A Toolkit of Mechanism and Context Independent Widgets

Murray Crease, Philip Gray & Stephen Brewster

Department of Computing Science
University of Glasgow
Glasgow
UK
G12 8QQ
<murray/pdg/stephen>@dcs.gla.ac.uk

Most human-computer interfaces are designed to run on a static platform (e.g. a workstation with a monitor) in a static environment (e.g. an office). However, with mobile devices becoming ubiquitous and capable of running applications similar to those found on static devices, it is no longer valid to design static interfaces. This paper describes a user-interface architecture which allows interactors to be flexible about the way they are presented. This flexibility is defined by the different input and output mechanisms used. An interactor may use different mechanisms depending upon their suitability in the current context, user preference and the resources available for presentation using that mechanism.

Introduction

Szekely [1] describes four challenges that need to be met by human-computer interface technologies. Interfaces need to be able to automatically adapt themselves to support the user's current task. Interfaces need to be able to support multiple platforms. Interfaces should be tailorable to the users current needs. Interfaces should be able to handle both input and output using multiple mechanisms. This paper describes a toolkit of interactors which are designed to provide a solution to the last three of these challenges, with an implementation which supports adaptable and tailorable output. An interface's support of multiple platforms and modalities combined with ease of tailorability is gaining an increased significance due to mobile devices becoming ubiquitous. It is no longer sufficient to design a human-computer interface for a static platform such as a workstation in an office. Rather, interfaces need to be able to adapt to different contexts of use. The interface may adapt in the way it accepts input, or in the way it is presented to the user. This adaption may occur for two reasons, a change in the resources available to the widgets (resource sensitivity) or a change in the context the platform is situated in (context sensitivity). The resources available to the widgets may vary due to a change in platform, for example from a workstation with a large monitor to a personal digital assistant (PDA) with a limited screen size; but it could also be due to the removal of a resource or

greater demand being placed on a resource. For example a MIDI synthesiser used to provide audio feedback could be disconnected or have a reduced number of free channels due to the play back of a MIDI sequence. The context of the platform, and therefore the widgets, could vary due to differences in the environment over time, for example the sun could gradually cause more and more glare on a monitor as the day progresses or a shared lab may have different ambient noise levels as the number of people in the room changes. The context may also vary as, for example, the location of a mobile device changes..

By using multiple mechanisms, the widgets can be more flexible in the way they can adapt themselves. The different characteristics of the different mechanisms, on both input and output, allow the demands made by the current context to be best met. Equally, with all the options available, it is important that users are able to tailor the widgets to their personal requirements. These requirements may be a high level preference, for example a preference for a particular colour, or may be based upon a need, for example if visual feed back is of no use to a visually impaired user.

Related Work

The Seeheim model [2] was one of the first user interface models to separate the user interface architecture into monolithic functional blocks. Three functional blocks were defined: the presentation system which handled user input and feedback; the application interface model which defined the application from a user interface's point of view and the dialogue control system which defined the communication between the presentation system and the application interface model. Like Seeheim, the toolkit architecture presented in this paper has a monolithic presentation component (albeit with separate blocks for input and output), although the dialogue control system is distributed through out the widgets. The toolkit architecture does not deal with application models because it is solely concerned with the input to the widgets and the output generated by the widgets.

MVC (Model View Controller) and PAC (Presentation, Abstraction, Control) [3] are both agent based models, where an agent is defined to have "state, possess an expertise, and is capable of initiating and reacting to events." [4]. An interface is built using hierarchies of agents. These agents represent an object in the application. In MVC, the model describes the semantics of the object, the view provides the (normally visual) representation of the object and the controller handles user input. In PAC, the abstraction describes the functional semantics of the object, the presentation handles the users interaction with the object, both input and output and the control handles communication between the presentation and the abstraction as well as between different PAC agents. The toolkit is object-oriented like both MVC and PAC, with each widget (or agent) encapsulated into different objects. Our toolkit, however, does not define the whole user interface in terms of a hierarchy of agents, but rather defines the individual widgets without specifying their organisation. Like the MVC model the toolkit separates input and output, although unlike MVC, the toolkit's widgets do not have a controller type object because it is concerned purely with input to and output from individual widgets. It would be possible, however, to

build an MVC type architecture around the toolkit. Like PAC, the toolkit abstracts the widgets, but unlike PAC, the toolkit's abstraction is only aware of the widget's state but is not aware of the underlying application semantics. This is because the toolkit is designed as an extension of the Java Swing toolkit [5] allowing it to be easily incorporated into existing Java applications.

Previous solutions to the challenge of a toolkit suitable for multiple platforms have included virtual toolkits. These toolkits layer the user interface architecture, extracting the generic components into portable layers which sit on top of platform dependent layers. The SUIT system [6] was designed to run on three platforms, Mac, UNIX and Windows. The user interface was split into two layers on top of the platform dependent toolkits. The toolkit layer provided the tools necessary to implement the interface. The graphics layer provided a well defined graphical layer which could be easily ported between platforms. The XVT system [7] added a single, platform independent layer to the toolkits of the two platforms (Mac and Windows) supported. This layer mapped XVT commands into appropriate commands for the platform. These solutions provide a means to produce user interfaces for multiple platforms. Our toolkit relies on the portability of Java to ensure the interface can run on different platforms, but extends the notion of portability to include the resources available to the platform and the context the platform is running in

The Garnet system [8] is a set of tools which allow the creation of highly interactive graphical user interfaces, providing high level tools to generate interfaces using programming by demonstration and a constraints system to maintain consistency. The Garnet toolkit allows the graphical presentation of its widgets to be easily modified by changing the prototype upon which the widget is based. Doing this will update all dependent widgets. This is analogous to changing the design of output for a widget in an output module of our toolkit.

The HOMER system [9] allows the development of user interfaces that are accessible to both sighted and non-sighted users concurrently. By employing abstract objects to specify the user interface design independently of any concrete presentation objects, the system was capable of generating two user interfaces which could run concurrently for the same application. This allowed sighted and non-sighted users to co-operate using the same application. Unlike our toolkit, the HOMER system developed two interfaces, using two separate output mechanisms rather than have one interface which can switch between multiple mechanisms as and when required, using several concurrently if appropriate.

Alty *et al*. [10] created a multimedia process control system that would choose the appropriate modality to present information to a user. This would allow more information to be presented by increasing the bandwidth the interface could use. Additionally, if the preferred modality is unavailable if, for example, it is already being used for output, the system would attempt to present the information using an alternative. It was found, however, to be almost impossible to specify how these switches should be made due to their general nature. To limit the complexity of the system, a user-interface designer would supply it with valid options for output modalities. Our toolkit avoids this problem by avoiding generic solutions, but handling specific situations individually.

The ENO system [11] is an audio server which allows applications to incorporate audio cues. ENO manages a shared resource, audio hardware, handling requests from

applications for audio feedback. This shared resource is modelled as a sound space, with requests for sounds made in terms of high level descriptions of the sound. Like ENO, our toolkit manages shared resources, although the toolkit extends the concept by switching between resources according to their suitability and availability. Similarly, the X Windows system [12] manages a shared resource, this time a graphics server. Again, our toolkit extends this concept by managing resources in multiple output mechanisms and switching between them.

Plasticity [13] is the ability of a user interface to be re-used on multiple platforms that have different capabilities. This would minimise the development time of interfaces for different platforms. For example, an interface could be specified once and then produced for both a workstation with a large screen and a mobile device with limited screen space. This is achieved by specifying the interface using an abstract model, and subsequently building the interface for each platform using that platform's available interactors and resources. Like the toolkit, plasticity allows user interfaces to adapt to available resources, although the mechanisms used are different. Plasticity allows an interface to be specified once and subsequently generated for multiple platforms. The interfaces for each platform may use different widgets to suit the resources available. For example, a chart may be used on a workstation monitor, but a label may be used on a PDA display. The toolkit, however, adapts the input and output of an existing widget to handle differing resources. For example, a widget may be reduced in size according to the available screen space. Additionally, the toolkit attempts to adapt the interface across multiple output mechanisms whereas plasticity is only aimed at visual feedback.

Toolkit Architecture

Here, we describe the architecture of the toolkit which allows the widgets to be resource and context sensitive. To enable this, the behaviour of the widgets is separated from the input and output mechanisms used. This allows the widgets to switch between mechanisms without affecting their behaviour. Similarly, the widget's presentation options are controlled by a separate module to allow the easy tailorability of the widgets. Initially, we describe how input to the widgets is handled. The second section describes how the widget are presented and finally, we compare input and output, highlighting any symmetries or differences, and describing how the two are combined in a consistent fashion.

Input Architecture

To be flexible the widgets need to be able to handle multiple input mechanisms, with the potential for new input mechanisms, such as speech or gesture, to be added dynamically. Fig. 1 shows the architecture employed to enable this.

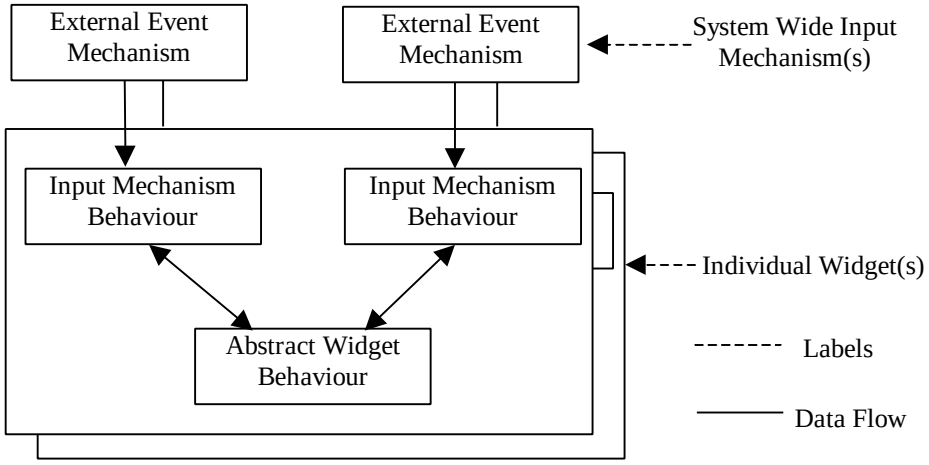


Fig. 1. Input Architecture

The abstract widget behaviour describes the generic behaviour of the widget. For example, a button would have the generic behaviour described by the state transition diagram shown in Fig. 2.

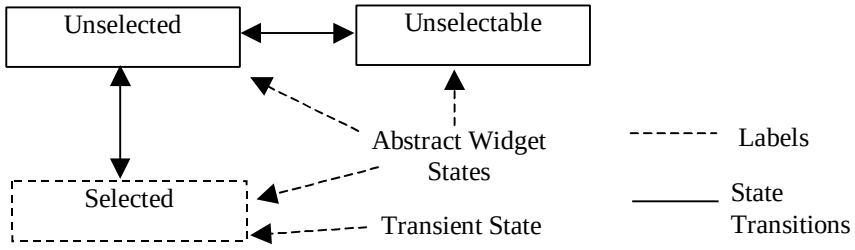


Fig. 2. Abstract behaviour of a button.

This behaviour is expanded upon by the input mechanism behaviour(s) for the widget. These behaviours conform to the abstract behaviour for the widget, but may include sub-states which more accurately specify the behaviour of the widget for a given input mechanism. For example, Fig. 3 shows a simplified state transition diagram describing the behaviour of a button when using a mouse as the input mechanism.

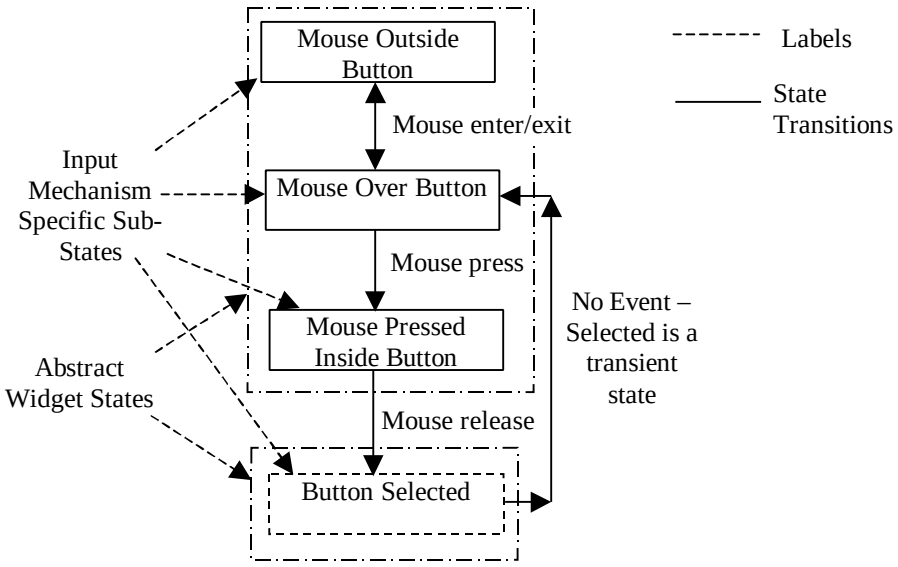


Fig. 3. Button behaviour for the mouse input mechanism (simplified).

The input mechanism behaviour(s) of a widget receive events from the relevant input mechanism(s). These events are then filtered appropriately according to the current state of the widget. If a relevant event is received, it is passed to the abstract widget state. If an event received by an input mechanism behaviour causes a change in the abstract state of the widget, the abstract widget state ensures that all input mechanism behaviours are notified so that the states of all the input mechanism behaviours are consistent.

For example, a button is in the unselected state, and it employs two input mechanisms, a mouse and speech input. If a user moves the mouse over the button, the appropriate event is received by the mouse input mechanism and is passed to the abstract widget behaviour. This event, however, does not affect the abstract state of the widget, so there are no concerns over the consistency of the states of the input mechanism behaviours. If the user was then to select the button using a speech command, this event would be received by the speech input mechanism behaviour and passed to the abstract widget behaviour. This event would change the state of the abstract widget to “selected”, so the state of the mouse input mechanism behaviour would have to be changed to selected to remain consistent. As selected is a transient state, the abstract widget behaviour would automatically have to update the states of all input mechanisms to the unselected state. This differs from the notion of data fusion in the MATIS system [14] where pieces of data, from separate input mechanisms are fused into one command, for example a request for a plane ticket where the destination and departue locations are given in two different modalities. In the toolkit, the analogous situation is that the user moves the mouse over the graphical

widget used to select one piece of data, but then actually selects the destination using speech input. Although two input mechanisms were used by the user, only one was used to make the selection. The use of the mouse in this case was redundant. If the user were to proceed with the selection using the mouse and selected the departure location using speech input, the toolkit would handle the two selections separately, and a mechanism such as data fusion would be required at the application level to determine the meaning of the pieces of data selected.

By separating the abstract behaviour of the widget from the input mechanism behaviour(s), the widget is insulated from any changes in input mechanisms. Indeed, the differences in input mechanisms are irrelevant to the abstract widget, and as such it is possible to replace one input mechanism with another, or even add a new input mechanism, providing multiple input mechanisms without affecting the widget in any other way. Which input mechanism(s) a widget uses is controlled by the user using a control panel (Fig. 4). This allows the user to add or remove input mechanisms to/from a widget.

Output Architecture

As with the input architecture, widget behaviour has been separated from the widget presentation, to allow the widgets to be flexible. Fig. 4 shows the output architecture.

The abstract behaviour of the widget requests feedback appropriate to its current state. This request is passed to the feedback manager which splits this request into multiple requests, one for each mechanism being used. Each new request is given three “weights”, reflecting user preference for this mechanism, resource availability for this mechanism and the suitability of this mechanism in the current context. These mechanism-specific requests are passed to the appropriate mechanism mappers, where any user preferences are added to the request. These preferences may indicate, for example, a 3D effect graphical output style or a jazz audio output style.

These amended requests are passed on to the rendering manger. This monolithic component ensures that the feedback requested by different widgets will not conflict. Because it is monolithic, the rendering manager is able to oversee the requests made by all the widgets and consequently is able to highlight any possible conflicts. If the output mechanism is able to handle such clashes, then the requests are passed on, otherwise the rendering manager will change the request to a different, more suitable mechanism. As with the input mechanisms, the output modules are monolithic components. Consequently, it is possible to replace an output mechanism with a new one, or to add a new output mechanism without affecting the existing output modules.

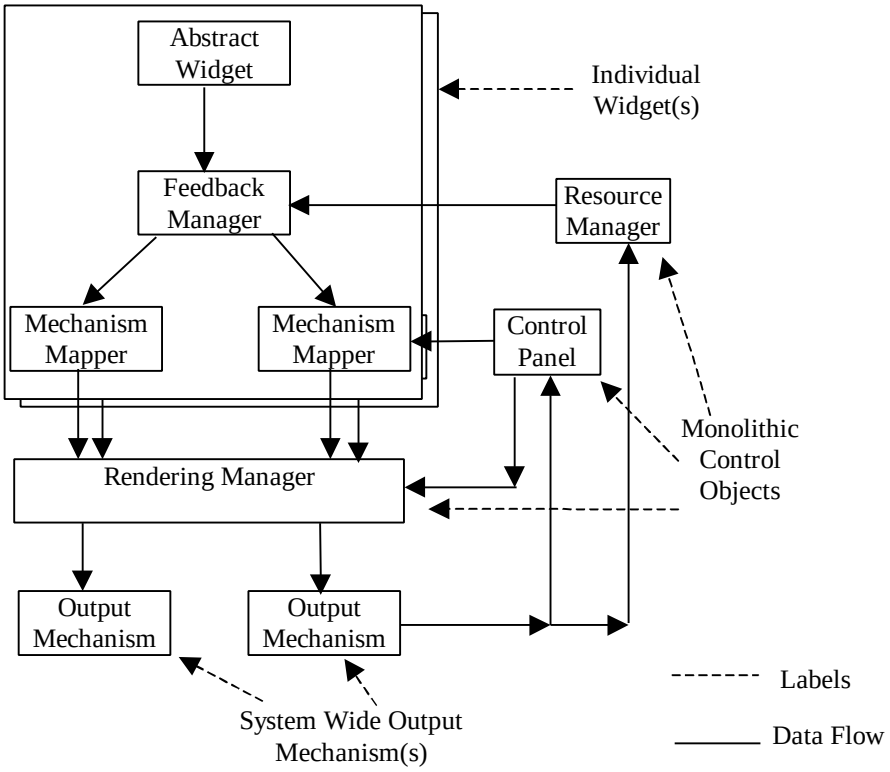


Fig. 4. Output Architecture

The widget's output can be modified in three ways. Users can specify a weight describing the importance they associate with a particular output mechanism for a widget. This could be set due to preference, the user prefers the audio feedback to be a little less intrusive, or user need, visual feedback is of no relevance to a visually impaired user. Output mechanisms can specify the resource availability weight. This could be affected by, for example, the screen size, or number of MIDI channels available. The suitability of a particular resource can be specified by external modules which can be added to the toolkit. These could, for example, measure the ambient volume, the brightness of the surroundings or the motion of a mobile device and adjust the feedback accordingly.

The control panel allows users to personalise the widgets output using parameters supplied by the output mechanisms. Users are also allowed to set a weight for their preference for each of the different output mechanisms here. The control panel also has an API allowing software control over these functions. The resource manager controls the values for the weights for the different output mechanisms. The user preference weight is received from the control panel. The resource availability weight is received from the respective output mechanisms, and the resource suitability weight

is received from software components via the resource manager API. It is thus possible to build software modules to extend the toolkit which can detect the suitability of a particular output mechanism and inform the toolkit when appropriate.

A Comparison of the Input and Output Architectures

Although the input and output architectures are largely symmetrical there is one difference due to the fundamental asymmetry of input and output, namely the output architecture has a monolithic control structure, the rendering manager, whilst there is no analogous structure in the input architecture. Whilst it would be possible to include an input analogue in the architecture, it is difficult to see what purpose it would serve. The rendering manager is necessary on output because the output for different widgets could potentially clash. The rendering manager avoids this problem by tracking the feedback generated system wide. An input analogue is unnecessary because such clashes on input cannot occur given the way input is currently handled.

Ensuring Input and Output Are Consistent

Because both the input and output behaviours are not encapsulated within the widget, there is a danger that the input and output behaviours might not be consistent. For example, a widget's graphical representation could change size, but because the output behaviour is separate from the input behaviour, the area deemed to be valid for input to the widget may not be changed. To avoid this scenario it is necessary for there to be some communication between input and output mechanisms. This communication, however, needs to be controlled as there is little point in, for example, an audio output mechanism communicating changes in its presentation to a screen-based mouse input mechanism.

To ensure that the appropriate input and output mechanisms communicate, each mechanism is associated with an interaction area. These areas define the space in which a user can interact with the widget. For example, a mouse input mechanism and a graphical output mechanism would share the same 2½D interaction area. Each mechanism that uses a particular interaction area will share common characteristics. For example, mechanisms that use the 2½D interaction area described above will share the same co-ordinate system. The communication between input and output mechanisms is brokered by a communication object in each widget. This object controls the communication between all the input and output mechanisms ensuring that they are all consistent in their behaviour. The communication object would receive messages from all output mechanisms, and pass these messages on to input mechanisms that share the output mechanism's interaction space.

There is a potential danger that some input mechanisms may require analogous output mechanisms or vice-versa. For example, a gesture input mechanism where you point to the location of a spatialised sound source to select the widget requires a suitable output mechanism. Equally, some mechanisms may not be suitable for a a particular widget. For example, many current haptic devices which require the user to actively grasp the device would not be suitable for feedback indicating the state of a

background task as it progresses. In this case, a haptic output mechanism would not be suitable. The solution provided by the toolkit is to supply it with default output mechanisms which are not platform dependent and are suitable for all widgets. A more generic solution would be to include constraints that mean all widgets must be able to be presented by at least one output mechanism, regardless of suitability or user preference.

Implementation

Java has been used to implement the toolkit. It was chosen due to its portability, adding to the flexible nature of the toolkit. The toolkit architecture has been fully implemented for the output, with three widgets (buttons, menus and progress bars) added to the toolkit so far. Output was implemented initially because we could take advantage of Java's built in event handling mechanisms. Although this isn't a suitable solution for the long term because it limits the use of different input mechanisms, in the short term it allows us to evaluate the effectiveness of the toolkit for output without the overhead of implementing the input architecture.

Rather than accepting events from independent input mechanism behaviour(s), the abstract widget encapsulates the input behaviour for the widgets using the Java AWT mechanism. The widget behaviours are specified using a state transition diagram which is hard coded into the abstract widget. Each node in the diagram listens for appropriate events, generating requests for feedback when the node is activated. When an appropriate event is received, the node is deactivated and the appropriate new node is activated. The requests for feedback are then passed to the widget's feedback manager where the request is split into multiple requests that are eventually received by the output mechanisms. Currently two output mechanisms are used, a graphical one based upon the standard Java Swing widgets and an audio one using earcons [15, 16]. Swing widgets are used as the basis for the graphical output because this allows us to use the AWT input mechanism, but it does have the disadvantage of not allowing as much flexibility in the way the widgets are presented. Additionally, this means that some of the output mechanism is encapsulated within the abstract widget along with the input mechanism.

The graphical presentation for the buttons and menus are generated in a similar way, by changing the size and colour of the widget appropriately. The progress bar, however, does not rely upon the AWT event system to receive events. The events are passed to it by the parent application. This allowed us to handle the graphical presentation of the progress bar in a manner more consistent with the design of the architecture, and to be more flexible in the way the widget is presented. The output module paints the progress bar from scratch in accordance with the request received, meaning the presentation can be changed in more interesting ways. Fig. 5 (a-c) show the same progress bar with different weights associated to the amount of resource available for graphical presentation.

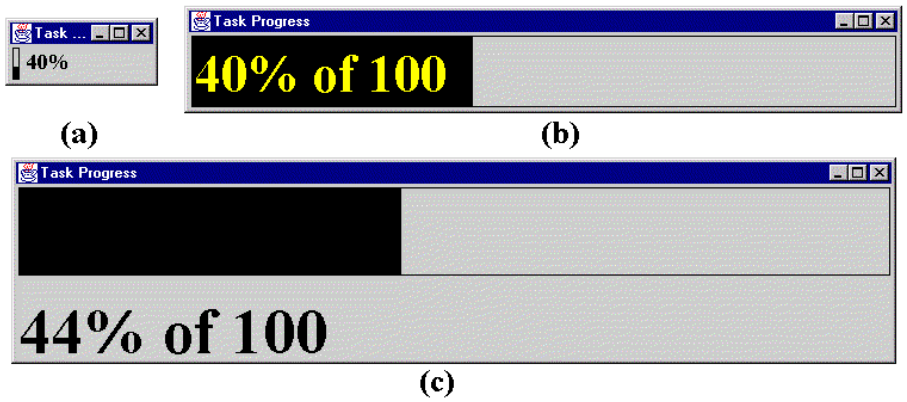


Fig. 5. Progress Bar Representations With Different Graphical Weights

The audio presentation for all the widgets is based upon previous work done on evaluating audio feedback in graphical user interfaces [17-19]. The audio presentation for all the widgets is flexible because it does not rely upon the Swing graphical output. Because of this it was possible to develop two different audio modules, which can be changed dynamically during use. The output modules are stored in Java jar files which are loaded as required. Should a developer wish to build a new output module, all he/she need do is build a java object which conforms to the API for an output module and load it into the toolkit.

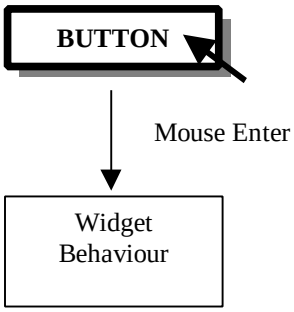
An interesting issue that arose as a consequence of using Swing widgets and therefore the AWT event mechanism was that it became apparent that assumptions had been made by the designers of Swing regarding the importance different events to software engineers using their widgets. For example, if a user presses the mouse button outside a JButton, the JButton is in a different state than if mouse button remains unpressed. Similarly, mouse releases outside a JButton are of relevance to the JButton. These events, however are not deemed to be of relevance to software engineers using these buttons and consequently are not readily available. The solution used in the toolkit is to resolve this problem is to have a global listener for mouse presses and release which listens for such events occurring on all components and passes them on to all interested widgets.

To further enhance the flexibility of the toolkit, an external module has been developed which measures the ambient audio volume of the environment around the device running an application built using the toolkit and adjusts the weighting for the suitability of the audio feedback, for example reducing the volume of sounds played if the environment is quiet (and *vice versa*).

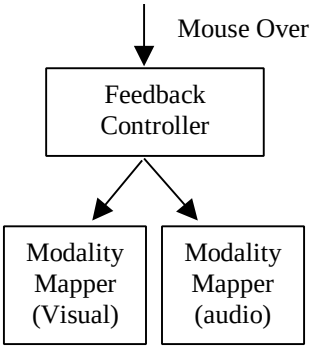
Worked Example

An audio/visual button is in its default state. The button is drawn as shown, with the cursor outside the area of the button. No sounds are played.

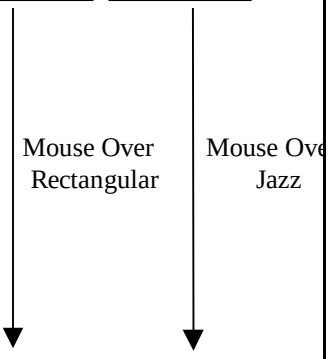
The mouse enters the button. This generates a Java AWT MouseEnter event which is passed to the abstract widget behaviour by the mouse input mechanism. The event is translated into a request for the appropriate, MouseOver, feedback.



The request is passed to the feedback controller. This widget has a weight of 30 for audio feedback, 50 for visual feedback and 0 for haptic feedback. Two requests are generated with appropriate weights, one for audio feedback and one for visual feedback. No request is generated for haptic feedback. Each request is passed onto the appropriate modality mapper.

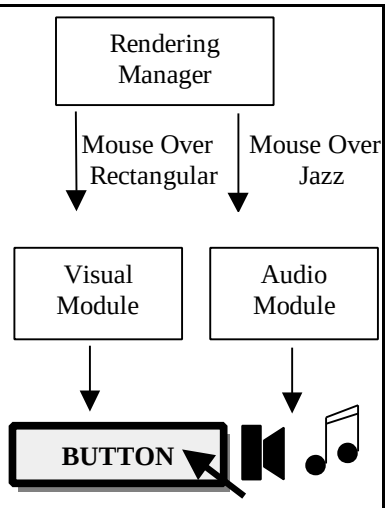


Each modality mapper modifies the event in accordance with user preferences set in the control panel. In this case, the style Rectangular is applied to the graphical request and Jazz is applied to the audio request. Each request is passed onto the rendering manager.



The rendering manager checks for potential clashes with these requests. In this case there are no clashes so the requests are passed onto the appropriate output modules.

Each output module receives the request and translates the request into concrete output. The visual module draws a rectangular button to match the user preference and shaded to indicate mouse over and the audio module plays a persistent tone at a low volume to indicate mouse over in a Jazz style to match the user preference.



Discussion and Future Work

This paper describes a user interface architecture that we have implemented which allows widgets to be flexible in several ways, using multiple input and output mechanisms, with the ability to add or remove mechanisms dynamically. The widgets can adapt their presentation according to the resources available for presentation, the suitability of different mechanisms given the current context and in accordance with user preference. This is achieved by separating the input and output mechanisms from the behaviour of the widget, allowing the widget to be used regardless of the mechanism. With the increased use of mobile devices, being flexible in this way allows the toolkit's widgets to be suitable for multiple platforms in multiple contexts without requiring any changes to be made.

Because the input and output mechanisms are no longer encapsulated in the widget, but are separate objects outside the widget, there needs to be some communication between them to ensure that the input and output mechanisms remain in a consistent state. However, there is also a need to limit this communication so that it only occurs between appropriate input and output mechanisms. To ensure this is the case, each mechanism is defined to operate within an interaction area. Only mechanisms sharing an interaction area will need to communicate. The implementation, shows that the architecture is effective, with widgets being able to modify their feedback according to resource and context requirements.

Acknowledgement

This work was funded by EPSRC grant GR/L79212.

References

[1] P. Szekely, "Retrospective and Challenges for Model-Based Interface Development," in *Proceedings of DSV-IS*, Namur, Belgique, 1996.

[2] G. E. Pfaff, *User Interface Management Systems: Proceedings of the Seeheim Workshop*. Berlin: Springer-Verlag, 1985.

[3] J. Coutaz, "PAC: An Object Oriented Model for Implementing User Interfaces," *ACM SIGCHI Bulletin*, vol. 19, pp. 37-41, 1987.

[4] J. Coutaz, L. Nigay, and D. Salber, "Agent-Based Architecture Modelling for Interactive Systems," *Critical Issues In User Interface Engineering*, pp. 191-209, 1995.

[5] Sun-Microsystems, "The Swing Connection", http://java.sun.com/products/jfc/tsc/index.html, as on 10/01/2000.

[6] R. Pausch, I. Nathaniel R. Young, and R. DeLine, "SUIT: The Pascal of User Interface Toolkits," in *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology, UI Frameworks*, 1991, pp. 117-125.

[7] R. Valdes, "A Virtual Toolkit for Windows and the Mac," *Byte*, vol. 14, pp. 209 - 216, 1989.

[8] B. Myers, D. Giuse, R. Dannenberg, B. Zanden, D. Kosbie, E. Pervin, A. Mickish, and P. Marchal, "Garnet: Comprehensive Support for Graphical Highly Interactive User Interfaces," *IEEE Computer*, vol. 23, pp. 71-85, 1990.

[9] A. Savidis and C. Stephanidis, "Developing Dual Interfaces for Integrating Blind and Sighted Users: The HOMER UIMS," in *Proceedings of ACM CHI'95 Conference on Human Factors in Computing Systems*, vol. 1, *Papers: Multimodal Interfaces*: ACM Press, Addison-Wesley, 1995, pp. 106-113.

[10] J. Alty and C. McCartney, "Design Of A Multi-Media Presentation System For A Process Control Environment," in *Eurographics Multimedia Workshop, Session 8: Systems*, Stockholm, Sewden, 1991.

[11] M. Beaudouin-Lafon and W. W. Gaver, "ENO: Synthesizing Structured Sound Spaces," in *Proceedings of the ACM Symposium on User Interface Software and Technology, 1994, Speech and Sound*, Marina del Ray, USA: ACM Press, Addison-Wesley, 1994, pp. 49-57.

[12] R. W. Scheifler and J. Gettys, "The X Window System," *ACM Transactions on Graphics*, vol. 5, pp. 79-109, 1986.

[13] D. Thevenin and J. Coutaz, "Plasticity of User Interfaces: Framework and Research Agenda," in *Proceedings of Interact'99*, vol. 1, Edinburgh: IFIP, IOS Press, 1999, pp. 110-117.

[14] L. Nigay and J. Coutaz, "A Generic Platform for Addressing the Multimodal Challenge," in *Proceedings of ACM CHI'95 Conference on Human Factors in Computing Systems*, vol. 1, *Papers: Multimodal Interfaces*, 1995, pp. 98-105.

[15] M. M. Blattner, D. A. Sumikawa, and R. M. Greenberg, "Earcons and Icons: Their Structure and Common Design Principles," *Human-Computer Interaction*, vol. 4, pp. 11-44, 1989.

[16] S. A. Brewster, P. C. Wright, and A. D. N. Edwards, "An Evaluation of Earcons for Use in Auditory Human-Computer Interfaces," in *Proceedings of ACM INTERCHI'93*

Conference on Human Factors in Computing Systems, Auditory Interfaces, 1993, pp. 222-227.

[17] S. Brewster and M. Crease, "Making Menus Musical," in *Proceedings of IFIP Interact'97*, Sydney, Australia: Chapman & Hall, 1997, pp. 389-396.

[18] S. Brewster, P. Wright, A. Dix, and A. Edwards, "The Sonic Enhancement Of Graphical Buttons," in *Proceedings of Interact'95*, Lillehammer, Norway: Chapman & Hall, 1995, pp. 43-48.

[19] M. Crease and S. Brewster, "Making Progress With Sounds - The Design And Evaluation Of An Audio Progress Bar," in *Proceedings Of ICAD'98*, Glasgow, UK: British Computer Society, 1998.