Architectures for Widget-Level Plasticity

Baha Jabarin and T.C. Nicholas Graham

School of Computing
Queen's University, Kingston, Ontario, Canada K7L 3N6
{jabarin,graham}@cs.queensu.ca

Using model- and language-based tools to develop plastic applications requires developers to become familiar with abstract modeling concepts or difficult language syntax. This is a departure from traditional visual interface development tools, in which developers select the widgets that will appear in the application and write the code that defines the widgets' functionality. We present WAHID, a widget-level approach to plasticity in both new and legacy applications that conforms to traditional interface development techniques. WAHID provides internal and external architectures for integrating plastic widgets in an application. The internal architecture provides plasticity in new applications and requires that the application code be available for the architecture to be deployed. The external approach uses gesture handling for widget activation in legacy applications. We demonstrate the viability of these architectures through example scroll bar and menu widgets.

1 Introduction

Recent years have seen a proliferation of new device types such as mobile telephones, personal digital assistants, tablet personal computers and electronic whiteboards. Such devices differ greatly in the interaction modalities they afford. For example, mobile telephones provide input via a microphone and a small numeric keypad, and output on a tiny display. Input to electronic whiteboards and tablet PC's is via a stylus, supporting freehand drawing and gesture-based input. On an electronic whiteboard, traditional interaction techniques may be cumbersome due to the whiteboard's physical size.

It is often desirable to develop software that runs on a range of devices. For example, a bank may wish to allow customers to access their accounts via a mobile phone, a PC-based web browser or a tablet PC. The ability for an application to mould itself to new platforms while retaining usability is called user interface *plasticity* [5].

Building plastic applications is difficult. Target platforms differ so greatly that it is hard to avoid creating separate designs and implementations for each. This leads to problems of creating consistency of function, style and branding from one platform to another, particularly as the product is modified after release.

An alternative to handcrafting platform-specific versions of user interfaces is to generate different versions from some kind of common development artifacts. Two broad strategies exist for such generation. The *model-based* approach [16,20,11,13] is to generate user interfaces from task, domain and interactor models. From these, platform-specific user interfaces are semi-automatically generated. The model-based approach is still experimental, and not yet supported by production-quality tools. Model-based development faces the barrier to adoption of differing greatly from the traditional UI programming techniques to which developers are accustomed.

A second approach is the *abstract language* approach, where programming notations are used to develop abstract user interfaces. Abstract languages such as WML [8] and XSL/XSLT [23,22] have been widely used to develop real products, but are based on cumbersome syntax.

In this paper, we explore the approach of *widget-level plasticity*. Pioneered by Crease et al. [6], this approach builds plasticity into the widget set. Application developers need only combine plastic widgets drawn from a toolkit, and may use traditional tools such as Visual Basic or Visual C++. Widget-level plasticity is easier than the previous approaches, since the problem of plasticity is devolved to the creator of the widget set. Widget-level plasticity has limited expressiveness. For example, the widget-level approach could not reasonably allow the development of an application that runs on both a whiteboard and mobile phone, as these platforms differ too fundamentally.

In this paper, we present WAHID (*Widget Architectures for Heterogeneous I/o Devices*). WAHID consists of two software architectures for widget-level plasticity, with the benefits of:

– Allowing the development of plastic applications using traditional development techniques such as supported by Visual Basic or Visual C++;
– Allowing the plastic adaptation of existing (or *legacy*) applications whose source code is not available.

To achieve these goals, WAHID limits the scope of plasticity to standard PC widgets (scroll bars, menus, etc.) and widgets appropriate to electronic whiteboards and tablet PC's. To illustrate the architectures, WAHID has been applied to the development of plastic scroll bar and menu widgets.

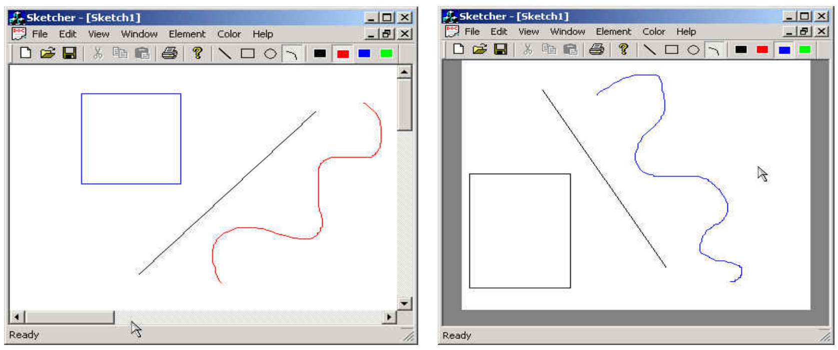
This paper is organized as follows. We first review the notion of Widget-Level Plasticity, and provide an example of an application built using this approach. We then present the WAHID architectures. Finally, we discuss related work in the context of a novel taxonomy of approaches to plasticity.

2 Widget-Level Plasticity

Widget-level plasticity allows applications to mould themselves to different platforms through change in appearance and behaviour of the application's widgets. Widget-level plasticity is less expressive than handcrafted plasticity, but considerably less burdensome to the programmer. Plastic widgets can be used in new applications by writing code in the same way as traditional PC applications are written, or even can be added to existing "legacy" applications which were not written with plasticity in mind. To motivate the concept of widget-level plasticity, we present two examples of plastic widgets: a plastic scroll bar and a plastic menu. These examples are intended to adapt to the PC, tablet PC and electronic whiteboard platforms.

2.1 Plastic Scroll Bar Widget

We demonstrate the behavior of the plastic scroll bar widget in *Sketcher*, a simple sketching application (figure 1.) The scroll bar can take on two forms: (a) a traditional scroll bar, as defined via the Microsoft Foundation Class (MFC) library, and (b) the *Horseshoe* scroll bar, based on a design by Nakagawa et al. [12] The *Sketcher* program is based on an application developed by Horton [9] and is implemented using the MFC document/view architecture.


Sketcher - [Sketch1]
File Edit View Window Element Color Help
Ready
Sketcher - [Sketch1]
File Edit View Window Element Color Help
Ready

(a) (b)

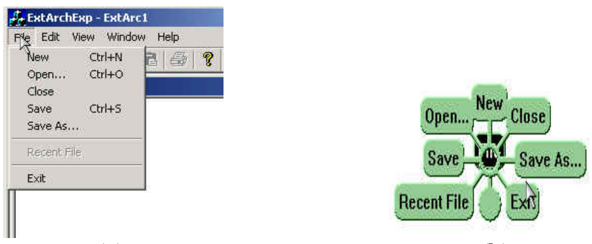
Fig. 1. Two versions of a plastic scroll bar widget, as used in the *Sketcher* application. Part (a) shows a traditional MFC scroll bar rendered for the PC platform. Part (b) shows the *Horseshoe* scroll bar [12] rendered for the electronic whiteboard platform.

The scroll bar adapts its form automatically to the platform being used. In figure 1(a), the scroll bar is rendered as the standard MFC scroll bar used on the PC platform. In figure 1(b), the scroll bar is rendered for an electronic whiteboard as a Horseshoe scroll bar. This version appears as a gray area along the three edges of the application window. To use the Horseshoe scroll bar, the user drags a stylus in the gray area in the desired scrolling direction.

The benefit of a plastic scrollbar widget is that the standard PC scroll bar is inconvenient for electronic whiteboard users. This is because the user's physical location around the whiteboard may not allow her to comfortably reach the scroll bar arrows and thumb and tap them with the stylus. The Horseshoe scroll bar is designed to allow the user to scroll from any position around the whiteboard [12].

2.2 Plastic Menu Widget

Pull-down menus [19] used in PC applications are also inefficient for use on an electronic whiteboard. As with the scroll bar, the user's physical location may not



(a) (b)

Fig. 2. A plastic menu widget. Part (a) shows a standard PC pull-down menu and its items. Part (b) shows the equivalent Pie menu [4].

allow her to comfortably reach for the menus to tap them with the stylus and access their items. Additionally, navigating pull-down menus involves too much arm movement as the user reaches for the menu and navigates through its items on the large whiteboard display.

Pie menus [4] provide an alternative to pull-down menus on electronic whiteboards and tablet PC's. Pie menus can be invoked anywhere in the application window, typically through a gesture such as a tap-and-hold. This eliminates the need for users to change their physical location around the whiteboard to access the pull-down menu. Pie menus are also faster and more reliable than pull-down menus because the large area and wedge shape of pie menu slices makes them easily targeted with little hand motion [4]. This reduces the amount of arm movement needed to find the required menu items versus pull-down menus.

Similarly to the scroll bar above, the plastic menu widget renders automatically either as a PC widget (figure 2(a)) or as a pie menu (figure 2(b)), depending on what platform is being used.

2.3 Deploying Plastic Widgets

There are two fundamental approaches to deploying plastic widgets. The first approach is to use plastic widgets within the application so that each widget appears exactly once, in a form appropriate to the platform. We call this the *internal* approach, since the plastic widgets are an integral part of the application. The second approach is to represent the widgets *externally* to the application. Plastic widgets intercept inputs before they are sent to the application, and then send results of the widget's activation to the application. This approach is appropriate when access to the application's code is not available. Two highly successful examples of the external approach include Microsoft's *Input Method Editor,* used to enter text in Asian languages, and WIVIK [18], an assistive technology allowing text entry for people with poor motor skills.

The internal approach has the benefit of smooth integration with the application, but requires access to the application's source code. The external approach can be applied to existing (or *legacy*) applications, but requires the widgets to run separately from the application. External widgets cannot modify the appearance of the application. For example, using the external approach, it is not possible to replace an MFC scroll bar with a Horseshoe scroll bar.

3 WAHID: Architectures for Widget-Level Plasticity

WAHID proposes two software architectures for widget-level plasticity. The *internal architecture* is used to integrate plastic widgets into new applications. The *external architecture* shows how to couple plastic widgets with legacy applications whose source is not available.

The WAHID architectures support plasticity at the widget level only. Therefore, they are not suitable for developing plastic applications for highly restrictive platforms such as personal digital assistants (PDA) or mobile phones.

3.1 The Internal Architecture

The principal goal of the internal architecture (WAHID/I) is to allow the development of applications whose widgets are plastic. When run on a given platform, these applications should mould themselves to the platform by the widgets taking on appearance and behaviour appropriate to that platform.

WAHID/I has in addition the following goals:

– Developing a plastic program should be as easy as developing a platform-specific program.
– Plastic programs should be developed using tools and methods with which developers are already familiar.

The first goal is met by the mechanism of widget-level plasticity itself. Plastic widgets are simply inserted into programs without specific programming for plasticity. The widgets then automatically adapt to the platform on which the program is run.

The second goal is met by designing the architecture of the plastic widgets to be compatible with existing development tools. In WAHID/I, we aimed for compatibility with the Microsoft Foundation Classes (MFC) [15]. MFC is the dominant user interface development framework on the Windows platform, and is primarily supported by the Microsoft Visual C++ programming environment. MFC is an event-based implementation of the Model-View-Controller architecture [10], where a *Document* implements the model, and a *View* implements the View-Controller. MFC provides an extensive class library, including a container class for the view (*CView*), and a variety of widget classes (*CScrollBar, CButton, CComboBox,* etc.) Visual C++ provides "wizards" to develop graphical user interfaces. Wizards allow high-level specification of parts of the user interface, and automatically generate the C++ code to implement them.

To meet the second goal, it was therefore necessary to:

– Maintain compatibility with the MFC Document-View architecture so that developers don’t have to use an unfamiliar architecture in order to gain plasticity;
– Maintain compatibility with Visual C++ wizards so that generated code runs in a plastic manner.

The following sections show how WAHID/I solved these problems in the context of MFC/Visual C++. This serves as a demonstration that widget plasticity can solve the mismatches between architectures required for plasticity and those of the underlying development toolkit.

Using the Internal Architecture. The use of internal architecture widgets requires minimal changes to standard MFC code. If we consider the scroll bar of figure 1, the only change required is that the main container component for the view be called *CViewP* instead of the usual *CView.* A scroll bar is created in the view by calling its *ShowScrollBar()* method. In *CViewP, ShowScrollBar()* creates an MFC scroll bar when running on a PC, or a Horseshoe scroll bar when running on a tablet or electronic whiteboard.

The Horseshoe scroll bar is designed to conform exactly to the event interface of the MFC scroll bar. This allows the code that controls scrolling (much of which is automatically generated) to use the scroll bar without knowing whether it is the MFC or Horseshoe version.

Therefore, from the programmer’s perspective, use of the plastic scroll bar widget has virtually no extra cost over using the standard MFC widget.

Implementing the Internal Architecture: The MFC framework has deep knowledge of the widgets being used and how they behave. For example, the *CView* component has embedded knowledge that scroll bars appear to the right and bottom of an application window. The implementation of plastic widgets must therefore solve this architectural mismatch between the plastic widgets and the MFC framework, allowing plastic widgets to be easily inserted into MFC programs.

The key idea behind the internal architecture is that the main window supplied by the programmer is wrapped inside a new “outer window.” The outer window is responsible for implementing the correct plastic behaviour of the application. Since the outer window is provided by the internal architecture, the programmer need not be aware of its presence. Figure 3 illustrates the appearance of the window structure of the Sketcher application after deploying the internal architecture plasticity framework.

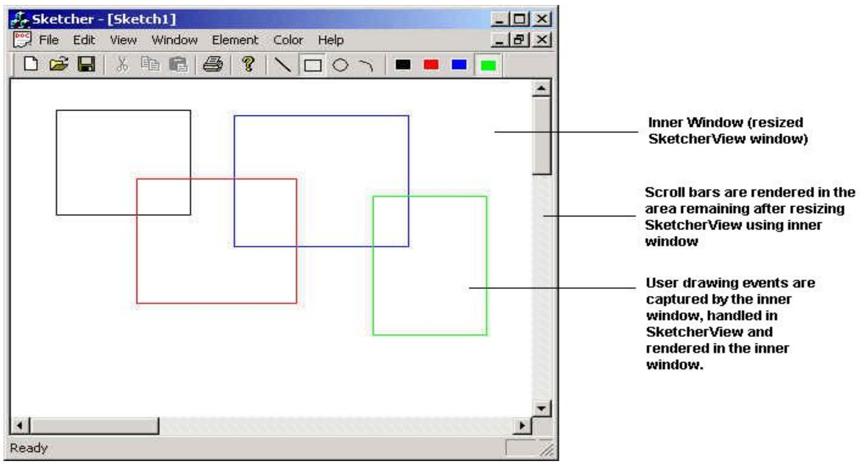

Sketcher - [Sketch1]
File Edit View Window Element Color Help
Ready
Inner Window (resized SketcherView window)
Scroll bars are rendered in the area remaining after resizing SketcherView using inner window
User drawing events are captured by the inner window, handled in SketcherView and rendered in the inner window.

Fig. 3. Illustration of the window structure in the Sketcher application after the internal architecture plasticity framework is deployed.

In the case of our scroll bar example, the outer window is responsible for resizing the application window depending on whether the MFC or Horseshoe scroll bar is deployed, and for handling scroll events and routing them to the appropriate scroll bar control.

Figure 4 illustrates the use of the internal architecture to deploy the plastic scroll bar in the Sketcher application. Figure 5 illustrates the flow of events among the internal architecture components. The internal architecture consists of the plastic view class. To deploy the plasticity framework in the Sketcher application, the developer allows the *SketcherView* class to inherit from the plastic view (*CViewP*). The *CViewP* class overrides platform-specific methods in *CView* in order to provide plastic functionality.

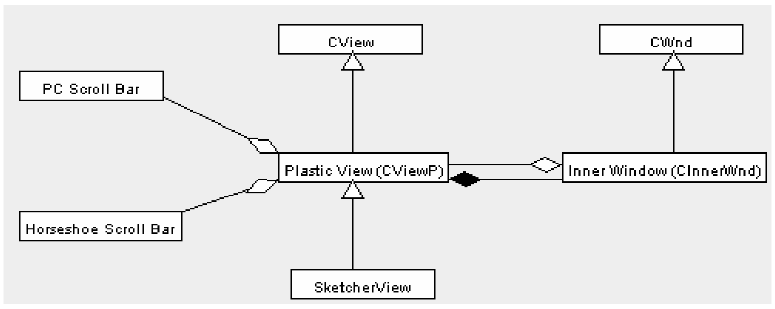


Fig. 4. A class diagram illustrating the deployment of the Horseshoe scroll bar in the Sketcher application based on the internal architecture approach.

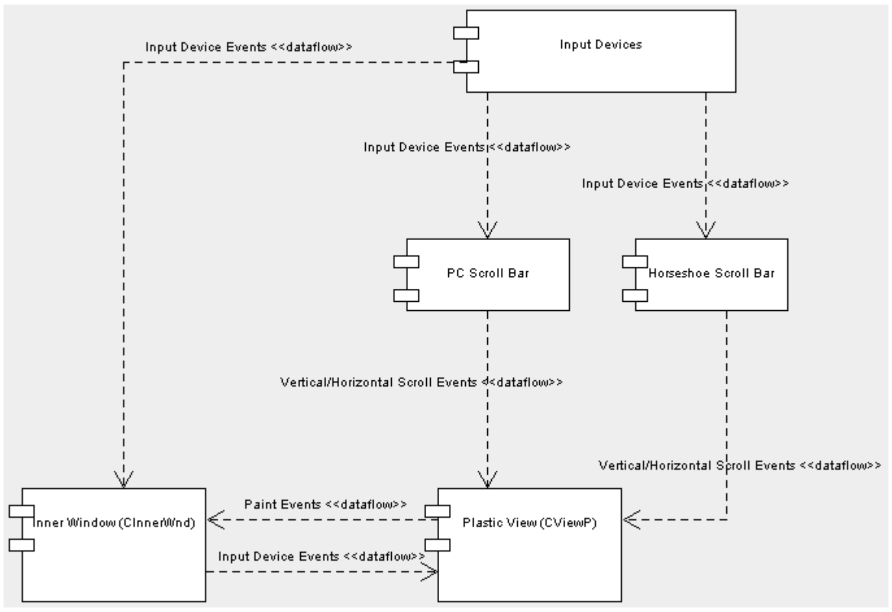

Input Devices
Input Device Events <<dataflow>>
Input Device Events <<dataflow>>
Input Device Events <<dataflow>>
PC Scroll Bar
Horseshoe Scroll Bar
Vertical/Horizontal Scroll Events <<dataflow>>
Vertical/Horizontal Scroll Events <<dataflow>>
Inner Window (CInnerWnd)
Paint Events <<dataflow>>
Input Device Events <<dataflow>>
Plastic View (CViewP)

Fig. 5. A component diagram illustrating the flow of events in the internal architecture.

The event flow depicted in figure 5 allows the internal architecture to be deployed without interfering with the functionality of the Sketcher application. Input device (mouse/stylus) events flow to the standard PC scroll bar or the Horseshoe scroll bar (depending on which is being used) and the inner window. The PC and Horseshoe scroll bars (see figure 1) generate vertical and horizontal scrolling events that are handled by the plastic view (*CViewP*). Since *SketcherView* inherits from *CViewP*, scrolling messages are handled in *SketcherView* according to scroll event handlers specified by the developer.

The *CViewP* class provides the plasticity framework for the Sketcher application by serving as the parent to the *SketcherView* class. The *CViewP* class provides new functionality for API functions such as those associated with scroll bar creation in the *SketcherView*. The new functionality allows the Sketcher application developer to call the scroll bar creation API functions in *SketcherView* in the usual manner. The scroll bar creation function, overridden in *CViewP*, will be able to render the PC or Horseshoe scroll bar.

The *SketcherView* class provides the handlers for mouse events, window sizing events and drawing and scrolling events. Most of the user's interaction with the Sketcher application (drawing, scrolling, etc.) is handled by the *SketcherView* class. The *CView* class defines the basic view/controller functions, such as handling window paint events [6]. Mouse handlers in *SketcherView* handle the drawing gestures the user makes in the view using the mouse. The *SketcherView* class is also responsible for displaying the scroll bars by making calls to API functions that set up the scroll bars. The scroll bars fire vertical and horizontal scrolling events, which are received and handled by *SketcherView*.

Upon deployment of the *CViewP*, the inner window (*CInnerWnd*) acts as a smaller version of the original *SketcherView* window before the latter derives from *CViewP*. As shown in figure 5, the inner window receives all input device (mouse/stylus) events related to drawing and handles them using the *SketcherView* mouse handlers. To accomplish this, the inner window forwards mouse events to the *CViewP* class. The inner window also receives window paint events which are fired to the *SketcherView* window when it is invalidated by actions such as user drawing, resizing, restoring or uncovering after being overlapped by another window. Each paint event carries a handle to a device context which refers to a particular window on the video display [15]. Paint events to *SketcherView* are routed to the inner window through *CViewP*. Upon receipt of a paint event, the inner window passes its own device context in a call to *SketcherView*'s paint event handler function. This forces all drawing and painting to be rendered in the inner window.

In summary, the difficulty of deploying plastic widgets in the internal architecture is a consequence of wishing to remain consistent with the architectural style of the user interface toolkit, in our case MFC. Our approach of wrapping the application (inner) window with a plasticity (outer) window solves this problem, allowing the plastic widgets to be deployed with only minimal changes to an MFC program. This approach has the added benefit that existing MFC programs can be easily converted to run in a plastic fashion.

3.2 The External Architecture

In addition to writing new applications that are designed for plasticity, it is also desirable to be able to use existing applications on novel devices. Commercial applications do not make their source code available, however, so the internal architecture cannot be applied.

The external architecture (WAHID/E) is designed to incorporate plasticity in legacy applications whose code is not available for modification. In the external architecture, electronic whiteboard widgets can be used in applications designed to run under MFC. The whiteboard versions of widgets are invoked via a gesture, such as tapping and holding the stylus. Example widgets that can be provided via the external approach are pie menus, soft keyboards and gesture-based scrolling.

The basic approach behind the external architecture is to run a process separately from the application that intercepts input events. If the input events are a widget invocation gesture, the process pops up an instance of the widget. Otherwise, input events are passed to the application.

The external approach has the obvious advantage that it can be applied to legacy applications whereas the internal approach cannot. The external approach has, however, the following disadvantages:

– External widgets are not part of the application. For example, it would not be possible to implement a Horseshoe scroll bar externally, as it is tightly integrated into the presentation of the application.

– The external whiteboard widgets do not replace the MFC widgets built in to the application. The original widgets will still appear on the screen, which may be unaesthetic or confusing.

The following section describes the deployment of widgets using the external architecture, using the example of an externally deployed pie menu (figure 2.)

3.2.1 Deploying the External Architecture: External widgets are added to an application by hooking input events and filtering them through a gesture router. The gesture router captures widget activation gestures, and routes input events to external widgets or the application as appropriate.

Figure 6 illustrates the flow of events in the external architecture. This example shows the deployment of a pie menu as a tablet/electronic whiteboard alternative to the standard pull-down menu used in PC applications. The pie menu is activated through a double-tap of the stylus inside the application window. A mouse hook intercepts all stylus events and forwards them to the gesture router. When a double-tap is detected, the gesture router fires a menu activation event to the widget container. The widget container processes the menu activation event by creating the pie menu and activating it in the application window. The pie menu is populated with the appropriate menu items obtained from the running application.

Using the pie menu in the external architecture shows the feasibility of obtaining information from a legacy application, such as the application menu hierarchy and commands associated with menu items, despite lack of access to the application code.

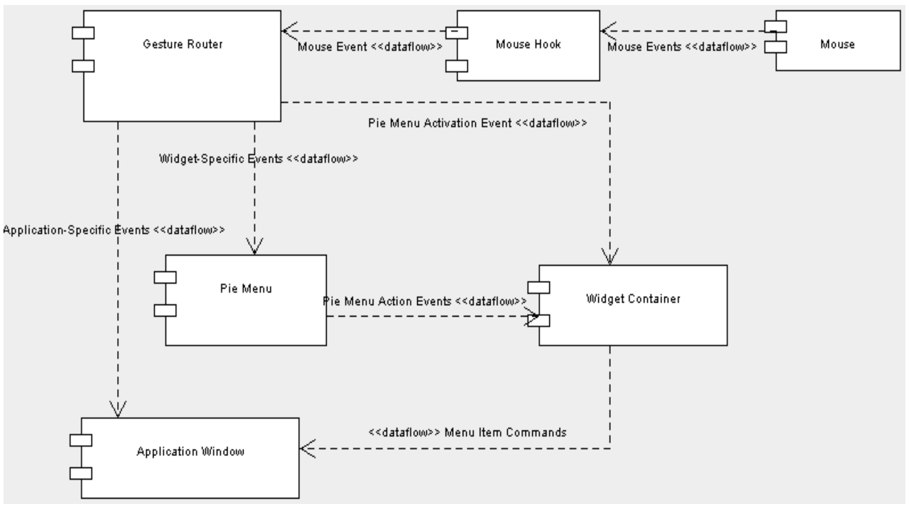

Gesture Router
Mouse Event <<dataflow>>
Mouse Hook
Mouse Events <<dataflow>>
Mouse
Pie Menu Activation Event <<dataflow>>
Widget-Specific Events <<dataflow>>
Application-Specific Events <<dataflow>>
Pie Menu
Pie Menu Action Events <<dataflow>>
Widget Container
<<dataflow>> Menu Item Commands
Application Window

Fig. 6. A component diagram illustrating the flow of events among the external architecture components.

The application window component represents the main window associated with the legacy application. A handle to the application window can be used to access the

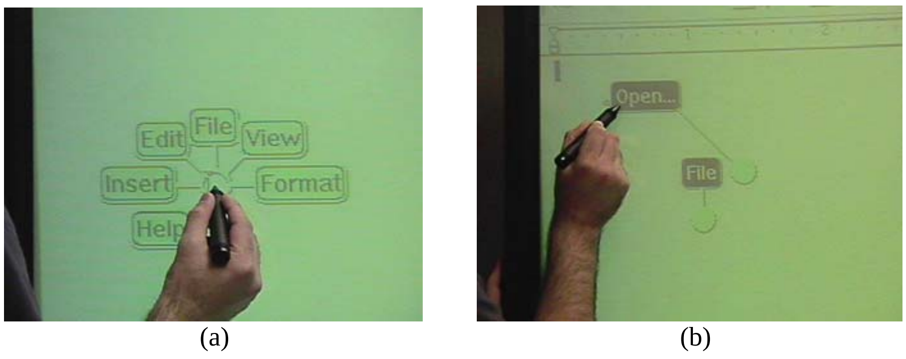


Fig. 7. A demonstration of opening a file in the WordPad application on the EWB. Part (a) shows the Pie menu populated with the top-level menu items of WordPad. Part (b) shows the user selecting 'Open..' from the 'File' menu.

application's menu hierarchy and the commands associated with menu items. The application window receives mouse (or stylus pen) events after they have been filtered through the gesture router. The application window also receives commands associated with Pie menu items through the widget container. The commands are processed exactly as if the application's standard menus were used.

Figure 7 demonstrates the WAHID/E deployment of *WordPad*, a simple word processing application provided with the Windows operating system. After double-tapping the stylus pen in the WordPad application window, the user is presented with a pie menu (figure 7(a)) populated with the top-level WordPad menu items (File, Edit, View, Insert, Format, Help). Figure 7(b) shows the user selecting the 'Open…' item from the 'File' menu. The pie menu fires the appropriate command associated with the 'Open…' item in the File menu to the WordPad application.

The external architecture provided in figure 6 can be generalized to allow the use of any electronic whiteboard widget in a legacy application. The widget container and pie menu components can be grouped together as an external widget component. The mouse hook and gesture router can be grouped together as a gesture handling component. The mouse can be replaced by a generic input device component. The flow of events among the new components remains the same.

4 Classifying Widget-Level Plasticity

Widget-level plasticity trades off usability versus expressiveness. Our approach is therefore appropriate for application development where it is possible to restrict plasticity to just the widgets. In applications intended for a wide range of platforms with fundamentally different properties (e.g., PC versus mobile telephone), the widget-based approach is insufficient.

To better understand the domain to which widget-based plasticity should be applied, we compare the WAHID approach to other model- and language-based approaches for plasticity. This comparison is summarized in table 1.

Level of UI Specification	Degree of Automation: Automatic	Automatic with Developer Intervention	Write-Once	Manual
Model-Based	ADEPT [11]	MOBI-D [16], HUMANOID [20], MASTERMIND [21], ArtStudio [5], TRIDENT [3]	OMMI [13]	*UAN* [7]
Abstract User Interface	WAHID, *WML*[8]	PIMA [2]	AUI[17], *UIML* [1], *XSL/XSLT* [23, 22]	CGB [6]
Concrete User Interface				Visual C++, Visual Basic, JBuilder

Table 1. A classification of interface development tools. Tools in italics are production quality.

Loosely following the framework of Calvary et al. [5], the classification is based on three axes: *level of user interface specification, degree of automation* and *production quality.*

Level of user interface specification describes the level of input artifacts to the development tool or method. This ranges over three levels:

– *Concrete User Interface:* A concrete user interface (CUI) precisely specifies the application's presentation and behaviour. CUI's are therefore platform-specific. The most basic form of plasticity is therefore to develop a separate CUI for each target platform, using traditional tools such as Visual C++ or Visual BASIC.
– *Abstract User Interface:* An abstract user interface (AUI) specifies the structure of a user interface's presentation and behaviour, but does not bind the behaviour to a specific platform. For example, the Wireless Markup Language (WML [8]) provides an abstract "*Select*" tag that allows users to select from a list of elements. The selection list may be rendered as a list of checkboxes or a menu depending on the platform on which the WML is rendered. An AUI must be transformed into a concrete user interface for use on a particular platform.
– *Models:* Domain, task, platform and interactor models are used to describe the purpose and context of use of the application. From these high-level models, an AUI is developed, from which a set of CUI's can be derived.

Degree of automation describes the level of developer input in the process of reifying one level of specification to the next. Reification involves translation from high-level models to the AUI, from the AUI to the CUI and to the interface implementation. Degrees of automation range over

– *Automatic*, where the developer provides no input to the process;
– *Automatic with developer intervention,* where the developer is able to influence the reification, or tune its output;
– *Write-once,* where the developer provides a specification for how to map from high-level model/AUI to a platform, and this specification is used each time the model/AUI is changed. For example, to transform XSL [22] abstract user interfaces to a specific platform, developers write an XSLT [23] transform once.

– *Manual*, where the developer manually performs the translation from one level of specification to the next with no tool support. For example, the User Action Notation (UAN [7]) is used to specify task and dialogue structure, from which a concrete user interface may be created by hand.

In table 1, tools indicated in italics are *production quality*, meaning that they have been applied to the development of commercial software.

The WAHID approach to widget-level plasticity is classified as an automatic tool that begins with an AUI specification of the user interface. Automatic tools are able to reify an AUI into a platform-concrete interface without developer intervention. MOBI-D [16], HUMANOID [20], MASTERMIND [21], ArtStudio [5] and TRIDENT [3] are all tools that provide a mix of automatic translation and developer input. Languages such as WML [8], UIML [6] and XSL/XSLT [23, 22] begin their interface specification at the AUI level. UIML and XSL/XSLT are write-once tools which allow developers to write transformations that can be reused in reifying AUIs to platform-concrete interfaces.

5 Evaluation of Widget-Level Plasticity

Widget-level plasticity allows developers to create plastic applications for a limited platform domain. The domain of applications supported by our WAHID approach consists only of desktop PC, electronic whiteboard and tablet PC applications. The limited domain supported by widget-level plasticity can be attributed to two factors. First, a widget-level plastic application can only change the appearance and behavior of its widgets, not its entire appearance. Supporting plasticity on platforms such as the PDA and cellular phone requires plasticity at the application- rather than widget-level. Second, as can be observed from table 1, automation in interface generation is generally provided at the cost of support for a limited domain. Providing automation on a large domain of platforms can bring about interfaces with unsatisfactory layout and appearance. This is because the less involved the developer is in the interface generation process, the less able a tool is in reasoning about which widgets to use and how to lay them out in the interface for a particular platform.

In the WAHID approach, the external architecture allows developers to incorporate plasticity in applications without changing the application code. The developer must only ensure that the external architecture components are able to communicate properly. The external architecture requires that the standard PC widgets continue to appear after the electronic whiteboard widgets are rendered. This wastes screen real estate and may confuse users. Further, some widgets, such as the Horseshoe scroll bar, cannot be implemented under the external architecture. The Horseshoe scroll bar must be rendered in a specific location when the application is first executed. It is not practical to allow the Horseshoe scroll bar to be activated through gestures.

The internal architecture allows developers to incorporate plasticity in new applications by requiring a minimal amount of change to the application code. Using the internal architecture, developers are able to incorporate plasticity in their applications using familiar interface development techniques. The internal architecture is specific to the MFC framework, but might be generalized to apply to other frameworks. The WAHID internal architecture deploys its plasticity framework between the application's view class and the MFC framework view class. Other framework-specific internal architectures must be able to deploy their plasticity frameworks at the appropriate level in the framework. A disadvantage of the internal architecture is that the application code must be available and open for change. The developer must also ensure that the application software architecture incorporates the internal plasticity framework. This is to avoid performing significant changes to the application code when deploying the internal architecture plasticity framework.

6 Conclusion

This paper has introduced WAHID, an approach to widget-level plasticity consisting of internal and external architectures designed to incorporate plasticity in new and legacy applications respectively. The internal approach relies on the availability of application code in order to allow deployment of its plasticity framework in the application. The external architecture allows widgets to cooperate with legacy applications to provide alternatives to the existing desktop widgets.

The widget-level plasticity approach is more automatic than model-based tools used in the development of plastic user interfaces. Further, widget-level plasticity allows developers to specify interfaces at the AUI level, a specification level less abstract than that of model-based tools.

Widget-level plasticity allows developers to achieve plasticity in applications over a limited domain. In this paper, the WAHID approach supports applications designed for the desktop PC, tablet PC and electronic whiteboard platforms.

Acknowledgements

We gratefully acknowledge the support of Communications and Information Technology Ontario (CITO) and the Natural Science and Engineering Research Council (NSERC) in this work. We would also like to thank Chris Wolfe and James Wu for their many helpful technical discussions, and Don Hopkins for the use of his pie menu ActiveX control.

References

1. Abrams, M., Phanouriou, C., Batongbacal, A. L., Williams, S., Shuster, J. E., UIML: An Appliance-Independent XML User Interface Language, *WWW8 / Computer Networks 31(11-16): 1695-1708*, 1999.
2. Bergman, L., Banavar, G., Soroker, D., Sussman, J., Combining Handcrafting and Automatic Generation of User-Interfaces for Pervasive Devices, *Proceedings of CADUI'02*, C. Kolski & J. Vanderdonckt (eds), Kluwer Academic Publishers, 2002, pp.155-166.

3. Bodart, F., Hennebert, A., Leheureux, J., Vanderdonckt, J., A Model-Based Approach to Presentation: A Continuum from Task Analysis to Prototype, *Proceedings of DSVIS'94*, F. Paterno (ed), Eurographics, 1994, pp. 25-39.
4. Callahan, J., Hopkins, D., Weiser, M., Shneiderman, B., An Empirical Comparison of Pie Vs. Linear Menus, *Proceedings of CHI'88*, ACM, 1988, pp., 95-100.
5. Calvary, G., Coutaz, J., Thevenin, D., A Unifying Reference Framework for the Development of Plastic User Interfaces, *Proceedings of EHCI 2001*, Toronto, 2001, pp. 218-238.
6. Crease, M., Gray, P., Brewster, S., A Toolkit of Mechanism and Context Independent Widgets, *Proceedings of DSVIS'00*, pp. 127-141.
7. Hartson, H. R., Siochi, A. C., Hix, D., The UAN: A User-Oriented Representation for Direct Manipulation Interface Designs, in *ACM Transactions on Information Systems, Vol. 8, No. 3*, July 1990, pp. 181-203.
8. Herstad, J., Thanh, D. V., Kristoffersen, S., Wireless Markup Language as a Framework for Interaction with Mobile Computing Communication Devices, *Proceedings of the 1st Workshop on Human Computer Interaction with mobile Devices*, Univ. of Glasgow, UK, GIST Tech. Report, G98-1, 1998.
9. Horton, I., Beginning Visual C++, Wrox Press Ltd., 1998.
10. Krasner, G.E., Pope, S.T.. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *JOOP*, 1(3):26-49, Aug./Sept. 1988.
11. Markopoulos, P., Pycock, J., Wilson, S., Johnson, P., Adept – A Task Based Design Environment, *Proceedings of the 25th Hawaii International Conference on System Sciences*, IEEE Computer Society Press, 1992, pp. 587-596.
12. Nakagawa, M., Oguni, T., Yoshino, T., Horiba, K., Sawada, S., Interactive Dynamic Whiteboard For Educational Purposes, *Proceedings of International Conference On Virtual Systems and Multimedia '96*, Gifu, Japan, 1996, pp. 479-484.
13. Paterno, F., Santoro, C., One Model, Many Interfaces, *Proceedings of CADUI'02*, C. Kolski & J. Vanderdonckt (eds), Kluwer Academic Publishers, 2002, pp. 143-154.
14. Petzold, C., Programming Windows, Fifth Edition, Microsoft Press, 1999, p.75.
15. Prosise, J., Programming Windows With MFC, Second Edition, Microsoft Press, 1999, p.503.
16. Puerta, A. R., A Model-Based Interface Development Environment, *IEEE Software, (14) 4*, July/August 1997, pp. 40-47.
17. Schneider, K. A., Cordy, J. R., Abstract User Interfaces: A Model and Notation to Support Plasticity in Interactive Systems, *Proceedings of DSVIS'01*, Glasgow, June 2001, pp. 40-59.
18. Shein, F., Treviranus, J., Hamann, G., Galvin, R., Parnes, P. and Milner, M., 1992, New Directions in Visual Keyboards for Graphical User Interfaces, in *Proceedings of the 7th Annual Conference Technology and Persons with Disabilities*, CSUN, CA, 465-469.
19. Shneiderman, B., Designing the User Interface: Strategies for Effective Human-Computer Interaction, 3rd Edition, Addison-Wesley, 1998.
20. Szekely, P., Lou, P., Neches, R., Beyond Interface Builders: Model-Based Interface Tools, *Proceedings of INTERCHI'93*, ACM Press, 1993, pp. 383-390.
21. Szekely, P., Sukaviriya, P., Castells, P., Muthukumarasamy, J., Salcher, E., Declarative Interface models for User Interface Contruction Tools: The MASTERMIND Approach, *Proceedings of EHCI'95*, L. J. Bass & C. Unger (eds), Chapman & Hall, 1995, pp. 120-150.
22. World Wide Web Consortium, Extensible Stylesheet Language (XSL)Version 1.0, W3C Recommendation, S. Adler, A. Berglund, J. Caruso, S. Deach, T. Graham, P. Grosso, E. Gutentag, A. Milowski, S. Parnell, J. Richman, S. Zilles (eds.), www.w3.org/TR/xsl, 2001.
23. World Wide Web Consortium, XSL Transformation (XSLT) Version 1.0, W3C Recommendation, J. Clark (ed.), www.w3.org.