly handled when triggered from a duplicate. The system also transparently manages the focus and stacking order according to standard window manager rules. In effect, the behavior of a duplicate is indistinguishable from the original source region to the user.

Parts of the above ideas have been previously presented by Tan et al. [22] (e.g. the ability to duplicate multiple screen regions into individual windows) and Hutchings and Stasko [14, 15] (e.g. the ability to duplicate windows). However, the ability to create new windows that seamlessly combine multiple screen regions and the ability to paste regions over arbitrary windows are unique to our work.

Cutting screen regions

In addition to supporting the creation of façade windows, the system also allows users to create holes in windows, via a context-sensitive menu that becomes active after a region on the screen has been selected. This can be used to remove uninteresting parts or to reveal other windows beneath. As an example, consider revealing a small utility, such as a calculator or calendar, inside an unused region of a primary application (Figure 3). As the keyboard focus follows the mouse position in Façades, the user can then simply interact via the keyboard with the partially covered calculator 'through' the hole. This is especially interesting if the primary application is run in full-screen mode, which is something that traditional window systems do not support. Holes created in a window with the Façades system can be deleted via a command in the window menu or with a keyboard shortcut.

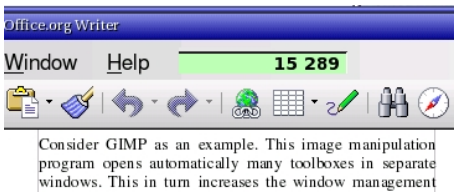

Figure 3: Accessing a calculator through a hole in a word processor.

Using external components to interact with applications

One idea that appears rarely in the discussion about adaptable user interfaces in the literature is that the user cannot only adapt the visual appearance of the interface, but also the *interaction* part of it. Façades allows the user to do this without any change to the code of the underlying application. One possible modification is to replace a component of a GUI with another GUI component, typically created by a third party. For example, with Façades the user can replace a dropdown list widget containing all countries of the world with a map widget or alternatively some radio buttons for the small set of countries that the user needs frequently in his or her work. Another modification allows the user to modify the interaction with standard components. For example, the user can integrate scrolling and zooming by remapping how mouse movements on a standard scroll bar are interpreted. These and other examples will be discussed in more detail later in the paper.

Managing Façades

To enable the quick recall of a façade, the user can give it a name and save it through a specific command in the window menu. When saving, the user can set options in a dialog: automatic recall, automatic hiding of source windows at recall time and the use of the window title in the saved description of the façade.

At a later time and if all relevant windows are open, the system can then recreate a façade automatically, or on user demand via the window menu. For this, Façades monitors all window related events and identifies matching configurations via window geometry, class, and resource names. If applicable, replacement widgets are automatically instantiated. A sub-menu of the normal desktop menu also contains a list of all saved façades for all currently active window configurations.

Contributions

In summary, we present the following new techniques for adaptable user interfaces:

- Seamlessly merge duplicated screen regions into new windows enabling the creation of new user interfaces for existing applications.
- The ability to create holes in windows and to seamlessly overlay duplicated content over existing windows.
- The ability to seamlessly replace widgets with other (potentially customized) widgets.
- The ability to seamlessly change the interaction with widgets, including the composition of widget behaviors, as well as the creation of toolglasses and other advanced user interface techniques.
- Implementing all of the above in a way that does not require any coding, with a simple-to-use interface based on drag-and-drop.

The following implementation section provides the technical details that make the system efficient and reliable and discusses related issues such as resizing.

IMPLEMENTATION DETAILS

In this section we describe how we implemented Façades and how it is integrated into a windowing system. Conceptually, Façades acts as a transparent layer over the window system that redirects input events and duplicates window regions as specified by the contents of each façade window. For seamless duplication it uses the off-screen buffer capabilities of Metisse [5], as well as its input redirection facilities. Façades determines widget

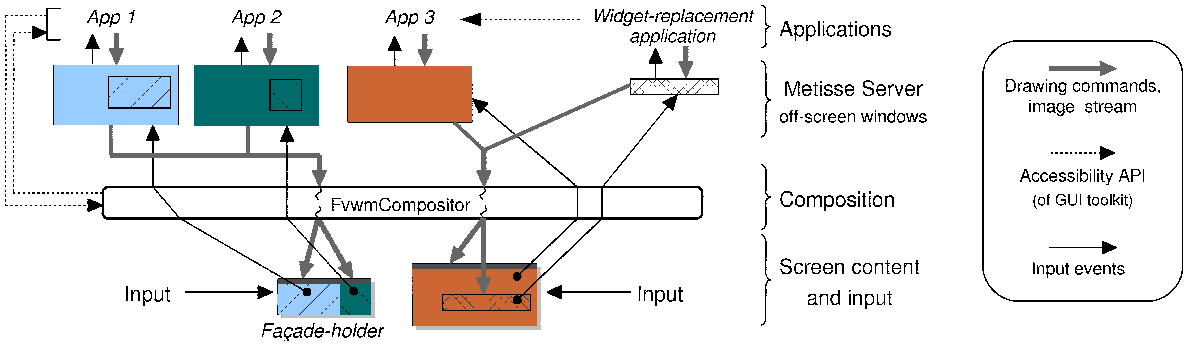

Figure 4: Illustration of input event and image flow in Façades system

positions through the accessibility API of modern GUI toolkits. Finally, widget replacement and interaction modification is achieved via the instantiation of simple replacement applications that are again based on accessibility API calls. Figure 4 illustrates how the various components of Façades work together. The left hand part shows a façade that composites two separate windows, whereas the façade for 'App 3' utilizes widget replacement. In the following subsections we first discuss how input & output are redirected and then mention how we access and replace widgets.

Basic input/output management using Metisse

Façades is implemented based on Metisse [5]. The Metisse architecture uses a compositing approach, making a clear distinction between window rendering and the interactive compositing process. The *Metisse server*, an enhanced X Window server, renders applications off-screen. In Façades, window images are composited by a separate application, *FvwmCompositor*, which is based on the window manager FVWM. Mouse and keyboard events received by FvwmCompositor are usually sent to appropriate applications through the Metisse server. In some cases, however, events are directly handled by FvwmCompositor itself, e.g. to implement façade region selection and window management commands, such as 'Alt-F4'. Specific façade commands in FvwmCompositor are accessible from FVWM to enable the creation of façade windows, holes, etc. Conversely, FvwmCompositor uses FVWM commands to handle pop up menus or to indicate the real mouse focus when the pointer is over a duplicate.

Each façade window is managed by an instance of a simple program, *façade-holder*, that keeps track of the duplicate regions it contains and creates a new X window to hold them (duplicates are then displayed by FvwmCompositor in that window). This program is invoked each time one or more duplicates are dragged from a source window and dropped onto the desktop. Each duplicate is described in *façade-holder* by a tuple of the following form: $(XID, src_x, src_y, src_{width}, src_{height}, dst_x, dst_y)$ where XID identifies the source window, $(src_x, src_y, src_{width}, src_{height})$ specifies the original region geometry relative to the source window, and (dst_x, dst_y) specifies its position in the façade window.

Façade-holders publish these tuples to other programs, including FvwmCompositor, through an X atom[1]. When a new duplicate is pasted into an existing façade window, FvwmCompositor sends an X client message with the source information for the duplicate to the *façade-holder*. Upon receiving this message, the *façade-holder* computes the local geometry of all its elements and updates its atom accordingly. FvwmCompositor catches this new layout and redraws the façade window.

FvwmCompositor maintains a list of duplicated regions for every window and handles updates for every content change. It also handles the necessary focus changes as the mouse moves from one duplicated region to another. Mouse and keyboard events for a façade window are normally sent to the appropriate source window. Similarly, clicking on a duplicate region raises the façade window, not the corresponding source window. FVWM handles these situations by distinguishing two types of focus: one for window management tasks, and the other for interacting with window content.

Transient overlay windows, such as popup menus or tooltips, are rendered in the right place. When such a window is mapped, FvwmCompositor computes its 'parent' window, i.e. the source window that is most probably responsible for this new window to appear. If the mouse pointer is over an element of the parent, FvwmCompositor positions the overlay based on the parent location and the element position and geometry. If the parent window is invisible, the overlay window is placed close to the pointer. Transient dialogs are placed so that their center is aligned with their façade window.

Iconification of source windows also poses specific problems. The usual way of iconifying X windows is to 'unmap' them in the server and replace them with a new graphical object. But unmapped windows do not get redrawn and cannot receive events. Consequently, when a source window is iconified in Façades, it is not unmapped, treated as iconified by FVWM and not rendered by FvwmCompositor. When a source window is closed, FvwmCompositor notifies the corresponding façades by sending them an X client message that specifies the region(s) to be removed. When its last element is removed, a façade either remains empty on-screen for

[1]Atoms are an X Window specific publish/subscribe mechanism

later use, or is automatically destroyed in the case of cloned windows. Façade and cloned windows are not resizable by the user. Cloned windows are automatically resized to match the geometry of their source window. Duplicated regions are kept visible in façades only if they are still visible in their source window.

All menus to manage façades are handled by FVWM. Some are statically defined in configuration files. Others are dynamically created by FvwmCompositor (e.g. the list of previously saved façades for a window). Saving a façade generates a human-readable description of its elements on disk. FvwmCompositor uses the geometry, class, resource names, and optionally the title of the source windows of a façade to create a heuristically-unique identifier. Widget-related information obtained from accessibility APIs can also be used to make this identifier more robust. FvwmCompositor loads all saved façade descriptions at startup and whenever windows are created or resized, it checks for matching façade descriptions and creates them accordingly.

Taking advantage of accessibility services

Widget-related information is very useful for creating façades. Knowing the position, size, type, and current state of each widget as well as having access to its actions offers a number of interesting possibilities. As an example, knowing the boundaries for each widget can facilitate the selection of widgets via snapping. There are several ways to obtain widget-related information and control widgets from the outside. In the current implementation of Façades, we use the accessibility APIs supported by most modern GUI toolkits.

Apple defined Universal Access APIs for its Carbon and Cocoa toolkits, Microsoft the Microsoft Active Accessibility & System.Windows.Automation frameworks, and X Window the Assistive Technology Service Provider Interface (AT-SPI), a toolkit-neutral way of providing accessibility services supported by GTK+, Java/Swing, the Mozilla suite, StarOffice/OpenOffice.org and Qt. All of these APIs can query the current position, size, type, and state of all widgets of an application. Furthermore, all possible widget actions can be activated via these APIs (e.g. one can cause selection events, trigger buttons, etc.). The following pseudo-code segment illustrates this for the example shown in Figure 9 via the AT-SPI accessibility API.

```
# Event handler for click at (x,y) on map
# Input: x, y, app_name (application name),
#        comp_name (widget name), comp_type (widget type)

# Map click to combobox list index
index = get_province_for_point(x, y)

# recursively find the accessible component in widget tree
application = desktop.find_app(app_name)
comp = application.find_component(comp_name, comp_type)
# get accessible action interface object
selector = comp.queryInterface("Accessibility/Selection")

# "aaaaand: Action!": fire event to widget
selector.selectChild(index)
```

For resizing there are two issues to consider. Any GUI application may resize it's window or widgets at any time or the user can resize the façade window itself. While the Façades system can detect the first kind of resize events via the accessibility API, any *automatic* change to a façade might break the layout of the façade as constructed by the user. This is clearly undesirable. Hence, we currently warn the user in this case and require that he/she fixes the problem manually. Second, a user can actively resize a façade window. While we could search for widgets that are resizable and try to adapt the layout accordingly, this would require an easy-to-use interface for specifying widget layout. As current layout methods typically have (too) many options, this is a research topic of it's own. Hence, we currently choose to disallow resizing of façades.

Other possible implementations

We have implemented the Façades system using Metisse and the accessibility API. The Metisse compositing architecture permits dynamic rendering of interface elements and handles input redirection. Furthermore, the FvwmCompositor interprets window management activities directly, while it passes interaction with façade content to the original applications.

It should be possible to implement Façades on other systems since accessibility APIs are now widely available. Moreover, the compositing approach is available under Mac OS X, Windows Vista and X Windows. However, neither OS X nor Vista have APIs flexible enough to freely redirect rendering output. For this reason WinCuts [22], called the PrintWindow function every second to update cut contents. In X Windows the full rendering API is accessible and documented. Even though this API is very complex (compared to Metisse) it seems possible to implement the rendering redirection part of Façades with it.

For input redirection, Mac OS X and Windows have no public API. As a workaround, WinCuts [22] draws a cursor over the interface elements, and the source window is kept in front of the true cursor. X Window has the X Event Interception Extension (XEvIE), but this extension is not powerful enough. For example it is not possible to send pointer events to a window, which is covered by another. A future X extension [20] may provide enough control of input redirection to implement something similar to Façades.

There are several other alternatives to extract widget related information and to activate widgets. For non-accessible GUI toolkits, one can extract information about widgets by modifying the dynamically linked toolkit library and adding functionality that returns (part of) the current widget hierarchy state on demand. Interaction with non-accessible widgets can be simulated via appropriate mouse and keyboard input events on the appropriate areas of a widget. E.g. to enter a particular string into a text-field, the system selects the field via a simulated mouse click, selects all old text and erases it via appropriate key sequences, and then

simulates entry of the new string. Most other widgets can be controlled with similar strategies. However, this is only a temporary workaround, as most GUI toolkits have already or are being retrofitted with an accessibility API, due to the strong need to add accessibility to all applications.

Alternatively, we can implement Façades via an intermediate layer in a window system. Such intermediate layers already exist today, e.g. in the form of user interface description languages (UIDL's). These are used to describe the user interface and how it activates the functionality of the application. XUL and XAML are two recent examples. If this intermediate layer is accessible from the outside, it is possible to implement Façades as an 'UIDL filter', which selectively replaces or duplicates widgets in the UIDL stream and adapts the calls to the application as appropriate.

DETAILED EXAMPLES / USAGE SCENARIOS

In the following section we present several examples of useful façades and explain how they were created.

Widget duplication

One application of 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 two sides of the work area (or even on all four sides), which has the potential to significantly decrease average tool selection time. Figure 5 shows a file browser - Konqueror - with an additional toolbar at the bottom.

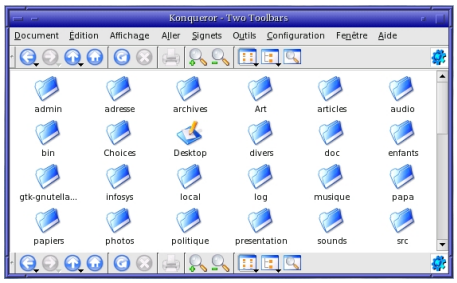

Figure 5: File browser with duplicated toolbar (bottom).

Façades also support the full duplication of whole windows, similar to [14, 15]. This functionality is activated via a titlebar menu. 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 example is the duplication of the GIMP toolbox window: toolboxes can be duplicated for each drawing window. We can even have two toolbox windows on each side of a drawing window to accelerate access to tools. Figure 6 illustrates such a layout.

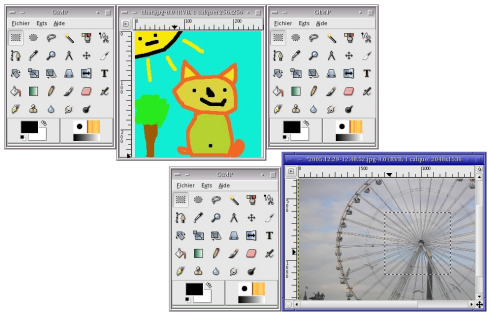

Figure 6: GIMP screen layout with duplicated toolboxes.

Another application of Façades is to duplicate useful notification areas into the area of an arbitrary window. As an example, consider duplicating the taskbar clock into the title bar or another unused area of a window (Figure 7). This is clearly interesting for full-screen applications and also for multi-monitor setups.

Figure 7: Duplication of taskbar clock into an unused area of Mozilla (near top right).

Widget duplication can also be used for 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 [14]. 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 his work in real time. For this Paul first creates a façade consisting of the two reload buttons and the two vertical scrollbars of the browsers. Then he places this façade on his main monitor just to the right of the web editor. This allows Paul to quickly test his design by interacting with the façade and has the advantage that his mouse never needs to leave the main monitor.

We already presented an example of the power of combining elements above. Another example is the creation of a notification 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' notifier façade. 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 including the sender name. Users can then 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 without switching to the full mail reader window as he/she can right-click on an e-mail's header line. One disadvantage of such a notification window is that it uses more screen space than the rather minimal taskbar notificators. However, Metisse has the ability to scale windows. Hence, such notifications can be also scaled (e.g. by reducing by 30%, which still maintains readability).

Widget replacement

Façades also targets the replacement of standard GUI widgets with other widgets. Consider a scenario where a user frequently uses a few options in a long list widget and only rarely uses other entries. A classical example is a call-center where data about each incident is recorded, and where the client base consists of many users in a small set of countries, but also a few others from around the world. Instead of having to choose every time from the list of all countries on the planet in the incident-entry form, it is much more efficient to have quick access to the subset of frequently used countries and provide a separate way to access the full list. As the call-center software developer cannot foresee which countries will be used frequently and how large that set will be, it is advantageous to give the user control of this GUI aspect. Figure 8 depicts an address entry form application for specifying addresses in Canada, the dialog that lets the user specify the provinces that appear in the façade, and the façade itself.

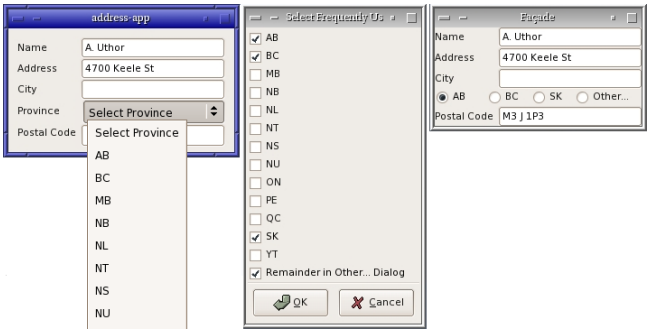

Figure 8: Original address entry application (left), the façade construction dialog, which lets the user select frequently used entries (middle) and the final façade for the application with a set of radio buttons (right).

In Façades, a user can access this functionality by first selecting a widget, then accessing a context-sensitive menu and selecting the appropriate entry. This will show a façade creation dialog with appropriate options. Once the user confirms their choice, Façades creates the custom replacement widget, which can be placed into a façade. The following pseudo-code illustrates the main parts of a generic combobox replacement widget. Code related to the dialogs for façade construction and 'Other...' functionality is not shown for brevity.

```
function combo2radio(app_name, combo_name):
  app = desktop.find_app(app_name)
  combo = app.find_component(comp_name, "combo box")
  # show dialog to user and return selected entries on close
  selection = SelectFromDialog(combo.items)
  # create a new window with the selected radio buttons
  radiobox = Window()
  for item in selection:
    radio = RadioButton(item)
    radio.bind("toggle", selectCallback, item.id)
    radiobox.add(radio)
  radiobox.display()

function selectCallback(widget, id):
  selector = widget.queryInterface("Accessibility/Selection")
  selector.selectChild(id)
```

Another option is to replace the provinces combobox in Figure 8 with an interactive map that allows direct selection of provinces in a map of Canada (see Figure 9). This is achieved via a replacement widget that maps click locations to selection events on the combo box. While this replacement widget is not as generic as the one depicted in Figure 8, it offers a better visualization, which some users may find easier to use. Depending on the user's needs, he or she may prefer one alternative or the other.

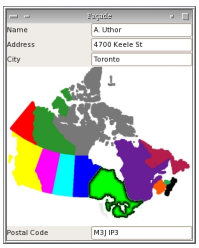

Figure 9: An alternative façade for the application from Figure 8.

As a different example for the replacement of standard widgets, consider a text-area widget and its enhanced replacement that adds syntax highlighting to make the contents easier to comprehend. With this replacement widget the user interface of *any* application with un-enhanced text-area widgets can be improved via Façades.

Similar to the shown examples, one can imagine many other replacement widgets and the code behind them will follow the general structure of the pseudo-code shown above, but tailored to the specifics of each pair of source and replacement widget. Consider e.g. enhancing an existing date entry field with an automatic pop-up calendar widget, whenever it is selected. Note however, that not all potentially possible widget replacements are 'good' from a UI designer standpoint, but this topic is beyond of the scope of this paper.

Interaction composition

Previous research has shown that toolglasses can improve user performance [16]. They 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 a function at the current cursor location. In Façades, the user can associate another (extended) input devices to a selected window or façade via the Façade window menu to create a toolglass. This causes the window to become semi-transparent and to remain always

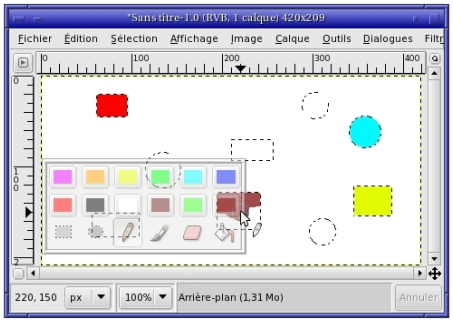

Figure 10: Using a palette façade as a toolglass.

on top over normal windows. The second input device, typically held in the nondominant hand, then controls the position of the toolglass window. Whenever the user presses a button on the device in the dominant hand, a click is sent to the toolglass (to activate the correct mode) and the press as well as subsequent drag events are sent to the window under the toolglass. This allows, for example, positioning the toolglass over the drawing area to select a tool and at the same time to start using that tool. For an illustration see Figure 10.

Moreover, a right click with the non-dominant device allows toggling between toolglass and normal window mode. This mode permits users to change tools with their non-dominant hand and to work with the selected tool with the dominant hand. We follow here the toolglass implementation of the post-WIMP graphical application CPN2000 [2]. One of the attractive features of Façades is that no change to the underlying application is necessary to fundamentally improve the user interface via toolglasses.

The OrthoZoom technique [1] combines scrolling and zooming. Mouse movement along the scrollbar direction results in scrolling, while orthogonal movements result in zooming. In addition, when the user releases the mouse button the original zoom factor is reestablished. This technique has been proven to be efficient and allows simultaneous control of scroll and zoom with a mouse. As a variation, one can map movement in the orthogonal direction to modulation of scrolling speed: the further the cursor is from the scrollbar, the slower the speed. This allows for very fine position control. Moreover, the two techniques can be combined: one orthogonal direction (e.g. left) is mapped to OrthoZoom and the other direction (e.g. right) modulates scrolling speed.

We have implemented these techniques for any accessible application. When the user right clicks with the façade modifier on a window, the system goes thought the widget tree and checks if the widget under the cursor has an accessible value. If yes, entries for scrolling are made available in the Façades menu. Additionally, if accessible zoom actions are available, OrthoZoom is made available, too. During interaction, the system then captures all events on the scrollbar and controls the application via the accessibility API. Clearly, the fluidity of the OrthoZoom techniques depends on the ability of the applications to rapidly zoom in and out, but this issue is beyond the scope of this paper. One unexpected benefit is that with this idea *any* widget with an accessible value can become scrollable - even a value text box can be controlled via this technique.

DISCUSSION

The current implementation of Metisse and the Façades system is fast enough to duplicate a 1590x860 video window at 25Hz on a recent laptop. Due to their complexity, accessibility APIs take some time to understand. However, once this has been mastered, replacement widget applications are very easy to generate. Modifying the interaction at the event level (e.g. remapping the action associated with a right click on a canvas), is also reasonably easy. The accessibility APIs provide all necessary data for Façades, but better access to graphical widget information could simplify some issues.

The ability to snap the selection to widgets is arguably the first thing that users notice positively about Façades. However, once users get used to the idea of freely adapting user interfaces of existing applications, they quickly come up with novel uses. One user, who uses a graphical editor in combination with command-line tools, has created a replacement widget with a "Save all" button that he places adjacent to the terminal window. The functionality behind the button activates the save function for all open editor windows to deal with the common problem of forgetting to save changes.

Another application of Façades is to monitor a larger set of mailboxes. As the user is waiting for different kinds of messages at different times, he creates a façade that monitors only those that are currently "interesting" and adapts that façade on demand to changed requirements. Yet another good use of Façades is to fix problems with suboptimally designed user interfaces. The search box of Thunderbird, for example, has a (barely visible) drop-down menu that allows changing between searching the subject, sender, and/or body. With Façades one can create a set of radio-buttons adjacent to the search box to make it easier to select the desired functionality.

Finally, Façades has the ability to make even static visualizations interactive by mapping mouse actions in certain regions to activations of other widgets, which is yet another way to enhance existing GUI's. However, we have to point out that not all modifications possible via Façades will improve the usability of a user interface. This is the trade-off faced by any user of a general-purpose tool.

CONCLUSION

In this paper, we presented a new approach to adaptable user interfaces. User Interface Façades allow end-users to quickly, flexibly and seamlessly change the interface of any application without coding. The system supports cutting, copying and pasting of screen regions, combined with the facility to overlay screen regions over other windows. We have shown how this approach supports both ad-hoc opportunistic customizations as well as persis-

tent ones. Furthermore, we demonstrated that Façades also supports deep customizations, such as the modification of the interactive behavior of arbitrary applications, something that previous work has not supported. We also presented several examples that demonstrate and extend the basic concept in several interesting directions (e.g. window management, multiple monitors, cross-application customizations, new scrolling techniques).

From a global perspective, we believe that Façades offers a good complement to direct programming of user interfaces. From the user's view, it greatly increases the flexibility of any GUI. From the programmers view, it is transparent, as no programming is required to give the user the ability to change the user interface. In the future, appropriate APIs to the Façades system may even enhance the interface programmer's or designer's ability to create good user interfaces.

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 façades even further. For future work, we plan to explore the Façades concept further and investigate how it can be integrate with UI description languages such as XUL & XAML. Furthermore, we will evaluate the adaptation facilities of Façades with user studies, similar to [8, 12].

In this context it is interesting to realize that User Interface Façades extend Apple's vision of the window system as 'a digital image compositor' [13]. More precisely, we can say that the addition of Façades to the standard window management and user interface paradigms allows us to put forth the vision of the window system as a fine-grained interactive graphical component compositor.

ACKNOWLEDGMENTS

Many thanks to the reviewers for their insightful comments, which led us to improve the paper. The initial part of this research was performed while the first author was on a sabbatical stay at In Situ, and Michel Beaudouin-Lafon's support is gratefully acknowledged. This work has been partially funded by NSERC and the French *ACI Masses de données* (Micromégas project).

REFERENCES

1. C. Appert and J.D. Fekete. Orthozoom scroller: 1d multi-scale navigation. In *Proceedings of CHI '06*, pages 21–30. ACM Press, 2006.
2. M. Beaudouin-Lafon and H. M. Lassen. The architecture and implementation of cpn2000, a post-wimp graphical application. In *Proceedings of UIST '00*, pages 181–190. ACM Press, 2000.
3. R. Bentley and P. Dourish. Medium versus mechanism: Supporting collaboration through customization. In *Proceedings of ECSCW '95*, pages 133–148. Kluwer Academic, September 1995.
4. L. Berry, L. Bartram, and K.S. Booth. Role-based control of shared application views. In *Proceedings of UIST '05*, pages 23–32. ACM Press, 2005.
5. O. Chapuis and N. Roussel. Metisse is not 3D desktop! In *Proceedings of UIST '05*, pages 13–22. ACM Press, October 2005.
6. S. Chatty, S. Sire, J.L. Vinot, P. Lecoanet, A. Lemort, and C. Mertz. Revisiting visual interface programming: creating gui tools for designers and programmers. In *Proceedings of UIST '04*, pages 267–276. ACM Press, 2004.
7. K. Edwards, S.E. Hudson, J. Marinacci, R. Rodenstein, T. Rodriguez, and I. Smith. Systematic output modification in a 2d user interface toolkit. In *Proceedings of UIST '97*, pages 151–158. ACM Press, 1997.
8. L. Findlater and J. McGrenere. A comparison of static, adaptive, and adaptable menus. In *Proceedings of CHI '04*, pages 89–96. ACM Press, 2004.
9. J. Fogarty, J. Forlizzi, and S.E. Hudson. Specifying behavior and semantic meaning in an unmodified layered drawing package. In *Proceedings of UIST '02*, pages 61–70. ACM Press, 2002.
10. J. Fujima, A. Lunzer, K. Hornbæk, and Y. Tanaka. Clip, connect, clone: combining application elements to build custom interfaces for information access. In *Proceedings of UIST '04*, pages 175–184. ACM Press, 2004.
11. D. Funke, J. Neal, and R. Paul. An approach to intelligent automated window management. *IJMMS*, 38(6):949–983, 1993.
12. K.Z. Gajos, M. Czerwinski, D.S. Tan, and D.S. Weld. Exploring the design space for adaptive graphical user interfaces. In *Proceedings of AVI '06*, pages 201–208. ACM Press, 2006.
13. P. Graffagnino. OpenGL and Quartz Extreme. Presentation at SIGGRAPH OpenGL BOF, Apple, 2002.
14. D. Hutchings and J. Stasko. An Interview-based Study of Display Space Management. Technical Report 03-17, GIT-GVU, May 2003.
15. D. Hutchings and J. Stasko. mudibo: Multiple dialog boxes for multiple monitors. In *Extended abstracts of CHI '05*, pages 1471–1474. ACM Press, April 2005.
16. P. Kabbash, W. Buxton, and A. Sellen. Two-handed input in a compound task. In *Proceedings of CHI '94*, pages 417–423. ACM Press, 1994.
17. E. Kantorowitz and O. Sudarsky. The adaptable user interface. *CACM*, 32(11):1352–1358, 1989.
18. T. Kühme. A user-centered approach to adaptive interfaces. In *Proceedings of IUI '93*, pages 243–245. ACM Press, 1993.
19. J. McGrenere, R.M. Baecker, and K.S. Booth. An evaluation of a multiple interface design solution for bloated software. In *Proceedings of CHI '02*, pages 164–170. ACM Press, 2002.
20. K. Packard. Coordinate transform redirection for composited window environments. Unpublished talk, FOSDEM 2006, Brussels (Belgium), 2006.
21. N. Roussel. Ametista: a mini-toolkit for exploring new window management techniques. In *Proceedings of CLIHC '03*, pages 117–124. ACM Press, August 2003.
22. D. Tan, B. Meyers, and M. Czerwinski. Wincuts: manipulating arbitrary window regions for more effective use of screen space. In *Extended abstracts of CHI '04*, pages 1525–1528. ACM Press, 2004.