Behaviour Objects: Taking the Feel out of Look and Feel

Mike Lopez
Manukau Institute of Technology
mike.lopez@manukau.ac.nz

One of the many challenges facing cross-platform developers is that of producing an application that has the "look and feel" (appearance and behaviour) of one that is designed specifically for the target platform. The developer also wants to use a common code base for as much of the application as possible. Conventional approaches dictate the use of an n-tier architecture in which the look and feel is encapsulated in a user interface layer, separate from the business and persistence objects of the application. This approach allows the use of a single code base for the latter with only the UI being platform dependent.
There are ample tools and guidelines available to support getting the appearance of the UI right, but few are available for managing behaviours.
This paper introduces the concept of behaviour objects and describes an approach in which user interface behaviours may be abstracted from the application and encapsulated in a set of behaviour objects that may then be plugged-in to an application object model.

1. INTRODUCTION

We often say that a well designed Windows application should look and feel like a Windows application and we say the same thing for Macintosh and Unix applications. There's a lot of commonsense in this. Users can learn to use applications much faster if the application conforms to the "look and feel" guidelines for the platform. Moreover, vendors such as Microsoft and Apple invest substantial resources in usability testing when developing these guidelines and the application developer can exploit this investment simply by following the guidelines.

Best practice has also dictated that the look and feel of an application has been separated from the core business logic and encapsulated in a separate User Interface or Presentation layer. This facilitates cross platform development by enabling a common code base for the core business logic with separate UI layers for each platform. This has been accepted best practice for so long that we have been conditioned to use the words "look and feel" together as if they were natural siblings.

Recently, this packaging together of look and feel has been challenged by the ability to define and apply "Skins" (Microsoft 2002) to the application. These skins control the appearance of the user interface without affecting its core logic. Current work such as Longhorn (Microsoft 2003) attempts to further separate out appearance from the program by using XML specifications to control appearance.

This paper raises the question of whether we can take a similar approach with the "feel" (or behaviours) of an application. If we abstract these behaviours into a standard set of objects then we could take a standard Windows application and add a Macintosh feel, or a Tablet PC feel without having to change the application code.

2. THE CONCEPT

The "feel" of an application is broadly defined by the set of behaviours that it exposes to the user. Typically, these behaviours are hard coded into each application. This paper first uses two examples (Soft Input Panel and VerticalScrollability) to look at how individual behaviours can be abstracted from application code and encapsulated into separate objects. The paper next looks at how a set of these behaviours can be packaged together as a "feel" and finally addresses the issue of how this feel can be implemented as a plug-in so that the application can be configured to take on different behaviours without recoding.

The sample code for all the examples in this paper uses Visual Studio.Net to target the DotNet compact framework on the Pocket PC. The con-

cepts illustrated can of course be applied to all OO environments. The sample code can be found on the author's Website.

3. POCKET PC SOFT INPUT PANEL

The Pocket PC does not have a keyboard so data entry is accomplished by handwriting recognition or by using a stylus to pick out letters from a keyboard image displayed on the screen.

Because of the limited screen real estate, best practice (Microsoft 2003) has evolved to suggest that the soft input panel should be displayed whenever the program is expecting text input and hidden otherwise. In practice, this means that the programmer traps the got focus and lost focus events of textboxes displaying and hiding the panel as required.

The sample code shows how this behaviour can be abstracted from the application code and implemented in a separate object. The object hooks up all the events necessary to implement the behaviour. This behaviour management object can then be added to any form without the need for coding event procedures for individual controls.

4. SCROLLABILITY

When developing the DotNet compact framework for smart devices such as the Pocket PC, Microsoft went to great lengths to minimise the memory footprint. A consequence of this is that many familiar objects lack the functionality of their counterparts in the full framework. In particular, forms and panels do not have any built-in capability for scrolling.

Although reliance on the stylus means that excessive use of scroll bars is probably a bad idea, it is important to keep the focused text box visible whenever the soft input panel would otherwise obscure it. Limiting text boxes to the upper part of a form is unnecessarily restrictive and the ability to scroll the main form programmatically is an easy way to achieve this functionality without that restriction.

The sample code shows how we can implement on demand scrollability as a separate behaviour object. We can then just add the behaviour to a form (or indeed to any other container).

4. PACKAGING BEHAVIOURS INTO FEEL

Although trivial, the two examples given above illustrate how behaviours can be implemented externally to the application code. We could of course package these as toolbox components that the programmer could drop onto a form to get the behaviours, but we'll take the idea a little further first.

The next step on our journey is to package together a number of behaviours to give a "Feel". To keep things to the essentials, we'll just use our scrollability and input panel behaviours. We will need to coordinate these two behaviours so that they work seamlessly together; it is this ability for behaviours to work in a coordinated and seamless manner that transforms a set of disparate behaviours into a "feel". To facilitate this coordination, it is important that the behaviour objects themselves expose a well thought out event model.

When we add this feel to a form, a scrollbar will appear automatically whenever the form needs it, the soft input panel will appear whenever the focus reaches a text box, and the form will automatically scroll to keep the text box visible if the soft input would otherwise obscure it.

The code for this is trivial; the constructor just initialises the two behaviour objects and sets up listeners to the SIP panel events. A real-world Feel object would not be much more complex; just creating a few more behaviours, and possibly configuring then from an "appSettings" xml file.

5. FEEL AS A PLUG-IN CAPABILITY

The final stage in this journey is to add plug-in capability. Let's assume that that there is a configuration utility that enables a user to choose from several "Feels" that are available for a platform. This utility could apply this choice by modifying a global configuration file, or a local application file.

Rather than adding the feel directly to a project, the programmer adds a FeelManager object. This object then identifies the Feel configured by the user and applies it to the application.

The sample code demonstrates the techniques used to achieve this plug-in based on configuration

in an appSettings file. Writing the configuration utility is left as an exercise for the reader.

6. SUMMARY

In general, behaviours can be abstracted from any system of objects that has a well designed event model and implemented in separate behaviour objects. These behaviour objects can then be packaged into a "Feel" object that implements a standard set of coherent behaviours. Plug-in technology enables different Feels to be applied to an application after development.

7. POSSIBLE FUTURE WORK

The next logical step is to develop sets of standard behaviours that conform to platform guidelines. This would facilitate cross-platform development.

Further development would enable CHI specialists to develop "feels" that represent best practice in usability and to implement these feels in software that can be applied to any existing applications that follow these guidelines without the need to modify, or even have access to, the application source code.

REFERENCES

The author's Web site can be reached from http://www.manukau.ac.nz/

Microsoft, 2002, "Explanation of User Interface (UI) Skins", Downloaded on May 11th 2004 from http://support.microsoft.com/default.aspx?scid=kb;en-us;253739

Microsoft, 2003, "What is Windows "Longhorn"?", Downloaded on May 11th 2004 from http://www.microsoft.com/windows/longhorn/default.mspx

Microsoft, 2003, "Using the Input Panel Component", Downloaded on May 11th 2004 from http://samples.gotdotnet.com/quickstart/compactframework/doc/inputpanel.aspx