Scientists in the MIST: Simplifying Interface Design for End Users

Saurabh Bhatia, D. Scott McCrickard, Tripp Lilley, Chris North
Center for Human Computer Interaction and Department of Computer Science
Virginia Polytechnic Institute and State University (Virginia Tech)
Blacksburg, VA 24061-0106
1-540-231-7409
{saurabhb,mccricks,tlilley,north}@vt.edu

Paul Kienzle
NIST Center for Neutron Research
National Institute of Standards and Technology
100 Bureau Drive, MS 8562
Gaithersburg, MD 20899-8562
pkienzle@nist.gov

Abstract: This work seeks to assist in the building and modification of the user interface by defining appropriate tools and toolkits for use in the development process. While our ultimate goal is to allow end users with little or no programming experience to be able to develop and modify interfaces themselves, a more attainable short-term goal is to empower developers to quickly create such interfaces, building and maintaining a sharable repository of tools and techniques during the development process. This poster paper describes our Malleable Interactive Software Toolkit (MIST), an architecture and toolkit that provides development support, and outlines the ways in which we expect it will assist developers and end users in rapid and flexible interface creation.

Introduction

Professionals with little or no interactive software development background have a recurring need to create usable interfaces for algorithms and analysis methods that they develop. The lack of suitable tools and infrastructure to support this need is a persistent problem in the software development and interaction design communities. The distinct requirements of diverse groups of users further complicates the issue, making satisfying all of these needs with a flexible tool is a difficult task.

By taking advantage of the unique juxtaposition of a collection of dormant research, contemporary ideas, and a burgeoning body of stable, widespread standards for information exchange (e.g., XML and its many children,) we seek to produce an environment in which the initial creation, and long-term maintenance/modification of interactive software carries with it significantly less cognitive overhead than does the present, conventional practice.

The ultimate (if perhaps unattainable) goal is to allow end users with little or no programming experience—the scientists who need the applications—to be able to develop and modify interfaces themselves, ideally by leveraging and reusing the work of prior designers. While some successes have been seen in some visualization toolkits (e.g., Snap (North & Shneiderman, 2000)), it has proven difficult to encourage non-technical users to make use of these tools. A more attainable short-term goal is to empower developers to quickly create such interfaces through tools like Snap, building and maintaining a sharable repository of tools and techniques during the development process. It is here where exist our ongoing efforts and contributions, described in this paper and to be elaborated in our poster.

Current Conventions

There has been considerable pioneering research in user interface (UI) creation and maintenance in the past several decades, including the seminal work reflected in Self's Morphic toolkit (Maloney and Smith 1995), Garnet and Amulet (Myers et al. 2001). Visual Basic is probably the most successful commercial product that attempts to simplify designing and programming graphical user interfaces. Each body of work has offered a distinct combination of solutions to the core problems, and brought forth new techniques to be appropriated by later projects, such as our own. These techniques have ranged from fundamental work on the way we construct applications to make them more amenable to interaction and integration (e.g., reflection in Smalltalk, Self, Java, etc.,) to concrete

progress on the mechanisms by which we actually manage interaction, whether in the form of a Graphical User Interface (GUI), Interactive Voice Response (IVR), a text-mode terminal, or some other means of communicating with the user.

A typical process for constructing user interfaces proceeds from the developer's understanding of the underlying application logic continuing onto deciding how the interface to the logic should look like. To enable user interaction with the application logic, developers have access to a set of standard widgets which allow them to create graphical interfaces to the application logic. Starting with the idea of what the interface should look like, the developer creates the visual interface consisting of widgets and links them with the application logic through code encapsulated in callback routines.

The standard widgets enable the visualization and interaction with the application data and logic. A phone number, a weight in pounds, a network address, and a birth date are all, ultimately, a series of digits which are presented differently on an interface so as to allow the user to understand their meaning. From a programmer's perspective, a series of digits is just a string or an integer value. In the conventional view because each of these data structures are essentially a series of characters, any text widget will suffice to accept user input and present output. Either the application logic, or the callback routine, is responsible for any validation and conversion necessary to make sure that the user input is acceptable, and to convert an internal representation into the string of characters the text widget is expecting to display.

The unfortunate consequence of this is the easy tendency to slip into patterns of development in which callback routines contain logic that ought to be in the application, application logic contains UI-specific details that ought to be a part of the callback routines, and in the middle are pieces of code that ought to be shared amongst callback routines, but not installed within the core application logic to effect this sharing. As a result of this, the spaghetti of call-backs (Myers 1991), small changes in either the application or the user interface components can have a potentially widespread and damaging effect on the entire system.

Declarative User Interfaces

Declarative user interfaces examine techniques for separation of user interface logic from the application logic. User interface concerns are written declaratively as a set of constraints. The constraints are satisfied at runtime usually by a renderer that is also responsible for rendering the declared UI model into an actual UI. Technologies like Mozilla's XML-based User Interface Language (XUL), Microsoft's Extensible Application Markup Language (XAML), and Harmonia's User Interface Markup Language (UIML) are exploring the concept of declarative user interfaces. By allowing high level UI specification, declarative user interfaces are able to separate and isolate the UI code from application code. Declarative UIs show great promise of avoiding the problems of callback spaghetti and hence we choose to build our architecture based on this approach.

The Malleable Interactive Software Toolkit (MIST)

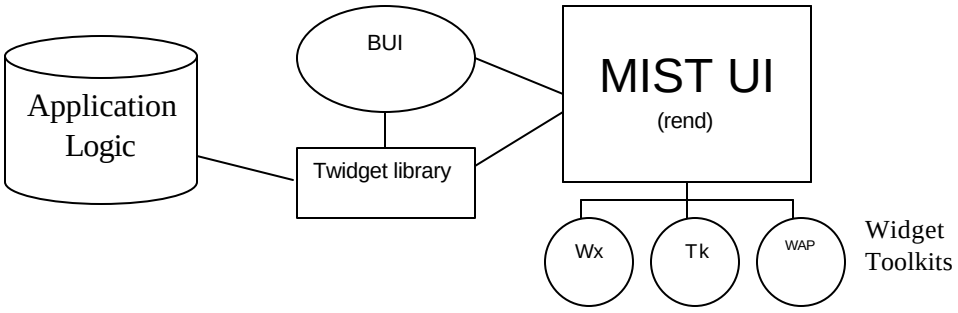

Figure 1. The MIST UI Framework – Users can select Twidgets to represent core concepts from the application logic and create a Binding User Interface which is then rendered into a typical user interface using standard widgets.

We propose an architecture which would make the process of creating and maintaining the user interfaces a less demanding endeavor. Utilizing the power of Declarative UI our architecture allows a user to go from concept to realization of a User Interface with minimal resistance from the toolkit. Our architecture is especially useful for

scenarios which tend to reuse application logic. Examples of this would be data analysis tools where you have a standard set of algorithms to analyze the data but need different user interfaces to manipulate the input to those algorithms.

Twidgets: Type-aware widgets

One of the key features of the MIST framework is the concept of Twidgets, or Type aware widgets. Twidgets act like an adapter between the application logic and the interface logic. They encapsulate higher level concepts of the underlying data and encode interface logic within themselves. Instead of working with individual widgets and tying them in with underlying data through callbacks, a Twidget allows the designer to represent the data directly.

By encapsulating the presentation and UI logic the Twidget is aware of the type of data it represents. For example, considering the traditional UI development wherein a programmer will first define a text entry widget and then write code that will populate the widget with some data like a telephone number. The programmer also has to write a considerable amount of code to define how the number would be presented on screen and what kind of input constitutes a valid telephone number. A Twidget on the other hand is aware that a telephone number will only consist of numbers and will automatically do the necessary formatting to display it effectively. The constraints defined in the Twidget template provide the *type* awareness of a Twidget. Twidgets are defined in XML based templates very similar to those used by various declarative UI standards.

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<twidget xmlns="http://xmlns.hci.vt.edu/nist/twidgets/0.0#">
  <!-- trivial twidget for representing an angle in degrees -->
  <numentry type="float">
        <label xml:lang="%(lang)s">%(label)s</label>
        <range>%(range)s</range>
  </numentry>
</twidget>
```

Figure 1. Twidget Template for an angle.

The angle Twidget above encapsulates the concept of an angle. It defines an input field for numeric entry, a label to display the name of an angle and a range that restricts the value that can be entered into the input field. Label and range are two variables that can be specified by the designer while creating the UI. The simple XML notation allows designers to define additional variables for a Twidget. The variables and constraints together help build a Twidgets type awareness.

A Twidget template is not tied to any particular widget representation. The Twidget merely defines the input/output and interaction requirements of the underlying concept. The angle Twidget can be represented by a label and a simple TextEntry widget or a label and a SpinBox etc. Essentially, the angle Twidget can be represented with any combination of widgets that are capable of displaying some text and accepting a numeric input. Theoretically this abstraction can even go beyond widgets. For example, the angle Twidget can also be represented by a physical device like a joystick which can be tilted to input the angle value while a LED display shows the label of the angle. This is a concept UIML explores and is out of the scope of our environment.

BUI: Binding User Interface

The Binding User Interface (BUI) is an XML representation of a UI that provides the necessary specification in order to use a Twidget in an interface. The BUI can be considered to define instances of the Twidget templates and specify/modify any of the Twidget parameters.

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<twidgets>
  <!--<!ENTITY % HTMLlat1 PUBLIC "-//W3C//ENTITIES Latin 1//EN//HTML"> %HTMLlat1;-->
  <twidget>
        <type>angle</type>
        <name>incidence</name>
```

```
        <values>
          <lang>en</lang>
          <label>incidence (<!--&Theta;-->)</label>
          <range><![CDATA[<gt>0</gt><lte>180</lte>]]></range>
        </values>
 </twidget>
 <twidget>
        <type>angle</type>
        <name>reflection</name>
        <values>
          <lang>en</lang>
          <label>reflection (<!--&Theta;-->)</label>
          <range><![CDATA[<gt>0</gt><lte>180</lte>]]></range>
        </values>
 </twidget>
 <placement>
 </placement>
</twidgets>
```

Figure 2. Binding User Interface (BUI) using two instances of the angle Twidget.

The above example shows a BUI defining a user interface that consists of two angle Twidgets. This simple example of the BUI does not contain any interaction information and thus the UI created from this BUI will only be a static display. The two Twidgets represent an angle of incidence and an angle of reflection. As defined in the Twidget template the BUI specifies a label and range for both instances of the angle. The range tag in particular defines a constraint on the possible values the angle can cover. While this is a constraint on each individual Twidget the BUI can also define constraints between two Twidgets. One example would be that the angle of incidence should also be equal to the angle of reflection. This would mean that the BUI defines a constraint between the two Twidgets that requires the values of the two Twidgets to always be equal. This example BUI also does not contain any placement information. In the absence of any placement information the widgets associated with the UI are simply packed in a flow layout.

Rend: The Screen Renderer

A Screen Renderer (rend) renders the actual MIST UI which is defined by the BUI and Twidget templates. The renderer is responsible for tying in the abstract concepts encapsulated in the Twidgets and BUIs with a real world interface. The Screen Renderer modifies the Twidget templates as specified by the BUI and then connects the Twidgets with appropriate widgets that can be rendered according to widget toolkit being used.

MIST currently uses a renderer written in Python to create a user interface from the BUI and Twidget templates. The interface is rendered using the Tkinter toolkit. Since the BUI is represented with XML, it is easily portable to many other languages and toolkits.

Application Logic

The Twidgets+BUI framework of the MIST UI is capable of capturing all UI logic within itself. This is made possible by the constraints defined in both the Twidget templates and the BUIs. This approach allows for a clear separation of the application logic and UI logic. The MIST UI, while encapsulating all of the necessary UI logic in the Twidget+BUI model, interfaces with application logic through event triggers invoked by user interaction. Thus using our framework the call backs are solely restricted to contain application code.

E-m: The Eclipse based MIST UI builder

The current implementation of the system uses Eclipse as a development platform for creating the MIST UI. Using custom plug-ins we were able to extend Eclipse to support the MIST framework. The necessary application logic is

provided by a beta version of a related software development project for the distributed Data Analysis for Neutron Scattering Experiments (DANSE). The MIST UI acts as a front end for the DANSE components. The components analyze the underlying data while the MIST UI interacts with the data. Users can tie the attributes exposed by the DANSE components to Twidgets which are automatically converted to interface representations. Any existing application logic written in a programming language that supports reflection can also be used with Twidgets. The UI builder thus allows users to design with abstract concepts present in the application logic and automatically generate the widgets required for the concrete realization of the user interface. By letting users work with the concepts in domains that they are familiar with, the Twidgets offer the path of least resistance for creating user interfaces.

Conclusions and Future Work

The goal of the MIST UI is to ease the process of creating and maintaining user interfaces. By separating the UI logic from the application logic the MIST UI avoids code entanglement and makes it easier to modify the UI logic and application logic independent of each other. The use of constraints in Twidgets+BUI coupled with reflection on application logic provides live UIs that can dynamically reflect changes in underlying application code. The malleable architecture enables reusability of application code which is especially useful for certain application areas like scientific data analysis where the scientists tend to use similar algorithms to analyze different datasets.

Although our current architecture does not ease the process of creating a user interface from scratch it does simplify the modification of existing UIs. If a large repository of application code and user interfaces is created using our architecture, we feel that users will be able to easily modify existing components to satisfy most of their needs. Furthermore, as Twidgets are written in simple XML, they have a low learning threshold. Once users become familiar with the Twidget concept they should also be able to create their own Twidgets with relative ease. When coupled together with a visual builder like E-μ we feel that users should also be able to create their own UIs from scratch. In fact our contention is that be allowing users to first modify existing UIs will ease the complex process of learning how to design and program an UI.

Our future work involves developing a self hosting UI builder where the UI builder itself is built using the MIST framework. This will allow users to extend the UI builder with the same level of simplicity as that of any other UI created in MIST. Our planned upcoming vision of MIST in the development of interfaces will guide the creation of robust and a reusable development environment.

References

Maloney, J.H., Smith, R.B. (1995). Directness and liveness in the morphic user interface construction environment. *In Proceedings of the 8th Annual ACM Symposium on User interface Software and Technology*. Pittsburgh, Pennsylvania, United States. UIST '95. ACM Press, New York, NY, 21—28.

Vander Zanden. B.T., Halterman. R., Myers, B. A., McDaniel R., Miller. R., Szekely. P., Giuse. D.A., Kosbie. D. (2001). Lessons learned about one-way, dataflow constraints in the Garnet and Amulet graphical toolkits. ACM Transactions Programming Language Syst., 23(6), pp. 776-796.

Myers, B. A. (1991). Separating application code from toolkits: eliminating the spaghetti of call-backs. *In Proceedings of the 4th Annual ACM Symposium on User interface Software and Technology*. Hilton Head, South Carolina, United States, November 11 - 13, 1991. UIST '91. ACM Press, New York, NY, 211-220.

North, C., Shneiderman, B. (November 2000). Snap-Together Visualization: Can Users Construct and Operate Coordinated Views, *International Journal of Human-Computer Studies*, 53(5), pp. 715-739,

DANSE: Distributed Data Analysis for Neutron Scattering Experiments <http://wiki.cacr.caltech.edu/danse/index.php/Main_Page> Accessed on April 14th, 2006.