UNIVERSITY
of
GLASGOW

A Toolkit Of Resource-Sensitive, Multimodal Widgets

Murray Crease

Submitted for the degree of Doctor of Philosophy

December 2001

Abstract

This thesis describes an architecture for a toolkit of user interface components which allows the presentation of the widgets to use multiple output modalities – typically, audio and visual. Previously there was no toolkit of widgets which would use the most appropriate presentational resources according to their availability and suitability. Typically the use of different forms of presentation was limited to graphical feedback with the addition of other forms of presentation, such as sound, being added in an *ad hoc* fashion with only limited scope for managing the use of the different resources.

A review of existing auditory interfaces provided some requirements that the toolkit would need to fulfil for it to be effective. In addition, it was found that a strand of research in this area required further investigation to ensure that a full set of requirements was captured. It was found that no formal evaluation of audio being used to provide background information had been undertaken. A sonically-enhanced progress indicator was designed and evaluated showing that audio feedback could be used as a replacement for visual feedback rather than simply as an enhancement. The experiment also completed the requirements capture for the design of the toolkit of multimodal widgets.

A review of existing user interface architectures and systems, with particular attention paid to the way they manage multiple output modalities presented some design guidelines for the architecture of the toolkit. Building on these guidelines a design for the toolkit which fulfils all the previously captured requirements is presented. An implementation of this design is given, with an evaluation of the implementation showing that it fulfils all the requirements of the design.

Contents

iii

Table of Figures

List of Tables

Acknowledgements

I would like to thank my supervisors Stephen Brewster and Philip Gray for their support and friendship throughout my PhD. Without their guidance and encouragement this thesis would not have been possible. Without Stephen Brewster's knowledge and assistance the sonically-enhanced progress indicator described in Chapter 5 would never have been developed and similarly without Philip Gray's vast experience of user-interface architectures and systems the toolkit of multimodal widgets may never have come to fruition. In both cases enough freedom (or should that be rope) was given to allow me to explore my own ideas.

I would also like to thank the various office-colleagues who over the years have put up with my peculiar whims and rants. Although too many to mention (and for fear of omission) you all know who you are. I would also like to thank the various Java gurus in the department who have helped me in my many hours of need whilst implementing the toolkit. Thanks to your patience I now hope to be ranked as a (junior!!) member of your number. I would also like to thank Jo Lumsden whose patient and thorough reading of this thesis has saved several red-faces at the very least.

Finally, I would like to thank my family for their support throughout my work on this thesis. Especial thanks goes to my wife Linda, who has had to endure the worst of the lows and suffer the long nights necessary for the completion of this work. Without her support none of this could have been done.

Declaration

The design and evaluation of the sonically-enhanced progress indicator presented in Chapter 5 has been published at ICAD'98 and Interact'99 [44, 45]. The design and implementation of the toolkit of multimodal, resource-sensitive widgets presented in Chapter 6 and Chapter 7 has been published at Interact'99, DSVIS'00 and HCI'00 [46-48]. These were jointly authored papers with Stephen Brewster and Philip Gray. This thesis only exploits those parts of these papers that are directly attributable to the author.

Chapter 1: Introduction

1.1 Motivation

The question that motivates this thesis might be stated as follows:

> How can the use of audio feedback best be employed at the human-computer interface to maximise the effectiveness of the interface, and what user interface architecture features are required to enable this?

In this section, the way different users of existing user interface architectures interact with them is described in the form of different scenarios. Two different users are considered: the designers of feedback and the end users of the interface. From these scenarios a more detailed description of the problem area is given and the approach this thesis uses to try and resolve these problems is briefly described.

1.1.1 Scenario 1: Designing Feedback

The design of user interfaces can be considered the combining of existing interface components to form an effective and easy to use interface which allows a user to perform certain tasks. This scenario describes the steps a designer may take when building such components from scratch or, as described below, modifying an existing component. In this case, the scenario describes work previously done to add audio feedback to an existing interface component: the pull down menu [18].

The first step undertaken was to analyse the problems with the existing, graphical, feedback for the pull down menus. This analysis looked at the way users interact with pull down menus and what feedback should be provided. The analysis then looked at the feedback that was actually provided to the users and attempted to discover any discrepancies between the ideal and actual feedback. In the case of the pull down menu it was found that it was possible for a user to make an error which went unnoticed because the difference in feedback between the different possible outcomes when making a selection from a pull down menu can be easily overlooked[1].

---

[1] Brewster [22] defines a piece of feedback that a user can overlook as avoidable. A glossary of specialised terms used in this thesis is given in Appendix B.

Based upon this analysis, the design of additional feedback was undertaken. This design took the analysis described above and chose sounds to fix the problem; i.e. to differentiate the different possible outcomes of a menu selection. Audio feedback was chosen as the analysis indicated that additional graphical feedback would also be overlooked unless it was overly intrusive. Four sounds were created to solve this problem. Sounds were played when the cursor was over the menu, when the cursor was over a valid menu item and when a menu item was correctly selected. The fourth, error, sound was played instead of the selection sound when the cursor had been over the selected menu item for less than a specified length of time[2]. The pitch of the over item sound depended upon which item the cursor was over (the pitch alternated between items) and the pitch of the selection sound was the same as the pitch of the selected item's "mouse over" sound.

Once the design of the new feedback had been decided, an application using the new menu had to be implemented to enable the new design to be evaluated. The implementation was done on an Apple Macintosh computer using Apple's Toolbox interface objects. The standard pull down menu was implemented in the Toolbox as a single method. When the mouse was pressed in the menu bar the application called the `menuSelect` procedure which handled all the user's interaction with the menu as well as any graphical feedback produced. Once the user's interaction with the menu was completed the `menuSelect` procedure returned an integer indicating which menu item, if any, had been selected. This implementation of the pull down menu allowed no leeway for modifying the feedback produced as there were no hooks upon which to hang any new pieces of feedback. The design described in the previous paragraph, for example, required the menu item the cursor was over at any given time to be revealed but this was not the case. It was necessary therefore to implement a new procedure to replace the existing `menuSelect`. This new procedure duplicated all the mouse event handling and graphical feedback produced by the existing procedure but in addition provided the new audio feedback described above. This was a non-trivial task requiring the entire widget to be re-implemented at high cost in terms of programmer time.

Once the evaluation of the addition of audio feedback to pull down menus was concluded it was shown that the sounds were an effective enhancement to the standard graphical pull down menus allowing the experimental participants to recover more quickly from any errors they made. The addition of the sounds to the existing graphical pull down menus was not simple, however. All the existing work that had been done in producing the graphical feedback had to be reproduced in addition to the work required to produce audio feedback.

1.1.2 The Problem: Assumptions About What Information is Important

Although it was possible to build pull down menus which had audio as well as graphical feedback, it was not as simple as adding the sounds to the existing graphical feedback. Rather, it was necessary to re-implement the graphical feedback to enable the audio feedback to be added. This was because the designers of the existing graphical pull down menu made some assumptions about the way it was going to be used. Firstly, by

---

[2] This time was defined as 117 msec. A fuller description is given in Section 3.4.1.

encapsulating the graphical feedback for the menu within a procedure that could not be modified they assumed that the graphical feedback would not be changed or added to. Secondly, by only providing the information about which menu item was selected (if any) they assumed that only some of the information about the user's interaction with the menu would be of any interest to the application containing the menu.

By making these assumptions (consciously or otherwise), the designers of the existing pull down menu had made the modification of the existing interaction object harder. Additionally, they had reduced the likely use of any new designs of feedback (e.g. audio feedback) for this interaction object. Because a new procedure had to be written to allow the addition of audio feedback, to change an existing application from using the existing interaction object to the new one would require a recompilation of the code. If, for example, there were multiple designs of feedback for the same interaction object a new version of the application would have to be built for each design.

1.1.3 Scenario 2: The Use of Such a User Interface

In this section, a brief scenario is described which highlights the way a user interface built with sonically-enhanced interaction objects could be expected to work. Consider a user running a range of standard applications built with sonically-enhanced interaction objects on his desktop machine. His machine has a large, colour monitor and a high quality MIDI synthesiser attached. It is possible, therefore, for the user interface to use both graphical and audio feedback that is of a high quality. Perhaps the office he is working in is empty apart from him so the audio feedback can be at a low volume and still be discernible. Although he is using a large monitor it is still cluttered because of the number of applications running. Because of this the progress indicators for some background tasks being performed are occluded. To ensure the progress indicator's information is available to the user, they need not be presented graphically, but may be presented only using audio feedback. Because there are several progress indicators running simultaneously the audio feedback they provide has been minimised to reduce the amount of interference between the different sounds.

Suppose that as the day progresses, the ambient noise level in the room increases. Perhaps more people come into the office and the number of cars driving past the window increases. The volume of the audio feedback is now too low to be discerned so the interface increases it slightly[3]. It is now at a level where it can still be discerned by the user but it is still quiet enough not to annoy others in the room. Should the ambient volume in the room return to its earlier, lower level the interface will in turn reduce the volume of its audio feedback. If, on the other hand, the ambient volume increases to such a level that the audio feedback is unable to be discerned and the volume control is at maximum, the progress indicators would revert to being presented visually. The user now switches machines to a laptop he is using to prepare for a talk later in the week. He still uses the same applications as he was using on his desktop machine, but the resources available for presentation have reduced. The resolution possible on the smaller screen is less than that on the desktop machine's monitor. The laptop is not connected to a high quality MIDI synthesiser, but instead uses an internal sound card which has a lower quality synthesiser. The interface compensates for these limitations in

---

[3] The threshold volume for sounds will vary for different users so presumably the level has been set appropriately for the user.

the resources available for presentation by changing the way they are used. The reduction in screen size is compensated for by making the interaction components smaller. The way these components are drawn is also changed. On the large monitor they were presented in a 3D style, but on the smaller laptop screen they are drawn with simple outlines. Similarly, because the quality of the sounds that were played when the application was running on the desktop machine are no longer possible, simpler sounds are played which are not outwith the capabilities of the laptop's sound card.

Because he is running behind schedule, the user decides to continue working on this problem on the train on the way to his presentation. He has been allocated a seat in what has been designated the quiet coach. The laptop detects this (perhaps there is a Bluetooth[TM] [1] transmitter in the coach) and so all audio output is directed exclusively to the headphone socket. But, the user does not have any headphones connected to the laptop so the interface assumes that audio feedback is not possible and therefore switches all feedback to the visual channel. The same restrictions of limited screen space still apply, but they cannot now be compensated for by the use of audio feedback. The interface, therefore, has to decide how to handle this lack of resource. In this case, the only feedback that was previously not presented visually were the progress indicators. Because there is insufficient visual resource remaining to allow these to be displayed, their visual presentation will probably be occluded by other windows. To compensate for this, the most important piece of information about a background task, its completion, is presented unavoidably to the user, with the rest of the progress information remaining in the background. The interface chose to present the completion of the tasks in this way with the assistance of the user who felt that because the completion of the background tasks was of such a priority it was important to be informed of their completion.

Once the user arrives at the venue for his talk he links his laptop to the digital projector. This projector has a resolution greater than that available to the laptop, and the interface is able to take advantage of this increase in the available resource for visual presentation. It therefore increases the size of the visual presentation for the interaction objects displayed by the digital projector as well as rendering them in a 3D style as before. The display on the laptop screen remains unchanged.

1.1.4 Discussion of Scenario 2

The scenario described in Section 1.1.3 is an idealistic account of how a human-computer interface may one day be. It is a wide-ranging account which touches on many different aspects of the design of such an interface that has previously not been done. The first aspect is that the use of audio feedback is on an equal footing with the use of graphical feedback. Indeed, the two are seen to be interchangeable in some instances. This is a novel development because the use of sound at the human-computer interface has primarily been seen as an addition to the use of graphics, not as a potential equal or indeed, a potential replacement.

The second aspect of interface design which needs further development is the concept of presentational resource and its effect on the way the interface is presented. Some applications (e.g. Quake II [68] allows the visual and audio presentation to be modified according to the available resource - Figure 1.1) allow the user to specify the available resources and consequently change the presentation appropriately. The novel idea presented in Section 1.1.3 is the way the interface adapted to the availability of the presentation resources as

appropriate. For example, the progress indicators were presented sonically due to a lack of screen space. A further novel aspect of the scenario is the way the interface used the available resources differently according to their suitability. For example, the volume used to present audio feedback was adjusted according to the ambient volume of the environment. This implies a need for sensors which can be used to monitor the environment to enable the interface to be presented appropriately. In Section 1.1.3 several sensors are used to monitor different aspects of the environment such as the ambient volume and location as well as more system orientated sensors which were used to detect the presence of hardware (e.g. screen size, headphones and digital projector) or otherwise.
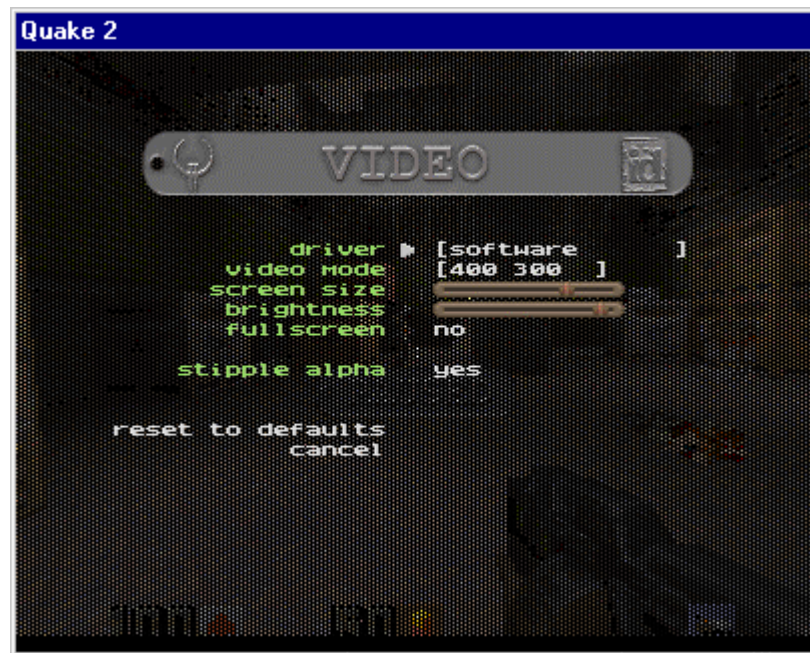
Figure 1.1 – Screenshot of Quake II from id Software showing how it allows the user to modify the presentation according to the available resources.

A third aspect of note is that of user control. Although not made explicit in the scenario, the scope for user control is great. For example, the system was unable to present the progress indicators sonically on the train, so the decision was made in association with the user to use visually demanding feedback only to indicate task completion. This decision was arrived at because the user decided the information was important enough to merit the interruption of the primary task. The system had no way of knowing if this was the case[4] and so could not have arrived at this decision independently, but could have, perhaps, suggested several alternatives from which the user could have selected one.

The final aspect requiring further work is that of the rules which control the way the interface is presented. The presence of such rules would imply the existence, either explicitly or implicitly, of models regarding all aspects of the interface and the context relevant to the interface. These models would be required to cover the

---

[4] A model of the user's task could enable the system to automatically make such decisions, but task modelling is outwith the scope of this thesis.

abilities of the different interface objects, the different presentation forms, the resources the presentation uses as well as any external influences which can effect the way these presentation resources can be used.
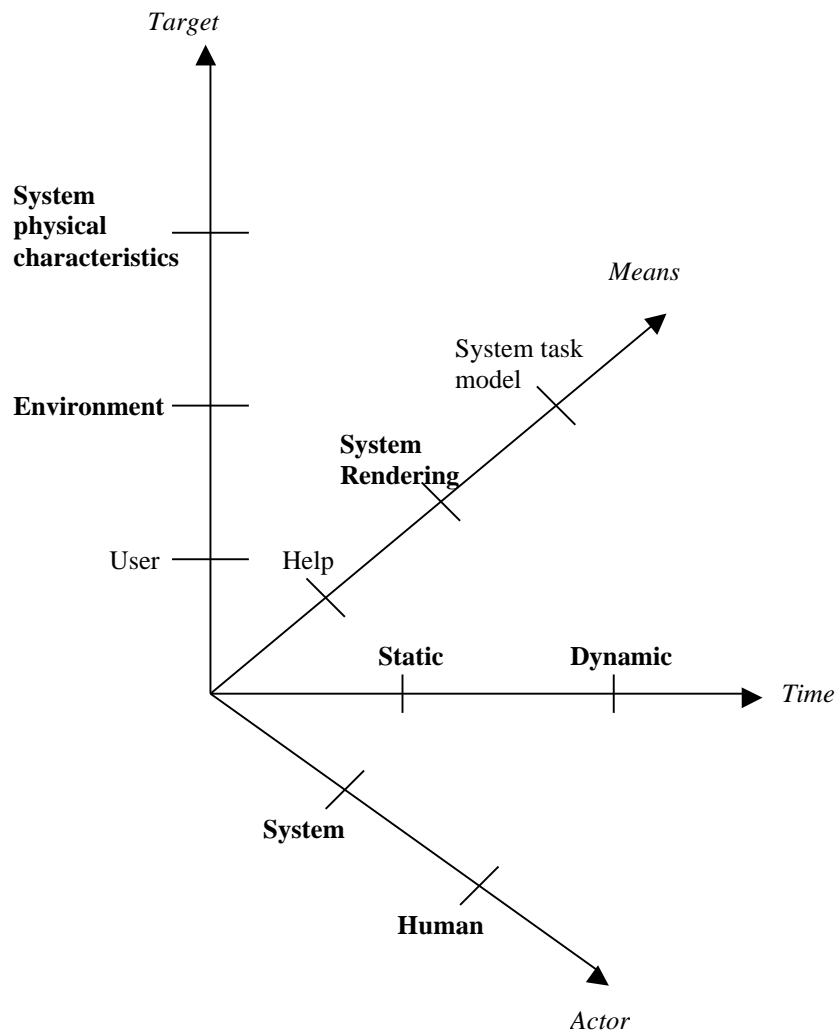
Figure 1.2 – A design space for adaptation at a high level of reasoning (from [109]). The areas of the design space which this thesis is concerned with are given in bold.

It is beyond the scope of this thesis to discuss all these interesting areas of research. Thevenin *et al.* [109] describe a design space of adaptation (Figure 1.2) which can be used to outline the scope of the thesis. The design space has four axes: Target, Means, Time, Actor. The Target axis describes the factors to which the system must adapt. Thus, the toolkit described in this thesis is concerned with variations in the system physical characteristics and the environment in which the system is running. Although it is feasible that it may in the future, the toolkit does not currently adapt to user behaviour. The Means axis describes how the system effects these adaptations. The toolkit is solely interested in the adaptation of the system rendering. The Time axis describes the temporal nature of any system adaptations. The toolkit is primarily interested in dynamic adaptations. Finally, the Actor axis describes the agent that undertakes the adaptation, a user or the system. i.e. whether the changes are made automatically or manually. The toolkit allows both these groups to

make changes. A short discussion on how the toolkit could be adapted to incorporate user and task models is given in Section 9.4.2.

A further area of work which this thesis does not concern itself with is the use of audio feedback in collaborative environments because this thesis is solely concerned with single users. Examples of audio used in such a way are described in Chapter 3 (e.g. [37, 62, 90]) but the issues such systems raise are not considered. Similarly, the related issue of audio feedback directed at one user being attention grabbing for a nearby user is not considered although the toolkit described in this thesis could be used to quickly implement systems which could allow the evaluation of potential solutions to this problem.

1.2 Thesis Aims

In this section the main aims of the thesis are summarised. The overall aim of this thesis is to describe a framework which allows the use of both sound as well as visual feedback at the human-computer interface. To achieve this two questions must be answered: How best to use sound at the human-computer interface and what architectural features are needed to build a framework or toolkit of widgets which supports this use of sound at the human-computer interface?

How Best to Use Sound at the Human-Computer Interface?

Although there are many examples of the use of sound at the human-computer interface (described more fully in Chapter 3) there has been no analysis done on how to support the incorporation of such sounds in the human-computer interface to maximise the effectiveness of the interface. The main aims of this part of the work are:

- To investigate existing uses of sound at the human-computer interface to determine the requirements the use of sound places on the designers of such interfaces.
- To undertake further work in the area of sonically-enhanced user interfaces that is required to complete this set of requirements.
- To investigate the ways audio and visual feedback can be used most effectively at the human-computer interface when the platforms used to support the interface may have differing abilities to generate different forms of presentation.

What Architectural Features Support the Use of Sound?

Many examples of user interface architectures and systems exist (described more fully in Chapter 4) but most do not explicitly support the use of sound at the human-computer interface. The main aims of this part of the work are:

- To review existing user interface architectures and systems to determine what architectural features, if any, they have which may be used to support the requirements described in the previous section.

- To design a toolkit of widgets which incorporates any architectural features found to support the use of sound at the human computer interface into an architecture with any new features found to be necessary.
- To implement this design and demonstrate that it meets its design goals.

1.3 Thesis Structure

The goal of this thesis is to investigate how best to use sound at the human-computer interface and to design a toolkit of multimodal widgets supports this use of sound. There are therefore two distinct strands of research that are intertwined in this thesis. Figure 1.3 summarises how the different chapters of the thesis relate to each other and to these two strands of research.

The Use of Sound at the Human-Computer Interface   **Toolkit of Multimodal Widgets**

Chapter 2: Thesis Concepts

Chapter 3

Review of the Use of Sound at the Human-Computer Interface

Chapter 4

Review of User Interface Architectures and Systems

Chapter 5

Design and Evaluation of a Sonically-Enhanced Progress Indicator

Chapter 6

Design of a Toolkit of Multimodal Widgets

Chapter 7

Implementation of a Toolkit of Multimodal Widgets

Chapter 8

Evaluation of a Toolkit of Multimodal Widgets

Figure 1.3 – Relationship of the different thesis chapters to each other and to the two main strands of research carried out.

Chapter 2 discusses some of the concepts that are used throughout the thesis. These concepts cover widgets, the basic user interface components; presentational resources, the resources used by the widgets to present feedback; output modalities, the different forms of feedback that make up presentation resources; and resource suitability, the suitability of the different output modalities according to the current source.

Chapter 3 reviews some of the most significant uses of sound at the human-computer interface. These uses range from a detailed evaluation of designs of sound for one widget to the implementation of a multi-user system which employs sound to enhance the visual interface. Chapter 4 reviews some of the most important user interface architectures. These architectures range from simple functional compartmentalisation to multi-threaded, multi-user agent-based architectures. This chapter also reviews some existing user interface systems, with an emphasis placed on existing systems that incorporate sound.

Chapter 5 describes the design and evaluation of a sonically-enhanced progress indicator. Before the design of a toolkit of widgets which used audio as well as visual feedback could be undertaken it was necessary to complete the set of sonically-enhanced widgets described in Chapter 3. The progress indicator completed the set of sonically-enhanced widgets. Chapter 6 describes the design of the toolkit of multimodal widgets and demonstrates how this design would handle a variety of scenarios. Chapter 7 describes the implementation of this design, highlighting any issues that arose during this implementation. The process of creating a new widget is also detailed. Chapter 8 reviews the implementation of the toolkit and discusses how well it meets the requirements of Chapter 6. Chapter 9 summarises the work described in this thesis, highlighting significant contributions it makes. This chapter also discusses some areas of future research that arise from the work described in this thesis.

Chapter 2: Thesis Concepts

This chapter defines the primary concepts used throughout this thesis. Three concepts are the primary foundation of this thesis:

- **Widget Toolkit** A widget is a user interface object that defines a specific interaction behaviour within a user interface and a model of information presented to the user. A widget toolkit is a collection of different widgets packaged to allow their simple inclusion in an interface.
- **Multimodal** The term multimodal is used to describe many different concepts. For the purposes of this thesis, a modality can be thought of as a form of presentation which uses a distinct lexical structure and presentation mechanism. So, multimodal refers to the use two or more such forms of presentation.
- **Resource-sensitive** For the purposes of this thesis, a resource is considered as being closely linked to a modality. The resource for a modality can be thought of as being the means to generate presentation in a particular modality. Being sensitive to a resource can mean both being sensitive to its availability and its suitability.

2.1 Widget Toolkit

Myers [86] defines a widget as "a way of using a physical input device to input a certain type of value". Furthermore, Myers [80] states that a toolkit of widgets will typically provide "an architectural framework to manage the operation of interfaces made up of these components". By combining these two facets, Myers reports that toolkits are successful because they "focus on just the low level aspects of the user interface, that are well recognised as important to have consistent and undesirable to re-implement". This section discusses these two facets of a widget toolkit in more detail.

2.1.1 Widgets

The Myers definition of a widget implies an inherent association with a physical input device but seems to separate the widget from its presentation. This would imply, for example, that a button can be selected using a pointing device such as a mouse without constraining how the button should appear on the screen. This, however, does not tell the complete story. A button can also, perhaps, be selected by tabbing into it and pressing return. Conversely, the presentation of the widget is typically not totally unconstrained. It is not usually possible to change the modality used to present the widget either dynamically or statically. For the purposes of this thesis, a widget shall be defined as an interactive component that encapsulates a user interface concept, its interactive behaviour and a model of the information presented to a user. Using this definition a widget is not fully abstract because the interactive behaviour, or input to the widget, is defined

but the presentation, or output of the widget, is not. Thus, a button could be considered as a selector that encapsulates the concept of a procedure that can be activated and can be selected by 'clicking' it with the mouse button. How it is presented, e.g. as a rectangular area on a screen, is not specified.

2.1.2 Toolkit

Myers states that a toolkit should not only contain the widgets used to build a user interface, but also an architecture to manage an interface built using these widgets. There are several aspects of the interface that such an architecture needs to manage. The presentation of the widgets needs to be managed so that the different widgets do not interfere with each other. Similarly, input to the widgets needs to be managed so that the different widgets receive only the events that are relevant to them. Finally, applications that use the toolkit to provide their interface must receive the appropriate notification of user input to the interface.

2.2 Multimodal

The term multimodal can be thought of as the use of multiple modalities. The definition of modality, however, is not as clear cut. One dictionary definition of modality is "a prescribed method of procedure" (the Concise Oxford Dictionary) and so using this definition multimodal can be considered as using multiple methods of procedure. It is therefore necessary to define what these methods of procedure can be. Coutaz *et al.* define multimodality as being the use of different input modalities [42] to a system. Brewster, however, defines multimodal as being the use of different sensory modalities to present information to a user [22]. Clearly, the term multimodal can be used for many different, and possibly conflicting, meanings. For the purpose of this thesis, multimodal shall be considered to mean the use of different output, or presentation, modalities.

2.2.1 Modality

The definition of multimodal given above states that it means the use of different output modalities. It is therefore necessary to consider just what an output modality is. Alty *et al.* define three standard terms as follows [4]:

- **Channel** The human sense used to perceive and communicate information, i.e. olfactory, taste, auditory, tactile and visual.
- **Media** The substance or agency by which information is conveyed to the user and vice-versa.
- **Mode** The style or nature of the interaction between the user and the computer incorporating the appropriate control actions both sides of the interaction may take.

The distinction between media and mode, however, is somewhat blurred. Alty *et al.* use the example of text and speech to highlight this. They suggest one possibility where natural language can be considered a mode of communication and text or speech are different media used to transmit the information. Conversely, they suggest that text and speech could be considered as different modes of communication and the display or

loudspeaker the media used to present the information. For the purposes of this thesis, a modality shall be defined as a pair consisting of the representational system (or mode above) and the physical i/o device (or media above) as proposed by Coutaz *et al.* [43]. It is not necessary to consider the channel as part of the definition as this will be encapsulated in the physical i/o device. Consider, for example, a bar graph presented both visually and haptically [114]. The visual modality is defined by the pair of the screen and the graphical notation used for the graph (in this case the height of the bars on that screen). The haptic modality is defined by the haptic device and the spatial representation used for the bars of this graph (in this case the height of the bars in the 3D space represented by the haptic device).

2.3 Resource-Sensitive

In the context of this thesis, 'resource-sensitive' means the ability to adjust the way a particular modality is used or which modality or modalities are used according to their availability and suitability. This section describes these two concepts more fully.

2.3.1 Presentation Resources

A presentation resource (or simply a resource) is the means required to generate output. The means vary according to the modality being used. A graphical, screen based display requires some form of screen which will have various attributes such as size (both in terms of physical size and resolution) and the number of colours available. Similarly an auditory display requires some form of audio hardware such as a MIDI (Musical Instrument Digital Interface) synthesiser or an internal sound card again having various attributes such as the number of notes that can be played simultaneously and the number of different instruments available. Although resource will typically be associated with a particular modality, different modalities may share some resources. Graphical and textual interfaces, for example, may share the resource of a screen.

2.3.2 Resource Availability

The availability of a resource describes the ability of the system to produce output in a particular modality that uses that resource. This could be determined, for example, by the available screen space or MIDI channels. The availability of a resource, however, may also be determined by underlying factors of the system. A machine with limited processing power, for example, may not be able to generate sounds of such high quality as a machine with more processing power. As implied by this last sentence the availability of a resource is not a binary condition. That is, it is not simply a question of saying a resource is available or it is not. There are, rather, degrees of availability that indicate, for example, not just how much but what quality of presentation is possible. For example on a hand-held device with a small visual display visual feedback is still available, but the amount of information that can be presented in this way is limited.

2.3.3 Resource Suitability

To determine whether or not a particular modality should be used requires more than simply determining whether the appropriate resources are available. It is also necessary to determine whether the use of these

resources is suitable. Alty *et al.* recommend that a particular form of presentation be chosen so that it maximises the user's comprehension [5]. This implies some form of dynamic allocation of resource to allow this to occur. It is not feasible to imagine, except in exceptionally rigid circumstances, that the pre-allocation of presentational resource would allow the maximisation of user comprehension. It may be, for example, that the user's context demands that a different modality be used. Perhaps in a situation where audio feedback would normally be used, an alternative modality is required because the ambient volume is too loud to permit the use of audio feedback. Similarly, perhaps the amount of information being presented to a user in a particular modality necessitates the use of an alternative. If, for example, the screen being used for display is too cluttered, more information presented visually would not be effective. Perhaps the modality required is being used to provide some other information. The task the user is performing and, indeed, the users themselves, can also effect the suitability of a resource. If a user is, for example, deaf then audio feedback is of no use to him/her. Equally, if a task is safety critical, the use of attention grabbing feedback which may, for other tasks, be found to be inappropriate could be deemed suitable.

The availability of a resource can, of course, be considered as a reason why it is not suitable. If the resource is not available then it is clearly not suitable for use. Because resource availability is such a fundamental piece of contextual information, this thesis distinguishes it from suitability. It is valid, however, to consider resource availability as no different from any other piece of contextual knowledge and therefore to assume that resource suitability is not a binary condition just as resource availability is not. The use of a particular modality may not be effective as it is currently being used but could be effective if the use of that modality is modified in some way. Audio feedback, for example, may not be discernible because the ambient volume of the environment is too loud. This is not to say that audio feedback should not be used, but rather that the way it is being used should be modified. In this case, perhaps simply increasing the volume of the feedback would be sufficient.

2.3.4 Resource-Sensitive versus Adaptive

In the previous sections, the concept of resource sensitivity was introduced. One of the aims of this thesis is to build a toolkit of widgets that can adapt the way widgets are presented in a resource-sensitive fashion. In many respects this can be considered an adaptive interface.

Schlungbaum [101] describes three different times that a user interface can be adapted to the end user:

- *adapted user interface* – the UI is adapted to the end user at design time.
- *adaptable user interface* – the end users themselves may adapt the UI.
- *adaptive user interface* – the UI changes its characteristics dynamically at runtime according to the end users' behaviour.

A resource-sensitive interface is clearly not an adapted user interface as the adaptation is not done at design time. A resource-sensitive interface could be adapted by the end users meaning that it could be classed as an adaptable user interface. Equally, however, the adaptation could be done dynamically at runtime implying that a resource-sensitive user interface is an adaptive user interface. The definition of an adaptive interface

13

given above, however, states that the UI changes according to the end user's behaviour. Although the adaptation of the interface may be as a result of a user's behaviour (e.g. moving from a quiet environment into a noisier environment) this behaviour is separate from the user's interaction with the interface. I therefore propose that for the purposes of this thesis the definition of an adaptive user interface be extended:

- *adaptive user interface* – the UI changes its characteristics dynamically at runtime according to **changes in its context of use.** The context includes the user, the task and the environment.

Using this definition, a resource-sensitive interface can be considered as an adaptive user-interface as it responds to changes in the environment, namely the availability of presentational resources and their suitability according to environmental constraints such as the ambient volume of the surroundings.

2.4 Conclusions

In this section, the three terms that are used throughout this thesis are defined. A widget toolkit is defined in terms of two concepts: widget, an interactive component which encapsulates user interface concepts but not their interactive behaviour or presentation, and toolkit, a set of widgets and an associated architecture which manages any interfaces built with these widgets. Multimodal is defined as using multiple output modalities with a modality defined as a pair consisting of the representational system and the physical i/o device. Finally, resource sensitivity is defined as the ability to adapt the presentation of widgets according to the availability and/or suitability of the different presentational resources, where such resources are the means required to produce feedback.

Chapter 3: Audio Human-Computer Interfaces

3.1 Introduction

This chapter provides some examples of existing human-computer interfaces that have been enhanced with audio. It begins with a discussion about the advantages and disadvantages of the use of sound at the human-computer interface, describing the different ways sound can be used, either as a supplement to any existing (typically visual) feedback or as a replacement for existing feedback. It then goes on to define different forms of audio feedback; speech, earcons and auditory icons, describing their benefits and drawbacks. Throughout the chapter several examples of actual interfaces and interface components that use sound to provide feedback are described. The lessons learned from these examples about the way sound should be used at the human-computer interface are highlighted in the discussion and are discussed at the end of this chapter with a view to their inclusion in the toolkit described in this thesis.

3.2 The Use of Sound at the Human-Computer Interface

Although most modern computers are capable of producing high quality audio feedback, most interfaces do not take advantage of this ability. Indeed, the majority of applications are silent. The one area where this is not the case is games. Many games produce high quality audio feedback, with the quality of this audio feedback often being one of the metrics used to evaluate games. Indeed, Buxton [32] reported that expert players were able to manage a higher score with the sound available than with the sound turned off. Walker *et al.* [113] report that not only is sound used as a means of monitoring off-screen events, but in one game the player is trained in the art of moving quietly to avoid detection.

This section will investigate reasons for using sound, as well as any problems that may be encountered if audio feedback is used. It will then describe different ways of using sound and their attendant advantages and disadvantages.

3.2.1 Why Use Sound?

The first question to ask is why use sound at all at the human-computer interface? We have been used to largely silent interfaces for many years now, so why change? One reason can be extracted from the work reported by Buxton that is described above. Why should it be the case that expert game players are able to achieve higher scores with the audio enabled? The implication is that the players are able to extract useful information from the audio feedback. This information may or may not have been presented to the players visually, but if it was the audio enhanced representation was found to be more accessible. This may well have been because of the typically visually demanding graphical presentation of a game where there are too many pieces of feedback for a player to be able to assimilate them all. Because the audio feedback can be processed concurrently with the messages processed via the player's limited visual focus, the player is able to receive more information using the two sensory modalities. This scenario can easily be extended to more everyday use of computers. The multi-tasking nature of most modern computers allows users to have multiple windows running with multiple applications open. This is despite the fact that the users will still have the same limited visual focus. There is still, therefore, scope for visual feedback going unheeded because the cues are outwith the user's visual focus. Thus, audio feedback can be used to enhance the effectiveness of a visual user interface.

There are, of course, instances when visual feedback provides no useful information to a user. If, for example, the visual cue is occluded behind another window then its information cannot be passed to the user. This will often be the case when processes which are running in the background try to convey some information to the user. One solution to this is to bring this process to the front, and consequently to the attention of the user. This, however, can be very annoying as the user's primary task is continuously interrupted by the background task grabbing attention. A second alternative is to alert the user visually without forcibly interrupting him/her. An example of this is the task bar on Windows™ machines. The task bar is a narrow strip running along one edge of the screen, which contains, amongst other things, information about the currently running tasks. If an application wishes to alert the user, it can place an icon into the task bar. Unfortunately, because the task bar is so narrow, these icons are typically very small. Even if they are flashing they will often go unnoticed by the user. To compound matters, it is possible that the task bar may itself be occluded by the foreground applications. In these instances, the visual feedback is of no benefit to the user. Audio feedback is a potential solution to this problem because of its omnipresent nature. This means that regardless of where the user's visual focus is, or how cluttered the screen is, audio feedback can still be discerned by the user.

Another reason to use audio feedback is that for the majority of users, the use of multiple sensory modalities is natural. For example, when crossing a road a pedestrian will typically both look and listen for oncoming traffic. Although this does not automatically imply that the use of multiple sensory modalities at the human-computer interface is the correct thing to do, it certainly gives an indication that it is worthy of investigation.

3.2.2 Some Problems With Sound

Although the use of sound can be beneficial, as described above, its indiscriminate use can be ineffective. This section describes some of the potential hazards in the use of sound at the human-computer interface. One issue that arises with the use of sound is that it is omnipresent. Although this has been described as an advantage in the previous section, potentially it can be a disadvantage. Unlike a visual display which typically will be directed at a single user and consequently only seen by that user unless others choose to look at it, audio feedback cannot be directed to one user. If the feedback provided to the user is in an audio form there is a risk that it will be discerned by others nearby. Although the audio feedback may provide a user with information that is relevant to his/her task, it is unlikely to be doing so for anyone else and so it may prove to be annoying. Conversely, because audio feedback cannot be directed at a single user, the information provided is no longer private, but may potentially enter the public domain. These problems can be solved by presenting the sounds at the appropriate volume level; loud enough so that the user can hear them, but not so loud that they can be widely heard. Another issue that arises from the omnipresent nature of sound is the question of annoyance. Visual feedback can be ignored, unlike audio feedback which cannot be avoided. Again, this problem can be overcome with the appropriate use of volume. If the volume of the feedback is sufficiently low, the user will be able to habituate, or allow it to fade into the background of consciousness, if he/she so desires[5].

Edworthy [52] suggests that the excessive use of sound may have more serious consequences for the user. She suggests that although the addition of sound may well improve a user's performance the audio feedback may well become annoying after long term exposure to it. Although many of the uses of auditory feedback have been evaluated, often with annoyance specifically in mind, most of these evaluations have not been carried out over the long term or in shared environments where annoyance is most likely to occur. Rather more seriously, however, Edworthy speculates that the use of auditory feedback could in the worst case affect the hearing of users in the long term or, more likely, add to the stress of the users. As with the question of annoyance, long term evaluations would have to be undertaken to determine the validity of these claims. A third concern raised by Edworthy is the danger that some sounds may mask other, more important sounds. The masking sounds may be part of the same system as the masked sounds or perhaps part of a nearby user's system. Part of the work described in this thesis is to try and prevent such masking by ensuring that the interaction objects which make up the user interface are sensitive to both the environment and feedback being produced by each other.

3.3 Different Forms of Sound

Up to this point, sound has been classed as one form of feedback. However, as with visual feedback, audio feedback can take several different formats. This section describes some of the main forms audio feedback can take, looking at the advantages and disadvantages of each form.

---

[5] Multi-user environments are beyond the scope of this thesis, but a brief discussion of such environments is included in Chapter 9.

3.3.1 Speech

Perhaps the most obvious form of audio feedback is speech. Most of us communicate using speech everyday and so it is natural to assume that this would be an appropriate format for audio feedback to take. Although screen readers, devices that can translate text to synthetic speech, are widely available speech is not always the most appropriate form of feedback to use. The major issue with synthetic speech is that it is very slow. To understand the information being presented, the user typically has to listen to the entire message from beginning to end. This is analogous to the problem text interfaces have where the user has to read through a list of items before finding the item they wish to select. Barker and Manji [8] claim that another limitation in the use of text is the lack of expressive capability which means that it often takes many words to describe relatively simple concepts. This idea can be extended to speech, compounding its lack of speed.

One possible way to improve synthetic speech would be to increase the presentation rate. Brewster reports work done on increasing the rate of presentation of synthetic speech [3, 104]. Both of these pieces of work found, however, that increasing the rate of presentation decreased the level of recognition, with the optimal rate for recognition being about 150 words/minute which is about normal reading rate. Arons [6] describes a system that allows users to skim recorded speech at varying rates. Users were able to skim through recorded speech at four different speeds according to their requirements. The slowest speed was simply the recording played back without any processing. At the second level, the speed of the playback was increased by selectively shortening or removing pauses in the speech. The third level allowed a user to gain quickly an overview of the content of the recording. Only the first 5 seconds of segments of the recording occurring after a long gap were played as such gaps typically signify a new topic or content words. The fourth level was similar but played the segments of the recording where the pitch of the speakers voice was highest as this typically signifies that key topic is being introduced. An evaluation found that the level four skimming was effective at extracting relevant points from the recording and for finding information although some users reported that the 5 seconds of speech played back at each segment was too short and they would have preferred some control over this length. The second level of skimming, the removal of pauses was the preferred method of skimming, however, as this technique did not remove any information from the playback, only the redundant pauses between people talking. Although the tasks used to evaluate these techniques (the retrieval of information from a lecture) did not allow the rate of presentation at which the playback no longer became perceivable, this work does indicate that the use of such techniques may be effective in allowing the presentation of speech to be sped up[6].

Another problem with speech is that it is attention grabbing. This means that it is difficult for users to habituate the sounds being presented if they are persistent. Patterson reports that this is because speech has a wide dynamic range, with vowels often being 30 dB more intense than consonants [94]. This means that if the level of the feedback was set to an appropriate level for the vowels, the consonants would be inaudible. This problem would also effect those nearby who are not using the interface as the intensity would not be at a level suitable to allow a sound to fade into the background. The combination of the slow rate of presentation

---

[6] It is not clear whether the techniques used would apply to languages other than English.

of speech with this difficulty in habituating it means that speech is typically ineffective at presenting continuous information.

There are, however, some occasions when speech can be effective at the interface. Speech is good for providing absolute information that more abstract forms of audio feedback are unable to do. If, for example, the values in a table are being presented sonically, speech would be able to give the exact values, whereas other forms of auditory feedback could give only approximations. This is because most auditory parameters (e.g. volume, pitch, *etc.*) are only suitable for the presentation of a few different values. Speech does not rely on these auditory parameters to provide information, rather the information is encoded in the words used within the sound. Speech, of course, is also very effective at presenting textual information sonically. Another advantage of speech is that it is typically easily understood because it is a form of audio feedback most people are familiar with. Unfortunately, it is not universal and as such needs to be localised.

Speech, therefore, is typically a poor choice for many types of auditory interface. It is slow, and can potentially be annoying, both to the users and any third parties nearby. Speech can be used effectively when exact information is required, because other forms of auditory feedback can only give approximations to values, and when textual information has to be presented. Although it is easy to understand when presented in the appropriate language, it is not universal and consequently needs to be localised.

3.3.2 Auditory Icons

An alternative to speech is the use of non-speech sounds. As with speech, these are used by most people in everyday life to aid in the gathering of information about their surroundings. Sound, for example, is used in addition to visual feedback to aid in many everyday tasks such as crossing a road. Gaver coined the phrase 'everyday listening' to describe the concept of identifying the attributes of the source of a sound rather than the attributes of the sound itself [60]. For example, when a door is shut, the listener will not perceive the sound in terms of the loudness, pitch and other such attributes of the sound, but rather will perceive the force with which the door was closed, the material the door was made of and perhaps the size of the room the door was closed onto. Furthermore, the sound produced will change according to the object that is the source of the sound and the interaction that produced the sound. In the above example, a very different sound will be produced if the door is replaced with a different, metal one. Similarly, if the door is kept the same, but is shut less forcibly a different sound will be produced. It is this combination of intuitive mapping of information onto sounds combined with the ability to parameterise these sounds along certain dimensions that inspired the creation of Auditory Icons [58, 59]. By mapping the information to be presented to the user onto sounds which are analogous to this information, it can be presented in an intuitive manner. For example, if a user selects an icon on the desktop, a different type of sound can be played dependant upon the type of object it is. Furthermore, this sound can be altered according to the properties of the object, such as is size, and the way the object is being interacted with. For example, in the SonicFinder which is described in more detail later in this chapter, Gaver suggests that a file could sound wooden with the wooden sound changing according to the interaction with the file and its properties. Applications could sound metallic with the metallic sound changing in a similar way according to the interaction and properties of the applications.

Gaver proposed a framework which allows the parameterisation of everyday sounds (Figure 3.1). This framework describes sounds in terms of three basic categories - solids, liquids and gases - with the attributes and events of these basic categories described towards the edge of the diagram. Towards the centre of the diagram, more complex events are described. At the boundaries between these categories, more complex events are found describing possible ways these categories can interact. These complex events are built using the basic attributes of their components thus allowing families of sounds to be created. Although this framework is only an initial attempt at categorising everyday sounds, it is useful when thinking about the way everyday sounds can be combined and consequently used to create families of auditory icons.
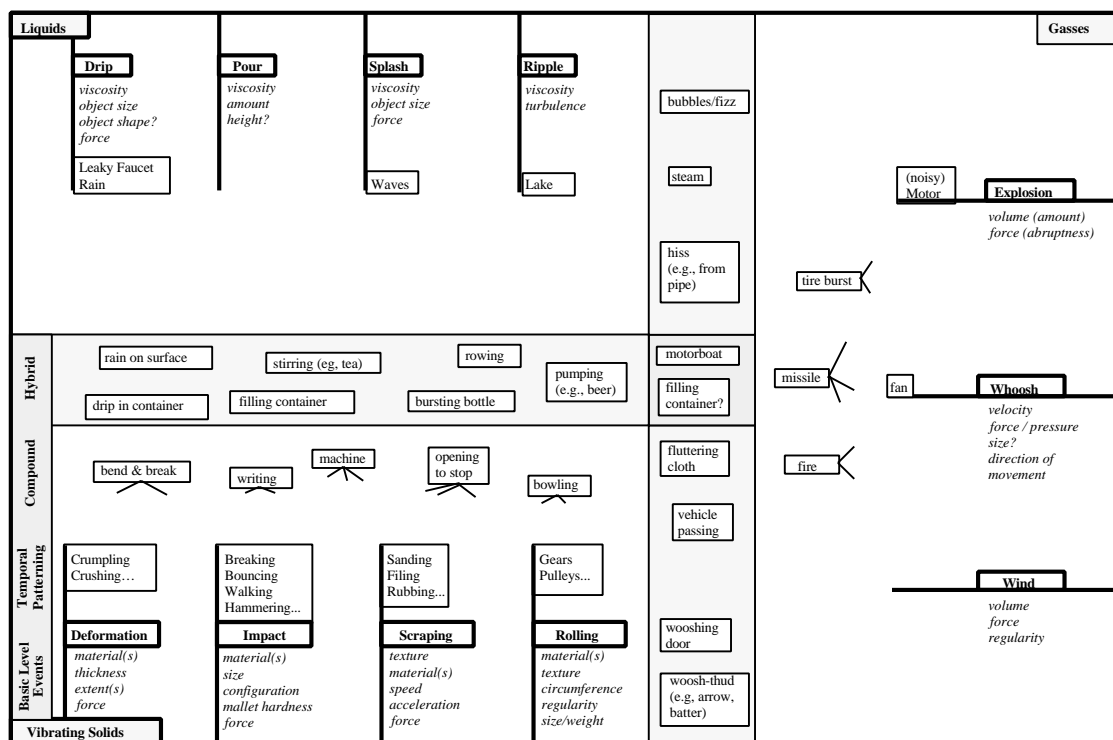
Figure 3.1 – A framework describing how different everyday sounds can be described (From [60]).

The problems with the use of auditory icons are closely linked with their main benefit: the fact that they are analogous to their meaning. This can have several disadvantages. Some pieces of information may not have any obviously analogous sounds. For example, in the previous paragraph it was reported that Gaver suggested that files could be wooden sounding. This is not an obvious analogy, but rather is an arbitrary decision. Once learned, the metaphor should be effective in providing the relevant information to the user, but is not initially obvious. Similarly, many interactions with objects in a system do not have an obvious real world analogy. A second problem is that if real world sounds are used, these sounds can have different meanings to different users. This problem can manifest itself in different ways. The source of the various sounds may be perceived differently according to the locale and what metaphors may be suitable. For example, in the Sharemon system [35] a knocking sound followed by the sound of walking was used to indicate that a user had logged onto a machine and was copying a file. One user, however, felt that the sound was representative of someone, an expectant father for example, pacing up and down outside a room after gaining no response to a knock. Furthermore, different individuals may perceive the sounds differently according to their different experiences and memories. Thus, there is a danger that the analogies used to give

the sounds meaning are not universal. For example, in the ARKola simulation of a drinks bottling factory [62] fully described later in this chapter, the sound of breaking bottles was used to indicate that bottles were not being correctly packed at the end of the production line but instead were falling off the end of the conveyor belt. The sound of breaking glass was found to be very compelling by the users of the system because in real life this sound is often associated with an event that requires immediate attention. The event in the system that was causing the generation of the sound was therefore attended to more rapidly than other events which may have been more important but whose audio representation was less compelling.

The SonicFinder

The SonicFinder [59] was an auditory interface which used auditory icons to enhance the standard graphical feedback given by the Finder on the Apple Macintosh. To ensure that the sounds used were presented consistently across the interface a mapping was defined which linked events that occurred in the interface to different sounds. For example, the selection of an object was associated with a tapping sound and dragging an object was mapped to a scraping sound. To further enhance the sounds, secondary attributes were used to convey further information about the event they represented. For example, different types of objects were associated with different materials. Files were associated with wooden sounds so when a file was selected a wooden tapping noise was generated. In addition, the size of the object selected was associated with the frequency of the sound, with larger objects producing a lower pitch sound. So, the selection of a large file would generate a low pitch, wooden, tapping sound. The addition of the sounds to the graphical interface was felt to be worthwhile because they provided the user with information that was either not provided by the graphical feedback (for example, the size of a file selected) or was only ineffectively provided by the graphical feedback (for example, the highlighting of an object which is occluded under the cursor). Gaver also felt that the addition of the sounds would increase the satisfaction experienced by the users.

No formal evaluation of the SonicFinder was undertaken, but informal evidence would seem to indicate that it was favourably received by its users. Indeed, Gaver reports that some users complained when they had to use the standard, silent Finder. Gaver hypothesised that this was because the way the sounds worked within the interface seemed natural and obvious. Gaver reports that there were two major benefits the addition of the sounds provided. Firstly, the users would experience more feelings of direct engagement with the world of the computer because the experience of hearing, as well as seeing, the objects in the computer made the computer objects seem more tangible. Secondly, the sounds allowed the users to be more flexible in the way they interacted with the system. For example, the sound played when selecting a group of files would give an indication of their size and consequently whether they would fit into the available disk space.

Although the SonicFinder was received favourably by its users, there were some problems with its implementation. The generation of the sounds was limited by the available technology of the time which would not allow the sounds to be manipulated using, for example, filtering or reverb. This meant that the number of parameters by which the sounds could be manipulated was limited, therefore limiting the range of sounds that could be produced. A second problem was the amount of memory used to store and play the sounds. This problem is eased by the ever increasing availability and decreasing cost of memory, but cannot be disregarded entirely. Gaver limited the quality of the sounds used as well as the number of sounds to

alleviate the pressures on the available memory. One solution to this problem is the *ad hoc* generation of auditory icons based upon the requirements of the sound. Conversy [39] proposed such a system which permitted the *ad hoc* synthesis of auditory icons for the monitoring of background tasks. The system allowed the generation of three different sounds: wind and wave sounds that could be used to indicate the state of a task and Sheppard-Risset tones, an auditory technique which gives the illusion of having a continuously rising or lowering pitch, to indicate speed.

ARKola

The ARKola system [62] was a collaborative system which used auditory icons to complement the graphical feedback. The SharedARK [61] (Shared Alternative Reality Kit) was used as the framework upon which the ARKola system was built. SharedARK allowed two types of sound to be played: global sounds which could be heard everywhere within the system and localised sounds which were louder as their location was approached. The ARKola system was a virtual soft drink factory which had a series of nine interconnected machines, of which the users could control eight. By controlling the rate at which ingredients were dispensed and combined, the amount of soft drink produced could be modified. The more soft drink produced, the more money the factory made. Two users controlled the factory, with each user able to see approximately one third of the whole plant. This form of plant was chosen because it allowed Gaver *et al.* to investigate how the sounds would effect the way users handled the given task and how they affected the way people collaborated. It was also an opportunity to investigate how different sounds would combine to form an auditory ecology. Gaver *et al.* relate the way the different sounds in the factory combined to the way a car engine is perceived. Although the sounds are generated by multiple distinct components, these combine to form what is perceived as a unified sound. If something goes wrong, the sound of the engine will change, alerting the listener to the problem, but in addition, to a trained ear the change in the sound would alert the listener to the nature of the problem. The sounds used to indicate the performance of the individual components of the factory were designed to reflect the semantics of the machine. For example, the heater made a whooshing sound like a blow torch and the bottle dispenser a clanking sound. The rhythm of these sounds indicated the rate at which the machines were working and if the machine stopped working, the sound stopped. In addition to these sounds, some alarm sounds were played to indicate materials were being wasted. For example, the sound of broken glass indicated that bottles were being lost. Finally, any user interaction with the system, for example button presses, were confirmed by simple, discrete sounds.

An informal evaluation was undertaken where pairs of users were observed controlling the plant, either with or without sound. The order of conditions was counter-balanced. These observations seemed to indicate that the sounds were effective in informing the users about the state of the plant and that the users were able to differentiate the different sounds and identify the problem when something went wrong. In the visual condition, however, the users were not as efficient at diagnosing what was wrong even if they knew there was a problem. Despite this, there were still some problems with the sounds used. Some of the alarm sounds were more semantically compelling making their cause seem more urgent than it really was. The sound of breaking bottles, for example, was so compelling that the users would attempt to stop the sound without first trying to understand the cause or ignoring more urgent problems. Conversely, some of the sounds were not sufficiently compelling and would occasionally go unnoticed. This was particularly true when a sound was

stopped because its source machine had stopped working. This was intended to alert the users to the cessation of that machine, but often went unnoticed. These latter problems indicate the difficulties of creating combinations, or an ecology, of sounds where the different sounds used may mask each other. From this, it is therefore reasonable to suggest that in an ecology of sounds the absence of a sound is insufficient feedback to alert a user to a background problem.

Guideline 1	If several background sounds are playing simultaneously the absence of one of these sounds may not be sufficient feedback to alert a user to a problem.

This work showed that although sounds can be combined to form auditory ecologies, there are some extra problems inherent in this. In such cases, the absence of a sound is not such a compelling piece of feedback and extra care must be taken to avoid the masking of sounds by other sounds. The ARKola system did show, however, that the addition of sounds can be effective in collaborative systems and for the monitoring of background tasks.

Summary

Auditory icons are non-speech sounds that use everyday sounds to represent their meaning. Because they use everyday sounds their meanings can be intuitively understood but, conversely, they can potentially have different meanings according to the listener's past experiences. Two examples of systems which have been built using auditory icons, SonicFinder and ARKola, were discussed. Although neither of these systems were formally evaluated, anecdotal evidence indicates that users liked the sounds and found them to be a useful addition to the graphical interface. The ARKola system in particular, however, showed that unwanted additional contextual information (in this case an increased perceived importance) can be applied to some sounds. Furthermore, it was found that the absence of a sound would not necessarily be sufficient feedback to alert a user to a problem if several sounds were playing.

3.3.3 Earcons

An alternative approach to the addition of non-speech sound at the human-computer interface is earcons. Whereas auditory icons use everyday sounds that allow the listener to perceive the information being conveyed by determining the source of the sound, earcons are abstract sounds where the musical qualities of the sound hold the key to the information being conveyed. Blattner *et al.* [12] suggested that by creating short motifs that can be used to represent a piece of information, complex sounds conveying complex pieces of information can be designed. A motif is a short melody which can be recognised as an individual entity. For example, the short melody associated with Intel in television and radio adverts is a motif. Blattner *et al.* suggested that simple waveforms (such as sine and square waves) with a narrow pitch range of a single octave should be used

Using these motifs, Blattner *et al.* suggest that it is possible to create families of earcons in a similar way to the parameterised auditory icons described in the previous section. For example, a family of error messages can be generated in the way shown in Figure 3.2. The root of the family is the error rhythm at the top of the tree. Two different forms of error, operating system and execution errors, can then be distinguished by

adding a different melody to the end of this rhythm. Finally, by playing a further melody in different timbres the specific subclass of error can be distinguished. An alternative to these hierarchical earcons are compound earcons. In this case, motifs are assigned to actions such as create or delete and objects such string or file. These motifs can then be played consecutively to represent an interaction with an object. Thus, to represent the creation of a file the motif for create would be followed by the motif for file.
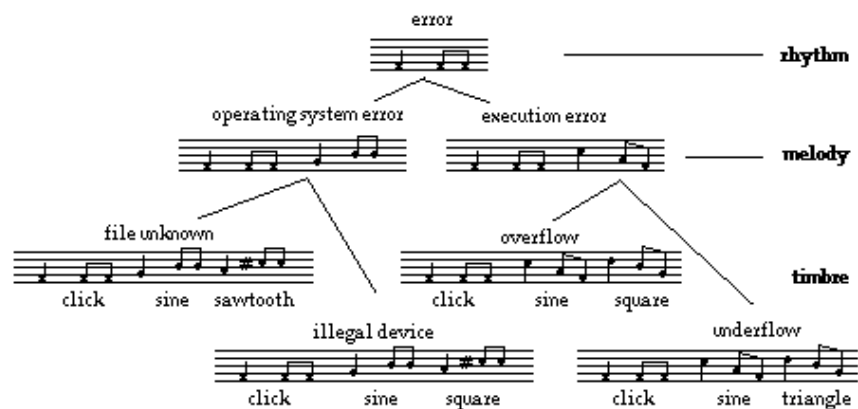
Figure 3.2 – A family of error messages, with each level in the hierarchy being distinguished by the addition of a new motif to the earcon and the different sounds on each level are distinguished by changing a musical parameter of the new earcon (from [12]).

Further work by Brewster *et al.* [27, 28] showed that the use of complex waveforms, or timbres, were more effective in allowing user to discriminate sounds, and that a wider range of pitch (of five octaves) was discriminable, especially when used in conjunction with rhythm differences. Because earcons are arbitrary, their meaning must be learnt. This has the advantage of removing any contextual mis-understandings that can occur with auditory icons and also ensures that the meanings for the earcons are truly universal, but conversely, potentially makes their learning and understanding harder. The work described above by Brewster *et al.* [27, 28] showed that earcons are an effective means of communication using sound, with experimental participants capable of high rates of recognition for different earcons.

Although very clear, the concatenation of new earcons onto the sound at each level of the hierarchy may well result in very long sounds. This may potentially cause difficulty if earcons are to be used at the human-computer interface as the speed of interaction may mean that the sounds cannot be played fast enough to keep up with the interaction. To combat this, Blattner *et al.* suggested that transformed earcons could be used. These would simply be the motif that was added at the node being played rather than a complete traversal of the hierarchy to the node. In Figure 3.2, for example, rather than traversing the whole hierarchy, a file unknown error would be indicated by the operating system motif played in a sawtooth instrument. This would still lead to unique earcons for the different nodes of the hierarchy because although some information would be lost, the final motif of the hierarchy shown in Figure 3.2, for example, would have a unique combination of timbre and melody. Another solution to this problem was suggested by Brewster *et al.* [21]. It was suggested that rather than playing the components of the earcon in series, they could be played in parallel thus reducing the time taken to convey the relevant information. An experimental evaluation was run to determine the participants' ability to recall and recognise either parallel or serial earcons. The results showed that there was no significant difference in the ability of participants to correctly recognise and recall

the meaning of parallel and serial earcons, meaning the earcons could be reduced in length without losing any of their information.

Blattner *et al.* suggested that five parameters can be manipulated to differentiate different earcons. Rhythm and pitch are the primary parameters that should be used, with timbre, intensity and register as the secondary parameters. Brewster *et al.,* however, found that timbre was much more important than previously suggested whereas pitch on its own was actually quite difficult to differentiate. Rather, the use of different registers (different pitch ranges for the same timbre. e.g. bass, alto, etc) was recommended. In both instances, wide differences were recommended to allow simple discrimination. They therefore suggested as primary parameters timbre, register and rhythm with secondary parameters being pitch, intensity, stereo position and effects.

Earcons, therefore, are short non-speech sounds which are capable of being played in parallel thus reducing their length without reducing the information they can convey. Because of their abstract nature they are not intuitive but must be learned although experimental work has shown that they can be recalled and discriminated with relative ease.

3.3.4 Summary

In this section three different forms of audio feedback were described and their advantages and disadvantages were discussed. The most obvious form of audio feedback is speech as this is the medium most of us use to communicate everyday. It is not, however, always the most appropriate form of audio feedback at the human-computer interface due to its slow speed and serial nature. Additionally, because of its wide volume range it is very difficult to set it to a level that would allow a user to habituate the audio feedback making it both attention grabbing and annoying. It is useful, however, at giving detailed information due to its very descriptive nature. Auditory icons are non-speech sounds that convey information by describing the source of the sound. This makes them easy to learn although any analogies that are used to map objects in a computer system to real life sound generating object have to be learned. These sounds can be parameterised by altering the source of the object used to generate the sound thus altering the sound produced. This allows families of auditory icons to be generated. Because the sounds used are real life sounds, they may well have different meanings to different listeners depending upon any contextual connotations the listener associates with that sound. Finally, earcons, which are abstract non-speech sounds were described. These sounds convey their information by mapping the musical properties of the sound onto different meanings. Because these mappings are abstract they have to be learnt, but they do not have any of the contextual limitations of auditory icons. Studies have shown that earcons can be recalled by listeners with the differences in meaning understood making them suitable for use at the human-computer interface.

3.4 Examples of Sound at the Human-Computer Interface

In this section several examples of the use of audio at the interface are given. In Section 3.4.1 work done on the design and evaluation of sonically-enhanced widgets is discussed. First, a framework used to evaluate the sonically-enhanced widgets is described before the evaluation of individual widgets is discussed in some detail. By examining previous work done in this area, some guidelines about the way sound should be used at the human-computer interface can be extracted and taken as requirements to which the toolkit described in this thesis must adhere. In Section 3.4.2 two systems which provide users with information about their environment are discussed. These systems provide guidelines about the use of sound as a means to allow users to elicit background information which can then be applied to the design of the audio progress indicator described in Chapter 5

3.4.1 Sonic Enhancement of Widgets With Earcons

To allow the consistent evaluation of any sonic enhancements of widgets, Brewster specified a framework for the evaluation of widgets [22]. The framework was based around three metrics for evaluation suggested by Bevan and McLeod [11]: effectiveness, or the accuracy and completeness with which tasks or sub-tasks are completed; efficiency, or the level of effectiveness achieved compared to the expenditure of resources; and satisfaction, or the perceived usability of the system by its users. In his framework, Brewster measured effectiveness by examining error rates, efficiency by time or amount completed and workload. This latter measure was acquired by using an amended NASA TLX workload scale [66]. The six standard measures (mental demand, physical demand, time pressure, effort expended, performance level achieved and frustration experienced) were used along with an additional measure: annoyance. This measure was added because a common argument against the use of sound at the human-computer interface is that it would be annoying to the user. By including this measure the truth in this argument could be determined. Finally, the participants overall preference, or rating, of the two conditions was taken to give an indication of the participant's high-level impression of each condition. A two-condition, within-subjects design was specified. This allowed each of the participants to perform each condition in the experiment and have their results compared. By counterbalancing the order in which the conditions were presented, any learning effects were removed. Prior to each condition the participants were given training and after each condition they were given workload scales to complete. The format of the experiments is shown in Table 3.1.

Participants	Condition 1		Condition 2	
Half the Participants	Sonically-enhanced widget train and test	Workload Test	Standard visual widget train and test	Workload Test
Half the Participants	Standard visual widget train and test		Sonically-enhanced widget train and test	

Table 3.1- Format for the experimental evaluation of sonically widgets from [22].

When discussing the enhancement of widgets with audio feedback, Brewster [22] defines several useful terms. The duration of a sound can be either transient: it lasts for a discrete length of time; or sustained: it lasts for an indefinite length of time. Feedback can be either demanding or avoidable. Demanding feedback cannot be avoided by a user whereas avoidable feedback can go unnoticed. A sound can be habituated if a user can let it fade into the background of his/her consciousness. In the following discussion the pitch of the sounds is described in terms of the western diatonic system which has eight octaves of seven notes [102]. The note's name is followed by its octave number. So, C_3 is the note C in the third octave. This is commonly referred to as middle C and has a frequency of 261 Hz. C_2 has a frequency of 523 Hz and C_4 has a frequency of 130 Hz. The note A above middle C has a pitch of 440 Hz and would be referred to a A_3.

Sonically-Enhanced Scrollbars

The first widget to be evaluated was the scrollbar [14, 22, 29]. An analysis of the use of scrollbars indicated two areas where the use of sound could prove beneficial. The first of these areas was when a user clicked in the scrollbar area to move the thumb wheel towards the location clicked. If the user continually clicks the scroll bar area, the thumb wheel may pass beyond the location of the cursor and a subsequent click will cause the direction of the thumb wheel to reverse. Brewster *et al.* called this 'kangarooing'. This error can happen because the only feedback given by the scrollbar to indicate the occurrence of this error, the location of the thumb wheel, is avoidable due to its small size. The second area where the use of audio feedback was found to be useful was in determining the position within the document being scrolled. If the indicator which gives the current position within the document was not within the visual focus of the user its feedback was avoidable. It was therefore possible to lose awareness of the current position in the document.

To counter these problems, two sounds were used. The first sound was used to indicate window scrolling and thumb movement. A fixed tone of 150 msec. was played to indicate a scroll event. A low pitched note (C_3) was played to indicate scrolling down. A high pitched note (C_0) was played to indicate scrolling up. By differentiating the feedback given for these two distinct scroll events, when a kangarooing error occurred it was hoped that the change in pitch of the audio feedback would alert the user. The second sound was used to indicate page scrolling and position. A low intensity continuous tone indicated the approximate location of the current page. The pitch of the tone varied from B_1 to C_4 when scrolling from top to bottom of a document. The notes played cycled through the scale of C major giving a total of 21 different pitches. When the document being scrolled crossed a page boundary the new sound was played at a slightly higher volume for 150 msecs to alert the user to this event and was then returned to a low level to allow the sound to be habituated. By giving the user demanding feedback when the document crossed a page boundary, it was hoped that users would be more aware of the current position in the document.

An experimental evaluation of the sounds added to the scroll bar was undertaken following the framework described above. The participants were presented with a text editor with a twelve page file of random characters. The page size was longer than the display area of the window meaning that if the display was scrolled the file did not automatically cross a page boundary. The participants had to perform a series of tasks that would require the document to be scrolled, with timing and error data recorded as well as the subjective workload of the participants taken with the NASA TLX tests. Two types of task were undertaken. Navigation

Tasks where the participants had to find a specified location in the document and Search Tasks where the participants had to locate a specified piece of text in the document. The results showed that the participants found the mental demand in the audio condition to be greatly reduced compared to the non-audio condition. Brewster *et al.* suggested that this was because it was easier for the participants to determine the page boundaries due to the demanding audio feedback. Additionally, the participants expressed a significant preference for the audio condition perhaps indicating the audio condition required less effort on their part. Finally, the participants reported no significant difference in the annoyance experienced.

The timing data showed that participants were able to navigate through the document faster in the audio condition than in the visual condition, indicating that the audio feedback was useful in assisting the participants. This result highlights how sound can be used as a means of reinforcing avoidable visual feedback and, more interestingly, negating misleading tactile information. By indicating when a page boundary had been crossed rather than merely indicating the basic result of the interaction (the direction scrolled in) the sounds provided relevant information to the users. In this case, the page size was larger than the screen size meaning that a mouse click on the scroll bar did not necessarily represent a crossing of a page boundary and equally, the page boundaries' visual representation was avoidable as the user scrolled through the document. No significant change was found in the time taken to complete the search tasks. This may have been because the search task was an intrinsically visual task with the participants having to visually scan the document to find the specified string so the addition of sounds was of limited benefit to them. There were no significant differences in the number of errors recorded in either condition. This may have been because the number of errors generated was in fact very low making an analysis very difficult. There was, however, some anecdotal evidence to indicate that the participants found the audio feedback useful in identifying when an error had occurred.

Guideline 2	The use of sound can provide relevant information by being mapped to the data model as well as the interaction technique.

The design and evaluation of a sonically-enhanced scrollbar showed that the addition of sounds to a standard graphical scrollbar could enhance the effectiveness of the standard graphical widget. Two potential problem areas were identified and sounds were added to try and alleviate these problems. The experimental evaluation showed that the sounds were effective in aiding the participant's navigation through a long document and reduced the mental demand experienced by the participants allowing them to complete navigation tasks significantly quicker. Importantly, there was no significant change in the annoyance experienced by the participants. This demonstrates that by careful design it is possible to add sound to the human-computer interface without causing the user annoyance. This is significant because one of the primary criticisms about the use of audio is its potential for being annoying. There was no reduction in the error rates, however, partially because so few errors were made and partially because the sounds did not warn users of impending errors, but rather notified users that potentially an error had been made. One limitation of the sounds added to the scrollbar was that the page position sound was only suitable for documents up to 21 pages in length. Brewster *et al.* suggests a solution to this could be to extend the range of the sounds by using rhythm and/or

intensity as well as pitch to differentiate pages. A solution suggested by Beaudouin-Lafon *et al.* [9] was to use Shepard-Risset tones, an auditory illusion similar to M.C. Escher's drawing of a perpetually rising staircase. These tones can be designed to increase (or decrease) in pitch perpetually. If a kangarooing error were to occur the user would hear the tone 'moving' in the wrong direction. Unfortunately, due to the nature of the tones, it is necessary to use simple sine waves and it can be difficult to ensure that the illusion of perpetual motion is retained. Also, because the tones are an illusion giving the impression of perpetual motion they are unable to give an indication of the current location within the document. Although no evaluation of the sounds was undertaken and despite the limitations of the sounds, the Shepard-Risset tones may be an effective way of providing directional information to a user scrolling through a large document because they will change tone when a kangarooing error occurs.

Sonically-Enhanced Graphical Buttons

Brewster *et al.* also investigated the addition of sound to the graphical button [14, 20, 22]. Again, an analysis of the way buttons are used and any potential problems that may arise was undertaken highlighting some potential usability problems. It was found that the existing, visual feedback was insufficient because in many instances when it was given the user's visual focus would be elsewhere. Consider, for example, the selection of a graphical button (Figure 3.3). The feedback given to the user during his/her interaction with the button is the highlighting of the graphical button when it is pressed down (Figure 3.3 (1).B and Figure 3.3 (2).B. The only difference in feedback between a correct selection (Figure 3.3 (1).C) and a non-selection, where the user moves the cursor off the graphical button before the selection is complete (Figure 3.3 (2).C), is the location of the cursor. Because the difference in location may be very small, the difference in feedback may go unnoticed especially if the cursor is moving. Additionally, Brewster *et al.* suggest that there is a large likelihood that the user will not be looking at the graphical button when the feedback is presented. Dix and Brewster [49] suggest that this sort of error, a slip-off, typically occurs when three conditions are met:

1. The user reaches closure after the mouse button is depressed and the graphical button has highlighted. Dix *et al.* [50] define closure as the user having a feeling of completeness and so moving on to his/her next task. In this case, the user feels his/her interaction with the graphical button is complete when it highlights.
2. The visual focus of the next action is at some distance from the graphical button.
3. The cursor is required at the new focus.
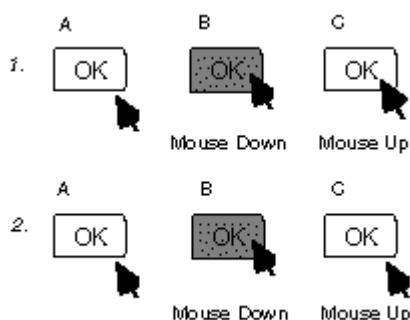
Figure 3.3 – The visual feedback presented by a graphical button when selected. (1) shows a correct selection and (2) shows a slip-off (from [20]).

These conditions occur when the user is an expert and will often perform tasks automatically without monitoring the feedback given. In this case, when the button highlight feedback is detected closure is reached. Whilst the user releases the mouse button to complete the interaction and select the graphical button, the mouse is moved to the next location it is required, possibly far away from the current location. If the user's visual focus is at the new location, the difference in the feedback presented by the graphical button may go unnoticed. This investigation indicates that a requirement for the toolkit is the exposure of the behaviour of the button that may not necessarily appear relevant to the interaction. In this case, the release of the mouse button outside the graphical button is relevant to the behaviour of the widget (as it determines whether or not it has been selected) but typically this event is not exposed as part of the behaviour of the widget.

Requirement 1	The full behaviour of the toolkit's widgets should be exposed allowing suitable forms of presentation to be generated.

A second problem with graphical buttons is that it can sometimes be difficult to determine whether the cursor is over the button's input area or not. Often, the cursor itself will obscure the graphical rendering of the button making this task harder, especially when the button itself is very small. Standard graphical buttons often give no feedback about whether the cursor is over the button or not and if the visual presentation of the button changes to indicate the cursor is over the button, this change may be obscured by the cursor.

Three sounds were used to improve the effectiveness of the graphical button. A continuous tone was played at C_4 at a volume just above the threshold level (the volume at which a sound is perceivable by a listener) when the cursor was over the graphical button. This allowed the sound to be habituated by the user but would also inform the user the cursor was over the selection area of the button. When the mouse button was pressed down over the graphical button a continuous tone was played at C_3 at a slighter higher level than the previous tone. This sound was designed to be more attention grabbing because it indicated an interaction was taking place. The third sound used indicated that the graphical button had been successfully selected. This sound consisted of two short tones with a pitch of C_1 and duration of 40ms. The duration of the sound was kept short so that the sounds could keep up with the pace at which the user was interacting with the system, but at the same time was made attention grabbing to ensure that it was perceived by the user. If the user's interaction with the button was very short only the success sound was played.

An experimental evaluation of these sounds was again undertaken [20] following the framework specified earlier in this section. The participants were presented with a graphical number pad consisting of graphical buttons. Using this number pad, they had to enter as many five digit numbers as possible within 15 minutes. The numbers to be entered were presented in an adjacent text field. As with the scrollbar evaluation, both quantitative data (timings and errors) and qualitative (NASA TLX ratings) were gathered.

Of the TLX ratings, only overall preference was shown to have a significant difference between conditions. The participants indicated a preference for the sonically-enhanced buttons. Brewster *et al.* suggest that this

was because the audio feedback allowed the participants to recover from errors with greater ease. The fact that the annoyance experienced by the participants showed no significant differences between the conditions confirmed that the use of sound at the human-computer interface can be implemented without annoying the users. It was suggested that the fact that the workload experienced by the participants remained unchanged was because the recovery from errors was seen as a separate activity from the main task. This result was backed up by the timing and error data that showed the participants could recover from errors significantly faster in the audio condition. This indicates that the sounds were effective in alerting the users to a correct selection and that no sound, when a slip-off occurred, was an equally demanding piece of feedback. This contradicts the finding discussed in Section 3.3.2 where it was found that the absence of a sound in the ARKola simulation often went un-noticed. In that case, however, the sounds were part of an ecology of sounds providing background information whereas in this case, the sound would have been as a direct result of a user action. A better analogy in this case is the supermarket checkout, where the checkout assistant expects to hear a confirmation sound when passing an item in front of a bar code reader and the absence of such a sound is a demanding piece of feedback, alerting the assistant to an error in the scan. Strangely, despite this the number of codes entered did not vary significantly in either condition. This was because the participants made more errors in the audio condition cancelling out the benefit gained by recovering quicker from the errors. Brewster *et al.* suggest that because the users knew they could recover from the errors easily they became careless and consequently made more errors. There was, however, no significant increase in the number of errors made and it was hoped that this was simply an artefact of the experiment.

Guideline 3	The absence of a sound where one is expected is demanding feedback if the sound would have been as a result of a user interaction, not as a piece of background information.

Although all the sounds used were found to be effective, the results indicate that the most important sound was the selection sound. This implies that it is possible to place the sounds in order of importance. If it was not possible to play all the sounds, for example there were insufficient audio resources to play all the sounds, or the use of all the sounds would mask other, more important sounds then this information could be used to determine which sounds should be played and which should be stopped. This also implies a requirement of a toolkit of multimodal widgets that it should be able to determine when it is inappropriate to play all the sounds and modify the feedback accordingly.

Guideline 4	If possible, the different sounds used to represent an interaction should be ranked in order of importance. In this way, it is possible to determine which sounds should be played if it is not possible to play all the desired sounds.

Requirement 2	The toolkit should be able to determine if it is not possible to play all the requested sounds and, if so, modify the feedback appropriately.

This work was reprised in a later experiment which evaluated the benefit of adding sound to graphical buttons on a hand held device where the screen space is limited [17]. The same tasks were performed as

before, with the experiment using a similar framework to that described at the start of this section. Two conditions were used, standard buttons (16x16 pixels) and small buttons (this time 8x8 pixels) both of which are standard button sizes on mobile devices. Each condition had two 7-minute treatments: visual only buttons and visual plus sound buttons. Due to the limitations of the device upon which the experiment was performed (a 3Com PalmIII handheld computer with stylus input) three simple sounds were played. The standard PalmIII key click sound was played when a button was successfully selected; a higher pitched version of this sound was played when the stylus was pressed in a graphical button; and a lower pitched version of this sound was played when a button was mis-selected. The results found in general confirmed the results of the previous experiment. For the standard buttons, the workload experienced by the participants in the audio treatment was reduced in all categories except time pressure. For the small buttons, the workload experienced in the audio treatment was reduced in all categories except time pressure and physical demand. Finally, in both conditions, the addition of sound allowed the participants to enter significantly more 5-digit strings than in the silent treatment.

The results of the two evaluations described here further validate the idea that the use of sound can enhance the usability of the human-computer interface. An interesting point to note is that the use of no sound when a sound is expected can be as demanding as the use of a sound. Although this was not used for the experiment run on hand-held devices, the first experiment on a desktop machine showed that the participants could easily recognise a slip off due to the demanding nature of the success sound *not* being played. The effectiveness of no sound versus error sound could easily be determined by running these experiments again with the independent variable being the use of error sound or not. As with the sounds used to enhance a scrollbar, the sounds used did not reduce the number of errors the participants made, but rather alerted them to an error once it had been made. Finally, it was shown that through careful use of sound, the participants did not experience anymore annoyance than with the silent conditions, confirming that sound can be effective in enhancing user interfaces.

Sonically-Enhanced Pull Down Menus

Brewster & Crease did a similar study to that done on graphical buttons for pull down menus [14, 18, 25]. An analysis of the way users interact with pull down menus indicated that the interaction can end in one of three ways: a correct selection, an incorrect selection or no selection. An incorrect selection could occur for one of two reasons. The user could simply select the wrong item by mistake (a *mis-selection*) or the cursor could slip into an item adjacent to the correct selection accidentally as the mouse button was being released (an *item slip*). Similarly, no selection could occur due to one of three reasons. The user could simply decide not to select an item, the cursor could slip out of the menu before the mouse button was released (*menu-slip*) or the user could select a disabled item or a separator. This last occurrence could occur due to either an item slip or the user simply deciding not to select anything. Figure 3.4 shows examples of (i) a correct selection, (ii) an item slip and (iii) a menu-slip.
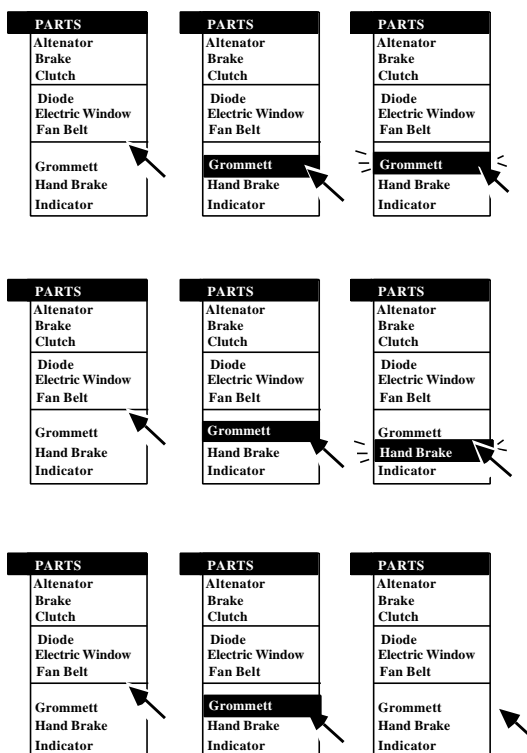
Figure 3.4 - Examples of (i) a correct selection, (ii) an item slip and (iii) a menu-slip (from [25]). (i) A correct menu selection. The cursor is moved into "Grommett" and released. Grommett flashes before the menu disappears. (ii) An Item Slip. The cursor is moved into "Grommett" and as the button is released the cursor slips into "Hand Brake". "Hand Brake" flashes before the menu disappears. (iii) A Menu Slip. The cursor is moved into "Grommett" and as the button is released the cursor slips off the side of the menu. The highlight disappears. The same happens if the cursor slips into a divider or disabled item.

This analysis showed that the problems that can occur with pull down menus are very similar to those that occur with graphical buttons, namely slip-offs. When the user moves the mouse over a valid menu item, it is highlighted and the user reaches closure. Thinking that the menu item has been successfully selected he/she may move the mouse to the next location whilst releasing the mouse button. This may cause the cursor to leave the menu item before the selection is complete. If the user's visual focus is away from the menu item being selected the feedback given by the menu may go unnoticed. Although this feedback varies across systems it is typically avoidable. On some systems, when a menu item is selected the menu simply disappears. This is identical to the behaviour of the menu if the cursor slips off the side of the menu before the mouse button is released. On other systems, the menu item selected may briefly flash before the menu disappears. In both instances the feedback is avoidable. If an incorrect selection occurs, the item selected may flash, but if the user's visual focus is elsewhere this feedback will go unnoticed.

Previous work had been undertaken to aid users with the navigation of menu structures. Barfield *et al.* [7] used simple earcons to indicate the position of a user in a menu. The earcons consisted only of simple, half-second tones which decreased in pitch as the user navigated down the hierarchy. An evaluation found that these sounds did not improve user performance. This may have been because the use of pitch to provide an absolute value is ineffective as it can be difficult to identify the pitch of a note when it is played in isolation. Karshmer *et al.* [69] developed a system of sonically-enhanced menus designed for visually impaired users. Both speech and earcons were used to provide audio feedback. Each menu had a different timbre and as the

user navigated down a menu the pitch of the earcon used to identify the current menu item decreased. If the menu item was clicked its name was spoken and if it was double clicked it was selected. An initial study found that the users found it hard to navigate the menus using the tones (again, probably due to the difficulty in identifying absolute values from pitch) but the addition of speech made the system viable.

Four sounds were designed to overcome these problems. The first, over menu, sound was analogous to the sound used for graphical buttons to indicate that the mouse was over the menu. In this case, the sound was played when the cursor was over the menu. A continuous note at pitch C_2 was played at a low volume. The sound continued as long as the cursor was over the menu. A second, over item, sound was played to indicate that the cursor was over a valid menu item. This sound was analogous to the sound used to indicate that the cursor had been pressed inside a graphical button, highlighting it. A continuous low intensity tone was again used with the pitch for the sound alternating for every item to ensure adjacent items were distinct. For odd numbered items the sound had a pitch of B_2 (987 Hz) and for even numbered items a pitch of E_3 (329 Hz) was used. Whilst the alternating pitch of this sound does not give any navigational cues to the user it does mean that the design is scalable to any length of menu. This design indicates that the toolkit should have some notion of parent widgets that allow the logical grouping of the children. In this case, the parent is the menu and the children are the menu items. The parent needs to be able to communicate with its children so they can be informed of their position in the menu. To prevent these sounds continually stopping and starting as the user navigated the menu, the over item sound was only played after the mouse had been over the item for half a second. A third, selection, sound was used to indicate the selection of a menu item. This sound was analogous to the success sound of a sonically-enhanced button. Two short tones of duration 40msec. were played. The pitch of these sounds was the same as the pitch of the highlight sound for that item, discriminating the selection sounds of two adjacent menu items. The final sound used indicated that an item slip had occurred. Following pilot evaluations, this sound was played instead of the selection sound if the cursor had been over the item for less than 117msecs[7]. when the mouse button was released. The sound consisted of three notes of duration 40 msecs. played in a fixed rhythm. The pitches of the notes (C_2, B_2 and F_2) were chosen to be discordant and therefore attention grabbing alerting the user to a potential error.

Requirement 3	It must be possible to group widgets together to allow their feedback to be co-ordinated appropriately.

The over menu sound was used because it was shown to be useful for graphical buttons and it was felt it would assist the users to detect menu-slips. The over item sound would assist users in determining if the correct selection had been made. If the incorrect item had been selected, then perhaps the pitch of the selection sound would indicate the error. The selection sound would alert the user to menu-slips by its absence in the same way as the absence of the sonically-enhanced graphical button's success sound was noticed when a slip occurred. Finally, by defining an item slip in terms of the length of time the cursor was over an item it was possible to play a sound indicating that such an error may have occurred, thus alerting the

---

[7] Pilot studies indicated that this time was an appropriate cut-off point for item slips. The exact figure of 117 msecs arose because the implementation was built on a Macintosh where timings are measured in 60ths of a second (117msecs = 7/60ths of a second).

user. The implementation of the sonically-enhanced menu using this design required that the menu widget had to be completely re-implemented, including the graphical feedback, because the existing widget did not expose any of the user interaction with the menu except for the final selection. This re-affirms Requirement 1 which stated that the full behaviour of the widget should be exposed to enable new forms of feedback to be added easily.

An experimental evaluation of the sounds was undertaken following the framework recommended by Brewster *et al.* Again, both quantitative and qualitative data was recorded. The participants were asked to select specified menu items from two different menus before confirming these selections using a menu item from a third menu. If the participants tried to confirm an incorrect selection or selections an error dialogue was displayed indicating to the participant that the error had to be corrected before they could continue. The participants had to select 150 pairs of menu items as quickly as possible. The average workload across all the categories was significantly reduced in the audio condition indicating that the effort required to perform the task was reduced in the audio condition. The number of errors corrected before the error dialogue was displayed was significantly increased in the audio condition indicating that the sounds were effective in conveying the relevant information to the participants. Similarly, the time taken to correct an error was significantly reduced in the audio condition. As before, the participants did not report the addition of sound to be annoying.

These results indicate that the addition of sounds to pull down menus can be effective in increasing their usability. As with the graphical button, the absence of a sound (in this case the selection sound) was shown to be a demanding piece of feedback. The sounds designed for the pull down menu varied slightly from the sounds used for the graphical button in as much as they did not map directly onto the way the user interacted with the widget. There was no explicit navigational information given by the sounds and an additional sound was used to indicate a high level event, the item slip. Navigational feedback was not given sonically because the navigation of a menu is a visually intensive activity and consequently any audio feedback given would potentially be annoying. It was found, however, that the item slip sound was played more often than necessary. This was because, for some users, 117msecs. was too long to indicate an item slip. Brewster & Crease suggest that a solution to this problem could be to have a control panel to allow users to control the minimum length of time the cursor needs to be over an item before a selection is classed as a correct selection. This is similar to the sort of control panel that that many system already have allowing users to specify the maximum length of time that can separate the clicks in a double click. The results indicate that, as with the graphical button, the most important sound was the success sound. Although the item slip sound did not provide correct information all the time, it should still be ranked highly as it did prove effective at alerting the participants to a potential error.

Sonically-Enhanced Tool Palettes

Brewster & Clark investigated the use of sound to improve the effectiveness of tool palettes [15, 16, 24]. Tool palettes are typically small windows or menus which contain a grid of buttons. Each of these buttons represents a mutually exclusive tool. When one tool is selected, the previously selected tool is unselected. It was found that although users interact with a tool palette in a similar way they would interact with a button or

menu, the behaviour of the tool palette can vary. Typically, if a user selects a button with a single click, the tool that button represents is used once before the tool palette automatically reverts to the default tool. If, on the other hand, the user selects a tool by double clicking its button the tool is active until the user selects a new tool. An example of such a tool palette is the drawing toolbar in Microsoft™ Word™ drawing toolbar. Often, the only feedback given to the user is a graphical outline round the selected tool. The style of this outline will vary according to the way the tool was selected, perhaps dotted for single use and solid for permanent use. As this graphical feedback is likely to be avoidable, especially if the user's visual focus is elsewhere, it could lead to errors as users may be unaware of the current tool being switched automatically.

Brewster & Clarke suggested that this problem could be overcome by the addition of sound. They suggested that the addition of a tool selection sound was all that was necessary. When a tool was selected by a single click a single tone of duration 100 msecs. and pitch C_3 was played. If the tool was selected by a double click, two tones of duration 100 msecs. and pitch C_2 were played. This was intended to highlight the different modes that were being used. After a tool had been used, a sound was played to indicate the currently selected tool. If the current tool had been selected by a double click, the double selection sound was played for that tool. If the tool had been selected by a single click then the single selection sound would be played for the default tool as the tool palette reverted to this tool. The selection sound for the default tool was played in a different timbre from all the other tools in the palette. This ensured that if the tool reverted to the default tool, the user would be aware of this because of the difference in timbre.

An experimental evaluation of the sounds used in the tool palette was undertaken following the framework specified at the start of this section. The participants were required to perform specified drawing tasks using a simple graphics package which employed a modified tool palette. Although the qualitative results showed no significant changes across the two conditions, the number of tasks performed with the wrong tool was significantly reduced in the audio condition. This indicates that the participants were able to determine they were using the wrong tool due to the audio feedback and consequently correct this error. Thus, sound was again shown to be an effective addition to the human-computer interface. The design of the audio feedback used here indicates that by not naïvely mapping sounds to events, the number of different sounds used can be minimised. In this case, two basic sounds were used to indicate the current tool despite the fact that there were 12 options on the tool palette. This was successful because an analysis of the use of tool palettes showed that after any drawing option the currently selected tool could be one of only two choices: the current tool or the default tool. Thus, it was possible to provide the solution described above which had the double advantage of being scaleable and limiting the number of sounds the user would have to learn.

Guideline 5	Limit the number of different sounds required by analysing the way a user interacts with an object rather than naïvely mapping a different sound to every different event.

This research confirmed the previous work which showed that sounds could be a valuable addition to the human-computer interface. The sounds added to the tool palette differed from those in previous experiments as they alerted the users to a potential error before it was made whereas previously the sounds would alert the users to errors they had already made.

Sonically-Enhanced Drag & Drop

Drag and drop is a common feature in direct manipulation user interfaces. It allows users to invoke commands on an object by simply selecting it and dragging it over another object where it is released. For example, a user can delete a file by dragging it over the wastebasket icon and releasing it there. Although this is an intuitive and natural way to interact with a computer, a study by Brewster [14, 23] showed that there were still some problems with the way the interaction worked. One of these problems was very similar to the problem he found with graphical buttons. When the item being dragged reaches the target object, the target typically is highlighted. When this occurs, the user reaches closure and starts to move the mouse away whilst releasing the mouse button. If, by the time the mouse button has been released, the mouse is no longer over the target, the interaction will not have been successfully completed. The second problem he found was that it can sometimes be hard to hit a target because the object being dragged obscures it, especially if the target is small. This problem had previously been identified by Gaver [59] in his SonicFinder interface which is described in more detail in Section 3.3.2 Gaver described the problem as "chasing the trashcan".

Gaver used auditory icons to provide audio feedback that reinforced the existing, but avoidable, graphical feedback. Two sounds were used, one to indicate that the mouse was over a target and a second to indicate that the interaction had been successfully completed. Although this system was never formally evaluated, Gaver reported that this was one of the "most obviously useful features of the SonicFinder". Brewster followed a similar course, but instead chose to use earcons as the means of sonification. Three sounds were used. The first sound, a continuous low intensity note at a pitch of C_4 and played with a reed organ timbre, was played when the mouse was over a target. The second sound, a short tone of 300 msecs. duration, a pitch of C_4, and a 'tinkle bell' timbre, was played to indicate that the target had successfully been hit. The third sound was used to indicate that a valid target had been missed. This sound was the same as the success sound except it used an orchestral hit[8] timbre.

An experimental evaluation was undertaken using the framework described at the start of this section. The participants were expected to perform a series of specified drag and drop tasks with both qualitative and quantitative data collected. The results showed that the participants were able to perform the tasks significantly faster in the audio condition. Additionally, the length of time the mouse was over the target before the mouse button was released was significantly reduced in the audio condition. The workload results showed a significant reduction in the effort expended and overall workload in the audio condition. These results indicate that the sounds were a useful addition, confirming Gaver's hypothesis.

As with the sounds used to enhance graphical buttons on a hand held device, a sound was played to indicate a negative result (in this case a missed target). This is in opposition to the other work done on, for example, menus and graphical buttons on a desktop machine which seemed to indicate that the absence of an expected sound indicating a positive result was demanding feedback in its own right. As with the button, the results described above do indicate an order of importance that the sounds can be ranked in: the 'success' sound is the most important, with the over sound being less important.

---

[8] An orchestral hit is a combination of several orchestral instruments playing a single note simultaneously.

Non-Visual Auditory Widgets

Mitsopoulos *et al.* proposed a methodology to allow the specification and design of non-visual widgets to be performed systematically [76, 77]. Although the work is aimed at developing non-visual widgets it can be generalised to the development of sonically-enhanced widgets also. Mitsopoulos *et al.* propose a three level methodology for the specification of auditory widgets. The first level, the conceptual level, consists of identifying the set of tasks associated with the widget and the abstract information necessary to perform these tasks. Mitsopoulos *et al.* cite Bregman's Auditory Scene Analysis [13] as the basis of the second, structural level. This specifies that the auditory structure consists of auditory streams, or perceptual entities, on which attention can be focused, and can be viewed across streams (in an instant) and within a stream (over time). It is easy to attend to a single stream over time, but it may be impossible to integrate the information provided by two separate streams if they are presented too quickly. Thus, the number of streams perceived by a user may vary according to the rate of presentation. Dependant upon the tasks to be performed, the sounds used and the rate at which they are presented can be altered to allow the structure to be viewed either across time and within streams or across streams at a given time. The third level, the implementation level, consists of specifying the physical dimensions of the sound so that they satisfy any restriction imposed by the conceptual and structural level.

Mitsopoulos *et al.* describe the definition of a group of check boxes to illustrate the methodology in action. At the conceptual level, tasks associated with the widget include identification of the status of the group (for example whether they are all selected or unselected), browsing the options available, identifying and/or changing the value of an option and the identification of the overall purpose of the widget. The information required to perform these tasks is the current status of the different check boxes. This information can be abstracted to a nominal dimension because the states of a check box are in no way ordered. A second, nominal piece of abstract information is the check box label. Because the conceptual specification stated that one of the tasks was the scanning of the group of check boxes this implies that the individual widgets should be semantically related, forming higher level widgets. This confirms Requirement 3 highlighted when discussing menus and tool palettes.

At the structural level, the auditory scene structure is specified. In this case, one of the tasks to be performed is a glance at the group of check boxes to give an overview of their states. A visual glance would give a high level indication of whether the check boxes were all checked, all unchecked or a combination of checked and unchecked. This could be presented as a series of tones representing the different check boxes in the group, with perhaps a different timbre used for selected and unselected. At a fast rate of presentation, representing a glance at the check boxes, the stream of sounds representing the different check boxes would segregate into two different streams of selected and unselected check boxes. If the rate of presentation was slowed down, a user could combine the two streams and attend to them as a single stream if desired. In this way, a hierarchy of streams can be built which forms the basis of the auditory scene.

The implementation of the sounds is limited by any restrictions placed at the conceptual and structural levels. The different streams in the auditory scene can now be realised. The discussion above indicated that two different sounds should be used to represent checked and unchecked boxes, with the two streams perceived at

slower presentation rates but integrated into a single stream at faster presentation speeds. This can be ensured by using the many factors that Bregman suggests affect the segregation of streams of sound. Several widgets have been implemented using this methodology, but as yet no evaluation of their effectiveness has been undertaken. Mitsopoulos *et al.* suggest that a comparison with the equivalent visual widgets would be the most desirable approach because the visual widgets are regarded as being efficient. This would also be the first step in comparing visual interfaces with non-visual interfaces.

Discussion

In this section, the design and evaluation of sonically-enhanced widgets using earcons was described. An important result that was shown in all the evaluations undertaken was that the participants did not find the sounds, albeit over only a short time, to be annoying. This was because the sounds used were providing them with useful information that was either not being presented visually or was being presented, outwith the participant's visual focus. An interesting piece of future research would be to reproduce this result over a long term evaluation. Another important result was that the omission of a sound was often shown to be a demanding piece of feedback. This is important because it allows the designers of sounds to minimise the number of sounds used, thus minimising the potential for the sounds to become intrusive.

There is still much work to be done, however. Although many widgets have been evaluated, these evaluations were of the widgets in isolation. There has been no work done to evaluate if the earcons used are suitable when they are used in combination where sounds may be masked or the absence of a sound could go unnoticed. Furthermore, although the annoyance of the primary user has been evaluated and shown not to be increased by the use of these sounds, no testing has been done to determine if the sounds are annoying (or useful) to any third parties nearby.

3.4.2 Audio Environments

In this section, two examples of audio environments are described. These systems provide users with information about their environment through the use of audio feedback. The Out to Lunch system [37] provides users with feedback about their work colleague's activities even if they are physically dispersed. The Audio Aura system [89, 90] provides user information about their email regardless of their location. In both instances, however, the systems are designed to provide their feedback in an unobtrusive manner. Other systems which perform a similar function include ShareMon [35, 36] which uses sounds to indicate file sharing activity and the RAVE system [57] which uses sound to indicate that a user is being monitored via a video link.

Out To Lunch

The Out to Lunch system [37] was designed to provide users with information about the activities (e.g. typing and mouse clicks) of other members of a group. This information was presented both sonically and visually on an electronic sign in order to promote a sense of group awareness amongst otherwise disparate co-workers. The first version of Out to Lunch used pre-recorded mouse clicks and keystrokes to indicate the activities of co-workers. These two aspects of user activity were totalled over a period of thirty seconds and the appropriate sounds were played for each user at a random point in the next thirty sounds. Similarly, the

distance travelled by the mouse was measured and a mouse rolling sound played as appropriate. The sounds for all the users were combined and could be heard by all members of the group. These sounds were supplemented by an electronic sign which was located in a public area and displayed summaries of the groups activities.

It was found, however, that the users of the system tired of both the audio and visual information provided. This was, in part, because the system provided very little information about the group's activities. The sounds, in particular, were found to be annoying because of aesthetic dissatisfaction with them. It was found that by including more mouse rolling sounds the users found the audio feedback to be less intrusive, although they still found it annoying in time. Cohen hypothesised that this improvement was because the mouse rolling sound included a white noise component and had a longer duration than the other sounds giving the overall mix of the sounds a more balanced feel. To overcome the first complaint about the information provided, the system was revised to reveal the general activity of individuals within the group rather than simply an indication of the overall group activity. The messages on the visual display were changed to allow this and the users within the group were represented by a specific theme. To prevent the audio feedback becoming annoying the sounds were changed from real-life samples to musical motifs. As long as someone in the group was active a guitar "drone"[9] was played in the background which established an ambience and prevented the other sounds from becoming too intrusive. The different users had different musical themes which were played at random periods if they were active. By giving the themes slow attacks and an echoing quality the annoyance due to the sounds was minimised. By playing the themes at random intervals the overall audio feedback was rarely repeated. If no users in the group were active, the guitar drone faded out and no sounds would be played at all. The users of the new system reported that the sounds were much more pleasant to listen to as well as more informative although it was unclear whether the users would still find the sounds pleasant after prolonged use. It is not always this clear as to what information different users will find relevant. It is therefore important that a toolkit of widgets that can provide audio feedback allows users to control what form the feedback will take.

Guideline 6	To prevent annoyance, audio feedback should be limited to that which provides useful information to users. If the information provided to the user is either irrelevant or provided by another source the sounds could become annoying.

Guideline 7	Blending sounds together rather than playing them in isolation can ensure that the audio feedback is less intrusive.

Requirement 4	A toolkit of multimodal widgets must allow users to control the form of presentation used for its widgets.

The Out to Lunch system fulfilled the purpose of fostering a sense of community amongst a group of disparate colleagues. As well as issues of privacy which are not discussed here, the implementation of the

---

[9] This was an arrhythmic, low-pitched, low-volume, seamless loop of guitar solo music lasting 13 seconds.

system highlighted some issues with providing non-intrusive audio feedback. For such a system to be effective the sounds must provide information relevant to the users to ensure that the sounds are not found annoying. Additionally, it was found that the users preferred the musical sounds as opposed to the everyday sounds. This, however, cannot be generalised because the comparison between the two different sounds is not a fair one. The musical sounds provided more information about the groups activities and were designed by a professional musician whereas the everyday sounds were basic samples which were not professionally produced. This does show, however, that musical sounds can be used to provide audio feedback effectively and unobtrusively.

Audio Aura

The Audio Aura system [89, 90] was designed to provide serendipitous information to users based upon their current location within their environment. This information was provided sonically via wireless headphones. The design of the system was based around three scenarios. The first scenario centred around the tension between going for a coffee with your colleagues between meetings and checking your email. To relieve this tension, the Audio Aura system provided the user an audio cue indicating how many unread messages with, perhaps, some indication of who they were from when they entered the coffee room. The second scenario was designed to give a user faced with an empty office an indication of whether the occupier of the office had been in that day and if so, if they have been out of their office for long. The third scenario was designed to give an indication about the activities of a disparate group of co-workers. This information would include indications of whether people were in work that day, whether people were working with shared objects or meeting face to face.

The information provided by the system is typically not qualitative in nature. The sound presented when a user is faced with an empty office, for example, would not give an exact indication of the length of time the person had been away from the office, but rather would be akin to seeing the person's light on and briefcase next to the desk implying that the person was probably not very far away. The design of the sounds was intended to provide a sonic ecology in the background which could be habituated. To this end, the information about the group was played continuously at a low volume which was the base out of which the other sounds 'rose'. This is similar to the approach taken by Cohen in the Out to Lunch system. Three different approaches were taken to produce this sound, but in both cases the sounds had slow attacks and decays, with low pitches and minimal high harmonic content to minimise their intrusiveness. One design used the sound of waves to indicate the group's activity. The closer the waves sounded the more active the group. The second design used a musical vibe instrument at a low pitch. As the group's activity increased the number of notes played at any given time was increased. The third design combined these two approaches. Information about the amount of email similarly used different approaches. One approach used the sound of seagulls to indicate the amount of email received, the more calls heard the more email received by the user whilst a second approach used a combination of pitch and length of melody to indicate the number of emails received.

Although the Audio Aura system has not been formally evaluated a brief, informal evaluation with the beach sounds indicated that users found the sounds both pleasing and informative. To effectively evaluate such a

system a long term evaluation of its use would have to be undertaken. The use of a background sound as the base for the other sounds to limit their intrusiveness confirms the guideline described in the discussion of the Out to Lunch System.

3.5 Guidelines and Requirements

This chapter has discussed many different systems which provide audio feedback to their users. Throughout this discussion many guidelines about the use of sound have been extracted and some requirements for a toolkit of widgets that are designed to provide audio feedback have also been extracted. In this section, these guidelines and requirements are discussed.

3.5.1 Guidelines for the Use of Sound

Brewster *et al.* [30] produced some guidelines on the generation of earcons for use at the human-computer interface. These guidelines, however, are largely limited to the manipulation of the musical qualities of the sounds (e.g. pitch, rhythm and timbre) to meet the designers requirements and do not detail how such sounds should be used at the human-computer interface. In this section, guidelines drawn from the research described in this chapter about how sound should be used at the human-computer interface are summarised.

- Unless the sound would have been generated as a direct result of a user action (e.g. the user pressed a button) the absence of a sound is not a sufficient piece of feedback to alert a user to a problem. This is especially the case if the absence of the sound is masked by other sounds.
- The absence of a sound where one is expected is a sufficient piece of feedback to alert a user if the sound would have been generated as a direct result of a user action. Although this guideline would appear to be a contradiction to the previous guideline the significant difference is that the expected sound is a direct result of a user action. It is unclear, however, whether this guideline would hold if other sounds were playing at the same time.
- The sounds used to inform the user of an event should be ranked in order of importance so that if it is not possible to play them all, only the most appropriate are played.
- Sounds should not exclusively be mapped to events directly related to a user's interaction, but can also be mapped to changes in the system's data model. Although sounds which indicate the status of a user's interaction with a widget have been shown to be useful, an equally useful alternative is to map sounds to events that occur in the data model as a result of these interactions. These sounds could simply be alterations to the interaction sound, as was the case with the scrollbar described above, or could generate completely different sounds.
- Limit the number of different sounds used by analysing the requirements of the task closely rather than naïvely mapping a different sound to every different event. Although an intuitive approach would be to assign a different sound to every variation of an event this may not be necessary to meet the users requirements and could, indeed, confuse the user due to the sheer volume of different sounds. The naïve solution is also not very scalable.

- Ensure that the sounds provide information that the users require and cannot get from other, less intrusive sources. If audio feedback is used to provide information that users are not interested in or can access as easily from other sources, the sounds will quickly become annoying due to sound's potentially intrusive nature.

- Blend background sounds together to form a cohesive audio ecology. Sounds can be intrusive if played in isolation. Although this can be desirable in some situations, this is not the case for background sounds.

3.5.2 Toolkit Requirements

Throughout this chapter, requirements for the toolkit described in this thesis are captured from the discussion on existing systems which employ audio feedback. These requirements are summarised here.

- The full behaviour of the toolkit's widgets should be exposed allowing suitable designs of feedback to be added simply. Often, the events that require audio feedback are not freely accessible to the designer of the sounds. It is therefore important that no assumptions are made about which events should be exposed and which should not be exposed.

- The toolkit should be able to determine if it is not possible to play all the requested sounds and if so modify the feedback appropriately.

- It must be possible to form logical groups of widgets so that their feedback can be co-ordinated appropriately. It may be that these groups of widgets will need to communicate with each other to allow the appropriate changes in feedback to be made (e.g. the tool palette) or perhaps the individual widgets will need to know their relative position in a group (e.g. the menu).

- It must be possible for users to control the form of presentation used for its widgets. It is unlikely that a designer will always know which information is relevant to a user. It is therefore important that the user is able to control the forms of feedback being used for different widgets.

3.6 Conclusions

This chapter has provided an overview of the existing use of sound at the human-computer interface. It has shown why sound can be a useful addition to the human-computer interface and discussed the three main alternatives that currently exist. Speech was shown to be slow and inefficient for most uses but is, of course, the best way to present absolute information. Auditory icons, or everyday sounds, use the properties of the source of the sound as the basis for the way they convey information. Because the sound is often analogous to their meaning, auditory icons can be easy to learn but there is a danger that different users may attribute different meanings to the same sounds. Earcons are abstract, structured sounds that convey information through the modification of their musical properties. Because they are abstract, earcons are not as intuitive as auditory icons but equally are more universal because they are not constrained by the user's background. The review described several examples of the use of sound at the human-computer interface. From these examples, a set of guidelines about the use of sound at the human-computer interface were extracted. Some of these guidelines are applied to the design of a sonically-enhanced progress indicator described in Chapter 5. A framework for the evaluation of sonically-enhanced widgets which is also applied to the design of the

sonically-enhanced progress indicator was also described. Finally, some requirements for the design of a toolkit of multimodal widgets (described in Chapter 6) were extracted from the systems described here.

Chapter 4: A Comparison of User Interface Architectures and Their Support of Multimodality

Coutaz *et al.* [41] describe a software architecture as "the organisation of computational elements and the description of their interactions". This chapter provides some examples of commonly used user interface architectures. The design of such architectures is important because if designed well they allow user interfaces to be modified and re-used easily. The primary aim of this thesis is to develop a toolkit of widgets that are multimodal and sensitive to the use of these modalities. This chapter will therefore explore these aspects of the user interface architectures discussed. Any architectural features that facilitate these objectives are highlighted throughout the chapter and are summarised at the end.

4.1 Introduction

The design of user interfaces can be thought of as the creation of a human understandable representation of the underlying system that allows easy and effective manipulation of the system. The reality, however, is somewhat more complex. Often, the internal representation of a system is not the best representation for user comprehension. There can therefore be a mismatch between the two views of the system. Similarly, different users may choose to view the same system in different ways, again leading to potential mismatches. Furthermore, the way a system is organised internally may change without requiring any changes in the way users view it. Conversely, the way in which users view the system may also change regardless of whether or not the system has changed internally. To cope with these complications, several basic user interface architectures have evolved allowing the modification of the user interface according to one or more of the different issues described above. This chapter describes some of the most important examples of user interface architectures, discussing their different features and advantages. Some notable examples of user interface systems are then discussed. Finally, some existing user interfaces which explicitly support multimodality are discussed. By reviewing existing user interface architectures and systems guidelines for a design of a toolkit of multimodal widgets are elicited. These design guidelines are summarised in Appendix D.2.

Of course, there are more tools available to engineers building user interfaces than just descriptions of architectures. There are toolkits of interaction objects, or widgets, consisting of reusable software

components which can be combined to form a user interface. These widgets will typically have an associated framework, or architecture, which allows this combination into a user interface. At a higher level are tools which reduce the amount of programming required to build a user interface. These tools are based upon a toolkit of widgets and allow the generation of the user interface to be accomplished with a minimal amount of programming, typically by allowing the direct manipulation of the widgets to create a display layout. This thesis is concerned with a user interface architecture and an associated toolkit of widgets, not a high level tool which can be used to manipulate such a toolkit. Consequently a review of such tools is not undertaken. For a comprehensive review of user interface tools see [86].

4.2 User Interface Architectures

In this section, several user interface architectures will be described. For each architecture, its strengths and weaknesses as well as its contribution to the field will be discussed. Although the architectures described in this section are not explicitly designed to enable multimodality or resource sensitivity, their management of these issues is also discussed.

4.2.1 Seeheim Model

The Seeheim Model [96] was one of the first explicit user interface architectures. The model split the user interface into three, distinct monolithic components (Figure 4.1). The application interface model describes the semantics of the system from the user's point of view. This model includes the data structures and procedures available to the user and any constraints in the order these procedures can be accessed. The presentation component defines the way in which the system can be manipulated by the user by providing both the concrete rendering of the output from the system and the means by which a user can input to the system. The dialogue control component manages the communication between these two components. Should there exist a need for the application interface model to communicate directly with the presentation component, for performance reasons, the dialogue control can initiate a direct link, the dialogue control bypass, between these components.
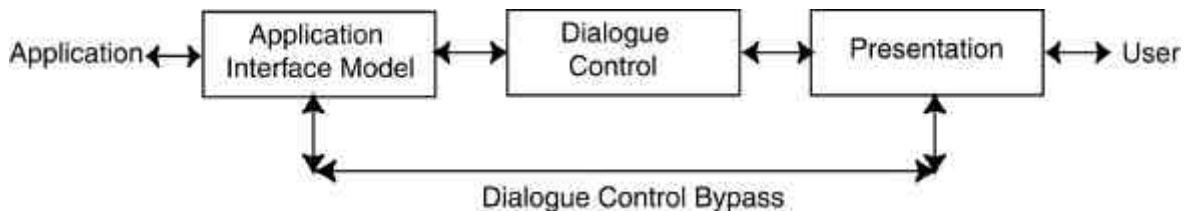
Figure 4.1 – Seeheim model showing the three components of the architecture.

The Seeheim model is notable because it separates the portion of the interface that the users interact with and the portion of the interface that translates these interactions into something the application understands. This acknowledges the potential for differences between the way the user interacts with the system and the way the system works internally. The presentation component handles user input to the system and presents any feedback defined by the system whilst the application interface model handles the mapping between interface

and application. The pipeline nature of the Seeheim model, however, does not handle the event driven model of user interfaces very well, as it does not allow the user inputs and system feedback requests to be interleaved.

The Seeheim model does not explicitly support multimodality as it specifies a single presentation component. However, the separation of the presentation from the application model implies the ability to change the presentation independently of the application model, perhaps simply to replace the existing presentation with a different form or perhaps, more interestingly, to supplement the existing presentation with an additional form of presentation, e.g. sound. The cost of doing this, however, is high because the dialogue control component is tightly coupled to the presentation component. Similarly, there is no architectural mechanism which has been specified to handle resource sensitivity. Seeheim does not specify, however, how the presentation component of the system should be implemented and it is here that resource sensitivity could be added.

Design Guideline 1	Separating the application model from the presentation enables the presentation to be changed.

4.2.2 The Arch Model

The Arch Model [108] was designed as an extension of the Seeheim model which acknowledged the problem of functional allocation between the different architectural components and enabled the use of different toolkits of interaction objects. The Arch model suggests separating the functionality of a user interface into five components (Figure 4.2).
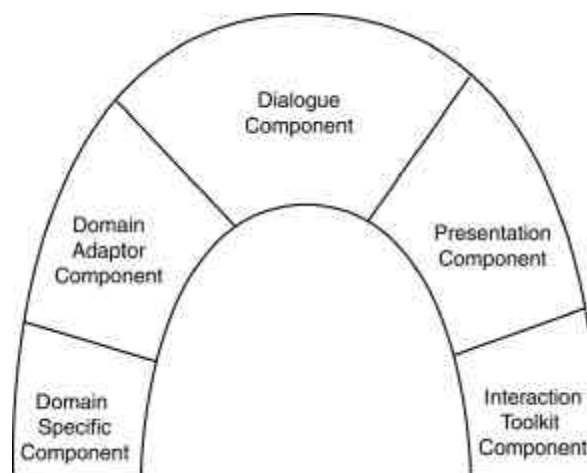
Figure 4.2 – The Arch Model (from [108]) showing the five components that comprise the model.

The domain specific component can be considered analogous to the application interface model in Seeheim. Similarly, the dialogue component can be considered analogous to the dialogue control in Seeheim and the interaction toolkit component can be considered analogous to the presentation of Seeheim. As such, the domain specific component performs domain-related tasks, the dialogue component controls the communication between the domain and the interaction toolkit which implements the physical interaction

with the end user. The two remaining components mediate the domain specific and interaction toolkit components' communications with the dialogue component. The domain adapter component implements any domain-related tasks required by the user of the system that are not implemented by the domain. It is responsible for triggering any domain initiated dialogue tasks as well as organising domain data and reporting semantic errors. The presentation component mediates the communication between the dialogue component and the interaction toolkit component. It does this by providing a set of abstract, toolkit independent interaction objects which the dialogue component can use. The decision about how these objects are represented by the interaction toolkit component is made in the presentation component.

By mediating the communication between the different components of the architecture the Arch model allows for changes in the system to be made. The primary intention of the Arch model is to allow different interaction toolkits to be used without necessitating wide-scale changes. By defining abstract interaction objects in the presentation component, the Arch model also defines a natural way of integrating virtual toolkits (e.g., XVT, described in Section 4.3) enabling applications to be ported more easily across platforms.

The authors of the Arch model, however, acknowledged that the use of different interaction toolkits may not be the primary reason for using a particular user interface architecture. They therefore generalised the Arch model into the Slinky meta-model. This meta-model allows for the functionalities of the different components in the architecture to shift from component to component. Giving the different components different functional emphasis allows the architecture to be geared towards different design goals. The Arch model described above can be considered as one instance of an interface architecture implemented using the Slinky meta-model. Consider, for example, an application written using the Arch model described above. The functions for opening and deleting files would be written in the domain specific component allowing different interaction toolkits to be used. If it was then decided that the ability to use different interaction toolkits was no longer important, a more sophisticated interaction toolkit, with a file selection widget, could be used shifting this functionality from the domain specific component to the interaction toolkit.

As with the Seeheim model, Arch does not explicitly support multimodal interfaces but does support changing the presentation. This is managed by the presentation component which maps virtual presentation objects used by the dialogue component to concrete interaction objects provided by the interaction toolkit component. The virtual presentation objects used by the dialogue component can be considered as abstract descriptions of the required presentation meaning they are independent of the concrete presentation. Again, there is no explicit architectural component for resource sensitivity but as the component responsible for mapping the abstract presentation objects to the concrete interaction objects the presentation component could be designed to perform this task.

Design Guideline 2	Describing the required presentation in terms of abstract, implementation-independent objects allows the separation of the presentation from the application model meaning the concrete presentation can be freely changed or supplemented.

4.2.3 Agent-Based Models

Agent–based models view the user interface as being composed of a collection of agents. Each of these agents has a state, an expertise and a capacity to initiate and react to events. The agents typically do not have a goal to achieve, but rather have the knowledge required to handle specific events. Coutaz *et al.* [41] highlight four advantages of agent based models:

1. An agent defines the unit for functional modularity, thus making it possible to modify its internal behaviour without having to change the rest of the system. This is analogous to an object in an object-oriented system which has an external interface or API but whose internal, private functionality is irrelevant to the rest of the system. As long as the external interface remains unchanged the internal functionality can be changed as often as required without necessitating changes throughout the rest of the system.

2. An agent defines the unit for processing, thus making it possible to distribute the system (i.e. distribute the agents) across a network.

3. An agent can be associated with one thread of the user's activity, thus allowing the user to suspend and restart that particular interaction at will. If a thread is too complex to be represented by a single agent, it can be represented by a collection of co-operating agents. A user-interface could be considered an agent which is composed of a hierarchy of co-operating agents, or widgets, each of which handle one thread of user interaction.

4. Agent models can easily be implemented in object-oriented programming languages. This is because the objects in an object-oriented program have many similarities to agents. They are both highly specialised processing units whose state is affected only by internal processing triggered by others. Furthermore, the subclassing mechanism of object-oriented programming allows objects (agents) to be modified with a minimum of effort as only the parts of the subclass needing to be changed require to be implemented.

The remainder of this section describes several of the more notable agent-based user interface architectures, highlighting their differences and similarities before briefly discussing the advantages and disadvantages of the different approaches.

The MVC Model

The MVC model (Model, View, Controller) [31] describes the user interface as a group of interaction objects, or agents. MVC models agents according to three functional perspectives. The model perspective defines the application oriented abilities of the agent. This can be considered analogous to the application interface model of Seeheim. The view perspective defines the perceivable output behaviour of the agent and the controller defines the perceivable input behaviour of the agent. The combination of the latter two perspectives can be considered analogous to the presentation component of the Seeheim model. MVC does not explicitly define a dialogue component with all three perspectives able to communicate with each other. Being agent-based these perspectives are distributed throughout the interface rather than being contained within monolithic components as in Seeheim. Consider, for example, the window shown in Figure 4.3. If this was built using an MVC architecture it could be considered as being composed of a hierarchy of three MVC

triads. The root node of the hierarchy would represent the window itself. Its view would manage the drawing of the window and its contents and its controller would handle input to the window. The view of the root triad, however, would delegate the rendering of the areas encompassed by the two buttons to the appropriate child triads. Similarly, the controller of the root triad would delegate the handling of input to the controller of the relevant child triad when the input was to one of the buttons.
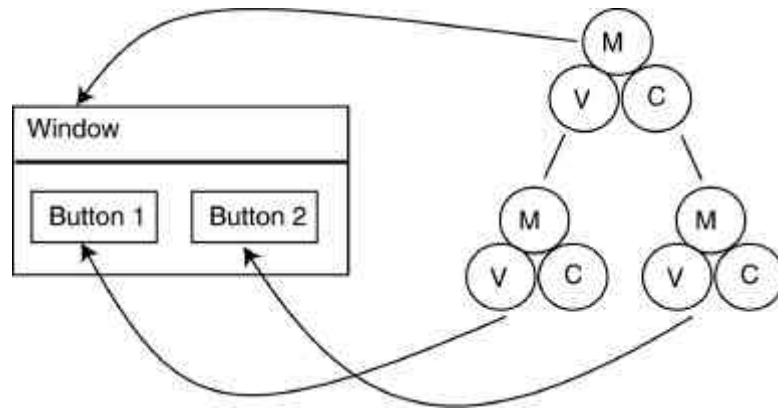
Figure 4.3 The MVC hierarchy for a simple user interface. Each of the three components of the window (the window and the two buttons) consist of a MVC triad which communicate with each other to ensure input and output are handled correctly.

MVC explicitly supports the use of multiple view-controller pairs for a single model, hence can support multimodality. Furthermore, because the output (view) is in a separate component from the input (controller) it is relatively straightforward to change the way the widgets are presented. Using an MVC hierarchy as described above, the MVC architecture is also capable of supporting resource sensitivity. Each node inherits an area in the screen in which it must display itself and any children, each of which are allocated an appropriate sub-section of the parent's area. This hierarchy, however, is tightly coupled to the visual presentation of the widgets and consequently is only suitable to manage sensitivity to graphical display resources. For example, the area used for the visual representation of a widget can be controlled by the widget's parent and because the parent is able to determine the requirements for all its children it is able to manage the distribution of the visual resource.

Design Guideline 3	Separating the output from the input makes it easier to modify the presentation of a widget (albeit with the overhead of communication between the input and output components).

The PAC Model

The PAC model (presentation, abstraction, control) [40] although similar to MVC, breaks down the functional allocation of the user interface slightly differently. The presentation component handles both the perceivable input and output behaviour of the agent. The abstraction represents the functional core of the agent and the control handles the communication both between the different components of the agent as well as the communication between different agents. By controlling the communication between the abstraction and presentation, PAC includes a mechanism to transform data between the abstract and concrete

perspectives of the user interface. By combining both the input and output aspects of the user interface into a single component, PAC enables the rapid feedback of any user interaction without the need for any extra communication between components, at the cost of making the task of changing the presentation of a widget more expensive. Because PAC agents can be grouped in a similar hierarchical structure to MVC agents, the PAC architecture is equally capable of supporting resource sensitivity. The explicit inclusion of an object (the controller object) to handle the communication between the agents, however, implies that it would be relatively straightforward to change the hierarchy, or perhaps utilise several different hierarchies, for different presentation modalities. It also implies that, as with the Arch model, there exists a defined syntax for the communication between agents meaning it should be easy to change one agent without affecting the others.

PAC-Amodeus [41] takes the PAC model and applies it to the Arch architecture. It decomposes the dialogue component of Arch into a hierarchy of PAC agents. The dialogue controller is decomposed in this way because it is the key element in the Arch architecture providing the bedrock for the communication between the application layers and the presentation layers. Furthermore, the other components in the architecture are often developed with the assistance of implementation tools meaning the developer has limited control over the architecture. A further extension of the Arch model provided by PAC-Amodeus is the provision of explicit boundaries between presentation and interaction toolkit components of Arch. The Arch model suggests what the functionality of these components should be within the framework of the Slinky meta-model, but PAC-Amodeus explicitly states how to define this functional decomposition. PAC-Amodeus states that the interaction toolkit component should be both device-dependent and language-dependent whereas the presentation component should be language dependent but device independent. All other components of the Arch model should be both device and language independent. The PAC-Amodeus architecture takes advantage of the way Arch enables style heterogeneity and allows for portability, modification and reuse of code as well as taking advantage of the way PAC supports the decomposition of tasks into multiple threads.

The ALV Model

The ALV model (abstraction, link, view) [67] has a similar functional breakdown to the PAC model. Where the PAC control handles communication between different agents as well as the communication between the different components of the agent, the ALV link is only concerned with the communication between the abstraction and the view. The link component is designed to allow multiple views to access the same abstraction. The abstraction component represents common data which can have multiple views. Every view has a link component which is its intermediary to the abstraction. The link component consists of constraints that maintain the consistency of the data between the abstraction and the view. These constraints ensure that multi-user applications where the application can be thought of as the abstraction and each user has a view remain consistent across all views.

As with PAC, multimodality is explicitly handled by the ability of an ALV agent to have multiple views to one abstraction, although it is similarly difficult to change the output as it is tightly coupled to the input in the combined view object. Resource sensitivity could be handled in ALV by the use of further constraints in the

link object. This, however, could prove to be computationally inefficient due to the high cost of constraint resolution as the number of constraints increases.

Summary

In this section, some of the most important user interface architectures have been discussed. The Seeheim and Arch models specify a function decomposition for the user interface. The Seeheim model implies that the presentation component can be changed by separating it from the domain, but because the dialogue controller object is tightly coupled to both the domain and the presentation it is, in fact, relatively expensive to do. The Arch model was designed as an extension of the Seeheim model which allows for the future modification of an interface built using the architecture. This is achieved by defining a syntax describing the required presentation in abstract terms which are then mapped to concrete feedback. To change the presentation it is therefore only necessary to change this mapping, meaning the dialogue component need not be altered. The Arch model was expanded by the Slinky meta-model of which Arch is a representative. This meta-model allows for the functional decomposition to vary between the different components depending upon the requirements of the design. This acknowledges that no one architecture will be able to fulfil all possible requirements.

Agent based architectures were then discussed. These architectures acknowledge the requirements of direct manipulation interfaces by allowing multi-threaded interaction and immediate semantic feedback. The four models discussed vary in the way they break down the functionality of the agents, but all separate the domain specific functionality from the presentation and input component(s). MVC separates input and output into separate functional components whereas ALV and PAC concentrate them into one. By separating input from output, MVC makes the changing of the output easier, but at the expense of the additional communication required between the separate components. PAC and ALV have a component explicitly designed to manage communication between components whilst MVC does not. By forming hierarchies of agents all the agent-based architectures are capable of supporting resource sensitivity. Typically, however, these hierarchies are based upon the visual presentation making the architectures only capable of being sensitive to the availability of the visual presentation resource.

4.3 User Interface Systems

This section briefly describes some of the most important User Interface Systems (or UISs) developed. A UIS can be considered as the implementation of a user interface architecture which typically is designed to meet certain, specific implementation requirements. Particular attention is paid to described UISs' support for multimodal and resource sensitive interfaces.

X Windows

The X Windows system [100] was developed at MIT to fulfil several requirements. Among the more notable of these are the requirements that the X system should be implementable on a variety of displays, the applications should be independent of the presentation device(s); the X system should support a wide range

of application interfaces and it should support high-performance, high quality text, 2D graphics and imaging. To fulfil these requirements, X was based on a client-server model. This allows it to be device-dependent by separating the device from the application. In X, the server can be thought of as the presentation device and the client as the application. The client and server can communicate across a network or on the same machine using inter-process communication. Both the client and server use a layered structure to ensure device and domain independence (Figure 4.4). In this example, both the application and the window manager are built on top of an X-Library layer. This layer manages the client's communication with the server. Similarly, the device library is built on top of a X-Server layer which handles the communication with the clients meaning that the communication between client and server is both device and domain independent.
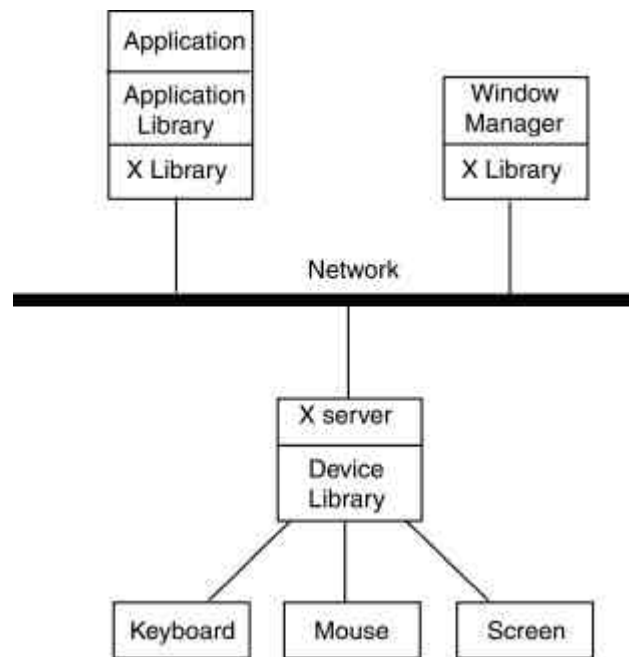
Figure 4.4 A typical X-Windows system structure. Both the application and device libraries are layered on top of standard X components so the X system remains device and domain independent (Adapted from [100]**).**

The server in X provides all the necessary resources (e.g. windows, fonts, mouse cursors and off-screen images) to allow the application to implement its user interface. The clients can request that the server creates an interaction object by supplying the appropriate parameters necessary for its formation and the server provides a handle to this interaction object in the form of a unique ID. Using this ID, the client can manipulate this object. The interaction object can be manipulated by any client thus allowing window managers to manage shared resources.

Multimodal interfaces are supported in X by allowing clients to communicate with multiple servers, each of which is a presentation device. In the server X handles the availability of the resource required for the presentation of the user interface. How this is handled depends upon the resource being used. In the worst case, the generation of an interaction object can fail due to the lack of available memory. It is up to the application to handle this failure appropriately, perhaps by requesting a similar service from an alternative server. The availability of colour on the display is handled by the server. The clients do not control the colour map being used, but rather request colours in a device independent way with the server translating this

request into the device dependent colour. Images are not described as low-level pixel manipulations, but rather as high-level drawing operations in terms of higher level abstractions such as lines and circles. The server is responsible for mapping these high-level requests into device dependent instructions. Fonts are provided by the device with the client creating the fonts by requesting a font with a given name from the server. No provision is provided for the mapping of one font to another if the source font is unavailable on a particular device. Automatic management of resources was enabled in X11 [63] by allowing servers to redirect requests made by a first client to a second client. This second client could be a presentation manager which could then modify the request as necessary. Conceptually, this is an additional layer in the X-architecture which can receive events from multiple clients and consequently control the feedback used across the different clients.

Design Guideline 4	A client server architecture facilitates multimodality by enabling the client to use multiple servers to provide a service at the cost to the client of determining which server to use.

Design Guideline 5	By providing a component which can intercept and modify requests for presentation global control over the resources used can be obtained.

Smalltalk

Smalltalk [31] was the first language and programming environment to support the development of interactive graphical user interface. It used the MVC architecture as its basis. Smalltalk views manage a portion of the available screen space, outputting graphical and textual feedback to that area whilst their associated controllers manage the user input to that area. The model manages the data represented by the view and manipulated by the controller. If a model has multiple views these are kept consistent by the model which maintains a list of dependent views and notifies them when a change occurs. Although the input and output to Smalltalk are separated into controller and view components, they are tightly coupled with each component having a reference to the other. This means that it is not possible to change the view without changing the controller. As Smalltalk is built using an MVC architecture, it shares the same advantages and disadvantages as MVC except with regard to the ability to change the view component without changing the controller.

InterViews

The InterViews system [71] is an agent based interactive toolkit built on top of X. Like PAC, InterViews combines the input and output behaviour of the agents into a single component, in this instance called the view (unlike the MVC view that only encapsulates the output behaviour of the agent). The abstract behaviour is defined in the subject component. This separation allows the use of different views for the same subject. The toolkit is composed of interactive components called interactors. These interactors can be both atomic interaction objects, such as buttons and menus, or compositions of interactors which form a scene. As with PAC, multimodality is possible but the overhead is greater than if they are decoupled.

The management of the available presentation resources (in this case the available screen real estate) in InterViews is handled by the interactors. Each interactor has a preferred shape and size as well as a stretchability and shrinkability which define the upper and lower limits to which the interactor is prepared to change its preferred size. A scene defines its preferred shape and size according to the preferred shape and size of its children. Scenes can use different algorithms to manage the presentation of the interactors they contain. Examples of these algorithms are the box and tray algorithms. The box algorithm tiles its child components, not permitting them to overlap whereas the tray algorithm allows overlapping. Although some form of resource management is in place, there is no scope for using an alternative modality should there be insufficient screen space. Thus, resource sensitivity is handled, but only for visual presentation.

Java AWT & Swing

The Swing user interface toolkit [55] is a user interface toolkit for Java. Swing succeeds AWT as Java's user interface toolkit, although many features of AWT were retained. Swing was built according to several requirements, most notably that it should be platform independent and support multiple look and feels. The look and feel of a Java application defines the appearance of the application, allowing the same application to, for example, mimic the appearance of the platform the application is running on, as well as allowing some of the concrete interaction details, such as accelerator key, to be changed. To facilitate this, the Swing architecture is based upon a modified MVC architecture. Unlike MVC, however, Swing combines the view and controller objects into a single user interface object, or component, which handles the input from and output to the user. If a model has several components representing it they are kept consistent by the model. As with Smalltalk, when a component is given a model it registers itself as being interested in any changes to that model and consequently is notified when the model changes. When such a notification is received the presentation is updated accordingly.
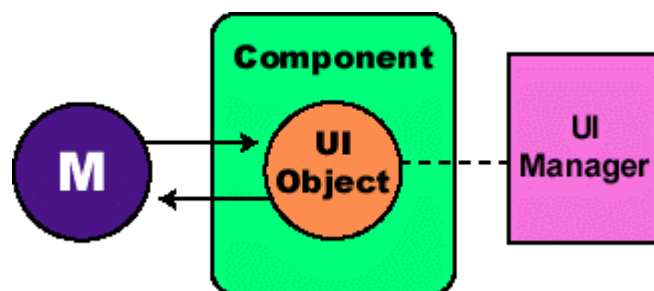
Figure 4.5 The Swing architecture (From [55]). The view and controller components of the MVC architecture it is based upon are replaced by a single component with the presentation's look and feel supplied by the UI Manager.

The look and feel mechanism of Swing handles multimodality by enabling the presentation of an interface to be changed. This change, however, is an interface wide change meaning it is neither possible to change an individual widget's presentation or apply multiple look and feels. Thus, if an additional modality is to be used, for example sound, this must be implemented in a look and feel which also implements the visual presentation. A forthcoming version of Swing does include the ability to include sound in a look and feel [107] but this is still limited in its approach. A table which maps events to sounds is loaded by the look and feel and the sounds are played when the appropriate events occur. The sounds played can be changed by changing this mapping, but there is still no ability to change the presentation of individual widgets. There is

also no scope for dynamically modifying the sounds played according to the availability or suitability of the sounds for the current context. This architecture is built around the visual presentation with the sounds as an addition to the presentation, not as an integral part of it.

As with the MVC architecture upon which it is based, Swing handles resource sensitivity by managing the visual resource required by the interface in terms of the widget hierarchy. Every component has a preferred, minimum and maximum size and the distribution of visual resource is determined by every parent requesting its children's preferred size. If there is insufficient visual resource, then the minimum sizes are used to determine the minimum allocation of resource possible. There is no explicit management of any other form of presentation resource although the audio look and feel mechanism described above does modify the sounds used according to the platform the application is running on. At start-up, the look and feel manager defines the mapping of sounds to events such that the sounds played mimic those that would be played for standard applications on that platform.

Garnet & Amulet

Garnet (Generating an Amalgam of Real-time, Novel Editors and Toolkits) [78, 79, 82-85] is a user interface architecture primarily aimed at allowing graphical and interactive behaviour to be quickly and easily built. The Garnet system includes high-level tools to assist in the building of user interfaces, but in this section only the Garnet toolkit, which is of most interest to this thesis, is discussed. The Garnet toolkit is composed of four layers built on top of the device dependent operating system. The bottom layer, the knowledge representation (KR) object system, is used to develop the application model. This allows the development of an object-oriented model of the application which uses a prototype-instance model rather than the usual class-instance model. This has the advantage of allowing any instance of an object to be the prototype for another, rather than only allowing a class to be used. Thus, it is possible to make new instances of currently instantiated objects rather than simply instantiating an object from a basic template or class. All the prototype slots, which store data or methods, are inherited by the new object. Furthermore, the new instance of the object is free to add or remove slots as required. The second layer in the Garnet toolkit is the constraint system that manages the relationship between the objects in the KR system and the interactors and graphical layer (described below). These relationships consist of constraints which define the way one value is changed when a second value changes. Garnet uses one way constraints which mean that if one object changes value, the other is updated, but not *vice-versa*. Garnet, however, does allow loops in the constraint dependencies, with the system traversing the loop exactly once every time a change occurs.
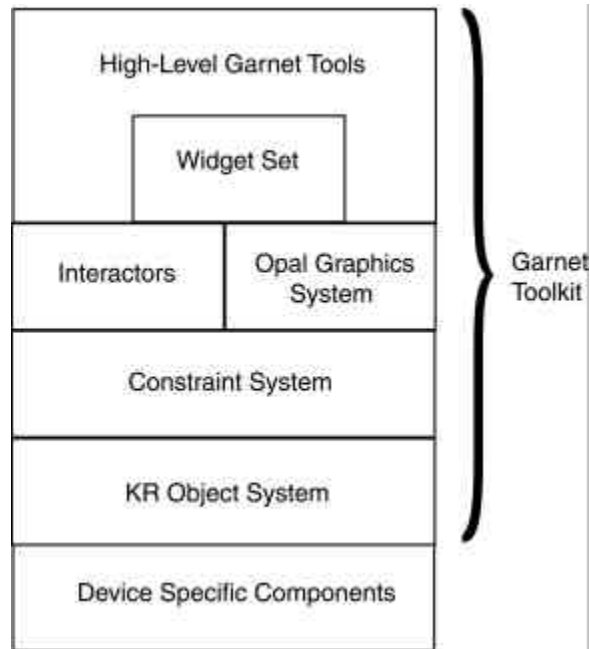
Figure 4.6 The Garnet toolkit architecture (Modified from [79]). Garnet mimics the MVC architecture with the view (Opal) and controller (interactors) managed by the constraint system which is also linked to the domain. The KR Object system provides a prototype-instance object-oriented model.

The third layer in the Garnet toolkit consists of two separate components, the graphics system and the interactors. Together, this layer handles input and output to the user but, like MVC, separates the functions into two components. The output is handled by Opal, Garnet's graphical system. Opal manages the drawing and erasing of visible objects, changing the presentation according to the arrangements of the objects on the screen. The graphical properties of objects can be specified in terms of constraints, with the presentation of the objects automatically updated if the values of other, relevant, objects change. Input to the Garnet toolkit is handled by interactors. These objects define the behaviour of an interactive component independent of its graphical representation. This allows the presentation to be changed without the behaviour changing, unlike Smalltalk which, although also based on the MVC model, has its views and controllers tightly coupled.

Garnet specifies six interactors which are sufficient to cover most of the interactive behaviours for a traditional graphical user interface:

- The menu-interactor allows items to be chosen from a set. It can be used to create standalone buttons as well as, for example, menus and button groups.
- The move-grow interactor allows objects to be moved or changed size using a mouse.
- The new-point-interactor allows an arbitrary number of points to be entered using the mouse to, for example, create a new line in a graphics editor.
- The angle-editor allows the angle at which the mouse is moving around an object to be calculated. This could be used to allow objects to be rotated for example.
- The trace-interactor allows the movement of the mouse to be captured to enable, perhaps, free-hand drawing.
- The text-string interactor allows the inputting of text.

Myers *et al.* acknowledge that whilst the set of interactors allow most standard forms of behaviour to be modelled, "radically different types of behaviours" such as gestural input or speech input require new interactors to be implemented. Multiple interactors can be used to provide complex forms of feedback.

Consider, for example, an object being dragged around an interface's desktop. Two forms of feedback are required here: the motion of the object being dragged and the highlighting of any objects which are able to accept the dragged object if it is dropped. This could be implemented in Garnet using a move-grow interactor to handle the dragging of the object and a menu interactor to perform the highlighting of the desktop objects. All interactors are driven according to the same state machine (Figure 4.7). There are three states defined for an interactor: start, running and outside. Once an interaction has started, it enters the running state. It can loop round this state indefinitely, or it can return to the start state if the interaction is stopped or aborted. When the interaction is running, the interaction can move outside the widget and enter the outside state. From here, the interaction can return to the running state or it can be stopped or aborted.
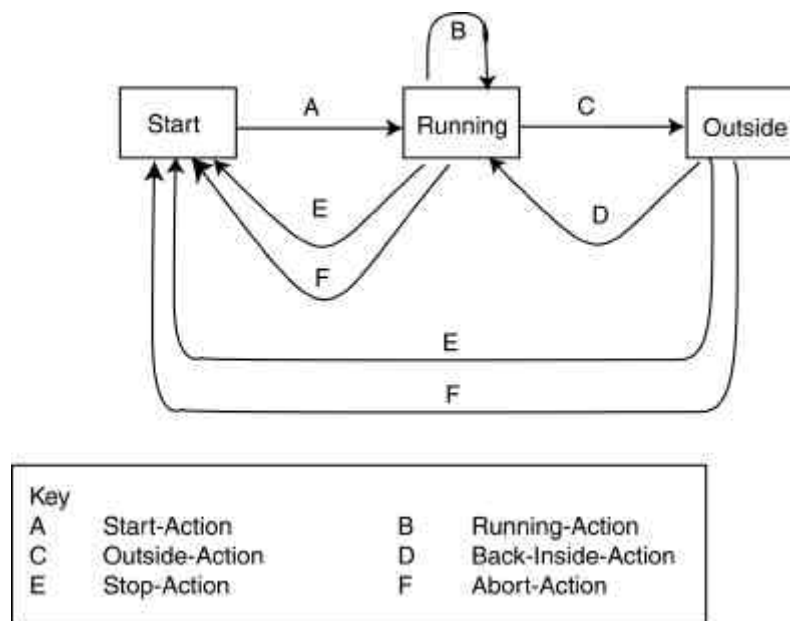
Figure 4.7 A simplified version of the state machine used to control every interactor in Garnet (modified from [82]). Once an interactor has started it can continue to run indefinitely, move outside and re-enter the widget, stop or be aborted.

The top layer of the Garnet toolkit provides widgets (called gadgets in Garnet) which applications can use to build user interfaces. These gadgets include commonly used widgets, such as menus, buttons and scroll bars, but also include more sophisticated, high-level widgets such as selection handles. If the application wishes to develop its own gadgets, it has access to the graphics and interactors layer below the widget layer.

The Amulet system [85, 87, 88] (Automatic Manufacture of Usable and Learnable Editors and Toolkits) is an improved version of the Garnet system now written in C++ (Garnet was written in Lisp). One of the changes between Garnet and Amulet is the addition of a new layer between the KR object system and the device. This layer, the GEM (Graphical and Events Manager) layer, provides a device independent interface to the device specific interface toolkit allowing Amulet applications to be ported easily between platforms supported by

Gem. The constraint system in Amulet includes a multi-way constraint solver as opposed to the one way constraint solver in Garnet, allowing more complex constraints to be resolved. A new interactor was developed for Amulet to allow the use of gestural input. Amulet also includes a mechanism which allows the addition of animations to a user interface [87].

Both Garnet and Amulet were designed to allow the easy modification of the graphical presentation and interactive behaviour of a user interface. The user interface toolkit part of the systems are split into a MVC-like decomposition with the consistency of these separate parts maintained by a constraint system. The controller component (in this case called interactors) defines standard behaviours to which the different widgets conform. Because the different behaviours have hooks onto which presentation can be associated it is possible to replace the presentation of a widget without necessitating any other changes.

Design Guideline 6	Providing hooks onto different aspects of the behaviour of widgets means that it is possible to change the presentation of the widgets without the explicit knowledge of the behaviour.

Andalusite [88] (Amulet's New Development Augments the Look-and-feel Using Sounds Including Text-to-speech and Effects) is an extension to the Amulet system that allows the easy addition of audio feedback to graphical user interfaces. Andalusite provides a high level, extensible interface to sounds, a platform-independent mechanism for controlling the use of the available audio hardware and an architecture for associating sounds with interactive behaviours. The Andalusite architecture provides two layers between applications and the platform dependent sound system. The low level interface provides sound objects which can be manipulated to provide a variety of different sounds whilst the high level interface provides the mechanisms which allow these sounds to be associated with interactive behaviours. Applications have access to both these levels.

The low-level interface provides sound objects to which various parameters can be applied. These parameters allow a variety of properties of the sound to be manipulated, including physical properties of the sound such as volume and balance as well as logical properties such as the number of times it is to be repeated and what sounds, if any, should follow upon completion. In this way, it is possible to create a sequence of sounds and, similarly, it is possible to control whether two sounds should be played together. If a sound is listed as a sound to be interrupted when a second sound is to be played, the first sound will automatically be interrupted when the second sound starts. The availability of audio resource is managed by defining sounds which should be pre-empted if there are insufficient resources available to play a sound. Once the sound has completed, the sound that was pre-empted continues.

The high level interface to Andalusite allows the sounds produced in the low level interface to be associated and synchronised with interactive behaviours and animations. To facilitate this, animation and interactor objects in Amulet have three extra slots which hold the sound to be played at the start of an interaction or animation, the sound to be played whilst the interaction or animation is ongoing and the sound to be played when the interaction or animation completes. Once the start sound has completed the interim sound is played

until the interaction is completed when the stop sound is played. If an animation is to be synchronised with the sounds being played a similar mechanism applies. Sound objects have three animation slots for start, interim and stop animations which are played appropriately. Currently, Andalusite provides pre-recorded sampled sounds and a text-to-speech capability, although it is hoped to extend this range to MIDI files and earcons (described fully in Section 3.3.3). It is not possible, however, to add sounds to any other point of the interaction without defining new interactor objects, limiting the way sounds can be added. Andalusite manages the audio hardware available by defining which sounds should be played in preference to others if there is insufficient resource available. There is no scope, however, for modifying the sounds themselves to manage the resource better or for deciding whether using audio feedback is the correct option to take given the current context.

XVT

The XVT virtual toolkit [110] was designed to allow application interfaces to be ported between the Microsoft Windows ^{TM} and Apple Macintosh ^{TM} platforms. This is achieved by providing a toolkit of platform-independent, virtual interaction objects. By specifying the interface to an application in terms of these virtual interaction objects, the application can be used with two different platforms without any changes. This conforms to Design Guideline 2 described in Section 4.2.2. The virtual toolkit is a thin layer between the application and the device specific environment. The toolkit provides an abstraction of the abilities of the platforms it covers, mapping these abstractions to the platform specific commands. By utilising the existing native components, an application built using the virtual toolkit has the advantage of appearing to be implemented in the native components. Conversely, it offers no opportunity for the modification of the presentation. Finally, because the toolkit relies upon the functionality of the underlying native system, to maintain portability XVT can only implement those functions that are available across all platforms.

SUIT

SUIT [95] was designed as a toolkit of interaction objects that are simple to use by an interface developer and portable across several platforms. The former requirement was met by basing the toolkit on a language that is simple to learn and by the development of tools that allow the effective and simple implementation of an application using the toolkit. The portability of the toolkit was assured by layering a cross platform graphics package between the SUIT toolkit and native toolkits such as X and Motif. This shares with XVT the advantage of ensuring applications built with the toolkit are portable but differed in that any applications built using SUIT would not look like native applications, whereas those built with XVT would. The advantage in the approach taken by SUIT is the ability to provide functions that are not necessarily available in the underlying system.

Summary

In this section, the design of seven different user interface systems was discussed, especially with regard to their ability to support multimodality and resource-sensitivity. By employing a client-server architecture X supports a multimodal interface by allowing clients (applications) to employ the services of multiple servers (output devices). Similarly, by allowing a second client (a window manager for example) to intercept all the

events sent by a client to a server it is possible to manage the use of a resource. Smalltalk implements the MVC architecture and so supports resource sensitivity through the hierarchy of widgets. However, because the view and controller components of the architecture are tightly coupled in Smalltalk it is harder to develop multimodal interfaces as it is necessary to reproduce both components if replacing the view used by a widget. InterViews decomposes the interface in a similar fashion, although the input and output components are explicitly defined in a single component. The Swing toolkit for Java is also based upon a modified MVC architecture. Again, the view and controller components are combined into a single component, but with the presentation of the widgets supplied by a look and feel manager. This allows the presentation of the widgets to be changed, but at an application wide level, not for individual widgets. Sounds can be added to the presentation of the widgets using the look and feel mechanism but the control over when the sounds can be played is limited. Amulet and its predecessor Garnet supported the quick adaptation of graphical interaction objects by using the MVC model and not tightly coupling the view and controller components. The controller, or interactor in this case, defines the behaviour of the different widgets in abstract terms with the view providing the concrete presentation which can take any form. Resource-sensitivity could be supported by using constraints in addition to those which currently ensure that the three components of the MVC model remain consistent. XVT and SUIT were both designed to allow user interfaces to be easily ported to different platforms. In both cases, the interface was built with abstract interface components which were mapped to platform dependent components.

4.4 Architectures That Support Multimodal Interfaces

In this section several user interface systems that support multimodal interfaces. In the discussion at the end of this section the advantages and disadvantages of the different systems are highlighted.

Promise

The PROMISE system [4] (PRocess Operator's Multi-media Intelligent Support Environment) is a system architecture designed to allow the presentation of process control environments. To enable this, a presentation architecture, the Multi-Media Man Machine Interface (M^4I) architecture was developed. The key goals of the PROMISE M^4I system were three-fold:

- To ensure the application is independent of the presentation system.
- To exploit the potential of multi-media interaction with the user by, for example, limiting the cognitive load on the user by presenting information in the most appropriate modality and increasing redundancy by providing multiple views of the same data.
- To support the most appropriate interaction style according to the current domain.

The PROMISE architecture (Figure 4.8) consists of four layers on top of the existing process control system. The asynchronous data manager mediates the communication between the existing system and the rest of the

PROMISE system. The advisory/diagnosis knowledge base systems analyse the data from the current state of the process. The two remaining layers comprise the multimedia toolset of PROMISE. The interaction data model describes and sets up the link between the application entities and their presentation in the user interface, although once this link has been established it is the role of the asynchronous data manager to ensure that the two sets of entities remain consistent. The presentation system is responsible for the presentation of display requests and the dispatching of user requests.
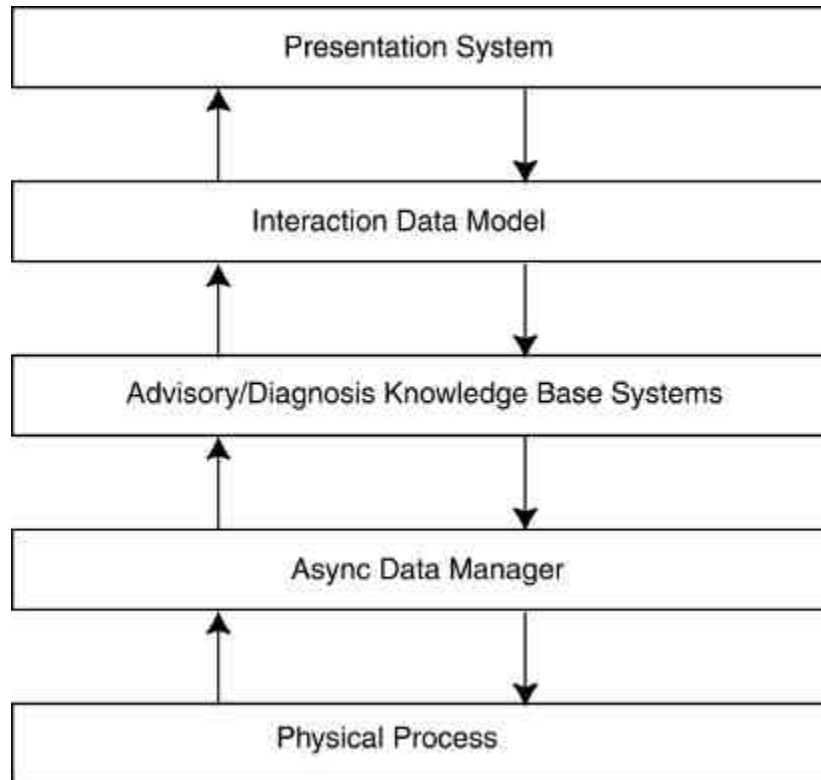
Figure 4.8 – The high level architecture of PROMISE (adapted from [4]).

To be able to successfully use different output modalities, the presentation system needs to be able to present the information using the available presentation resources. This implies that the presentation system needs to manage the allocation of these resources. Alty *et al.* consider this problem to be analogous to the problem of task scheduling with the exception that it is unlikely to be acceptable to postpone the presentation of a piece of information in the same way a task can be postponed. They propose that this can be resolved by the use of alternative modalities. Verbal output, for example, could be presented as text in a window or as a spoken output via a speech synthesiser. The selection of presentation made on the basis of availability alone, however, is not entirely satisfactory. There will almost certainly be some modalities which are more appropriate than others for a given interaction so a second consideration for the allocation of resources is the suitability of the modality. Alty *et al.* describe three pieces of information required to ensure the appropriate modality is selected:

- The characteristics of each resource;
- The nature of the interaction being presented;
- User preferences.

To encapsulate this information within the presentation system implies that models of this information need to be generated to allow the application of rules to determine use of different presentation modalities. Alty *et al.* acknowledge that the likelihood of the presentation system being able to automatically select the most appropriate resource or resources for presentation would at best introduce severe performance limitations and at worst be unfeasible. They propose, therefore, that the presentation system chooses between the alternate output modalities specified by a designer and according to the designer's rules. A second problem inherent in the use of multiple output modalities is the synchronisation of the different presentations of a piece of information (or a scene). Alty *et al.* suggest this should be handled by the synchronisation of concurrent processes, with each process controlling the presentation in one modality. They suggest that these processes should contain abstract barriers, called gates, at which all the processes forming a scene must synchronise before continuing. Similarly, when a scene is to be terminated all processes must reach a predefined point of sensible termination, a mark, before the scene can terminate. A similar mechanism can be used if one of the resources being used in a scene needs to be reallocated. The resource manager requests the termination of the process being reallocated and waits until it reaches a mark when it can sensibly be terminated.

The PROMISE architecture allows the presentation of a process control system to be done using the most appropriate output modalities available. The decision about which modality to use is taken by a resource manager based on criteria laid down by the designer of the system.

ENO

ENO [10] is an audio server developed to allow Unix applications to incorporate non-speech sounds. ENO has a very similar architecture to X, allowing applications transparent network access to the server. Rather than simply acting as a repository of sampled sounds, ENO controls the shared resource of a sound space shared by the client applications. The objects which inhabit this sound space are what ENO describes as sound sources. These sources represent possible causes for a sound rather than the sound itself. A sound is produced as the result of an interaction with a source. For example, a bell source will produce a different sort of sound when an impact interaction is applied to it as opposed to a scraping interaction. Every sound can have a distinct spatial location. This virtual location may be defined according to the type of sound (all sounds with a similar meaning may be given a similar location for example) or the sound's importance (more important sounds may be spatialised to sound closer for example). These sources are arranged in a tree where children inherit their parents attributes. In this way, for example, a group of sources could have their volume adjusted by the adjustment of the volume of a common ancestor.

The ENO server is entirely responsible for the rendering of the audio feedback according to the parameters given by the client. This has several advantages. Allowing the server, which may well have specialist audio hardware, to manage the physical use of the hardware necessary for the generation of the sounds allows the use of this limited resource to be shared across several clients. If the rendering of the sounds was distributed across the clients this would have the advantage of distributing the CPU load across the clients, but it would have the disadvantage of the forcing the clients to have to manage the resources required for the rendering of the sounds. The ENO server employs several techniques to manage the resources available to render the required sounds. Sounds are cached on the assumption that the same sound may be played several times

during the same interaction. The caching of the sounds is done to maximise the hit rate of the cache. If, for example, a sound is requested with a particular spatial location, the sound is cached prior to its spatialisation because the spatialisation of sounds does not involve much CPU overhead[10] and the chances of the sound being requested for the same location are deemed unlikely. Because it is possible to calculate the length of time required to render a sound, and this time is typically dependent upon the sample rate and hence quality of the sound, the sounds are produced to the highest quality possible in the time available to the server. This management of the hardware resource means that although some lower quality sounds may be produced, the audio feedback is always generated on time and without any clicks caused by the generation of the sound lagging behind its presentation. The disruption caused by this could be further minimised by reducing the quality of the sounds least likely to be noticed (those spatialised to be furthest away for example) first.

Mercator

The Mercator project [91] was designed to allow visually impaired users easy access to previously inaccessible graphical user interfaces by mapping a graphical interface to an auditory interface. This mapping was done transparently to the application. The need for such a mapping arose because, although graphical user interfaces are regarded as being easier to use for most users, they are almost impossible for blind users to use because they cannot be translated to text (and consequently speech) in the way that text-based interfaces can be. The Mercator system maps the graphical interaction components of X (described in Section 4.3) to its own Auditory Interface Components (AICs). As with X's graphical components, these AICs may be a single widget or a composition of several widgets. The mappings between X's widgets and Mercator's widgets are not necessarily one to one mappings, however. A text window, for example, may contain scrollbars as well as a text area in X, but because the concept of limited screen space which necessitates the scrollbars is not relevant to the AIC, it will only consist of a representation of the text. Mercator represents AICs as auditory icons (see Section 3.3.2 for a full description of auditory icons). Every type of AIC has its own auditory icon and each AIC has various attributes. Many AICs have text labels which can be presented using a speech synthesiser. Other attributes of AICs (e.g. button unavailable, menu item has sub menu etc.) are represented by modifying the auditory icon for that type. If, for example, a button was unavailable, the auditory icon is played in a muffled fashion to indicate this.

By allowing the audio presentation to be navigated Mercator enables the user to "scan" the interface and to interact with it. The former is achieved by allowing the user to navigate a tree hierarchy which breaks down the user interface into smaller AICs. The user can navigate this tree using the numeric key pad, and can interact with the interface by, for example, pressing Return to select a button.

To implement this audio interface, Mercator gathers two types of information about the interface from the application (in this case, an X Client). It was decided to do this rather than simply modify the X-Toolkit as it allowed the addition of auditory feedback to be achieved for existing X applications. Thus Mercator took a generic approach which would allow existing applications to benefit from the addition of audio feedback

---

[10] The processing cost of spatialising sounds is not necessarily cheap but because ENO used spatialisation as a means of distinguishing sounds rather than allowing users to accurate identify the location of a source the spatialisation could be done relatively cheaply.

without needing to be rebuilt. The first type of information gathered is the communication between the X client and server. This told Mercator both when graphical output and interaction events were taking place. This low level information regarding what was happening on the visual display was insufficient to generate AICs however. Higher level information was required from the X Toolkit Intrinsics layer which related the lower level graphics operations to higher level widgets. The combination of these two pieces of information allow the tree of AICs which models the auditory interface to be generated. By continuing to monitor the communication between the X Client and Server as well as continuing to update the higher level information it is possible to update the presentation as required and to allow user input in the auditory interface.

It was found, however, that this architecture did not allow the satisfactory generation of the tree of AICs representing the audio interface because insufficient information was gathered. Mercator was therefore updated to overcome this problem [51]. The XT Intrinsics library and XLib components were modified so that they would communicate relevant changes in the application to any interested external agent(s) allowing the model of the interface to be generated. Although this no longer makes the use of Mercator transparent to the application (it would need to be linked to the new libraries) it was felt the advantages of getting the extra information required to model the interface outweighed the disadvantages. Input is handled by mapping low level input events received by Mercator to the low level X events the application would expect for the input event if it had occurred using a standard input technique such as a mouse. To select a menu item, for example, a user uses speech input. Mercator would receive this event and generate the appropriate mouse events to simulate the selection of the menu item. The architecture of Mercator, however, is still based on the visual representation of the interface with all events having to be translated to and from the audio modalities used. There is also no scope for controlling when one output modality is used in preference to another.

Homer

Homer [98, 99] is a user interface architecture which supports the development of interfaces in dual modalities. Rather than simply extracting the model used for the visual interface and applying it to an audio interface, possibly after modifying it to suit, Homer builds separate models for each of the interfaces. Savidis *et al.* call such an interface a dual user interface. These models are developed from a single, abstract model of the user interface which does not rely upon any modality specific features. This approach has the advantage of developing interfaces in multiple modalities that are specifically designed for each modality but suffers from not being applicable to any existing applications as well as increasing the workload of the user interface designer.

The key construct which allows HOMER to produce the dual interfaces is the virtual interaction object. These differ from objects in virtual toolkits such as XVT (see Section 4.3 for a full description of XVT) in that they are not grounded in a particular interaction metaphor such as the desktop interface, but are independent of such metaphors, describing only the abstract behaviour. These virtual interaction objects can then be mapped to either visual or non-visual virtual interaction objects which are grounded in a particular presentation metaphor. These visual or non-visual virtual interaction objects can now be mapped directly to the concrete interaction objects. It is the responsibility of the designer of the interface to specify the mappings between the different virtual and physical objects, with consistency between the different objects maintained

by the use of designer included constraints. The physical interaction objects available to the designer depend upon the toolkits used to provide them. The Athena and Motif toolkits were used to provide visual interaction objects and the COMONKIT was implemented to provide audio interaction objects. This toolkit employed a room metaphor as its basis, with the different surfaces of a room capable of being used as containers for objects and navigation between rooms was possible. The physical layout of the room could be presented using 3D audio surrounding the user.
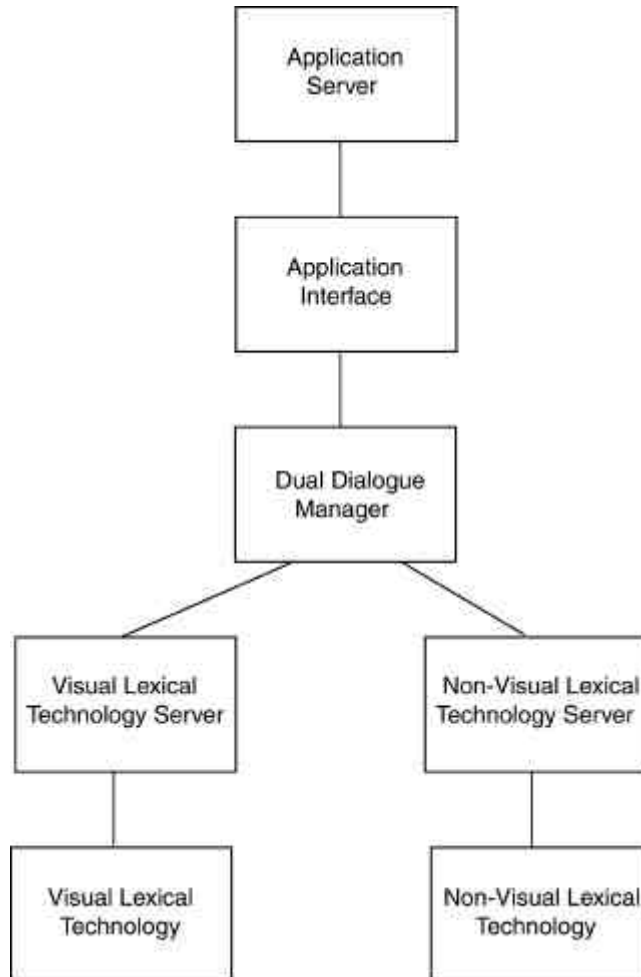
Figure 4.9 - The Homer architecture, which is based upon the Arch model but with two versions of the presentation and interaction toolkit components: the non-visual and visual lexical technology server (presentation) and lexical technology(interaction toolkit) components.

The runtime architecture of HOMER is a modified version of the Arch architecture (see Section 4.2.2 for a full description of Arch). As in PAC-Amodeus (see Section 4.2.3) the dialogue component (in this case called the dual dialogue manager) is populated with PAC agents. These agents are, however, modified with an extra presentation component to handle the extra interaction space being used. Correspondingly, there are two Arch presentation components (the visual toolkit server and the non-visual toolkit server) and two interaction toolkits.

By not taking the approach of adapting the existing model used for the visual interface, but generating a truly modality independent model, HOMER allows the development of both audio and visual user interfaces to an

application which may use the most appropriate presentation metaphor. This, however, comes at the cost of not being able to apply the method to an existing application. Because it is aimed at providing interfaces to applications for both sighted and non-sighted users, HOMER does not include any scope for applying different modalities to different parts of an interface, neither is there the ability to change the modality being used, or the way a particular modality is being used, due to the current context.

Fruit

The FRUIT system [70] (Flexible and Rearrangeable User Interface Toolkit) was developed to provide a mechanism that would allow for conditions when either a visual interface is insufficient for the user or the platform supporting the interface has only limited functionality. For example, visually impaired users would find a visual interface unsatisfactory and would require the interface to be presented in a different modality. Similarly, a user running an application on a powerful desktop machine may need to switch to a less powerful laptop but continue to run the same application. This may necessitate a change in the way the interface is presented. Furthermore, FRUIT was designed to be able to handle such switching dynamically, suspending one form of interface and reconnecting the application to a new interface. By providing this mechanism, it was hoped that FRUIT would bridge the gap between standard graphical user interfaces and user interfaces designed for those with special needs.

To achieve the flexibility described above, FRUIT makes some assumptions about the capabilities of the user interface. These assumptions were made to allow the implementation of everyday applications without too much difficulty. It was assumed that there would be some form of text input (e.g. keyboard or speech recognition) and pointing devices for input. Similarly, output was limited to either bitmapped displays, screen oriented displays or line oriented displays. These three displays form the different presentation modalities available with the possibility of audio feedback coming from the use of screen readers with screen oriented displays and speech synthesisers with line oriented displays.

The FRUIT toolkit is composed of abstract widgets that define the functionality of the widgets without defining how they should be rendered. Each widget is defined to have three groups of slots. The first group of slots holds any modality-independent information. One such slot could, for example, hold the callback procedure invoked when the widget is activated. The second group of slots holds the methods which handle the user interaction with the widget. These slots could, for example, hold the methods which handle the insertion and deletion of characters from a text area. The third group of slots can hold modality-dependent information about the possible rendering of the widget. These slots could, for example, hold foreground and background colours for the widget if it is rendered graphically. The developers of an interface define this widget set by extending the appropriate abstract widget and filling the different slots accordingly. Applications can now be built using these abstract widgets.

Three components are used to generate the user interface. A communication stub holds the interface in terms of abstract widgets and provides the API through which the application builds the interface. An interaction shell maps this abstract interface to a concrete representation and manages the user interaction with this representation. Applications will typically invoke a single interaction shell but may, if desired, use several

shells simultaneously. This would allow, for example, the presentation of an interface in both graphical and auditory modalities. The user specifies which shell(s) should be used. The third component of the architecture, a session manager, manages FRUIT applications, enabling the suspension and reconnection of different interaction shells.

FRUIT allows the use of different (albeit limited) output modalities to be applied to the same application's interface. The different interfaces can be changed dynamically according to the user's requirements. The use of different modalities, however, must be applied across the whole interface and cannot be switched for individual widgets. Equally, the decisions about which modality to use rest entirely with the user and there is no scope for adjusting the way a modality is used according to its availability.

Plasticity

Thevenin *et al.* [109] proposed an architectural framework that would allow user interfaces to be designed so that the usability of an application is preserved regardless of variations in either the system or environment. They define Plasticity as the attribute which would allow developers to specify an interface as usable and subsequently produce it for multiple platforms. The plasticity framework consists of seven components (Figure 4.10).
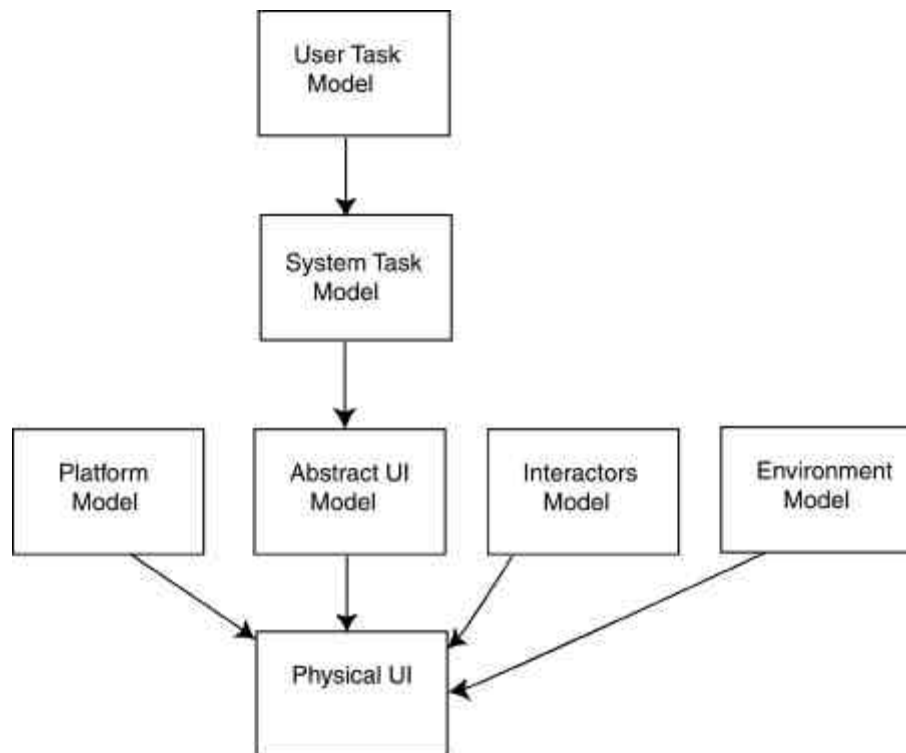
Figure 4.10 – The Plasticity Framework showing the seven components of the framework and their dependencies (from [109]). Arrows denote model dependencies.

The user task model is a formal or semi-formal transcription of the real world activities undertaken by the users. From this, the system model, which defines how these tasks can be accomplished by an application, is derived. From the system task model the abstract user interface model is derived. This describes an abstract rendering of the domain concepts and functions in accordance with the system task model. The abstract user

interface model is interactor independent and acts as the basis for the physical user interface. As well as getting input from the abstract UI, the physical user interface is based upon input from the platform model, the interactors model and the environment model. These three models define how the abstract UI is mapped to the physical UI. The platform model defines the capabilities of a particular platform. These include interaction capabilities such as the input devices (e.g. mouse, keyboard etc) and presentation devices (e.g. screen) available as well as the computational and communication facilities available. The interactors model defines the interactors available for a particular platform. These interactors are defined in terms of the data types they are able to handle as well as their appropriateness and rendering cost. The appropriateness of an interactor is defined by its user-centred effectiveness. In this way, interactors can be selected to optimise the usability of an interface and, should there be no wholly suitable interactors available, the degradation in effectiveness of the interface can be determined. The rendering cost of the interface is expressed in two ways. The cost to the representational system and the cost to the output device. For example, the size, or footprint, of an interactor in a screen could be the rendering cost of that component. The environment model specifies the context of use for the user interface. This is specified it terms of objects, people and events that are peripheral to the user's current task but may have an impact on either the system or the user's behaviour.

The transformation of the abstract UI to a physical user interface is achieved by splitting the abstract UI into different presentation units [111]. The interface is transformed into a hierarchy of these presentation units which model the interface in terms of domain concepts and information containers as well as describing navigational links between the units. Thevenin *et al.* recommend a two step transformation from an abstract UI defined in terms of presentation units to a physical user interface: correspondence matching between presentation units and the interactors available and subsequently a correspondence matching between the interactors and the physical resources provided by the system.

Thevenin *et al.* have provided a framework which allows the creation of multiple interfaces for different platforms with different capabilities based upon a single specification. The different interfaces use different interactors according to the available interactors for a particular platform and may divide the interface into different screens which can be navigated between if insufficient presentational resource is available. The framework has been tested by the implementation of a couple of applications, but further work is still required to develop all the models needed for the framework to be truly effective.

Discussion

In this section, six user interface architectures designed to allow the use of alternative modalities at the human-computer interface have been described. Three of the architectures: Mercator, Homer and Fruit allow interfaces in alternative modalities to be developed whilst two others - Promise and ENO - allow the use of alternative modalities as a complement to a visual interface. Plasticity proposes an architectural framework which allows different user interfaces to be developed which ensure that an application is usable regardless of any variations in the system or environment. The approaches taken by these different architectures will be compared with their advantages and disadvantages discussed.

Homer and Fruit introduce the idea of an abstract widget which is modality independent and from which the interface in alternative modalities can be generated. In both cases, different interfaces are generated which use different presentational modalities as opposed to having one interface which uses multiple presentation modalities. Furthermore, because these systems employ architecture specific components it is not possible to adapt existing interfaces to use these systems. Mercator, on the other hand, takes an existing visual interface and maps the model of that interface into a new model consisting of audio interface components meaning that it is possible to add sound to existing interfaces without any modification. This, however, is at the cost of an increased implementation cost. Promise provides the designer with the ability to specify the modalities to be used to present information and if the modalities should change, how these changes should be performed as the ability of the system to manage such changes automatically is at best limited and at worst would severely impair performance. ENO simply provides a mechanism for generating sounds without specifying how they should be included in the interface.

None of Mercator, Homer or Fruit provide any mechanisms to adjust the modality being used for presentation, or how that modality is used according to its availability. This may well be because they all provide alternative interfaces primarily to suit the user's needs rather than providing complementary feedback in an alternative modality. Promise manages the availability of presentational resources by presenting information in the most suitable resource available. ENO manages the audio hardware required to produce the sounds by caching sounds if it thinks they may be used again and by adjusting the quality of the sounds generated if there are insufficient CPU cycles available to generate the sounds in time. *None of the systems described in this section, however, allow the presentation of the interface to be modified, either in terms of **which** modality is used or **how** a particular modality is used according to the suitability of that modality.*

This thesis describes the design of a toolkit of resource-sensitive, multimodal widgets. Like Promise it enables the presentation of a user interface in the most appropriate modalities although, unlike Promise, the suitability of the modality being used is determined by the context as well as the availability of the modality. Like Mercator, it allows existing applications to be modified rather than requiring they be rebuilt although this is done by modifying the presentation of the existing interaction objects rather than the generation of a new, possibly different, hierarchy of audio interaction objects.

4.5 Summary

In this chapter, several user interface architectures and systems were described. They all have one thing in common which is the division of the user interface into different functional components dependent upon the requirement the architecture or system is designed to meet. The Seeheim model split the user interface into three distinct components to handle domain specific information, the presentation of the system to a user, the application model and a dialogue component to mediate the communication between these two components. Arch extended this model by introducing intermediary components between the dialogue component and the presentation and application components. This acknowledges the requirement of changing either the domain

or presentation components without affecting the rest of the interface. The model-based architectures take these ideas and apply them to an object-oriented organisation which enables the multi-threaded interfaces common today. All the models discussed separate the application component from the presentation component as with Seeheim, but some separate presentation into two separate components for input and output (e.g. MVC) and some have components specifically designated to handle the communication between components (e.g. PAC and ALV).

The systems described is this chapter take the architectures and apply them in a way to suit their own needs. X, for example, can be thought of as a form of Arch with the dialogue component handling network communication and the presentation component handling the mapping between different device dependent toolkits. Garnet and Amulet allow the easy changing of the presentation by ensuring that the model and view components of the MVC architecture they use are not tightly coupled. Similarly, Swing delegates the view portion of its modified MVC architecture to a user interface object provided by the currently installed look and feel allowing the look and feel of its interfaces to be changed. Mercator, Homer and Fruit all allow the development of parallel interfaces in different modalities, primarily aimed at allowing visually impaired users access to graphical user interfaces. Mercator builds a new model of the user interface based upon the existing visual model whereas Homer and Fruit promote the development of two or more interfaces in parallel based upon a modality independent model of the interface. Andalusite allows the addition of sound to an interface by providing hooks to which the sounds can be associated whereas ENO provides a mechanism to generate sounds. Promise allows the designer of an interface for a process control system to use the most appropriate of the available output modalities by specifying the most suitable modalities to use for a piece of information and the rules that should be obeyed if that modality is not available.

All the user interface systems perform some form of management of the presentational resources used. The X-Server manages the physical presentation resource of a device and similarly ENO manages an audio device modifying the quality of the sounds produced if insufficient resource exists to produce them at a higher quality within the time they are required. InterViews delegates the management of the screen real estate to its interactors which have a minimum and maximum size. Andalusite manages the availability of the audio hardware available to play sounds by allowing the designer to specify which sounds should be given preference if there are insufficient resources available. Promise allows the designer to specify how alternative modalities should be used if the preferred modality is not available.

Throughout this chapter, several design guidelines have been extracted. These guidelines describe the advantages of different architectural decompositions provide for the support of multimodality and resource-sensitivity. For easy reference, these guidelines are also listed in Appendix D.

- Separating the application model from the presentation enables the presentation to be changed.
- Describing the required presentation in terms of abstract, implementation-independent objects allows the separation of the presentation from the application model meaning the concrete presentation can be freely changed or supplemented.

- Separating the output from the input makes it easier to modify the presentation of a widget (albeit with the overhead of communication between the input and output components).

- A client server architecture facilitates multimodality by enabling the client to use multiple servers to provide a service at the cost to the client of determining which server to use.

- By providing a component which can intercept and modify requests for presentation global control over the resources used can be obtained.

- Providing hooks onto different aspects of the behaviour of widgets means that it is possible to change the presentation of the widgets without the explicit knowledge of the behaviour.

The user interface toolkit described in this thesis exploits many of the ideas described in this chapter. Like many of the systems described, it separates the presentation of the feedback for the interaction objects from their behaviour allowing different forms of feedback to be used, possibly in parallel. Like Promise it allows the inclusion of rules to specify how different modalities should be used. Furthermore, the use of sensors allow it manage the way presentational resources such as screen space and audio hardware are used. This is extended so that the notion of the suitability of particular modality can be monitored and consequently cause the way the interface is presented to be modified.

Chapter 5: Design and Evaluation of an Auditory Progress Indicator

5.1 Introduction

Progress indicators are a common feature of most modern graphical user interfaces. They allow users to monitor tasks which are not completed instantaneously (or at least not fast enough so that the user does not notice a delay). Such tasks include downloading files from the Internet or installing software from a CD onto a hard-drive. Although much work has been done on the design and evaluation of sonically-enhanced widgets (e.g. [14]) and work has been done on the use of sound to monitor background tasks (e.g. [2, 62]) the design of a sonically-enhanced progress indicator has never been formally evaluated. This is a significant omission as not only are progress indicators commonly used, but they represent a new challenge in this area. All the sonically-enhanced widgets that have been evaluated thus far have been driven by user input whereas progress indicators are driven by a background task, typically independently of user actions. Not only does the evaluation of a sonically-enhanced progress indicator provide an insight into how to sonify background tasks, but it means that the set of sonically-enhanced widgets is complete, paving the way for a toolkit of such widgets to be built.

This chapter begins by describing the two most common forms of progress indicators. It then discusses the importance of progress indicators and describes what information they should provide. A brief review of existing audio progress indicators follows, especially with regard to the information they provide to the user. A design for a sonically-enhanced progress indicator is then discussed. An initial evaluation of this design to determine whether these sounds are effective is described. The design for an additional sound and further evaluation is then discussed. Guidelines on the use of sound and requirements for a toolkit of multimodal widgets are extracted from these experiments and are summarised at the end of this chapter.

5.2 Commonly Used Progress Indicators

Typically, systems will employ one of two types of progress indicator. A 'work in progress' indicator (e.g. Figure 5.1) merely indicates that the task is being processed, but gives no indication of the current state of the task, or of the expected completion time. Figure 5.1 (a) show the progress indicator from Internet Explorer 5™ for Windows. This animated icon changes from the Windows logo to a picture of the Earth and back to

indicate that a file is being downloaded. Figure 5.1 (b) and (c) show two different forms of busy cursor on the Windows NT system. Figure 5.1 (b) shows a hand pointer with a wrist watch indicating that a background task is being processed. The user is able to perform other tasks in the meantime. Figure 5.1 (c) shows an hour glass pointer indicating that a foreground task is being processed and that the user is unable to perform any other tasks until the task being processed is complete.

(a)   (b)   (c)
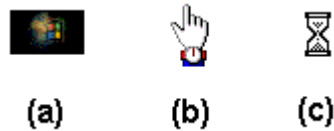
Figure 5.1- Work in Progress Indicators.

Another form of work in progress indicator provides the user with relevant information without giving any indication of the state of the task. An example of such progress indicators are the screens displaying product information which are shown whilst a program is being installed.

The second type of progress indicator, a 'percentage done progress indicator', will afford the user with information about the task being processed. Figure 5.2 shows an example of a percentage done progress indicator on the Windows NT operating system. The bar across the bottom of the window is painted blue in proportion to the percentage of the task completed. Textual information above this bar gives the user more detailed information about the task being processed, in this case a file transfer, whilst in the title bar of the window a summary of the task progress is given. This type of progress indicator, where the amount of the task completed is indicated by a bar filling up has led to the term progress bar becoming common.
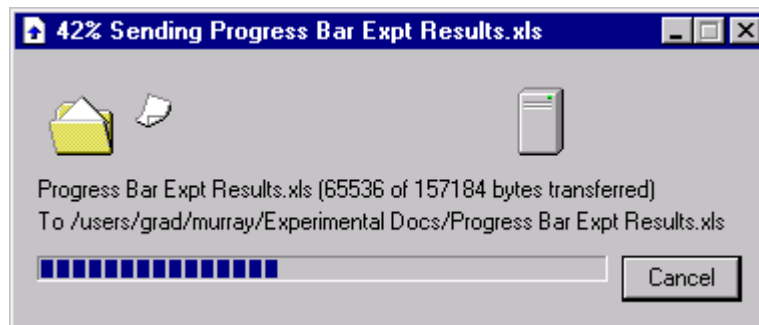
Figure 5.2 - Percentage Done Progress Indicator.

5.2.1 The Importance of Progress Indicators

Myers [81] ran an experiment to determine whether users preferred systems with or without progress indicators, and whether constant or variable response times affected the user's preference. The participants were required to answer eight questions, which typically required approximately fourteen database queries. When the query was made, a progress indicator may or may not have been displayed, and the response time was either constant (10 seconds) or variable (uniformly distributed from 1 second to 17 seconds). The forty-eight participants were randomly assigned to one of four equally sized groups :

- Constant time : "Progress" / "No Progress"
- Variable time : "Progress" / "No Progress"
- No Progress: "Constant Time" / "Variable"
- Progress: "Constant Time" / "Variable"

Each group was further sub-divided so that half the participants got one condition first, the rest, the other condition first. After each condition in the experiment the participants had to answer a questionnaire which was designed to capture their attitude towards the system. This questionnaire asked the participant to fill in 10 scales which represented a variety of measures. The measures, e.g. Anxious – Relaxed, Bored – Excited, had a range of 0 (negative) to 9 (positive).

The results of the experiment showed that the participants preferred the version of the system which employed progress indicators over that which did not. Surprisingly, users did not indicate a significant preference for the system which completed the tasks in constant time, regardless of the presence of a progress indicator or not. Myers reports that this last finding was contrary to previous work by Miller [74], which showed that users preferred constant response time over variable response time, although in this instance the evaluation was done on a system which never utilised progress indicators. Myers speculated that this may have been caused by differences in the degree of variability of the system response time. In the earlier work, Miller used the rate at which characters were displayed on the screen as a measure of system response time variability whilst Myers used the time taken to perform a query (1 to 17 seconds).

Myers suggested several reasons why users prefer systems with progress indicators. Firstly, he suggested that because the progress indicator is not displayed until the task has been accepted by the application, it provides the following information described as being important to users by Miller [75].

- That the request has been registered
- That the request has been accepted
- That an interpretation of the request has been made
- That the system is working on a response to the request

Further, Myers hypothesised that novice users like systems with progress indicators because it gives them confidence that the system has not crashed. Foley [54] stated that novice users are likely to believe that systems should operate quickly, so a seemingly unresponsive system caused by a background task being processed may cause the user to believe that the system has crashed. Although experts will typically have a reasonable idea about the length of time a task should take, Myers felt that they too would benefit from progress indicators because they are more likely to perform multiple tasks at the same time, and without progress indicators it is difficult to keep track of the state of these different tasks. The progress indicators allow the users to plan and monitor their work more efficiently. Finally, Myers felt that progress indicators allowed users to plan activities around the task(s) currently being processed without having to wait for the task to be completed. They could achieve this because the progress indicator allowed the users to choose a secondary task appropriate to the time available before the task currently being processed completed.

5.2.2 Information a Progress Indicator Should Present

Conn [38] expanded the discussion above by introducing the concepts of time affordance, delay, time tolerance window and task hierarchy. Conn describes a time affordance as :

> " … a presentation of the properties of a delay in a task or anticipated event that may be used by an actor (e.g. a user) to determine the need for an interrupting or facilitating action." (p188)

Conn describes eight task properties that a time affordance must provide to be complete :

- Acceptance: What the task is and whether it has been accepted with the input parameters and settings.
- Scope: The overall size of the task and the corresponding time the task is expected to take barring difficulties. (Once Acceptance and Scope are indicated the task may pause for the user to decide whether to initiate.)
- Initiation: How to initiate the task and, once initiated, clear indication that the task has successfully started.
- Progress: After initiation, clear indication of the overall task being carried out, what additional steps (or sub-steps) have been completed within the scope of the overall task, and the rate at which the overall task is approaching completion.
- Heartbeat: Quick visual indication that the task is still "alive" (other indications may be changing too slowly for a short visual check).
- Exception: An indication that a task has encountered errors or circumstances that may require outside (i.e. user) intervention.
- Remainder: Indication of how much of the task remains and/or how much time is left before completion.
- Completion: Clear indication of termination of the task and the status at termination.

The concept of acceptance can be related to Foley's requirement that the user be informed that a request has been registered and accepted, the concept of initiation that an interpretation of the request has been made, and the concept of heartbeat that the system is working on a response. The second concept described is that of delays. A delay in a task is simply the time period between the start of the task and its completion. A static delay is a period during which nothing appears to be happening even if the task is progressing normally. Such a delay occurs if the only feedback given to the user about a task in progress is the change of the cursor from a pointer to (for example) an hour glass cursor which is not animated nor gives any other indication of progress. A dynamic delay is a delay where there is an indication that the task is progressing normally. Such a delay occurs when the user is provided with reliable evidence of the task's progress.

The time tolerance window is the length of time a user (or system) is prepared to wait before deciding something has gone wrong. Several factors determine the length of the time tolerance window:

- Importance: If a task is particularly urgent or sensitive the time tolerance window may shrink dramatically.

- Estimations: If the user's estimations of task length are set appropriately, his/her time tolerance window can be increased.

- Individual Differences: Different users will have different tolerance levels for delays (i.e. some users are more patient).

- Familiarity: Once the user knows the level of dynamic or static delay in a given interface, he or she will know what to expect of the system.

If the system delays are entirely static, the user's time tolerance window will be static. If, however, the delays are dynamic, or contain dynamic elements, the start point for time tolerance window will be reset when new progress information is provided. This progress tolerance window is supplemented by a second time tolerance window: the scope tolerance window. This is a measure of the absolute length of time the user is prepared to wait for the task to complete regardless of indications of progress.

Figure 5.3 describes the relationship between the scope tolerance window and the progress tolerance window.
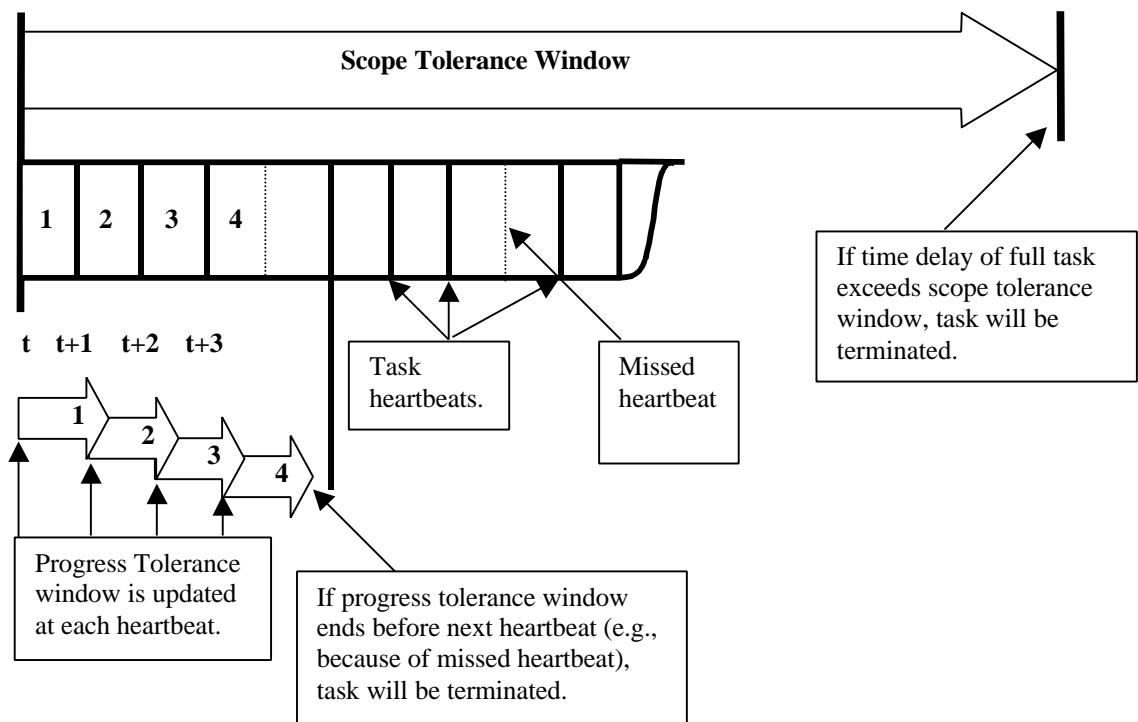
The progress tolerance window is elastic and could have been extended with other indications of progress.

Figure 5.3 – Relationship of progress and scope tolerance windows to dynamic delays which have heartbeats (from [38]).

Conn describes the concept of a task hierarchy as viewing any given task as a hierarchy of task steps, each of which has a corresponding delay. At the top level is the complete task, which after initialisation and before completion is a certain percentage completed. At the next level, the complete task is broken down into

several task steps, or sub-tasks. These sub-tasks can be recursively broken down until the sub-tasks can be completed well within a progress tolerance window. Thus, by presenting the complete task as a series of sub-tasks, each of which can be completed within a progress tolerance window, the user's progress tolerance window will be reset upon the completion of each sub-task allowing the whole task to be completed.

Using these three concepts, Conn derived four principles regarding progress indicators and the systems that support them.

- Every user analyses the task in progress and is prepared to stop the execution of this task if he/she feels something is wrong. The correctness of this response is dependent upon the individual, the nature of the delay and the context. Static delays provide no information and are the source of potentially incorrect responses. Dynamic delays are always preferable, with the quality of the dynamic delay determined by the information available during the delay. Dynamic delays are usually achievable by breaking the task down into suitably sized sub-tasks or by setting the expectations of the user appropriately.
- A system should provide good time affordance, ensuring that any delays are dynamic delays. A good time affordance will provide all eight of the task properties described above (acceptance, scope, initiation, progress, heartbeat, remainder, exception and completion).
- Progress indicators should be able to indicate that "something is happening". It is not necessary for the user to understand the nature of the progress being made, merely for him/her to perceive stages of progress.
- A system should provide a true estimate of time delay. If the information presented cannot be trusted by the user, it can do very little to expand his/her progress tolerance window.

The work described in this chapter concentrates on the second and third principles outlined above. The first principle refers to the breaking up of a task into sub-tasks and the fourth principle refers to the quality of the information presented. Both the breaking up of a task into sub-tasks and the quality of information presented concerns the "engine" providing the information presented to the user, whilst this chapter is concerned with the way in which that information is presented to the user.

Figure 5.4 shows how the progress indicator shown in Figure 5.2 meets the second principle above by providing all eight of the pieces of information required. Acceptance is shown by the name of the source and destination files being copied. Initiation and heartbeat are given by the textual description of how much of the file(s) has been copied. Scope and remainder are given by the size of the file to be copied and the amount copied so far. Progress and completion are given by the graphical feedback in the form of the rectangle filling up. Exceptions are handled at the start of the task, e.g. if there is insufficient disk space at the target destination, or during the task, e.g. if the network connection is lost, by popping up a dialogue box alerting the user to the exception.
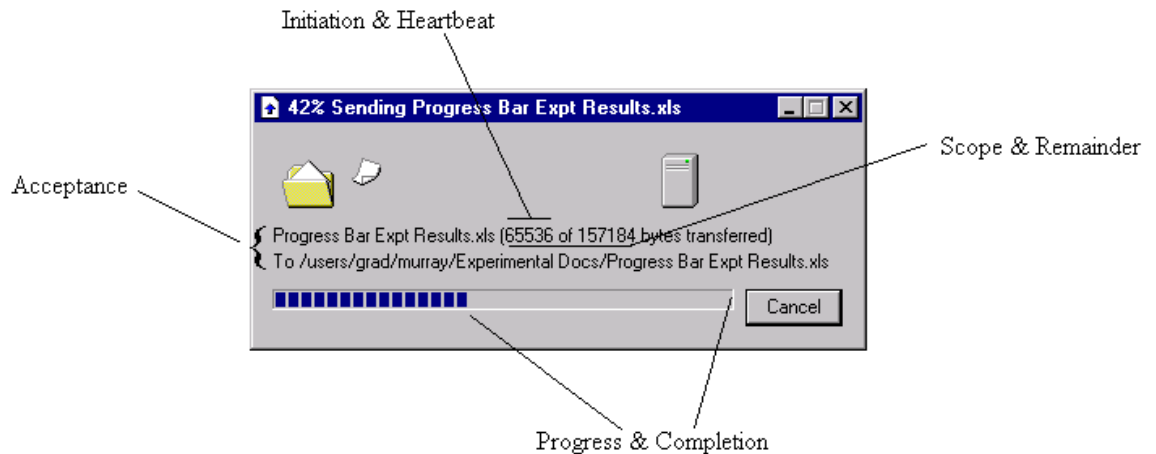
Figure 5.4 – An example of how a graphical progress indicator provides time affordance.

The third principle is met by the changing number of bytes transferred and subsequently the percentage completion figure in the title bar. N.B. The graphic at the top of the window showing a document "flying" from the folder to the hard disk does not indicate that something is happening. This graphic is displayed regardless of the state of the task and merely indicates that the system has not crashed.

Although the progress indicator in Figure 5.4 provides all the information required, this still may not be enough. If the task is going to take a long time, the user may push the progress indicator to the side of the screen, or perhaps even cover it up with another window. If this occurs, the user will not easily be able to monitor the download task, especially if he/she is performing a secondary task. The solution presented in this chapter is to use a different modality, audio, to present the progress information to the user. Using sounds to provide the information allows the user to concentrate his/her visual focus on the task he/she is currently performing. Indeed, Conn hints at such a solution when describing the properties of a task required for a time affordance: "These properties are often visual but may also have other (e.g., audible) components" (p188).

5.3 The Use of Sound To Monitor Background Tasks

The SonicFinder [59] a sonically-enhanced version of the standard Macintosh Finder graphical user interface, uses auditory icons [58] to reinforce the standard graphical feedback (for a full review of both the SonicFinder and auditory icons see Section 3.3.2). As part of the interface, the progress indicator was sonified using the sound of a jug being filled with water in proportion to the percentage completion of the task. As the jug filled (as the task progressed) the sound of the poured water would have a higher pitch. Eventually, when the task was completed, the sound would stop just as the jug sounded full. The sounds acted as a reinforcement of the standard graphical feedback without providing the user with any extra information. This did have the advantage of freeing the user's visual focus, but was not a complete audio

solution. Although there was no formal evaluation of the SonicFinder, users did report finding the SonicFinder to be a useful and appealing interface. The progress indicator did not, however, fulfil the two principles described in the previous section. The sounds used do not provide all the information required to give good time affordance, they only provided four of the required pieces of information about the task. The acceptance and initiation of the task was indicated by the sound starting. Heartbeat was indicated by the continuing sound and completion was indicated by the sound stopping. There was no information given about the scope of the task, the amount of the task remaining or the progress of the task. Some of this information could be gathered across time, for example the rate at which the pitch of the sound changed indicated the rate at which the task was progressing. There was, however, no obvious end point defined so the user was not able to easily determine the remainder of the task.

Another interface which used auditory icons to enhance the usability was the Audio Web system [2]. This system is a version of the Mosaic Web browser which uses sounds to enhance the information given to the users about links and downloads. Users can determine the scope of a file download by moving the cursor over the link to the file. A tick-tock sound was played for a time proportional to the expected transfer time and a second sound was used to indicate the file type. The size of the file was indicated by a piano note. The higher the pitch of the note, the smaller the file. Once the transfer had been initiated, pops and clicks were played indicating data transfer. If an error occurred, a breaking glass sound was played. Again, these sounds were not a complete solution, providing only information about the scope, initiation, heartbeat, exception and completion of the task. Because the sounds indicating data transfer were unchanging regardless of the speed of transfer or amount of the transfer done, no information about the progress or remainder could be gained from the sounds. Again, no formal evaluation of the system was undertaken.

Other work has been done on providing background information to users using audio. The ARKola system [62] used audio to allow users to monitor the status of a virtual drinks factory whilst the Out to Lunch [37] and Audio Aura [89] systems provide background information in audio to provide group information in a disparate group of co-workers. All of these systems are described in more detail in Section 3.4.2.

5.4 Design of Sounds for an Audio Progress Indicator

The progress indicator requires more complex sounds than the other widgets previously evaluated (for a full review of these widgets see Section 3.4.1). This is partly because the other widgets typically change their (audio) feedback as a reaction to a user action whereas a progress indicator's feedback changes as the state of the task being monitored changes independently of any user actions, and partly because a progress indicator is required to provide a lot of information. With this in mind, to determine whether the approach being taken was correct, an initial design for an audio progress indicator which provided only some of the information required by Conn was designed and evaluated. The sounds were designed using earcons [12, 28] because their structured nature allowed the complex information of a progress indicator to be presented concisely.

Brewster [21] also showed that earcons could effectively be played in parallel, allowing multiple pieces of information to be presented concurrently. Four sounds were initially designed to provide some of the information required by Conn. The sounds discussed in this section use the notation described in Section 3.4.1.

5.4.1 End Point Sound

This sound was used to indicate the target point of the task and could be considered analogous to the right hand side of the graphical component (labelled completion) of the progress indicator shown in Figure 5.4. A single bass guitar note (General MIDI instrument number 34) with a fixed pitch of C_5 and duration of 500 msec was played once every second. This sound provided the user with some of the information required to calculate the amount of the task remaining as well as providing heartbeat information. Discrete notes were chosen, as opposed to a continuous note, to minimise the annoyance felt by a user by minimising the number of different sounds played concurrently. A bass instrument was chosen because this sound is the root upon which the progress sound (Section 5.4.2) is based in a similar way to a bass line which is the root to the melody of a tune.

5.4.2 Progress Sound

This sound was used to indicate the percentage of the task completed and could be considered analogous to the right hand side of the "filled" portion of the graphical component of the progress indicator shown in Figure 5.4. A single organ note (General MIDI instrument number 20) was played for 250 msec, once a second immediately after the completion of the end point sound. The pitch of this note was used to indicate the percentage of the task completed. The pitch of the note started at C_3 and as the task progressed, the pitch was lowered in semi-tone steps, in proportion with the percentage of the task completed, until the pitch reached C_4 once the task had completed. The use of semi-tone steps meant the pitch of the sound followed a descent of the chromatic scale[11], giving twelve discrete steps before the pitch reached the final pitch, which had the same relative pitch (C) as the end point sound. By playing this sound immediately after the end point sound it was possible to make a relative judgement about the difference in pitch between the two sounds. As with the end point sound, this sound used discrete notes to minimise annoyance. The use of discrete notes also allows for better discrimination of the changing pitch. Several instruments were tried for this sound, with the most acceptable all sharing the characteristic of being easily envisaged as instruments that could lead a melody. This was important as the information this sound carries is conveyed in the pitch of the sound. It also tied in with the concept of the end point sound being similar to a bass line. The use of an organ was due to personal preference for the sound of the instrument, not because it was any more or less effective than the other candidates.

---

[11] The use of the chromatic scale is clearly based a familiarity with western music and would not necessarily be most appropriate choice in other cultures. The use of such a scale, however, allowed the easy use of MIDI synthesisers to generate the sounds.

5.4.3 Rate of Progress Sound

This sound was used to indicate the current rate at which the task was being completed. For example, if a file was being downloaded, this sound would indicate the rate, or number of bytes per second, at which the file was being downloaded. The sound consisted of a number of short piano notes (General MIDI instrument number 1), with a fixed pitch of C_5 and duration of 80ms, played every second. The number of notes played each second was dependant upon the rate at which the task was progressing. The use of rhythm as the means to convey information to the user meant that this sound did not interfere with the other sounds which used pitch to convey information. The number of notes played each second ranged from three, for the slowest progressing tasks, to twelve for the fastest; giving a range of ten values. The range of notes used started at three because one note would not be discernible and two notes would possibly be masked by the end point and progress sounds as they would use the same rhythm. An upper limit of twelve notes was used as this allowed a duration for the notes which satisfies the minimum duration recommended by Brewster [30]. The duration of the notes used in this sound (0.08 seconds) is marginally lower than the minimum duration recommended by Brewster (0.0825 seconds), although he says that shorter notes can be used if the earcon is very simple, as is the case here where the earcon consists of only one note played repeatedly. Additionally, the instrument used for this note, piano, has a very short attack time meaning it can be perceived in a shorter period of time. Although it was a continuous series of short notes, annoyance was minimised by playing this sound at a lower volume than the other sounds. This had the effect of making the sound fade into the background unless the number of notes played was changed indicating a change in the rate at which the task was progressing as change in rhythm caused by the change in the number of notes played can be easily perceived [30]. A piano was chosen as the instrument for this sound because its attack time is very short meaning the individual notes were still recognisable despite their short duration. The number of notes played per second for a given rate of progress for a task would depend upon the task itself as well as the metric used to measure the rate. For example, a file copy from one local disk to another would be expected to progress at a greater rate of bytes per second than a file download from across the Internet. Therefore, the number of notes played per second for a given rate of bytes per second would be greater in the latter case than in the former.

5.4.4 Completion Sound

This sound was used to indicate task completion to the user. Three chords were played immediately one after another, each chord consisting of two notes of pitch C_5 played for 250ms, with the third being played for 500ms. Two instruments were used for the notes in the chord: bass guitar, the end point sound instrument, and organ, the progress sound instrument. Because the information provided by the sound, that the task has completed, is typically of greater importance to the user than the other pieces of information conveyed by a progress indicator, this sound was played at a slightly higher volume than the other sounds to make it more demanding. Three chords were played in one second to further distinguish it from the progress and end point sounds which played two notes in one second. This is in accordance with Brewster's guidelines which recommend different rhythms as an effective means of distinguishing earcons. The final chord is longer than the other two chords as a means of signalling completion and finishing off the earcon.

A similar style of sound could be used to indicate an exception. A similar, but discordant sound was successfully used to indicate that the user had potentially made an error making a menu selection from a pull down menu [18].

5.4.5 How The Sounds Fit Together

This section describes four types of task and how these would be represented by the sounds described above.

- Task is completed at a steady, fast rate. One second after the task starts, the sounds start. During the first second no sounds are played because the task has only just started so no feedback about its progress can be given. First, a bass note of pitch C_5 (end point sound) is played for 500 msec followed by an organ tone of pitch C_3 (progress sound) and duration 250 msec. Concurrently, ten piano notes with duration 80 msec and pitch C_5 are played evenly throughout the second. This pattern is repeated with the progress sound's pitch gradually decreasing until the task is complete when the completion sound is played. The percentage progress can be ascertained by decreasing pitch of the progress sound. The absolute rate of progress can be ascertained by the number of rate of progress notes played each second (Figure 5.5).
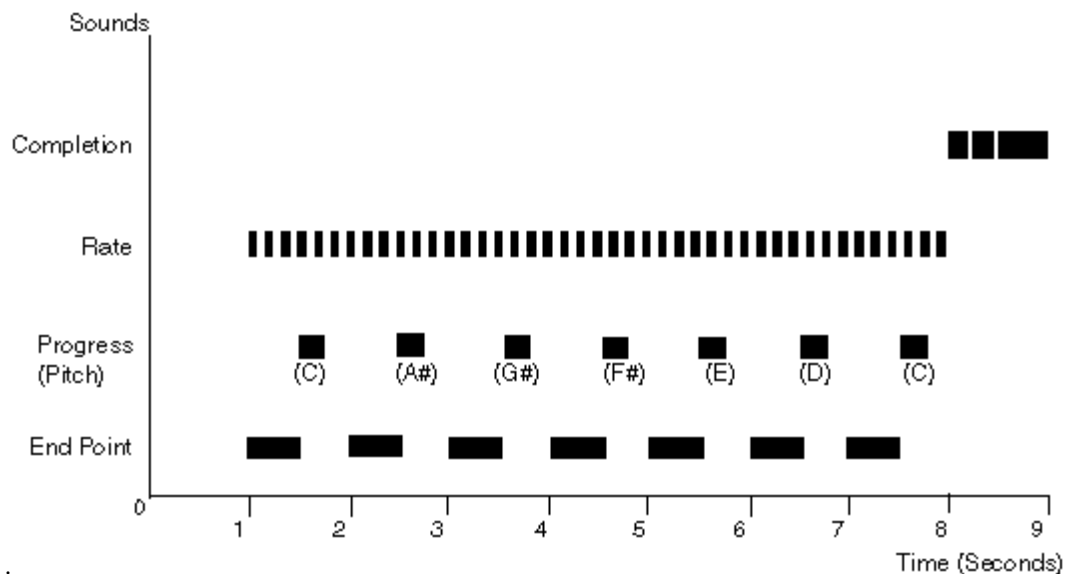
Figure 5.5 – The sounds played for a task that progresses quickly and steadily (the number of rate of progress sounds is indicative only).

- Task is completed at a steady, slow rate. The sounds in this example are very similar to the previous example except only four rate of progress notes are played each second. This would occur if, for example, the task was smaller than the task in the previous example, but because it progresses slower it takes the same time to complete, meaning the progress sounds would decrease in pitch at the same rate, with the difference in absolute rate of progress indicated by the number of rate of progress sounds played each second (Figure 5.6).
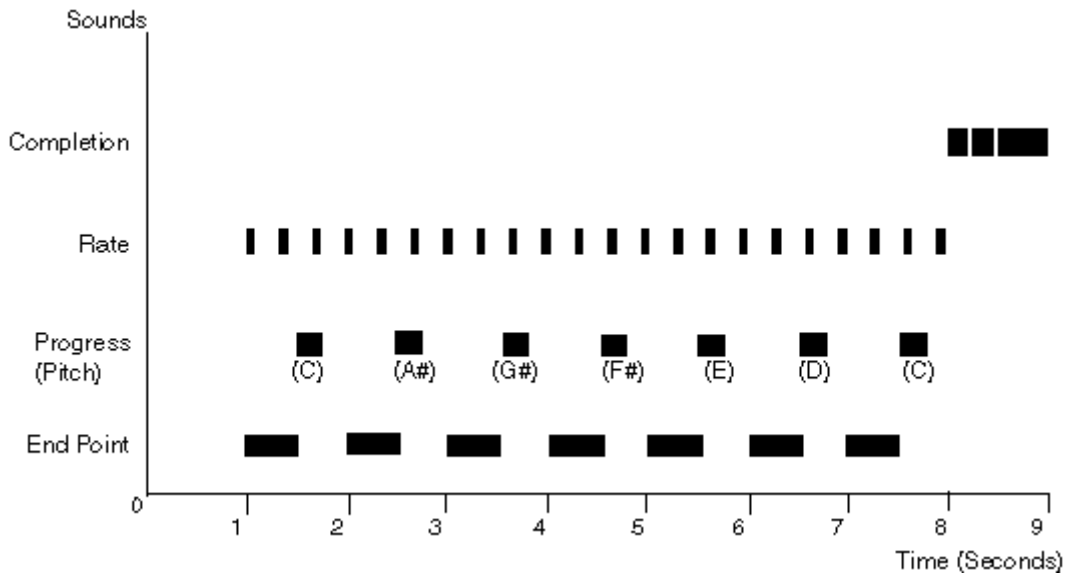
Figure 5.6 - The sounds played for a task that progresses slowly and steadily (the number of rate of progress sounds is indicative only).

- Task starts slowly, but speeds up as time go by. The sounds start as with the previous example. The slow percentage progress is indicated by the slowly decreasing pitch of the progress sound and the slow absolute rate of progress is indicated by the small number of rate of progress notes played each second. As the rate of progress gradually increases, more and more rate of progress notes are played each second and the consequently increased percentage progress is indicated by the more rapidly changing pitch of the progress sound. Eventually, the completion sound is played when the task completes (Figure 5.7).
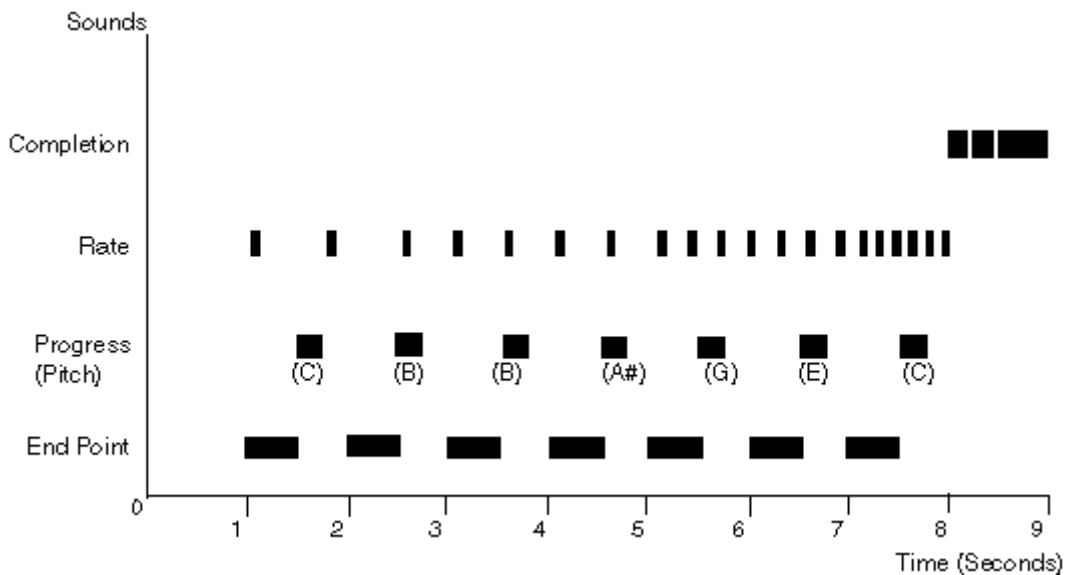
Figure 5.7 -The sounds played for a task that progresses quickly at first but slows down over time (the number of rate of progress sounds is indicative only).

- Task starts quickly and slows down as time goes by. The sounds start as in the first example, with the fast absolute rate of progress indicated by the number of rate of progress notes played each second and fast percentage progress indicated by the large jumps in the pitch of the progress sound. As time goes by, the number of rate of progress sounds played each second decreases and the progress sound changes pitch less frequently, indicating the slowing progress of the task. Eventually, the completion sound is played when the task is completed (Figure 5.8).
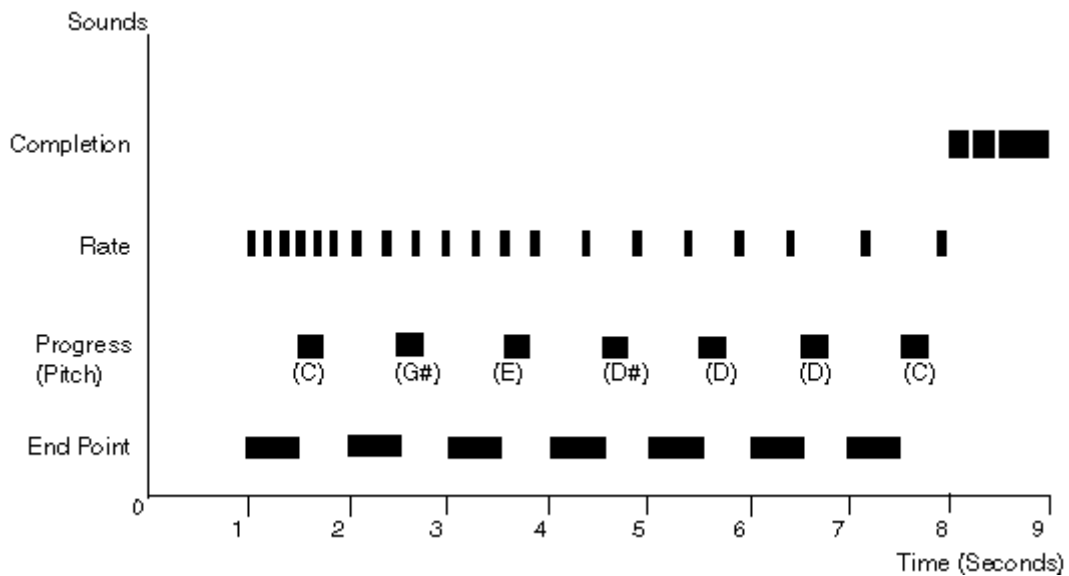
Figure 5.8 -The sounds played for a task that progresses slowly at first but speeds up over time (the number of rate of progress sounds is indicative only).

The sounds described above provide six of the eight pieces of information required by Conn, partially fulfilling his second principle. Initiation is indicated by the sounds starting, progress is indicated by the progress sound, heartbeat is indicated by the continuing sounds, remainder is indicated by the progress, end point and rate of progress sounds and completion is indicated by the completion sound. An exception could be indicated by an exception sound as described in section 5.4.4. The rate of progress sound indicates that something is happening, fulfilling Conn's third principle. The sounds fulfil several of the guidelines described in Section 3.5.1. The sounds combine to form a cohesive ecology of sound with no one component standing out. The rate of progress sound provides the continuous base out of which the end point and progress sounds to emerge. Indeed, the sounds used here introduce a new guideline. The use of instruments and the way they related to the information provided could be considered analogous to the way many pieces of contemporary music are presented. The bass instrument used for the end point sound could be considered as the root upon which the organ of the progress sound was based. Furthermore, the rate of progress sound provided a rhythmic background to these two sounds. Sounds continue to be played even if the background task has stalled, conforming with the guideline that states the absence of a background sound is not necessarily sufficient feedback to alert a user to a problem. In this case, the sounds are of course mapped to a data model, not user interaction conforming to a third guideline and finally, the sounds should not prove to be annoying because they provide relevant information to the user and although this information may well be available visually it will be more accessible using the sounds.

Guideline 8	When structuring complex earcons consider using instruments and rhythm in an analogous way to the structure used in music.

5.5 Experimental Evaluation of Sounds

An experiment was designed to determine the effectiveness of the sounds described in the previous section. To achieve this, the experimental design had to ensure the participants had to look at a progress bar, but at the same time require their visual focus elsewhere. This is a situation which can often occur in real life. If, for example, a user is downloading a file from the Internet he/she may perform a second task whilst waiting for the download to complete. If this task requires the user's visual focus, e.g. word processing, then the information presented visually by a progress indicator may go unnoticed. It was decided that the best way of satisfying these experimental requirements was to simulate the situation described above. The word processing task to be performed would have to engage the participant's visual attention, with the visual progress indicator visible, but not necessarily within the user's visual focus.

5.5.1 Experimental Design

To fulfil these requirements the interface shown in Figure 5.9 was implemented. The text area in the bottom left quarter of the interface was where the participants were to perform the word processing task. This task was to transcribe some poems from hardcopy. Poems were chosen as the text to transcribe because they would engage the participant's attention. The graphical representation of the progress indicator was positioned in the top right hand corner of the interface. This mimics the progress indicator being pushed to the side of the screen to allow a secondary task to be performed. The experiment was started by pressing the Start button located below the graphical progress indicator. This started the download of the first file and put the focus into the text area. The participant could then start entering the poems into the text area. The participant was required to enter as much of the supplied poems as possible in the time taken to download the current file.
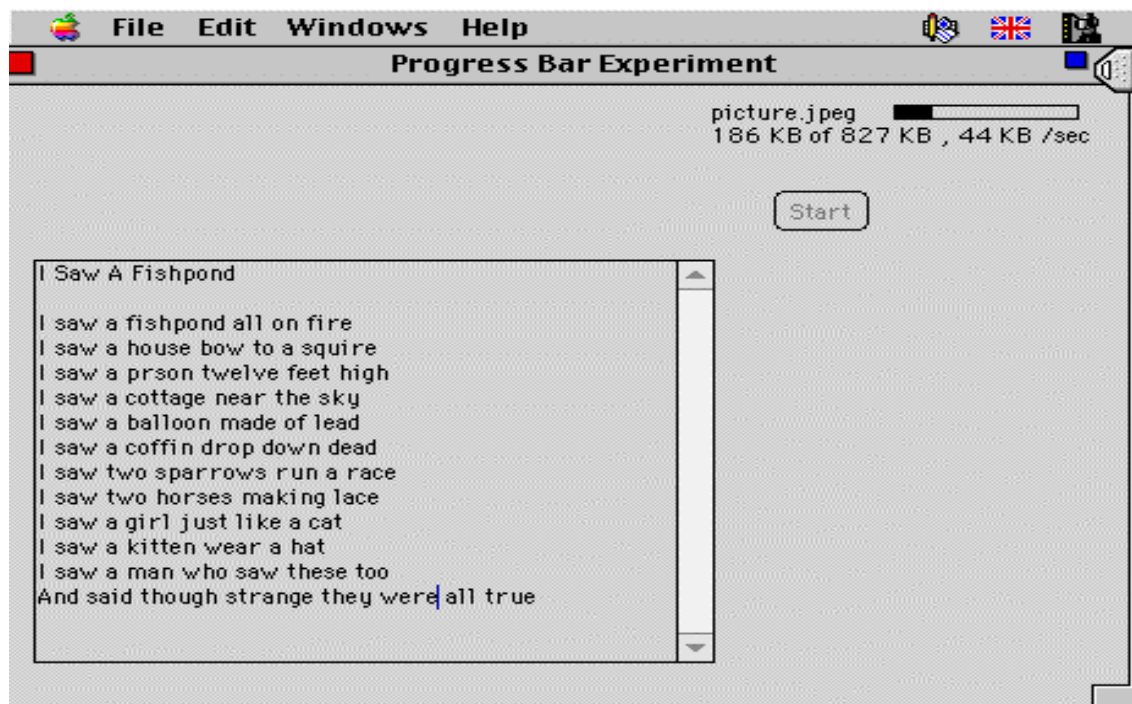
Figure 5.9 - Experimental interface for the evaluation of sonically-enhanced progress indicators. To save space the image has been edited removing much of the space between the text area and the progress indicator.

The experimental design follows that used previously to evaluate the effectiveness of audio-enhanced widgets as described in Section 3.4.1. The format of the experiment is shown in Table 5.1. A two condition, within-subjects design was used, allowing each participant to be used twice and their results compared. The ordering of the conditions, with half the participants doing the audio condition first, and half the visual condition first, eliminated any training effects from the results. After each condition, the participants filled in NASA TLX Workload Ratings [66]. A seventh rating was added to the six standard workload ratings: mental demand, physical demand, time pressure, effort expended, performance level achieved and frustration experienced. The new measure, annoyance, was added because this is a criticism often levelled at audio feedback – that it is annoying due to its omnipresent nature. An eighth factor, overall preference was added to allow the participants to indicate an overall, subjective preference. In addition to the workload data, the participants were video recorded allowing the number of times they looked at the progress bar to be recorded as well as physical data from the experiment such as times and number of words typed. A video camera was placed to the right of the monitor as the participants viewed it, facing the participants. The participants would spend most their time looking at the left of the screen, where the text area was located or to the left of the monitor where the texts they were entering were located. During these activities the participants would be looking away from the camera. When the participants looked at the progress indicator they would look towards the right of the monitor and hence towards the camera enabling the number of glances at the progress indicator to be counted.

Subjects	Condition 1		Condition 2	
Eight Subjects	Sonically-enhanced progress bar. Train & test.	Workload Test	Graphical progress bar. Train & test.	Workload Test
Eight Subjects	Graphical progress bar. Train & test.		Sonically-enhanced progress bar. Train & test.	

Table 5.1 – Format of Experiment

In the visual only condition, the participants were shown four different training downloads, as described in Section 5.4.5, and told how to interpret the information given by the graphical progress indicator. They were then given a short practice session to familiarise themselves with the task. The visual condition then followed. In the audio condition, the graphical feedback was still provided to ensure the evaluation focussed on whether the sounds could effectively enhance the graphical feedback. As before, the participants were presented with four different downloads and told how to interpret the audio feedback. The participants were given the same poems to transcribe in the first condition regardless of whether it was audio or visual condition. A second set of poems was given for the second condition. The "downloads" were read in from data files and were the same for both conditions.

5.5.2 Experimental Hypotheses

There were three main experimental hypotheses :

- The average workload felt by the participants should be reduced in the audio condition as the participants would find the additional information provided by the sounds useful and the overall preference should increase for the auditory progress bar as the sounds help the participant with the task.

- There should be no increase in frustration or annoyance due to the sounds as they provide feedback that is relevant to the participant.

- The time taken to press the "Start" button after a download has completed should be reduced in the audio condition as the completion sound alerts the user more quickly than the visual feedback alone. As a direct consequence of this, the overall time taken for the task should be reduced in the audio condition.

5.5.3 Results

The first two hypotheses listed above are concerned with the subjective workload felt by the participants. The average workload ratings are shown in Figure 5.10 for both the audio and visual conditions.
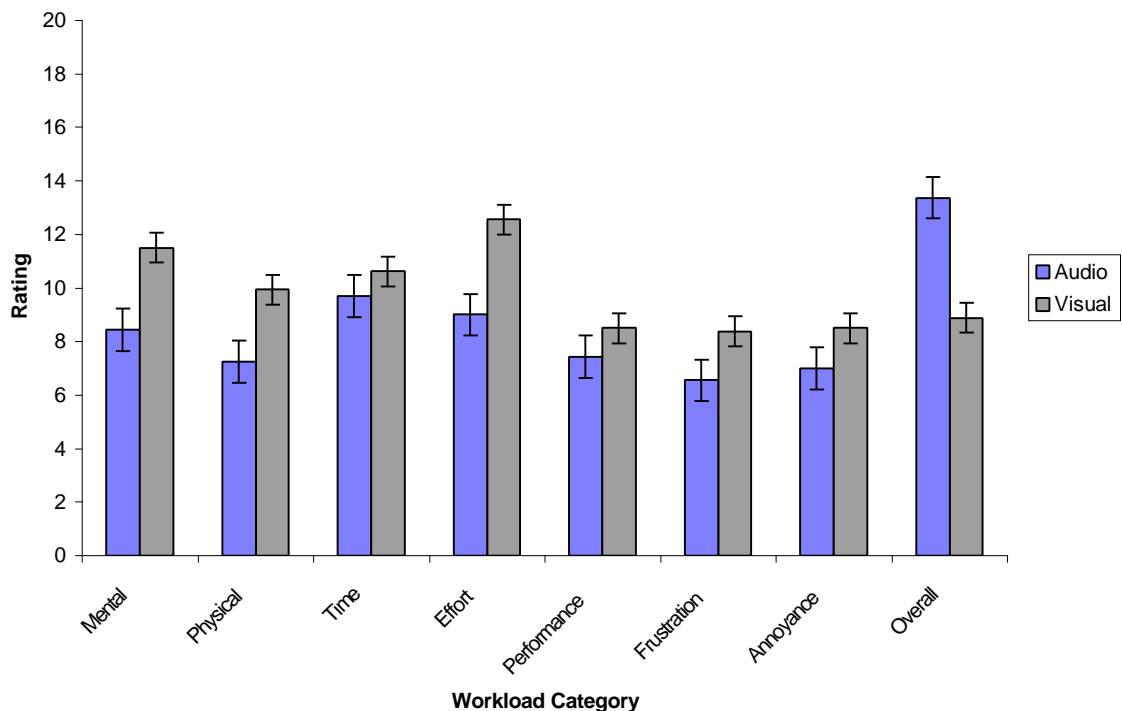
Figure 5.10 - Average TLX workload scores for the audio and visual conditions with standard error shown. The ratings for performance have been subtracted from 20 meaning that for all individual categories a lower score is better. The overall preference has not been adjusted so a high score is better.

The average overall workload, for the six standard measures, was reduced significantly from 10.25 in the visual condition to 8.06 in the audio condition (T_5=4.94, p=0.004), confirming the first hypothesis. The annoyance and frustration felt by the participants did not increase in the audio condition confirming the second hypothesis. In fact, the frustration felt was significantly reduced from 8.375 in the visual condition to 6.5625 in the audio condition (T_{15}=2.17, p=0.046) with the annoyance experienced remaining unchanged (7 in the audio condition and 8.5 in the visual condition, T_{15}=1.48, p=0.159). Of the remaining five workload measures, four showed significant improvement. The mental demand on the participants was reduced from 11.5 in the visual condition to 8.4375 in the audio condition (T_{15}=3.99, p=0.001). The physical demand on

the participants improved from 9.9375 in the visual condition to 7.25 in the audio condition (T_{15}=2.77, p=0.014). The effort expended by the participants was reduced from 12.5625 in the visual condition to 9 in the audio condition (T_{15}=4.66, p=0.0003). The performance level the participants felt they achieved improved from 11.5 in the visual condition to 12.5625 in the audio condition (T_{15}=2.51, p=0.024). Finally, the participants indicated an overall preference in favour of the audio condition of 13.375 compared to 8.875 for the visual condition (T_{15}=4.29, p=0.0006). The sixth workload measure, time pressure, displayed no significant difference (audio=9.6875, visual=10.625, T_{15}=0.97, p=0.346). The raw workload data for this experiment can be found in Appendix C Table 4 and Appendix C Table 5.

The average time taken to press the button after a download had completed (Figure 5.11) was significantly reduced in the audio condition from 5.3 seconds in the visual condition to 2.8 seconds in the audio condition (T_{15}=5.36, p<0.0001). This resulted in a corresponding improvement in the total time taken to complete the task from an average of 727 seconds in the visual condition to 687 seconds in the audio condition (T_{15}=5.36, p<0.0001).
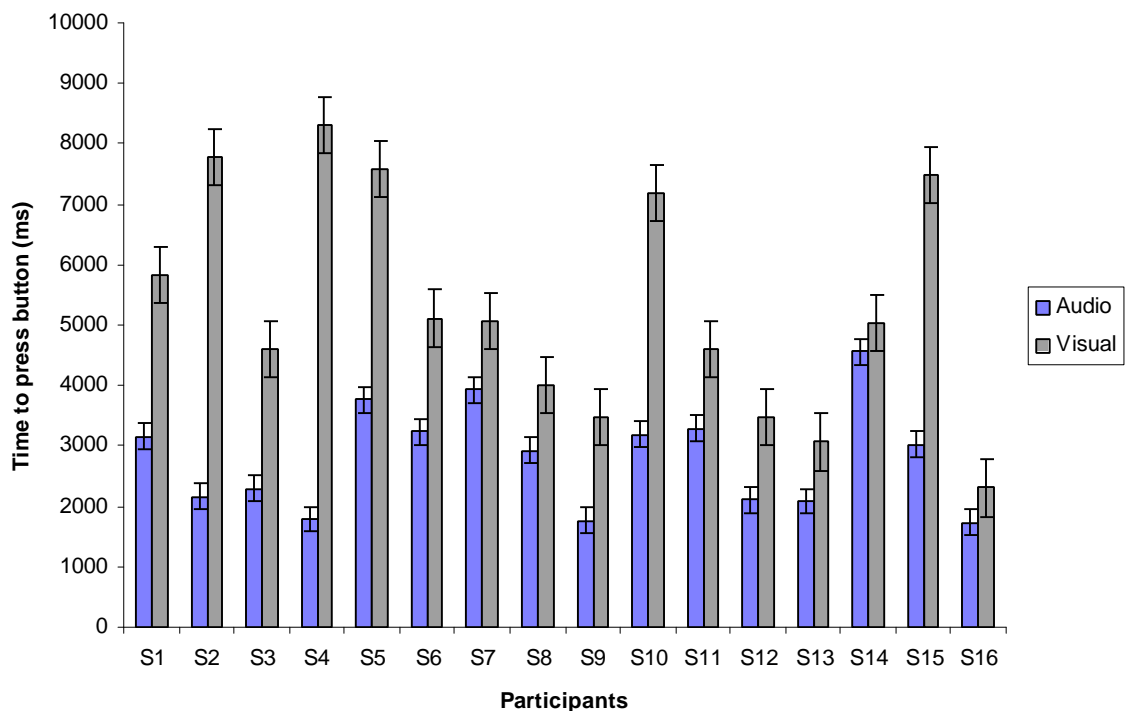
Figure 5.11 – Average time to press the "Start" button after a download had completed with standard error shown.

There was no significant difference in the total number of words typed (audio=343.75, visual=334.19, T_{15}=0.466, p=0.645) or the number of words typed per minute (audio=30.06, visual=27.63, T_{15}=1.35, p=0.196). The participants glanced at the progress indicator significantly fewer times on average in the audio condition (25.0625) compared to the visual condition (83.9375) (T_{15}=10.60, p<0.0001). The raw physical data for this experiment is in Appendix C Table 6 and Appendix C Table 7.

5.5.4 Discussion

The reduction in the average workload experienced by the participants confirms the first hypothesis. The reduction in the frustration felt by the participants combined with the constant annoyance experienced confirm the second hypothesis. The reduction in the average workload experienced by the participants is a reflection of the reduction in six of the eight workload ratings. The mental demand may have been reduced because the sounds were easier to interpret than the visual information. The task required the user to be aware of the completion of a download task, and the completion sound acted as an alarm whereas in the visual condition, the participant had to continuously monitor the download. The physical demand may have been reduced because the participants had to turn their heads to look at the progress indicator more often in the visual condition. The effort expended may have been reduced in the audio condition, subsequently leading to higher performance levels achieved because the participants could concentrate more fully on the typing task without the interference of having to monitor the progress indicator visually. The time pressure remained unchanged because the participants were performing the same task in both conditions, and the addition of sounds did not add any extra pressure or urgency to the task. The reduction in the frustration experienced by the participants in the audio condition may have been because they lost their place in the text less often because they did not have to visually monitor the progress indicator. The annoyance experienced by the participants did not significantly change because the sounds in the audio condition provided the participants with relevant information which they could take advantage of in the task.

The reduction in the time taken to press the button after a download had completed and the subsequent reduction in the time taken to complete the task confirmed the third hypothesis. The participants pressed the button after a download had completed sooner because the completion sound acted as an alarm. Interestingly, the more times a participant looked at the progress indicator in either condition, the sooner he/she pressed the button after completion (Figure 5.12), although the differences are less pronounced in the audio condition. This graph implies the importance of monitoring the task. The more closely it was monitored, the sooner the participant was aware of its completion. It also indicates that the sounds in the audio condition might be effective replacements for the visual feedback as the reduced number of glances in the audio condition contrasted with the improved time to press the button after a completion. There were eleven occasions in the audio condition where the user never looked at the graphical progress indicator. This was possibly because the focus was automatically returned to the text area after the button was pressed. Thus, some users learnt to keep the mouse over the button and pressed the mouse button when the download completed without looking at the graphical feedback. Although there was no significant difference in the number of words typed in either condition and there was a significant reduction in the time taken to complete the task, there was no significant reduction in the number of words typed per minute in the audio condition. This may have been because of a learning effect, as the number of words typed in the second condition was significantly greater compared to the number of words typed in the first condition regardless of whether it was the audio or visual condition (first=303, second=374, T_{15}=7.14, p<0.0001).
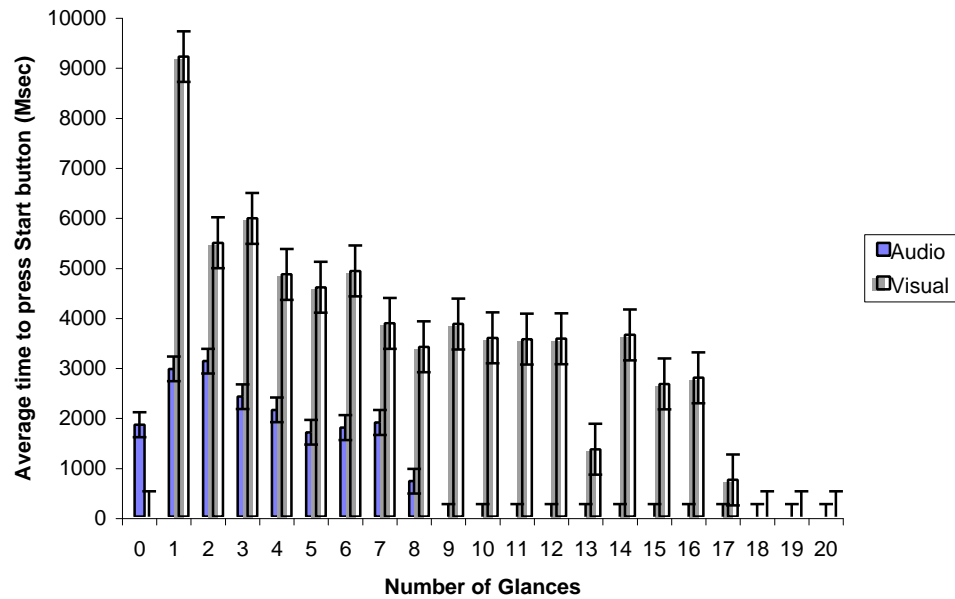
Figure 5.12- The number of glances at the progress indicator during a download plotted against the time taken to press the button after the download completed with standard error shown. For example, in the audio condition the average time taken to press the start button if the participant did not glance at the progress indicator during a download was just under 2000msec. In the visual condition, the participants never pressed the start button without glancing at the start button.

The experimental results showed that the participants found the sounds useful without finding them annoying. There was, however, only anecdotal evidence to show that the sounds were being used for monitoring the background task as the experimental task only required the users to be aware of the download completion. Figure 5.12 indicates that the users were monitoring the download using the sounds and the video evidence seemed to back this up. For the audio, some participants would closely monitor the graphical progress indicator when the download slowed down dramatically whereas normally they would ignore the graphical feedback given during a download. This highlighted the need to perform a second evaluation which required the participants to monitor the background task. The sounds evaluated were also incomplete, not providing all the pieces of information required by Conn. A second evaluation was required to evaluate any new sounds introduced.

5.6 An Additional Sound for an Audio Progress Indicator

Four sounds were described in Section 5.4 which met seven of the eight requirements for time affordance described by Conn. The remaining requirement, scope, required the design of an additional sound.

5.6.1 Scope Sound

This sound needs to fulfil a similar requirement to that of the rate of progress sound: to convey an indication of the magnitude of a task. The rate of progress sound refers to the absolute progress of a task, whilst the scope sound needs to indicate the size of a task. With this in mind, a design similar to the rate of progress sound was used for the scope sound. This sound consisted of a number of short piano notes, with a duration of 80msecs, played in the first second of a download. The number of notes played each second was dependant upon the size of the task. Unlike the rate of progress sound, the pitch of each subsequent note was a semi-tone higher than the previous one. Thus, the scope of the task was given by two cues: the number of notes played and the pitch of the last note played. The pitch of the sounds was increased every note rather than decreased so that for tasks with a large scope, the final note would have a high pitch and therefore be demanding [30]. A similar technique would not be suitable for the rate of progress sound as it is played in parallel with the progress sound which already uses pitch as its primary cue. As with the rate of progress sound, the scope sound can have between three and twelve notes giving a range of ten different sounds that could be played. This sound would be played in the previously silent first second of the download.

5.7 A Second Experimental Evaluation

The experimental evaluation described in Section 5.5 showed that the sounds designed for the audio progress indicator are effective at alerting the participants to the completion of a task. There is, however, only anecdotal evidence that they are effective at allowing the participants to monitor the task. This section describes two experiments designed to evaluate the effectiveness of the sounds at allowing the participants to monitor a background task and the effectiveness of the scope sounds.

It was decided to evaluate the effectiveness of the scope sound in isolation rather than in conjunction with either a distracter task or the other audio progress indicator sounds. This was done because unlike the sounds described in Section 5.4, the scope sound is not designed to allow background monitoring of a task. Nor is it designed to provide the user with information about his/her interaction with the interface like previous audio enhanced widgets [14]. The scope sound is designed purely as an alternative to the visual feedback which is typically used to provide the information to the user. Typically, the user will not be performing a second task. He/she will have initiated the task to which the scope sound is related and will be interested in the information provided. The user may use this information to determine whether it is worthwhile continuing with the task or whether it should be cancelled. Similarly, the scope sound does not provide the user with any information about his/her interaction with the interface, other than that a requested task is being initiated. After a brief explanation about the scope sound and a short training session, the participants were presented with a series of thirty scope sounds which they heard once. The thirty sounds consisted of three occurrences of each of the ten sizes of scope sound in a random order. After each sound, they had to categorise the scope of the task into one of four categories: small, medium, large or very large. Because the participants were asked to give a subjective opinion about the size of the task the scope sound was representing there was no concept of a correct answer. What would be expected, however, was that each participant would answer

consistently. Before each scope sound was played, an organ tone of fixed pitch (C_5) and length (2 seconds) was played. This served as both a warning that a scope sound was about to be played and to prevent consecutive sounds being judged relatively.

To evaluate whether the sounds were effective in allowing users to monitor a background task, the participants were asked to perform a modified version of the task described in Section 5.5.1. As before, the participants were required to type in poems whilst downloading files. This time, however, the participants were required to monitor the progress of the downloads. Previously, the participants had no control over the downloads. They were unable to stop the download and conversely, the downloads never stopped before completion. This time, they were able to stop the download and move onto the next one if they felt it was progressing too slowly and the downloads themselves could stall. This need to monitor the download was reinforced by the task which was to download a fixed amount (in megabytes) of completed files as quickly as possible. This compared with the previous task where the participants could not stop a download but were merely required to recognise when a download had completed. So that they could not try and calculate if a successful download would complete the task the participants were not told the total required, but knew that to complete the task as quickly as possible they would have to monitor the downloads. The monitoring of the progress conflicted with the requirement to type in as many of the poems as possible whilst downloading the files. The experimental interface for this part of the experiment is shown in Figure 5.13. In contrast to the previous experiment, the graphical progress indicator was not shown in the audio condition, meaning the participants would have to rely solely on the audio feedback.
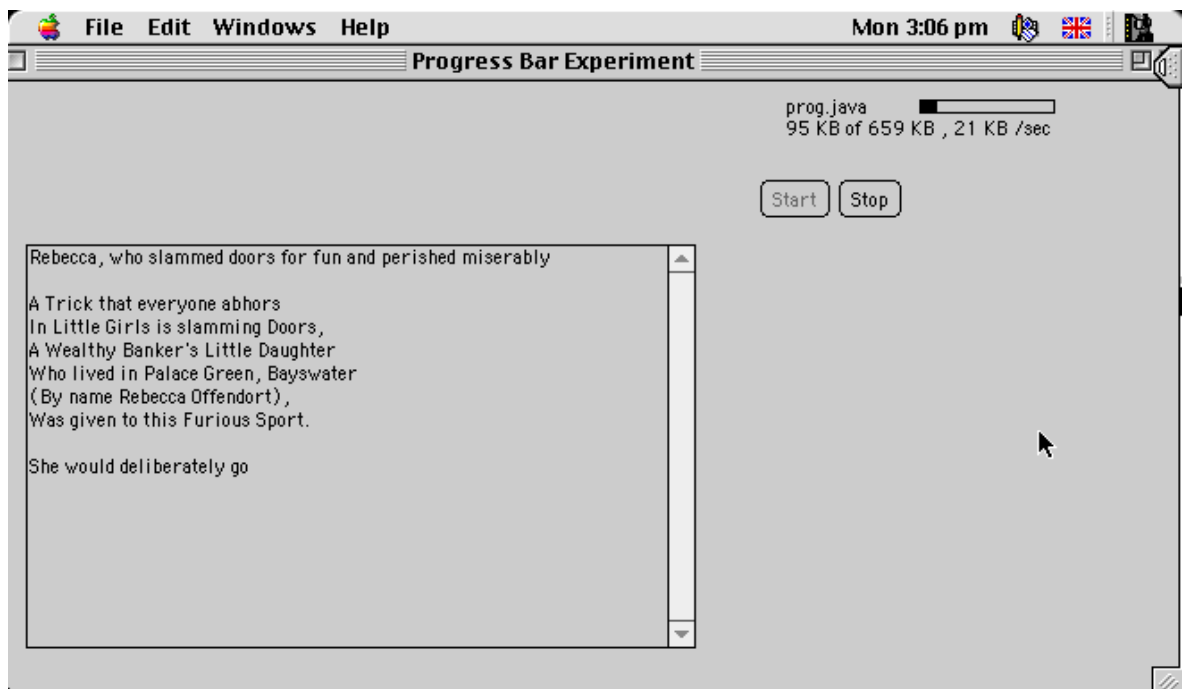
Figure 5.13 – The experimental interface used to evaluate the effectiveness of the sounds at allowing users to monitor a background task's progress. The diagram has been modified to save space; the gap between the text area and the progress indicator being significantly greater than shown.

The two experiments were performed concurrently, with the format shown in Table 5.2.

Subjects		Condition 1		Condition 2	
Seven Subjects	Evaluation of scope sound	Audio only progress indicator. Train & test.	Workload Tests	Standard visual progress indicator. Train & test.	Workload Tests
Seven Subjects		Standard visual progress indicator. Train & test.		Audio only progress indicator. Train & test.	

Table 5.2 – Experimental design for the second round of evaluations. All the participants evaluated the scope sound first followed by a two-condition, within subjects, design to evaluate the participants' ability to monitor a background task with sound.

5.7.1 Hypotheses

There were six main experimental hypotheses, one for the evaluation of the scope sounds and five for evaluation of the effectiveness of the sounds at allowing the participants to monitor a background task.

Scope sound evaluation :

- The participants would be able to correctly categorise the sounds in the scope experiment. The exact categorisation may vary across the participants, but each participant would consistently put each of the sounds into an appropriate category for themselves according to how many notes were played in the scope sound. The categorisation would vary across the participants because the different scope sounds were not specifically identified with any category.

Monitoring evaluation :

- The overall workload felt by the participants would be reduced as the information provided by the audio progress indicator would be easier to digest whilst performing the typing task as it does not interfere with the participant's visual focus. Consequently, the overall preference for the auditory progress indicator should be increased.
- There would be no increase in frustration or annoyance due to the sounds as they provide feedback that is relevant to the participants.
- The time taken to stop a download that has stalled would be decreased as the sounds more easily allow the participants to monitor the state of the downloads.
- The participants should be able to perform at least as well in the audio condition showing that the sounds were equally effective at enabling the participants to monitor the background tasks as the standard visual feedback.
- As with the first experiment, the time taken to start the next download after the previous one had completed should be improved in the audio condition. This will not necessarily lead to a reduction in the total time taken to complete the task, however, as there are more factors in the time taken to complete the task in this experiment: time to stop a stalled download, number of downloads stopped, which downloads are stopped and which are allowed to complete.

5.7.2 Results

Figure 5.14 shows the number of times each different scope sound was categorised as small, medium, large or very large. Because there were no correct answers for the categorisation of the sounds, the results show a bell curve for each of the categories. It is reasonable, therefore, to take the modal selection for each of the categories as a representative value for that category. The modal selections for small, medium, large and very large were three sounds, six sounds, nine sounds and twelve sounds respectively. This even distribution indicates that the participants were largely able to discriminate the sounds, confirming the first experimental hypothesis. The distribution curves for the different categories are also what would be expected. The overlap of the curves can be partially explained by different participants categorising sounds differently.
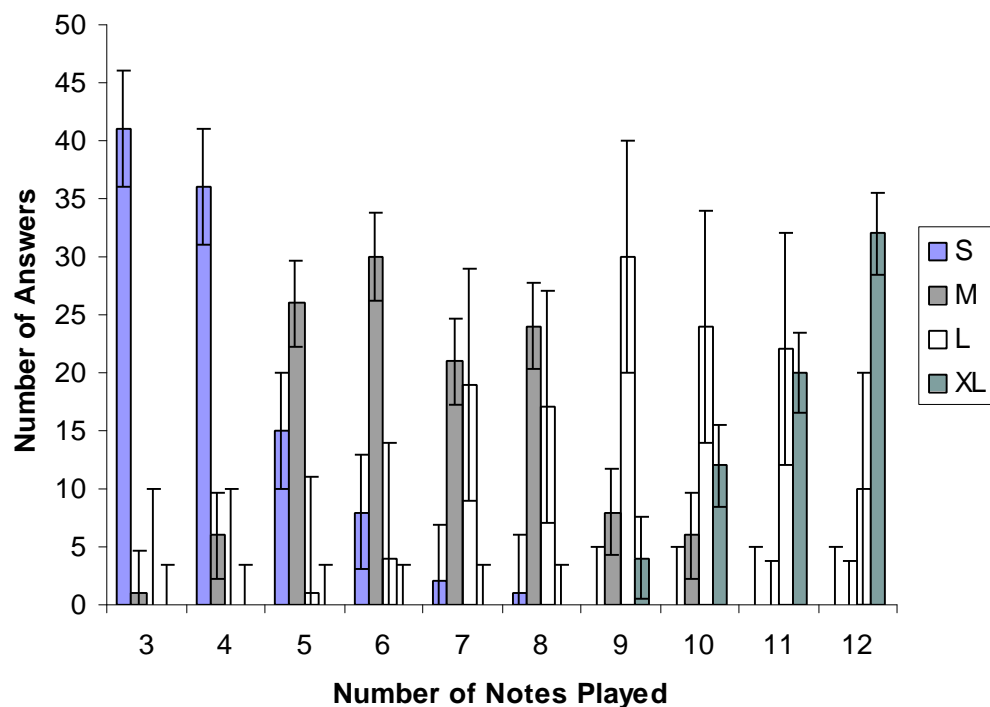
Figure 5.14 – The number of categorisations for different number of notes played in the scope sound with standard error shown.

The participants had to classify thirty sounds each, giving four hundred and twenty classifications across all participants. The total classifications made are shown in Table 5.3. Of these classifications, over half (56.4%) were defined as being correct. A correct categorisation was defined as being within or on the boundary of the appropriate range for that participant (i.e. it was classified as unique or shared). For example, if a participant classified sounds with 3,4,5 and 6 notes as 'small'; sounds with 5,6 and 7 sounds as 'medium' and sounds with 7, 8 and 9 notes as 'large' then sounds with 3 and 4 notes which were classified as small would count as unique. Small and medium sounds with 5 and 6 notes would be counted as overlapped. Medium and large sounds with 7 notes would be counted as shared. 84% of the sounds classified as small were classified correctly showing that the participants were able to distinguish small scope sounds from larger sounds. 54.1% of medium, 40.6 % of large and 50.8 % of very large classifications were classified correctly indicating that the participants had more difficulty distinguishing these sounds. The large classification was particularly poorly identified with only 40.6% of large classifications being correct. This may have been

because the participants found it difficult to distinguish sounds with a lot of notes. When the number of correct categorisations was compared against the number of notes played, the data shown in Table 5.4 was generated.

	Unique	Shared	Overlap	Total	% Correct
Small	69	15	16	100	84
Medium	57	9	56	122	54.1
Large	42	12	79	133	40.6
Very Large	27	6	32	65	50.8
Total	195	42	183	420	56.4

Table 5.3 – Total categorisation of different scope sounds (Correct = not overlapped)

This shows a reduction in the participants' ability to discriminate the sounds as the number of notes is increased from 3 to 9. This reduction is only reversed for sounds with greater than 9 notes, perhaps because these values were at an end point in the data and so could be more easily categorised. These results indicate that whilst the use of frequency can be an effective means of conveying information, it appears that users can only differentiate different numbers of notes effectively when there are fewer than 6 notes played. The raw data for the scope sound evaluation can be found in Appendix C - Table 14.

Guideline 9	When using frequency as a means to convey information avoid using more than 6 notes per second as users can find more than 6 notes difficult to differentiate.

No Sounds	Unique	Shared	Overlap		%Correct	%Incorrect
3	39	0	3		92.86%	7.14%
4	21	18	3		92.86%	7.14%
5	27	3	12		71.43%	28.57%
6	18	3	21		50.00%	50.00%
7	15	0	27		35.71%	64.29%
8	15	0	27		35.71%	64.29%
9	6	6	30		28.57%	71.43%
10	18	0	24		42.86%	57.14%
11	9	9	24		42.86%	57.14%
12	24	3	15		64.29%	35.71%
Total	192	42	186		55.71%	44.29%

Table 5.4 Categorisations of sounds with different number of notes (Correct = not overlapped)

The average overall workload (over the six standard ratings) felt by the participants in the monitoring experiment was 9.23 in the audio condition compared to 11.34 in the visual condition ($T_6 = 4.58$, $p = 0.0059$) confirming the second hypothesis. The frustration experienced by the participants (9 in the audio condition, 10.93 in the visual condition, $T_{13} = 1.42$, $p = 0.18$) was not significantly different in either condition, but the annoyance felt by the participants was significantly reduced from 10.93 in the visual condition to 7.93 in the

audio condition ($T_{13} = 2.18$, p = 0.05). This confirmed the third hypothesis which stated that there would be no significant increase in the frustration or annoyance felt by the participants. In addition to these hypothesised results, the mental demand (14.14 in the audio condition, 11.14 in the visual condition, $T_{13} = 3.93$, p = 0.002), the physical demand (8.42 in the visual condition, 5.71 in the audio condition, $T_{13} = 3.04$, p = 0.009), the effort expended (13.79 in the visual condition, 10.36 in the audio condition, $T_{13} = 5.32$, p < 0.001) and overall preference (8.42 in the visual condition, 5.71 in the audio condition, $T_{13} = 3.89$, p = 0.002) were all significantly improved. The performance level the participants felt they achieved (10 in the visual condition, 11.21 in the audio condition, $T_{13} = 1.23$, p = 0.24) and the time pressure experienced (10.93 in the visual condition, 10.36 in the audio condition, $T_{13} = 0.77$, p = 0.23) did not differ significantly. The raw workload data can be found in Appendix C Table 10 and Appendix C Table 11.

The remaining hypotheses were concerned with the timing data. The time taken to stop a download that had stalled was not significantly improved contrary to the third hypothesis (6.3 seconds in the audio condition, 8.6 seconds in the visual condition, $T_{13} = 1.94$, p = 0.07). The participants' overall performance in the task was not significantly different in either condition confirming the fourth hypothesis. The time taken to complete the task (762 seconds in the audio condition, 772 seconds in the visual condition, $T_{13} = 0.53$, p = 0.61), the number of words typed (371 in the audio condition, 347 in the visual condition, $T_{13} = 0.622$, p = 0.544) and the bytes downloaded per second (28736 in the audio condition, 29197 in the visual condition, $T_{13} = 0.99$, p = 0.34) did not change significantly.

The results of the experiment described in Section 5.5 were confirmed with the time taken to recognise that a download had completed significantly improved in the audio condition compared to the visual condition (2.42 seconds in the audio condition, 3.63 seconds in the visual condition, $T_{13} = 5.103$, p = 0.0002). The raw physical data for the experiment can be found in Appendix C Table 12 and Appendix C Table 13.

5.7.3 Summary

The first part of the experiment evaluated the new scope sound described in Section 5.6. The participants were asked to place different sounds into one of four different categories: small, medium, large or very large. The participants were able, in general, to categorise the sounds in a consistent manner with 56.4% of sounds appropriately categorised. The sounds which the participants found easiest to distinguish were those with fewest notes. 84% of sounds categorised as small were correctly categorised compared to only 54.1% of those categorised as medium, 40.6% of those categorised as large, and 50.8% of those categorised as very large. This implies that the participants found it easier to discriminate sounds with fewer notes. Additionally, because the small category was an endpoint, there was less opportunity to make an error. This was confirmed by the analysis of the categorisation of the individual sounds. Almost 93% of three and four note sounds were correctly categorised. Half of the sounds with six notes were correctly categorised with the rate of correct categorisation dropping off for sounds with greater than six notes. This trend is slightly reversed for sounds with ten or more sounds. This is most likely to have occurred because this was an endpoint and thus the opportunity for making a mis-categorisation was reduced. Despite these problems, the sounds did appear to be useful for giving a rough guide to the size of the task. This can be seen in Figure 5.14 which shows a standard distribution curve for each of the different categories.

The second part of the experiment evaluated the usefulness of the sounds at allowing the participants to monitor the background task. The experiment simulated a real life task, to download a file in the background whilst performing a second, foreground task. In the first experiment, the user was not under any pressure to monitor the progress of the downloads as they always completed. In this experiment, however, the user had to monitor the progress of the tasks because the downloads could stall. Additionally, because the task required the participants to download a fixed amount of files (in megabytes) as quickly as possible, if a download was progressing too slowly the participant could stop it and move onto the next one. The participants were able to complete the task just as quickly in the audio only condition as they were in the visual only condition, showing that the audio feedback provided was just as effective as the standard visual feedback. It was hypothesised that the participants would detect both the completion and stalling of downloads faster in the audio condition. This was the case for completed downloads, confirming the results of the first experiment, but the time taken to detect a stalled download did not improve significantly. This may have been because the completion sound acted as an alarm to the participants, but there was no explicit stalled sound because it was not treated as a separate state, but rather as a download which was progressing very slowly. The participants did seem to be aware that the download had stalled, but because the feedback indicated slow progress, they often waited to see if it improved. Conversely, in the visual condition the participants received compelling feedback in the form of the text indicating current transfer rate (in this case 0 bytes/sec) meaning there was no such uncertainty although the download could have, theoretically, sped up again. This is an indication that, whilst the audio feedback was effective, there were instances where the reinforcement of the visual feedback would have been useful. Another example of where such a reinforcement would have been beneficial was the size of the current task. Although the participants were given an indication of the scope of the task sonically at the start of each download, they would sometimes forget the size of the download and consequently if it started to slow down could not make a suitably informed decision about whether to stop the download or not. In the visual condition this was not such an issue as the size of the download was available textually throughout the download. This information could be given visually as was the case here or, if suitable, another modality such as speech could be used.

To try and gauge their subjective preferences the participants in the experiment were given workload ratings to complete. The average workload was reduced in the audio condition, possibly because the audio feedback allowed the participants to monitor the download without disturbing their visual focus which was required for the typing task. The frustration experienced did not change significantly, but the annoyance experienced was significantly improved in the audio condition. This may have been because the participants could keep track of where they were in the poem more easily in the audio condition as they did not require to visually monitor the progress indicator and consequently lose their place in the poem they were transcribing. Their frustration remained unchanged because in both conditions they were provided with all the information required to complete the task. The reduction in mental demand on the participants in the audio condition may be explained by a reduction in the cognitive load in the audio condition. In the audio condition a general indication of the state of the progress was easily ascertained from the sounds. In the visual condition, more detailed information could be ascertained from the graphical feedback, but the percentage of the task completed would have to be calculated, either from the percentage of the graphical bar filled or by calculating the percentage completed from the textual description of the size of the file being downloaded

and the amount currently downloaded. Although it seems reasonable to assume that the participants could get a good approximation of the state of the download from the graphical bar, perhaps they tried to ascertain a more exact idea of the state of the download by using the textual information. The reduction in the physical demand in the audio condition will have occurred because the participants would not continually have to turn to look at the progress indicator. The reduction in effort expended in the audio condition may be due to the reduction in the physical and mental demands. Together with the fact that the time taken to complete the task remained constant, these results indicate that the participants could use the audio progress indicator just as effectively as the standard graphical progress indicator, but with less effort.

5.8 Discussion

The two experiments described in this chapter show that the outlined design for an audio progress indicator is effective in allowing users to monitor a background task. By using the audio channel, the participants' visual focus was freed allowing a second task to be performed whilst waiting for the background task to be completed. However, there still remain some issues to be resolved with the use of audio feedback for monitoring background tasks.

5.8.1 Multiple Background Tasks

The experiments described above showed that users could monitor single tasks using audio feedback. There is no evidence, however, that this technique can be simply scaled up to allow the monitoring of multiple tasks. If this were tried, the sounds from the different progress indicators would interfere. This is an important area to be considered as the computers are capable of performing multiple tasks simultaneously and as Myers noted [81]:

> "… most people find it difficult to keep track of what is happening when multi-processing. Progress indicators help users plan and monitor their various tasks so their time can be more effectively used."

The problem arises because the information presented by the progress indicator is persistent whereas the information presented by other interface interaction objects tends to be discrete and typically in response to a user action. For example, the sonically-enhanced button evaluated by Brewster [20] produced audio feedback only when the user moved the mouse over the graphical button or when the user subsequently pressed the button. This state is user driven, and can only apply to one button at any given time thus the possibility of clashing audio feedback does not arise. This implies the need for some sort of mechanism to control the sounds generated by multiple audio progress indicators on the same machine. There are several possible approaches such a mechanism could take.

Requirement 5	The toolkit must be able to manage any interference between different pieces of feedback.

One possibility would be to use different timbres for different progress indicators. Whilst this would aid the differentiation of the different sounds, there is still a risk that they could still be confused, especially if there were several background tasks being monitored. Further, it would be necessary to assign different progress indicators different timbres. This could lead to the same task being assigned different timbres each time it is run, potentially leading to confusion.

A second solution would be to spatialise the sounds so that they could be perceived as coming from a different source. This would allow the same sounds to be used consistently for all progress indicators allowing for a more consistent use of sound across several progress indicators. This is the approach taken in further work at Glasgow by Walker *et al.* [112] in their design for an audio progress indicator for mobile devices with limited screen space. The design for the progress indicator uses the standard graphical progress bar as its basis, transforming its visual feedback into audio and its linear bar shape into a circular orbit around the user's head. Two sounds are used to indicate task progress. The first, a "lub" sound was played in front of the user. The second, a "dub" sound was spatialised onto the circular orbit around the user[12]. The percentage progress of the task was indicated by the angular position of the second sound. 0% completion was directly in front of the user, 25% to the users right, 50% directly behind the user, 75% completion to the users left and 100% directly in front of the user again. The absolute rate of progress of the task was indicated by the time delay between the two sounds. The longer the delay, the slower the rate of transfer, with an infinite delay indicating a stall. Completion was indicated by the two sounds sharing the same spatial location directly in front of the user.

An experimental evaluation of the sounds was undertaken, with the 16 participants performing a similar task to that described in Sections 5.5.1 and 5.7. The participants had to transcribe as many poems as they could in the time it took to download data from twenty files. To force the users to monitor the downloads, they were periodically stopped and given the option of restarting or moving on to the next one. This decision was made based upon the minimum percentage that had to be downloaded for each task. In the audio condition the users were given no visual feedback about the state of the download, and in the visual condition, all visual feedback was removed when a download was interrupted meaning the participants had to monitor the downloads throughout their progress. Analysis of the experimental results showed that the participants could monitor the downloads more effectively in the audio condition than in the visual condition; making more correct decisions regarding the continuing or cancelling of a stalled transfer.

Although the experimental evaluation only used one progress indicator at any one time, there remains the potential to use multiple indicators concurrently. This is the primary focus for Walker *et al.*'s future research. Another area in which the spatialised, audio progress indicator could be improved is in the quality of the sounds that it uses. Due to technical limitations it was not possible to generate high quality sounds. An improved version uses sounds that have been designed to be aesthetically appealing. Additionally, the spatialisation of the sounds has been linked to a head tracker meaning the spatialisation of the sounds is adjusted relative to the users head position. To allow the presentation of multiple progress indicators, the way

---

[12] The descriptors "lub" and "dub" are onomatopoeic.

the sounds are presented is changed, with an audio cursor sweeping round the user's head every second. The sounds for the different progress indicators are then played at the appropriate spatial location as the audio cursor sweeps round the circular orbit. A future evaluation of this design is planned.

5.8.2 Background Tasks That Take A Long Time

Each download task in the two experiments described above typically lasted less than thirty seconds. In a real life situation, a background task may take several hours or even days. In such an instance, the audio feedback could prove to be annoying, with a continuous sound playing giving very little new information to the user. One possible solution to this would be to fade out all the sounds if the task is progressing steadily. If this is the case, the sounds would be providing no new information to the user other than state of the task is changing at a steady rate. There is a danger, however, that if the user is not present for or unaware of the sounds gradually fading away, the lack of sounds could be misconstrued as a missing heartbeat sound. One solution to this would be to reduce the number of sounds played rather than removing all the sounds. A sound could be played indicating heartbeat with all other sounds being removed. If there is a change in the way the task is processing, its rate of progress decreases for example, the other appropriate sounds could be faded back in providing the user with the relevant information. This solution still has the problem, however, that not all the information required by the user is available at all times. It would be better, perhaps, to use all the sounds but to have them played in the background. This would be an audio analogue to pushing a graphical progress bar to one side of the screen, but with the advantage that the user is still able to attend to it. This could be done by a simple reduction in the volume of the sounds without removing them altogether, or perhaps by spatialising the sounds "away" from the user.

Requirement 6	The toolkit must be able to monitor the presentation of a widget over time and, if appropriate, adjust the presentation.

5.9 Guidelines and Requirements

This chapter has discussed the design and evaluation of a sonically-enhanced progress indicator. During this discussion some guidelines about the use of audio at the human-computer interface and requirements for a toolkit of multimodal widgets were extracted. This section summarises these guidelines and requirements

5.9.1 Guidelines on the Use of Audio

The guidelines described in this section complement the guidelines presented in Section 3.5.1.

- Using frequency, i.e. number of notes played per second, to convey information only works for a limited range of frequencies. In the evaluation of the scope sound (Section 5.7) the participants were quite efficient at differentiating sounds which have a small number of notes but were much less efficient at differentiating sounds with a lot of notes. An upper limit for effective discrimination was shown to be six notes per second.

- The use of different instruments and rhythm in complex earcons can be structured in a similar way that they are structured in music to provide effective feedback to users.

5.9.2 Requirements for the Toolkit

The requirements in this section complement the requirements presented in Section 3.5.1.

- The toolkit must be able to manage any interference caused by different pieces of feedback.
- The toolkit must be able to monitor the presentation of a widget over time and adjust it appropriately.

5.10 Conclusions

Myers showed that users found progress indicators useful as they indicate to novice users that the system has not crashed and provide expert users with information about the ongoing task that may allow them to organise other activities around the background task. Conn described the information that a progress indicator needs to present to provide users with good time affordance. Many graphical progress indicators provide all the required information, but these indicators are often pushed to the side of the screen or even occluded as the user performs a second task whilst the first is performed in the background. This means that the information provided by the progress indicator cannot be perceived by the user.

The solution to this problem, as suggested in this chapter, is to provide information about an ongoing background task using the audio channel, freeing the visual channel for any foreground task the user may wish to perform. Four sounds were initially designed providing users with some of the required pieces of information. An experimental evaluation was then under taken which anecdotally showed some participants were able monitor a background task using the audio feedback whilst performing a visually intensive foreground task. The participants were required to download some files as quickly as they could whilst typing in as many texts as possible. This forced the participants to concentrate their visual focus on a primary task whilst trying to monitor a second, background task. It was also shown that the participants were better able to detect the completion of a background task using the audio feedback. A second evaluation was carried to determine conclusively whether the participants were able to monitor background tasks using the audio feedback. A new sound, describing the scope of the task about to be undertaken was also evaluated. This evaluation showed that the users could effectively monitor a background task whilst performing a second, foreground task, using only audio feedback.

By performing this design and evaluation of a sonically-enhanced progress indicator the set of evaluated sonically-enhanced widgets has been completed, opening the way for the design and implementation of the toolkit of multimodal widgets described in subsequent chapters. Furthermore, by performing this work some guidelines for the use of sound at the human-computer interface and some requirements for a toolkit of multimodal widgets have been extracted to complement those presented in Section 3.5.1.

Chapter 6: Design of a Toolkit of Multimodal Widgets

6.1 Introduction

In the previous chapter, the design and evaluation of a sonically-enhanced progress indicator was described. There is now a wide range of sonically-enhanced interaction objects, or widgets, that have been shown to be effective in enhancing the usability of graphical user interfaces (see Section 3.4.1 for a full review of this work). Despite this body of work, audio-enhanced interfaces are still not widely available even though most new computers are capable of supporting such interfaces. One reason for this lack of availability is that it is relatively easy to build an interface using the graphical widgets supplied by a toolkit or interface builder[13] - no knowledge of the techniques required to design and build such graphical components is required as this knowledge is encapsulated within the widget by the widget designer. In Section 3.4.1 the knowledge that has been gained about the design of sonically-enhanced widgets was described. However, since no toolkit exists which encapsulates this knowledge, to build an audio-enhanced interface is much harder than a visual only interface. A second reason is that although growing, the body of knowledge about sonically-enhanced user interfaces is still limited. This may well be for the reason given above, *viz.*, it is hard to evaluate the design of audio widgets because they are hard to build.

This chapter describes the design of a toolkit of multimodal widgets. It starts by examining the requirements extracted from the work described in previous chapters. These requirements are discussed and further requirements are identified. A design for the toolkit which meets these requirements is then presented. The remainder of the chapter discusses this design, detailing how it meets the requirements outlined previously - in particular with reference to the design guidelines extracted from the work described in Chapter 4.

6.2 Requirements

In this section the requirements for a toolkit of multimodal widgets are discussed. All the requirements for the toolkit are located in Appendix D for easy reference. In Chapter 3 and Chapter 5 six requirements for the toolkit were extracted:

---

[13] Although it is easy to build such an interface this does not guarantee a high quality interface.

1. The full behaviour of the toolkit's widgets should be exposed allowing suitable forms of presentation to be generated.

2. The toolkit should be able to determine if it is not possible to play all the requested sounds and, if so, modify the feedback appropriately.

3. It must be possible to group widgets together to allow their feedback to be co-ordinated appropriately.

4. A toolkit of multimodal widgets must allow users to control the form of presentation used for its widgets.

5. The toolkit must be able to manage any interference between different pieces of feedback.

6. The toolkit must be able to monitor the presentation of a widget over time and, if appropriate, adjust the presentation.

The following sections discuss these requirements and elicits further requirements for the toolkit. Section 6.2.6 discusses a further requirement: that there should not be a large learning cost for engineers to include the toolkit's widgets into their applications. Section 6.2.7 draws all the requirements together.

6.2.1 Additional Modalities

Perhaps the most obvious way to create a toolkit of audio-enhanced widgets would be to take an existing toolkit of graphical widgets and add appropriate audio feedback. This, however, is not as easy as it might seem. When designing a sonically-enhanced pull down menu [18, 25] (described fully in Section 3.4.1) it was shown that the designers of graphical toolkits make assumptions about the way the toolkit is going to be used. These typically assume the users of such toolkits, i.e. the programmers building the user interface, are interested in a limited subset of events that can occur to a widget. The design and evaluation of a sonically-enhanced pull down menu was undertaken in Object Pascal on a Macintosh. The experimental evaluation was built using Macintosh Toolkit widgets. The menu widget had a very simple API. When the user pressed the mouse button in the menu bar, the program made a call to the `menuSelect` procedure. This procedure handled all the user interaction with the menu until a selection was made or the user finished the interaction with the menu. The only data supplied to the application was the integer value returned by the `menuSelect` procedure which indicated whether or not a selection was made and, if so, which menu item was selected. Information about the menu item the mouse was over, or indeed which menu the user had entered, was hidden within the `menuSelect` procedure. To enhance the pull down menu with audio it was necessary to write a new procedure which reproduced all the work previously encapsulated within the `menuSelect` procedure as well as including the addition of audio feedback. As well as showing the difficulties in adapting existing toolkits, this highlights the difficulties that exist in trying to add new modalities of feedback to the human-computer interface.

Another solution would be to build a new toolkit which encapsulates audio feedback in addition to any graphical feedback. Whilst this would meet the needs of an interface builder wishing to include audio feedback, this would not assist designers wishing to evaluate new types of feedback. Equally, it restricts the addition of further modalities. Haptic feedback devices are becoming more readily available and there is a growing body of work on the haptic-enhancement of user interfaces (e.g. [93]). It is therefore important that it should be simple to add new designs of feedback to a toolkit's widgets, either replacing any existing

designs or adding designs in a new modality, meaning that the behaviour of the widgets must be fully exposed and the presentation of the widgets not encapsulated within the widget. This confirms Requirement 1.

6.2.2 Controlling Intra-Modality Clashes of Feedback

One issue that arises when audio is added to an interface is how the audio feedback from different widgets combine with each other. For example, in Section 5.8.1 it was described how the audio feedback from two or more different progress indicators could interfere with each other. This could be the case for any two pieces of persistent audio feedback. The problem is not, however, limited to widgets that provide persistent audio feedback. If, for example, a user selects a button which presents a transient piece of audio feedback, the button's feedback may interfere with another, persistent piece of feedback (e.g. an audio progress indicator) that is being played at the same time. It is even possible to imagine, in a collaborative environment for example, a scenario where two pieces of transient feedback could interfere with each other if the multiple users perform actions that simultaneously generate audio feedback that interfere with each other.

The potential for a clash also exists using visual feedback but because it is possible to allocate individual widgets a specific area in the visual output space, such clashes can easily be handled. Typically, this is done by a windowing manager which controls the allocation of the limited visual resource in the shape of a screen. This manager controls the overlapping and consequent visibility of different visual components. An analogous manager needs to control similar clashes between competing pieces of audio feedback. The concept for this manager should not be limited to only handling audio feedback, but to all forms of feedback. This confirms Design Guideline 5 which stated that a component which can intercept and modify requests for presentation can provide global control over the resources used. A further requirement is that it should be possible to modify or add rules which manage these changes to match the flexibility provided in allowing the presentation of the widgets to be changed.

Requirement 7	The rules which manage the presentation of the toolkit's widgets must be modifiable to ensure that the flexibility afforded by the support for different output modalities is matched.

A second form of intra-modality control that the toolkit needs to manage is the presentation of persistent feedback. In Section 5.8.2 the problems associated with the use of audio feedback to provide information about a background task over long periods of time were discussed. In these cases, the toolkit needs to be able to monitor the use of different presentational resources and adjust the requests being made appropriately. These adjustments can be made using the same mechanism described above.

6.2.3 Controlling the Use of Different Modalities

Once an interface is capable of using multiple forms of feedback simultaneously, issues arise over control of which forms to use. A decision needs to be made about the feedback used based on the availability of the resources required to generate the output in different modalities. To make such a decision implies the need to

be able to gather the necessary information and to be able to change the use of the modality accordingly. Typically, the output module generating the feedback in a particular modality will be able to determine whether there is sufficient resource to produce the requested feedback. Thus, it will be able to inform the toolkit if it is no longer able to meet requests for feedback allowing the toolkit to change the requests appropriately. The availability of presentational resources, however, does not need to be treated as a binary condition. An alternative would be to detect the availability of resources and modify the requests made so that they use an appropriate amount of the available resource, thus maximising its use. This requires the monitoring of physical data to detect the availability of the different presentational resources. In many instances, the components translating the requests for presentation into concrete feedback will be the sensors as they will often be in the best position to detect the availability of presentational resources, but this need not be the case. In either case, the sensor passes the information to the toolkit which distributes this information to the individual widgets. This different mechanism is used as opposed to the mechanism described above where the output module informs the toolkit directly because in this case, the use of the output module is not stopped but rather it is changed. These changes do not necessarily apply to all widgets. Perhaps the designer of the interface wanted to ensure that certain widgets are not effected by such changes. Therefore, the sensor informs the control system which then informs all the relevant widgets. All subsequent requests are then given the relevant modifier to ensure that the presentational resource is used appropriately.

The availability of resources can be illustrated by considering a possible scenario. If a user stopped working on his/her desktop machine and continued working on a laptop machine with a smaller screen size, the availability of the resource necessary to display visual information will have been reduced. This can be overcome by reducing the amount of that resource used (in this case making the interface components smaller), or by using an alternative modality for some or all of the interface components. This scenario confirms the second listed requirement: the toolkit should be able to adjust the use of output modalities in accordance with the availability of the resources necessary to produce the output. This adaptation could take place within a modality or across modalities. As described in the previous section, it is important that the toolkit is not limited in the way it can make these changes.

The availability of a resource can be considered as one aspect of a broader problem faced by the toolkit: the suitability of a presentational resource. Clearly, if the resource is unavailable, it is unsuitable for use. Other examples of unsuitable resources are more subtle, however. In a quiet environment, audio feedback could be an appropriate means of presenting a user with information. In a very loud environment, however, audio feedback may not be perceivable by a user and so would be unsuitable. The situation is, however, more complex than the simple binary case described here. If the quiet environment described above becomes slightly louder the use of audio feedback may still be appropriate, but the way this modality is used needs to be modified. In this case, the volume of the audio feedback needs to be increased to ensure that it remains discernible. To make a decision about the suitability of a modality implies the need to be able to gather the information necessary to make the decision and to be able to change the use of a modality based upon that information. The information required will have to come from the environment, or context, in which the interface is located as it is contextual factors, such as the ambient volume of the surroundings or the location of the user (e.g. with respect to other users), as well as the task itself that determine which presentation

modalities are suitable. As before, it must be possible to change the rules which control this adaptation. Similarly, the sensors which detect the context should not be fixed, but rather, it should be possible to load new sensors as and when required.

Requirement 8	The toolkit should be able to modify the use of different presentation modalities according to their suitability.

Requirement 9	The sensors which are used to detect the context of the toolkit's widgets should be external to the toolkit so that they can be modified as required.

An interesting issue arises when the concepts of resource suitability and resource sensitivity are considered. The former has been described as the suitability of a particular output modality according to the current context whilst the latter has been described as the ability of the system to meet the demands of the interface in a particular output modality. As the availability of a resource can be considered as part of the current context it is reasonable to suppose, therefore, that a similar mechanism should be used to control both these attributes.

Design Guideline 7	Although information about the availability of presentational resources and the suitability of presentational resources will often come from different sources they can both be considered as contextual information and therefore can be managed using similar mechanisms.

6.2.4 End User Control of the Feedback Generated

It is important that the end users of applications built using the toolkit retain control over the feedback generated. Although many of the adaptations described in the previous section could potentially be performed automatically this may prove to be annoying to users. Shneiderman [103] talks about the problems that can arise with adaptive user interfaces:

> "… but even occasional unexpected behaviour has serious negative side effects that discourage use. If adaptive systems make surprising changes, users must pause to see what has happened. Then, users may become anxious because they may not be able to predict the next change, interpret what has happened, or restore the system to the previous state." (pp35-36)

There are many aspects of the toolkit over which the users should have control. The first of these is the use of output modules. If there are two or more possible output modalities that could be used for a widget's presentation, a decision has to be made regarding which will be used. The user could potentially wish to use different output modalities for different widgets or perhaps use the same output modality for all widgets. Different output modalities could be used, for example, to differentiate between different applications or,

perhaps, to highlight individual widgets that a user feels are important. Once the output modalities to use have been decided, there may be output modality-specific options that require to be set for each widget. Again, the user may decide to use the same options for all the widgets or use different options for different widgets.

For many users, however, this may not be a relevant, or even desirable task. Indeed, for such the ability to change the presentation may well lead to a degradation in the effectiveness of the interface. Typically, it is designers of the interface that will be particularly interested in the ability to change the settings to enable them to determine the best forms of presentation to be used. End-users of applications built using the toolkit, on the other hand, will not, typically, be interested in changing most of the settings. This therefore implies the need to provide multiple user interfaces to the toolkit, one for designers which affords total flexibility regarding the presentation used and one for end-users which provides only limited flexibility regarding the changes allowed. The latter interface may, for example, only allow the users to modify the high level options for the output modules in use (the style used, for example) but may not allow the user to change the output modules used by the different widgets.

Requirement 10	It should be possible to develop different tools for the toolkit, each of which allow the presentation of the interface to be modified in different ways.

6.2.5 Grouping Widgets Together

In many instances, what is viewed as a single widget by a user is, in reality, composed of several widgets. Such widgets include menus which are composed of multiple menu items. In other cases, multiple widgets can be grouped together logically. For example, a set of radio buttons, although distinct widgets, can be logically grouped together to form a single meta-widget. In both these cases, the individual widgets will need to have at least some concept of the other widgets in the group. Menu items, for example, may need to know their location in the list of menu items whereas radio buttons need to be aware of when another button in the group is selected to ensure that their mutual exclusivity is preserved. It is therefore important that groups of widgets can be formed in the toolkit.

6.2.6 Software Engineers Use of the Toolkit

It is important that software engineers are able to use the toolkit to build applications without having to overcome a large learning cost. If it is necessary to make large changes to existing code or learn new techniques when writing new code then the uptake of the toolkit may be reduced. This implies that the toolkit should be built in a popular, well-known programming language and conform to the API of an existing widget toolkit. By building the toolkit such that it conforms to an existing API and making only consistent and standard changes, it should be possible to automatically transform existing code into code which uses the toolkit, thus greatly simplifying the use of the toolkit.

Requirement 11	The toolkit should conform to an existing API to minimise the cost to engineers who build new applications and/or modify existing applications so they can use the toolkit's widgets.

6.2.7 Summary

There are now eleven requirements for the toolkit to meet:

1. The full behaviour of the toolkit's widgets should be exposed allowing suitable forms of presentation to be generated.
2. The toolkit should be able to determine if it is not possible to play all the requested sounds and, if so, modify the feedback appropriately.
3. It must be possible to group widgets together to allow their feedback to be co-ordinated appropriately.
4. A toolkit of multimodal widgets must allow users to control the form of presentation used for its widgets.
5. The toolkit must be able to manage any interference between different pieces of feedback.
6. The toolkit must be able to monitor the presentation of a widget over time and, if appropriate, adjust the presentation.
7. The rules which manage the presentation of the toolkit's widgets must be modifiable to ensure that the flexibility afforded by the support for different output modalities is matched.
8. The toolkit should be able to modify the use of different presentation modalities according to their suitability.
9. The sensors which are used to detect the context of the toolkit's widgets should be external to the toolkit so that they can be modified as required.
10. It should be possible to develop different tools for the toolkit, each of which allow the presentation of the interface to be modified in different ways.
11. The toolkit should conform to an existing API to minimise the cost to engineers who build new applications and/or modify existing applications so they can use the toolkit's widgets.

6.3 Toolkit Design

Eleven requirements for the toolkit were described in the previous section. The first decision about the design of the toolkit concerns the eleventh requirement above, that the toolkit should be simple to use to enable developers to easily modify existing applications or build new applications. There are two aspects to the solution presented in this thesis which satisfies this requirement: first, if the toolkit is to be simple to use it must use a language that is currently widely used; second, the toolkit must conform to an existing toolkit application program interface or API. It was decided to use the Java™ programming language [106] as the basis for the toolkit as it is a well known, platform independent language which has a broad existing user base being, for example, widely taught on programming courses. The toolkit's API is based upon the Java

Swing™ Toolkit's API as this is a commonly used Java widget toolkit and consequently is very well known. The remainder of this section discusses the design presented in Figure 6.1.

6.3.1 An Overview of the Toolkit Architecture

The proposed architecture for the toolkit is shown in Figure 6.1. The architecture can be considered analogous to the architecture of X-Windows (Section 4.3). Both employ a client-server architecture, in X the clients, or display areas, can use only one server whereas in the toolkit the clients can use multiple servers.. In both cases, the servers (or output modules in the case of the toolkit) provide the presentation for the clients. This follows from Design Guideline 4 which states that such an architecture facilitates the use of multiple output modalities by enabling the clients to use multiple different servers to provide feedback. Unlike the servers in X, the toolkit's output modules do not handle input, providing only output for the widgets. This follows from Design Guideline 3 which states that separating the input and output components makes it easier to change the presentation than if the two components were tightly coupled. Design Guideline 5 states that the provision of a component which can intercept and modify requests for presentation enables global control over the presentational resources used. Whereas X did not explicitly supply a component to do this, but rather allowed servers to pass requests from a client onto a different client (e.g. a windowing manager) before they were handled by the server, the toolkit does provide such a component, the rendering manager which receives all the requests for feedback before they are translated into concrete feedback and consequently is capable, along with the control system, of managing the use of different presentational resources.
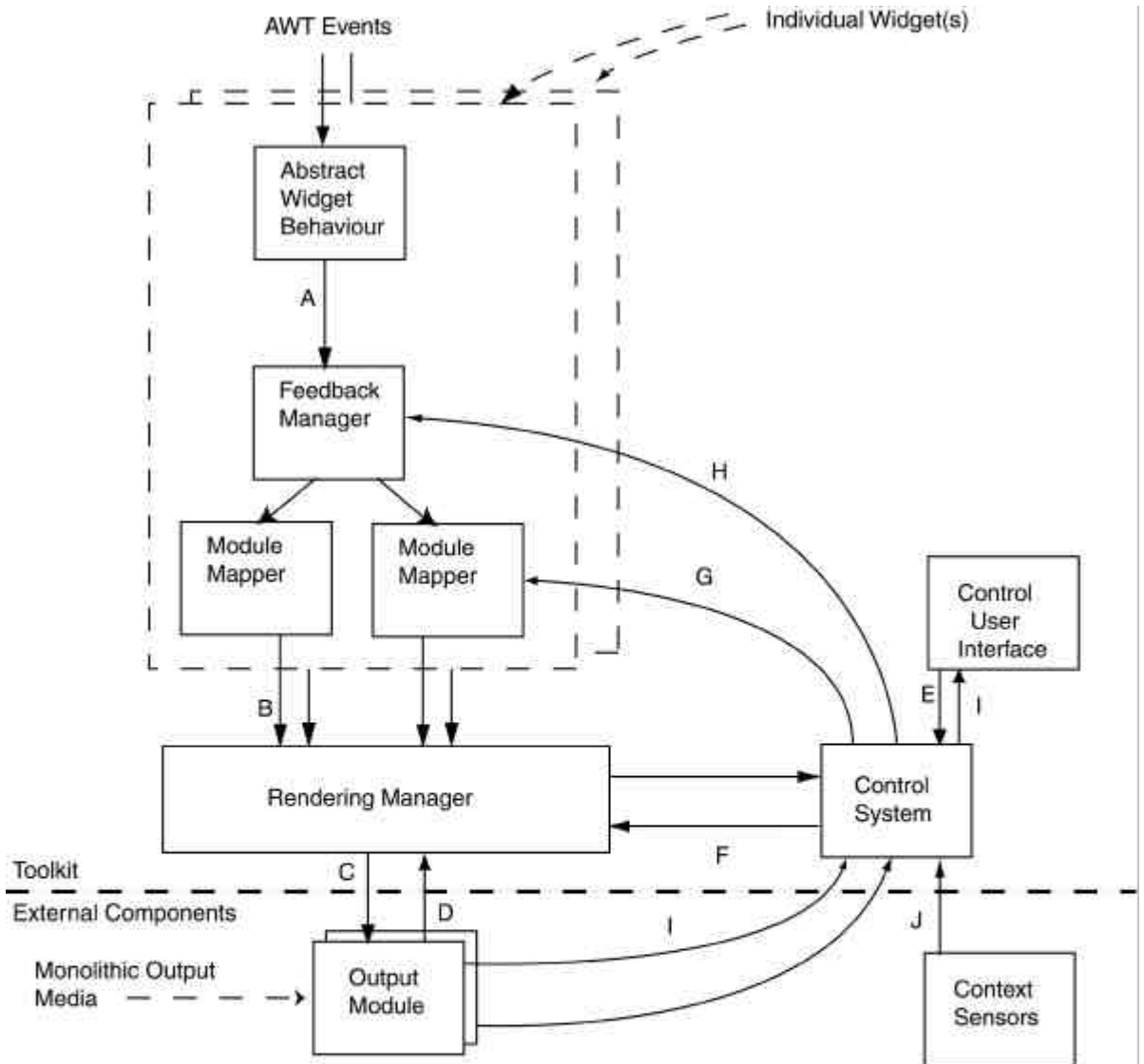
Key

A - Abstract request for feedback
B - Output media specific request for feedback
C - Output media specific request for feedback (potentially modified)
D - Output media flags stating unable to handle specific widget(s)
E - Commands to change the use of output media for specific widget(s)
F - Rules governing the transformation of widget presentation
G - Instruction to a widget to use a particular output media option
H - Instruction to a widget to use a particualr output media
I - Description of output media's options and abilities
J - Context Information

Figure 6.1 The toolkit's architecture showing both the components that belong to individual widgets and global components that manage the presentation of the individual widgets. Each widget can potentially use multiple output modules and equally each output module can potentially provide feedback for multiple widgets.

6.3.2 Ensuring the Simple Addition of Different Forms of Presentation

The first requirement listed in Section 6.2.7 was to fully expose the behaviour of the toolkit's widgets, making the task of replacing the existing presentation of a widget with a different design of presentation, or to supplement the existing presentation with an additional design, simpler. The component in the architecture shown in Figure 6.1 that supports this requirement is the *abstract widget behaviour*. This component defines the behaviour of the widget, accepts events that occur to the widget and translates them into requests for presentation. When the current state of the widget changes, the abstract widget behaviour must make a request for appropriate presentation, guaranteeing the full behaviour of the widget is exposed, following the recommendation of Design Guideline 6. These requests for feedback are translated into concrete feedback by the output modules, monolithic components that can handle requests from multiple widgets. This follows from Design Guideline 1 which states that separating the application model from the presentation enables the presentation to be changed. Furthermore, by separating the output from the input the toolkit adheres to Design Guideline 3 which states that doing this makes it easier to modify the presentation of the widgets. Output modules may provide any form of presentation deemed appropriate. An output module may provide feedback using only one output modality or may utilise multiple output modalities. Equally, an output module may be capable of providing feedback for a wide range of widgets or, perhaps, only one. Thus, it is possible to have an output module that provides both audio and graphical feedback, perhaps providing different "styles" for the feedback which effect the both the audio and visual presentation in a consistent manner. Equally, it is possible to have an output module which provides, for example, only audio feedback and requires the use of a second output module if graphical feedback is required. Similarly, it is possible to have an output module which provides feedback for all widgets and another which provides feedback for only one class of widget. In this way, if a new widget is designed it is only necessary to provide an output module which provides feedback for the new widget.

Each widget has a *module mapper* component for every output module it uses. These components provide a link between the abstract requests made by the abstract widget behaviour and the concrete feedback generated by the different output modules. Each module mapper stores any output module-specific information for requests for feedback made by the widget. The distribution of the requests to the different module mappers is controlled by the *feedback manager* which, when it receives a request from the abstract widget behaviour, passes a duplicate of the request onto all module mappers for that widget which then embellish these requests with output module-specific information. For example, when a mouse is moved over a graphical button causing its state to change, the abstract widget behaviour generates a request for appropriate presentation given its current state which is passed onto the feedback manager. The button employs two output modules to provide feedback, an audio module and a visual module so the feedback manager passes the requests onto the two, associated module mappers. These module mappers are aware of the output module-specific information required for presentation of that widget. Perhaps the visual output module can render buttons in two ways: as rounded rectangles or as ovals. The style currently used by the widget would be stored in the module mapper associated with the visual module. The output module-specific requests are then passed on to the appropriate output modules. By making the initial requests for presentation

from the abstract widget behaviour independent of any presentational information the architecture conforms to Design Guideline 2 which states that describing the required presentation in abstract terms means the presentation can be freely changed or supplemented.

6.3.3 Global Control Over the Presentation of the Widgets

Several of the requirements described previously require that the toolkit is able to manage the feedback of several different widgets. The second requirement states that the toolkit must be able to determine whether it is possible to meet all the requests being made. For example, if there is insufficient audio capability to meet all the requests for sound the toolkit must attempt to modify the requests so that widgets can all be presented. This can be considered as the widgets being sensitive to the availability of presentational resources, or simply being resource-sensitive. The fifth requirement states that the toolkit must be able to manage interference between the feedback of different widgets. In Section 5.8.1, for example, it was described how the audio feedback of two progress indicators could interfere with each other, meaning that for the presentation to be effective it would have to be changed, perhaps by playing fewer sounds. The sixth requirement states that the toolkit must be able to monitor the presentation being requested by a widget over time and adjust it if appropriate. In Section 5.8.2, for example, it was described how the persistent audio feedback of a progress indicator could become annoying, indicating that the use of audio feedback over long periods of time would be unsuitable. The eighth requirement states that the toolkit must be able to modify the use of the different presentational resources according to their suitability. The volume of audio feedback, for example, could be changed as the ambient volume of the environment changes. These last three requirements can be considered as the widgets being sensitive to the suitability of a presentational resource.

All these requirements can be satisfied by the inclusion of a global component which receives the requests for feedback before they are translated into concrete feedback, satisfying Design Guideline 5 which states that a component which can intercept and modify requests for presentation provides global control over the resources used. In the toolkit, this component is the *rendering manager*. This component is passed the requests for feedback from all the widgets, allowing it the opportunity to monitor the feedback being presented at a global level. The architecture of X-Windows allowed a component similar access to the requests for presentation by allowing servers to send requests they receive to a window manager client which then modifies the requests according to the availability of the presentational resources. In the toolkit the rendering manager communicates with the widgets via the *control system* component which can be considered the glue which keeps all the components of the toolkit together by keeping references to them all.

To allow the toolkit to determine whether there are sufficient presentational resources available to meet the requests of the different widgets, the toolkit needs to be supplied with rules which allow it to do this. Requirement 7 states that these rules must be modifiable to ensure that the flexibility afforded by the use of different output modalities is matched. If, for example, a new output module is designed and subsequently used with the toolkit, it is unlikely that its needs will be met by the existing set of rules. It will therefore be necessary to include new rules in the toolkit. One way to do this would allow these rules to be stored in a resource file loaded when the toolkit was started up. Because the output modules use the presentational resources managed by the rules, it is not unreasonable to assume that these rules could also be generated by

the output modules. The design for the DTD for these rules is shown in Figure 6.2. The DTD is specified in XML as this is a well known standard which can easily be parsed in Java.

```
<!ENTITY % objectName " name  CDATA #REQUIRED">
<!ENTITY % numericalValue " value NMTOKEN #REQUIRED">

<!ELEMENT Rule (OutputModule+, Widget+, Test, Condition+)>

<!ELEMENT OutputModule EMPTY>
<!ATTLIST OutputModule %objectName;>

<!ELEMENT Widget EMPTY>
<!ATTLIST Widget %objectName; >

<!ELEMENT Test EMPTY>
<!ATTLIST Test %objectName; >

<!ELEMENT Condition (((Min, Max) | Enumeration), Result+)>

<!ELEMENT Min EMPTY>
<!ATTLIST Min %numericalValue; >

<!ELEMENT Max EMPTY>
<!ATTLIST Max %numericalValue; >

<!ELEMENT Enumeration EMPTY>
<!ATTLIST Enumeration %objectName;
                      value   CDATA #REQUIRED>

<!ELEMENT Result (Enumeration)>
```

Figure 6.2 – An initial design for the DTD (Data Type Definition) of the toolkit's rules in XML. A rule consists of one or more output module(s) that the rule effects and one or more widget(s) to which the rule applies, the test to be applied and one or more conditions with associated results.

In this design, a rule consists of one or more output modules that it effects and one or more widget classes that it applies to. The rule also specifies one or more tests to be run with associated conditions to be met for the feedback to be modified. The output modules and widgets are identified using a string. The test is also identified by a string which maps to one of an extensible set of pre-defined tests included with the toolkit. A condition can either consist of a pair of integer values identifying the minimum and maximum values of a range or a string representing a named enumerated value. Similarly, a result consists of a named enumerated value. For example, consider the rule shown in Figure 6.3.

```
<Rule>
  <OutputModule name="Standard Earcon Module">
  <Widget name="All">
  <Test name="Number Requests">
    <Condition>
      <Min value=11>
      <Max value=999>
      <Enumeration name="timescale", value="1">
      <Result>
        <Enumeration name="Output Module", value="Visual
Module">
      </Result>
    </Condition>
  </Test>
</Rule>
```

Figure 6.3 The XML instance of a simple rule which monitors the use of the "Standard Earcon Module". If more than ten requests are made, the rule determines that the "Visual Module" should be used for presentation instead.

This rule monitors the number of requests for feedback made to the "Standard Earcon Module". If more than ten requests are detected in a second, the rule specifies that the feedback should be switched to the "Visual Module". The possibility of a race condition can be avoided by modifying the requests directly without making the widgets alter future requests. In this way, the number of requests for audio feedback remains constant, ensuring the rule is continually invoked. This has the advantage of not requiring a history mechanism - the widgets continue to make the same requests, meaning that their preferred options do not need to be stored. They can be recalled when the modification enforced by the rule is no longer required. The disadvantage to this approach is that the rendering manager needs to continually implement the same changes enforced by the rules, increasing the processing overhead.

Of course, as described in Section 6.2.3, some instances of resource sensitivity will not be as easy to detect using a rule and will require the use of a sensor. Consider, for example, a visual output module which uses a screen for its display. The number of requests made for feedback to this module does not have a fixed limit in the same way as an audio module may be limited by the number of available MIDI channels. In this case, the visual module would need to determine the availability of the resource, perhaps simply the screen size, and inform the toolkit either that it is unable to meet more requests or that future requests will need to be changed to maximise the use of the presentational resource. In the both cases, the output module informs the rendering manager directly, enabling the rendering manager to use different resources.

The management of resource suitability can be handled in a similar fashion. Information about the context is supplied by external sensors which inform the control system of the appropriate modifications to make to requests for feedback. This information is then passed to the rendering manager. By sharing the same mechanism with the mechanism used to support resource sensitivity, the design follows Design Guideline 7.

The case of the feedback requested by two or more widgets interfering with each other can be handled using similar rules to that shown in Figure 6.3. Because all the requests for feedback are received by the rendering manager it can keep track of the different requests being made and modify them as appropriate. Consider the case of the audio progress indicators which interfere with each other, first described in Section 5.8.1. In this

case, one way to ensure the feedback of the different progress indicators does not interfere with each other would be to reduce the number of sounds played for each progress indicator as the number of progress indicators increased. Thus, a rule to manage this would monitor the number of progress indicators requesting audio feedback and as this number changed, the rule would modify the number of sounds to be played by the output module for each progress indicator. If the audio feedback obeys Guideline 4, which states that if possible the different sounds used to provide a piece of feedback should be ranked in order of importance, then the output module need only follow this ranking to ensure that the most important information continues to be presented.

The requirement that the toolkit is able to track requests for feedback over time is also fulfilled by the rendering manager. Because all such requests are received by the rendering manager, it is able to build a history of previous requests where this is necessary. Using this history in conjunction with similar rules to that shown in Figure 6.3 the rendering manager will be able to manage changes in feedback over time.

6.3.4 Allowing the User To Control The Presentation

The fourth requirement described in Section 6.2.7 states that users should be able to change the presentation of the different widgets to suit their own needs. The tenth requirement, also described in Section 6.2.7, states that the toolkit should be able to support multiple control interfaces, each with different functionalities, designed to meet the needs different groups of users. Thus, an interface designer would require complete control over the presentation of different widgets while end-users of applications built using the toolkit may only require a limited subset of this functionality. These requirements are met by the control system and control user interface components. Because the control system is capable of referencing all widgets as well as the rendering manager and any sensors, it is able to control the presentation of all the widgets. By providing an API to the control system, the toolkit enables an external component: the *control user interface* to access these controls. Because the control UI is an external component, it can freely be swapped with other control UIs fulfilling both requirements described above.

6.3.5 Grouping Widgets Together

Requirement 3 states that it should be possible to group widgets together. This may be done to ensure that: they behave consistently (e.g. in a group of radio buttons only one may be selected at any given time); they are presented consistently (e.g. perhaps every radio button in the group should have the same background colour); and/or that they are presented appropriately (e.g. perhaps the presentation of a menu item is dependent upon its location in the menu). The abstract widget behaviour component ensures that a group of widgets behaves consistently by allowing global events. In the example of the group of radio buttons, when one button is selected, it is important that all other widgets in the group are made aware of this. Thus, an appropriate event is passed to all the widgets in the group when one of the group is selected.

The consistent presentation of a group of widgets is facilitated by allowing hierarchies of widget instances in a similar way to, for example, Smalltalk or InterViews. In the example of the button group, as the parent widget (the group) is merely a logical grouping it does not have any of the components shown in Figure 6.1

as it does not have any behaviour and so cannot request feedback. But if it, not the child widgets (the individual buttons), receive the commands from the control system regarding feedback to be used, it can them pass on these commands to all the child widget ensuring that they are all presented consistently. In other cases, it may be possible that the parent widget will possess a behaviour and so request feedback, but the mechanism would remain the same. The appropriate presentation of a group of widgets is facilitated in a similar way, with the parent widget informing the child of the relevant information. Thus, for example, when a menu item is added to a menu, the menu informs the menu item of its location in the menu.

6.3.6 Requests For Feedback

In the previous sections, the various components of the toolkit and the way they communicate with each other have been described. No consideration has been given to the form the widget's requests for feedback take, however. This section describes these requests. Requests for feedback by the widgets need to include three pieces of information: widget type, state, and the event that caused the transition to this state. This is the minimum information required by an output mechanism to allow it to provide useful feedback. Both the current state and the event that caused the transition to this state are required as this allows the feedback presented to be state dependent, event dependent, or perhaps both. If the request for feedback only encapsulated the current state of the widget, a feedback designer would have no way of distinguishing different transitions which lead to the same state. For some widgets this may be all that is required. For other widgets, however, more information may be required. A progress indicator, for example will have extra information about its state such as its current, minimum and maximum values. The requests for feedback therefore need to be able to include varying pieces of state information. As well as information about the widget, which is output mechanism independent, the request may well include information that is specific to the output mechanism. For example, the user may wish to set a look and feel option for an audio output mechanism which requests that the feedback be given in a jazz style. As with the information about the state of the widget, varying information about the output mechanism setting may be included in a request. Thus, the request for feedback from a widget needs to be able to carry four pieces of information:

- the type of widget
- its current state
- the event that caused the transition to that state
- any extra information regarding the widget's current state and any output mechanism preferences.

The latter two need to be scaleable as an unknown number of items may be passed. One way to achieve this would be to store the information in a dictionary. The keys for the dictionary would be the descriptions of the information stored and the values would be the actual information. In Java, the Hashtable class allows values of any type to be stored, meaning the requests would not be constrained in the type of information passed.

6.4 Analysis of Design

Section 6.2 described the requirements for a toolkit of resource and context sensitive multimodal widgets and Section 6.3 described an architecture designed to fulfil these requirements. In this section, the toolkit architecture is summarised, several scenarios which the toolkit may potentially have to handle are described, and the way the toolkit is able to handle these scenarios is described.

6.4.1 Toolkit Architecture – Summary

A summary of the objects used by the toolkit is given in Figure 6.4. For each object, several pieces of information are given: the scope of the object (i.e. whether it is a global toolkit object or a widget specific object); what knowledge is held by the object; and what other object(s) it communicates with.

Object	Scope	Knowledge	Communication
Abstract Behaviour	Widget	State of current widget	**In** Outside system: User input to widget **Out** Feedback Manager: Abstract, media independent feedback requests
Feedback Manager	Widget	Media used by widget	**In** Abstract Behaviour: Abstract, media independent feedback requests Control System: Commands regarding output media **Out** Module Mapper(s): Abstract, media specific feedback requests
Module Mapper	Widget	Media specific settings for widget	**In** Feedback Manager: Abstract, media specific feedback requests Control System: Commands regarding output media options **Out** Rendering Manager: Abstract, media specific, parameterised feedback requests
Rendering Manager	Global	Rules for transforming feedback requests	**In** Module Mapper(s): Abstract, media specific, parameterised feedback requests Control System: Transformation rules Output Media: Transformation flags **Out** Output Media: Abstract, media specific,

			parameterised feedback requests Control System: Transformation commands
Output Media	Global	Concrete renderings of abstract feedback requests	**In** Rendering Manager: Abstract, media specific, parameterised feedback requests. **Out** Rendering Manager: Transformation flags Control System: Abilities and rule definitions. Outside System: Concrete feedback.
Control System	Global	Rule definitions All widgets Sensor & output media options and abilities	**In** Rendering Manager: Transformation commands Output Media: Abilities and rule definitions. Sensors: Options and changes in context. Control UI: Transformation commands **Out** Feedback Manager: Output media to add/remove Module Mapper: Output media option to add/remove Rendering Manager: Transformation rules Sensors: Sensor commands Control UI: User options
Sensors	Global	Context information	**In** Control System: Sensor commands **Out** Control System: Options and changes in context
Control UI	Global	User Settings	**In** User options **Out** Transformation commands

Figure 6.4 - Summary of toolkit architecture objects. For each object its scope (global or widget), knowledge and communication links are described.

The knowledge about the presentation is distributed throughout the toolkit. The abstract widget behaviour only has knowledge about the current state of the widget, but has no knowledge of the presentation being used. This means that the requests generated by the abstract widget are abstract requests which are independent of presentational information. This conforms to Design Guideline 2. The module mappers, on the other hand, do have knowledge of the presentational specific information for the widget. Every widget has one module mapper for every output module being used. The module mappers, therefore, provide the link between the abstract requests and the concrete feedback provided by the output modules. A summary of the way the different objects in the toolkit communicate with each other is given in Appendix E.

6.4.2 Scenario 1: Modifying Audio Feedback in a Noisy Environment

A user is running an application built with the toolkit on a desktop machine capable of producing high quality audio feedback. Initially, the environment in which the user is situated is very quiet. Using the control panel the user specifies an audio output module for the application's widgets. The output module informs the control panel of the different abilities it has and options it can handle. One of those options is volume, where the value can have a range of between 0 and 100. The user leaves the volume at 50, the default value. The user also activates an ambient volume sensor[14] which detects the ambient volume of the surrounding environment. The sensor specifies to the control panel that it can vary its sensitivity, again where the value can have a range of between 0 and 100. The user adjusts the sensitivity of the sensor to 10, making the sensor insensitive to any sudden changes in volume. The user now starts to work with the application.

If the ambient volume of the surrounding environment was then to increase, the sensor would detect this change. Because the sensor has a low sensitivity value, it would not indicate a change to the toolkit until it had determined the change was persistent. It would then alert the rendering manager to the change in the ambient volume by giving it a modifier. This modifier would be tagged by the parameter it was affecting, in this case volume, and the sensor that provided it. All subsequent requests for audio feedback would then have their volume parameter modified appropriately as the output module had previously indicated that it could handle changes in volume. All subsequent audio feedback would be louder to compensate for the increased ambient volume. If the ambient volume continued to increase there may reach a time when the adjusted volume of the presentation requests go above the range of the output module. In this instance, the rendering manager would have to check the current rules to see if there are any switches that can be made. The user has selected a rule which states that in the absence of audio feedback, the size of the visual feedback should be increased 10%[15]. Because the user has flagged the rule as one that can be applied automatically, the rendering manager does not require user intervention before applying the rule. The rendering manager therefore blocks all requests to the audio output module being used, and applies a modifier to the requests made to the visual output module increasing their size by 10%.

This scenario shows how the toolkit can handle changes in the use of feedback according to contextual requirements and how it can handle switches between output modules when insufficient resources exist in the requested output module. An important aspect to note is that the user retained control at all times. The user defined which sensors would be used and how they would work. The output modules used to provide feedback were selected by the user as was the rule used to switch between them. Finally, the user had the option of confirming the application of the rule but instead chose to allow it to be applied automatically.

---

[14] Such a sensor has been built. It samples the ambient volume of the environment, damping out any sudden changes in the ambient volume (such as a door being slammed) and is also designed to avoid feedback loops by monitoring the audio output of the system.

[15] Although such a rule may seem nonsensical work done by Brewster *et al.* [19] indicated that larger graphical buttons were just as effective in allowing users to make selections as smaller, sonically-enhanced buttons. It has not been shown, however, that such a rule would apply in all instances!

Even if the user chose to do none of the configuration but rather let the system do everything automatically the option remains for the user to change anything that is not suitable for them.

6.4.3 Scenario 2: Handling Intra Modality Clashes

A user is downloading five files from the internet. The application being used to perform this task was built using the toolkit and uses audio feedback for the progress indicators. However, because five files are being downloaded, the audio feedback from the different progress indicators clash. The output module used to provide the audio feedback includes sounds which were built according to Guideline 4, which states that the different sounds used to provide the feedback should be ranked in order of importance. The output module is therefore capable of reducing the number of sounds played with only the minimum amount of information being lost to the user. The output module calls this property the "Fidelity" of the sounds and allows for three levels of fidelity: low, medium and high. The output module also provides a rule which describes the maximum number of progress indicators that can be presented at each fidelity. In this case, the rule states that if there are three or more progress indicators requesting feedback then the fidelity for the audio feedback must be set to low. As the files complete their downloads, the number of progress indicators requesting audio feedback is reduced and so the rendering manager modifies the requests so that the fidelity value is increased. Because the property "Fidelity" is specific to the output module it would provide this rule and, typically, would automatically activate it.

This scenario shows how the toolkit is capable of monitoring requests for presentation and consequently modifying requests if it detects a clash. This example highlights the interaction between an output module, which provided a rule based upon the presentation it was capable of providing, and the rendering manager which implemented the rule to ensure that the presentation provided was appropriate.

6.5 Conclusions

In this chapter, a design for a toolkit of multimodal, resource-sensitive widgets was described. Initially, the requirements that had been gathered earlier in the thesis were discussed, drawing out further requirements for the toolkit. A design for the toolkit was then presented followed by an in depth discussion of how the design meets the requirements, in particular with reference to the design guidelines gathered in Chapter 4. The design described uses a client-server architecture, similar in many aspects to that of X-Windows. In the toolkit, the individual widgets are the clients and output modules which provide concrete feedback for the widgets are the servers. This design allows the toolkit to be multimodal by allowing the widgets to send requests for feedback to multiple output modules. Similarly, the design allows the toolkit to be resource-sensitive by providing a global component which receives all the requests for feedback made by the widgets and so can modify any unsuitable requests. Finally, two scenarios were described which demonstrated how the toolkit would be able to meet the requirements presented earlier.

Chapter 7: Toolkit Implementation

7.1 Introduction

In the previous chapter the design for a resource-sensitive, multimodal toolkit was described. The requirements for this toolkit were discussed and the consequent design analysed. In this chapter the implementation of this design is outlined, with particular reference to how it satisfies the requirements of the design. Issues that arose about the design and its implementation are also discussed. Initially, the implementation of the individual components that are part of each widget are described, followed by how these individual components communicate with each other. Next, the implementation of the framework into which these widgets fit is described and, finally, the implementation of components that are not part of the toolkit *per se* but are necessary additions (e.g. output modules and sensors) is discussed. Throughout this chapter it is assumed that the reader has some knowledge of Java™ and its associated terminology. For an introduction to Java see [33].

7.2 The Use of Java as the Implementation Language

The toolkit was implemented in Java [105] because it is a well known programming language with a broad user base. The toolkit was based upon the Swing User Interface Toolkit. For the purposes of this thesis, the term 'Swing User Interface Toolkit' or simply 'Swing' will be used to cover the Swing, AWT and AWTEvent packages in Java. The AWT package is a user interface toolkit supplied with Java which forms the foundations upon which the Swing package is built. The AWTEvent package is the event handling mechanism supplied with Java which both Swing and AWT use. One decision that had to be made prior to the implementation of the toolkit was the level to which the toolkit's widgets would be based upon the standard Swing widgets. This decision impacts mainly on the input to and output from the widgets. Basing the toolkit's widgets upon the existing Swing widgets has the advantage of allowing the toolkit to use the existing input mechanism. This means that it is not necessary to implement methods that determine whether a mouse event, for example, occurs over a particular widget as this is automatically supplied by Swing. Conversely, this approach is limited to using events that are supplied by Swing, limiting the widgets that can be built using this approach. For example, it is impossible to build a slider widget using the existing

`JSlider` widget as a basis because many of the events required (e.g. "mouse below thumb wheel") are not exposed by the standard widget. Not using the existing Swing widgets as a basis for the toolkit's widgets means that this input handling does need to be built into the toolkit, adding a significant implementational overhead. The advantage would be the increased flexibility in the events that could be generated.

The presentation of the widgets also depends upon this decision. If the toolkit's widgets are based upon the existing Swing widgets then the flexibility of the visual presentation is limited to modifying attributes such as the colour and size of the Swing widgets whereas, if the toolkit's widgets are independent of the Swing widgets, their visual presentation has no restrictions - albeit at the cost of increased implementation time. It would, for example, be possible to change the shape of the buttons if the latter approach was taken.

It was decided to base the toolkit's widgets upon the standard Swing widgets. Although this means limited flexibility in both input and output the advantages in implementation cost made this worthwhile, allowing a prototype of the toolkit to be implemented at relatively little cost. Although the presentation of the widgets is limited, there is still sufficient flexibility to show the effectiveness of the architecture. Because the toolkit is built on top of the existing Swing architecture if a designer were to build a widget which did not use Swing as its basis he/she would need to provide all the underlying event handling that the toolkit relies upon Swing to provide.

7.3 The Implementation of the Widgets

In Chapter 6 the design for the widgets of the toolkit (as shown in Figure 7.1) was discussed. As can be seen, each widget has three components: the abstract widget behaviour, the feedback manager and one or more module mappers. The abstract widget behaviour receives external events and translates them into requests for concrete feedback. These requests are passed to the feedback manager which controls which output modules are being used. The feedback manager passes these requests onto the module mappers for each output module being used. The module mappers then apply any parameters that are relevant to that particular output module before passing the request on to the rest of the toolkit. In addition to these three components, an interface which describes the common behaviour of all the toolkit's widgets was designed. Finally, the actual widget object, which manages the different components of the widget was designed. In this section, the implementation of the different components that form the widgets, and the way they communicate, is discussed.
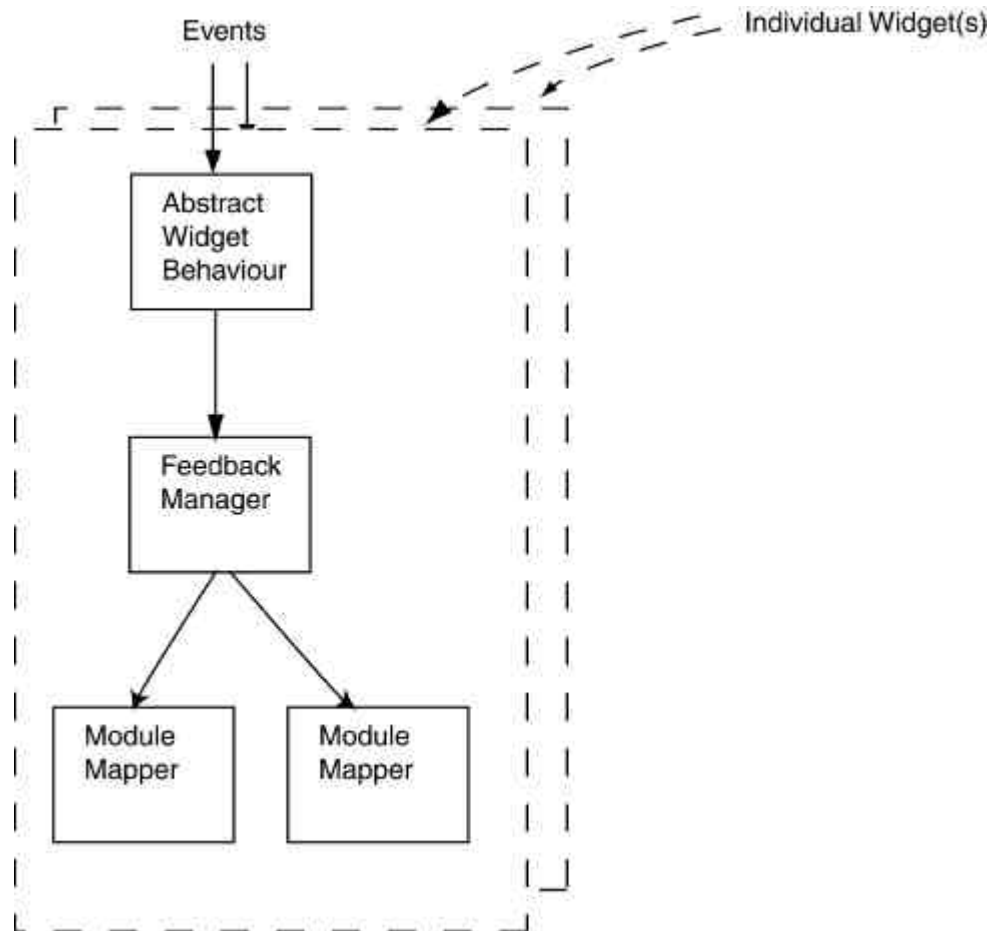
Figure 7.1 – Architecture of a widget in the toolkit.

7.3.1 MComponent Interface

The MComponent interface describes the behaviour that is common to all the toolkit's widgets. The name MComponent was chosen in reference to the JComponent interface which describes the behaviour of the standard Swing widgets with the 'M' representing 'multimodal'. The behaviour defined in the MComponent interface is with regard to the toolkit and is in no way concerned with the behaviour of the widget with regard to the user or the system. The behaviour defined in the interface can be split into three separate sections:

- adding and removing listeners (or objects which are interested in events that occur to the widget)
- adding and removing output modules or output module preferences
- adding and removing global feedback modifiers (i.e. make changes to the feedback according to resource availability or suitability).

The methods used in the first section can be seen in Figure 7.2.

```
// Add/remove event listeners
void addPrivateMouseListener(EventListener eventListener);
void removePrivateMouseListener(EventListener eventListener);
```

Figure 7.2 – MComponent methods allowing the addition and removal of private (i.e. toolkit) mouse listeners.

These methods allow the addition of private mouse listeners. In this context, private means listeners that are used within the toolkit to meet the requirements of the toolkit and these listeners should not be confused with mouse listeners which are added to the widget by an application which is using the widget and is interested in the events that occur to the widget. More detail about these private listeners can be found in Section 7.3.3.

The methods used to add and remove output modules or output module preferences can be seen in Figure 7.3.

```
void addModule(String name);
void removeModule(String name);
void setPreference(String module, String name, Object preference);
void clearPreference(String module, String name);
void clearAll(String module);
```

Figure 7.3 – MComponent methods allowing the addition and removal of output modules and output module preferences.

These methods allow the addition and removal of output modules from the widget (i.e. allowing one form of feedback to be totally removed or a new form added. The preference methods allow a preference for a particular output module to be set or cleared or all preferences for a particular output module to be cleared. More details about the way these methods work can be found in Section 7.4.2.

The methods allowing the setting of a global modifier can be seen in Figure 7.4.

```
// Add/remove a global modifier
void setModifier(String name, Object modifier);
void clearModifier(String name);
```

Figure 7.4 – MComponent methods allowing global modifiers to be set.

These methods allow a modifier to be set or cleared. More details about the way these methods work can be found in Section 7.5.2.

7.3.2 The MWidget Component

The MWidget component is the component that actually represents the widget. This object is a subclass of the Swing widget being extended and implements the MComponent interface. Thus, the toolkit's button object is an MButton: it extends JButton and implements MComponent. The MButton extends JButton allowing it to take advantage of the existing functionality and input mechanism. One exception to this rule so far is the MProgressBar. Because the MProgressBar does not rely upon the input mechanism used by standard Swing widgets it does not have to extend the JProgressBar. Thus, it is able to be truly flexible in the way it is presented.

All the MWidgets have the same structure as outlined here. Firstly, they all implement the MComponent interface. The `addPrivateMouseListener` and `removePrivateMouseListener` methods are called by the widget's abstract widget behaviour (Section 7.3.3). These methods are used to control a list of mouse listeners managed by the widget. These private listeners are used by the toolkit to manage the flow of events through the widget and changes of state of the widget. According to the current state of the widget, different listeners will be listening for mouse events.

7.3.3 The Abstract Widget Behaviour

The abstract widget behaviour component of a toolkit widget defines the behaviour of the widget and consequently translates external input events to the widget into abstract[16] requests for feedback (or GEL – Generic Event Language - Events described in Section 7.3.4) which are passed on to the feedback manager. This component of the architecture is managed by two objects in the widget: the `MWidget` and the `WidgetStateChart`. These two objects are the only two widget objects that are specific to a particular widget. All the other objects described in this section are generic to all widgets. The architecture for the abstract widget behaviour is shown in Figure 7.5.
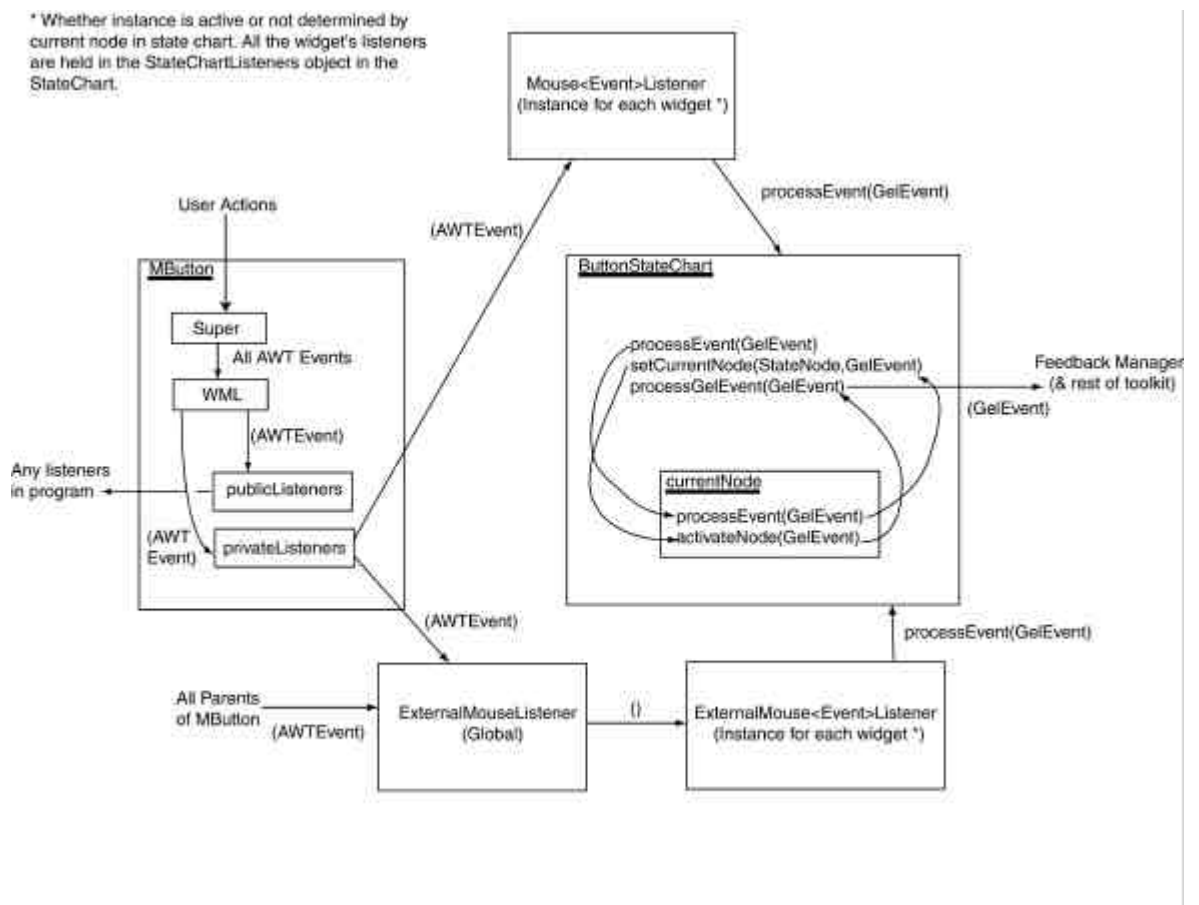
Figure 7.5 – Architecture of the abstract behaviour of a widget showing the data flow through the object and the objects that control this flow. All the components in this figure combine to form the abstract widget behaviour as shown in Figure 7.1.

---

[16] These requests do not specify any presentational information, merely a description of the current state of the widget.

At the heart of the `MWidget` object is the super class of the widget, the original Swing widget (the object labelled "Super" in Figure 7.5). This Swing widget has one listener associated with it (the object labelled "WML" – Widget Mouse Listener - in Figure 7.5). This is the only listener that is associated with the super class of the widget. Listeners added to the widget by the toolkit, or the application using the widget, are stored in lists of listeners and any events that occur to the widget are passed on to these listeners by the widget listener. The toolkit's listeners are stored in the vector `privateListeners` and the application's listeners are stored in the vector `publiclisteners`. This mechanism has two advantages: it allows control over the order in which the listeners are given the events and it decouples the behaviour of the Swing widget from the perceived behaviour of the `MWidget`. The private, or toolkit, listeners are processed first to allow the presentation of the widget to keep pace with the user's interaction. The application, or public, listeners are then processed. By decoupling the behaviour of the Swing widgets from the perceived behaviour of the `MWidgets` it is possible to control the perceived behaviour of the `MWidget`. If, for example, it was desired to change the behaviour of a button so that it was selected when it was pressed rather than when it was released an action event could be generated by the MButton when a press was detected. If the action listener were associated directly with the original superclass this would not be possible.

The `MWidgetStateChart` object implements the statechart defining the behaviour of the widget. Statecharts [65] were used to drive the behaviour of the widgets as opposed to, for example state-transition tables because they provide three advantages:

- *Depth* – Statecharts include the concept of sub-charts. These sub-charts allow behaviour to be organised hierarchically. This allows the grouping of several states to be treated as a single node in a higher level statechart, simplifying the specification of the behaviour.
- *Orthogonality* – Statecharts allow for sub-charts to run in parallel. Thus, statecharts can consist of two or more statecharts representing sub-behaviours of the system being modelled which, in combination, represent the complete behaviour. Without this ability, the definition of the system would require the statechart to have a node representing every combination of the states of the sub-behaviours.
- *Broadcast Communication* – Statecharts allow for events that drive state transitions to be broadcast globally. This ability, in conjunction with orthogonality allows for concurrency in the behaviour of the system. Thus, two sub-charts could both respond to the same event and consequently change state in parallel.

The use of sub-charts in defining the behaviour of a widget is advantageous when considering the use of multiple different input mechanisms to a widget. The superchart, for example, could define the abstract behaviour of the widget, with different sub-charts defining the mechanism-specific behaviour for these different abstract states. For a fuller description of the use of multiple input mechanisms to a widget see Section 9.4.2. As well as being useful when considering the use of multiple input mechanisms, the orthogonal property of statecharts is useful when considering groupings of widgets. Consider, for example, a group of radio buttons. The group can be modelled as a single superchart consisting of sub-charts for the different buttons running in parallel. In this way, the specification of the group is only trivially more complex that that of the individual buttons. Furthermore, the ability of statecharts to use broadcast communication can be used

to ensure that the states of different buttons are kept consistent by broadcasting, for example, selection events thus ensuring the mutual exclusivity of the different buttons.

At the heart of the `MWidgetStateChart` object is the statechart describing the behaviour of the widget. Figure 7.6 shows the statechart specification for the behaviour of a button. The enabled state consists of 7 sub-states which define the behaviour of the button with respect to input using a mouse or similar pointing device. The default state is State A where the mouse button is not pressed and the mouse is outside the button.
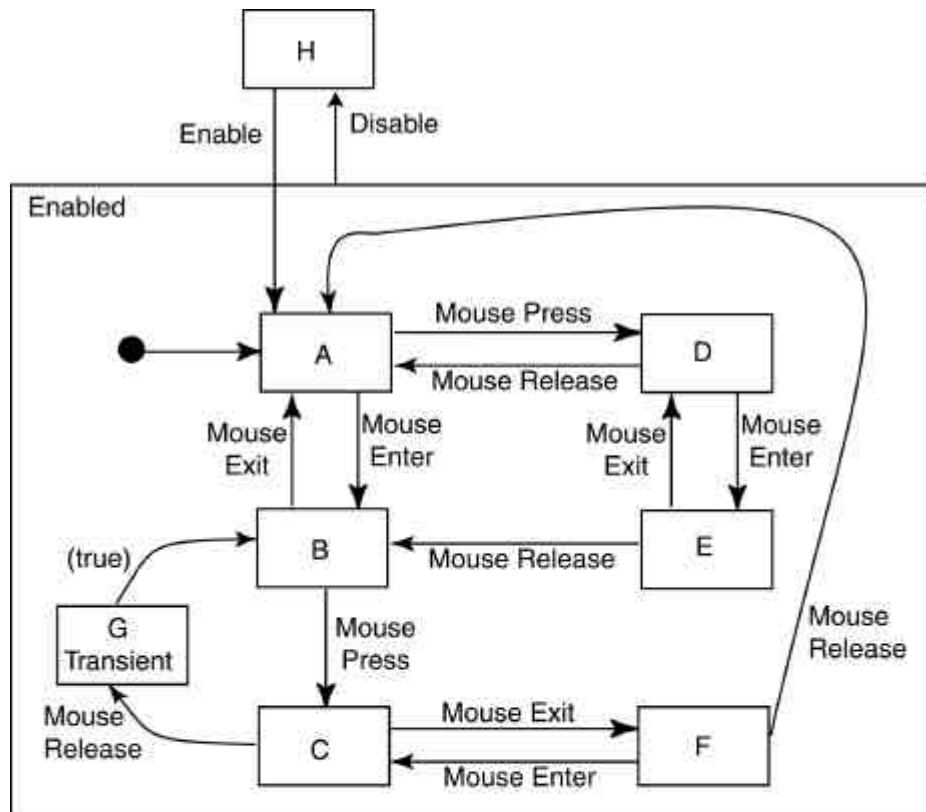
State Descriptions

State	Mouse Location	Mouse Pressed	Mouse Press Location
A	Outside widget	No	-
B	Over widget	No	-
C	Over widget	Yes	Inside widget
D	Outside widget	Yes	Outside widget
E	Over widget	Yes	Outside widget
F	Outside widget	Yes	Inside widget
G	Transient state ... widget selected		
H	Widget disabled		

Figure 7.6 – A statechart specifying the behaviour of a button. This behaviour is specified in terms of the location of the mouse, whether the mouse button is pressed or not and if so the location of the press. When a state transition takes place a new request for feedback is generated.

The nodes of this statechart control the events being listened for at that particular state and determine which becomes the new node should a particular event occur. The code defining State A in Figure 7.6 and its links is shown in Figure 7.7. Although the behaviour of the button has been specified as a statechart, it has been implemented as a state transition table in this case. This was done because the implementation of the state transition table was simpler than the implementation of two statecharts where one of the statecharts consists of only one node[17].

```
        // Not pressed and the mouse is outside.
        normalState = new
        StateNode(this,GelEvent.NORMAL,GelEvent.BUTTON);

        // Links from the normal state
        normalState.addEvent(GelEvent.ENTER,overState);
        normalState.addEvent(GelEvent.EXTERNAL_PRESS,outPressState);
        normalState.addEvent(GelEvent.SET_DISABLED,disabledState);

        // Links to the normal state
        overState.addEvent(GelEvent.EXIT,normalState);
        outsideState.addEvent(GelEvent.EXTERNAL_RELEASE,normalState);
        outPressState.addEvent(GelEvent.EXTERNAL_RELEASE,normalState;
        disabledState.addEvent(GelEvent.SET_ENABLED,normalState);
```

Figure 7.7 - Code used to define the normal state of a button. First, the actual node is defined with parameters of the statechart it is part of, the state it represents and the widget it is in. Then, links are added to other nodes with the event necessary to cause the transition. Finally, links are added to other nodes into the normal state node.

The StateNode class used to define the nodes of the statechart is a generic type that can be used for the statecharts of all widgets. The states (e.g. NORMAL), event types (e.g. ENTER, EXTERNAL_PRESS) and widget types (e.g. BUTTON) are all constants defined in the GEL_EVENT class. This class is the type of object used to make the abstract requests for concrete feedback (Section 7.3.4).

The MWidget and MWidgetStatechart objects are linked by the listeners employed by the state nodes. In the code snippet given in Figure 7.7 it can be seen that the normal state node listens for three events: mouseEnter, externalMousePress and disabled. Thus, when the normal state node is activated it will only listen for these three events. All other events are ignored. This is achieved by the activation and deactivation of the appropriate listeners by the StateNode when it is entered/exited. Thus, the StateNode representing State A in Figure 7.6 activates listeners for mouseEnter, externalMousePress and disabled events when it is entered and deactivates them when it is exited. This example highlights the three different ways events are handled by the toolkit's widgets. The first type, standard AWT mouse events, is exemplified by the mouseEnter event. The second, mouse events not typically recognised by AWT mouse handlers, is exemplified by the externalMousePress event and the third, events that are not AWT mouse events, is exemplified by the setDisabled event.

---

[17] The current implementation of the toolkit does not use statecharts. Future implementations should, however, benefit from the advantages of statecharts especially when the abstraction of input behaviour (discussed in Section 9.4.2) is considered.

Standard mouse events are transferred from the `MWidget` object to the statechart object by the appropriate `mouseListener` which has been activated by the current StateNode. Thus, in the case of the normal state, a mouse enter listener has been activated and when it receives the `AWTEvent mouseEnter`, it generates the appropriate `GELEvent` which is passed to the current state node. This causes the state node to deactivate itself and activate the appropriate new node. The `GELEvent` is then passed on by the statechart to the next component in the widget, the feedback manager. Listeners for standard mouse events are activated by being added to the vector of private listeners maintained by the widget. They are deactivated by being removed from this list.

Mouse events that are not typically handled by standard AWT mouse events, such as the `externalMousePress` (a mouse press which occurs outside the widget), are handled by special mouse listeners. In the case of external events this is implemented using a global, external event listener. This listener is a mouse listener which is added to every widget. In addition, it is added to every parent of every widget. This allows collection of events that occur to widgets that are not in the toolkit. The need for this to be done will be diminished as the range of widgets included in the toolkit is increased. The external mouse listener contains vectors of listeners for every type of external mouse event such as presses and releases. These listeners are the external mouse event listeners for the widget's statechart. When an external event is received by the global external event listener it indicates this to the appropriate external mouse listeners which generate the appropriate `GELEvent` which is passed on to the widget's statechart.

Events that are not AWT mouse events are passed directly to the widget's statechart. If, for example, a button is disabled (by the button's method `setEnabled(false)` being called), the button will pass the appropriate `GELEvent` to the button's statechart. This will deactivate the current node and activate the appropriate new node.

7.3.4 Generic Widget Objects

As well as the objects described in the previous section, all widgets utilise two objects that are common to all widgets. In this way, the overhead of developing a new widget is minimised. Widgets share two common types of component: the *feedback manager* and *module mappers*. A third generic object defines the requests for feedback made by the widgets. In this section the implementation of these objects is discussed.

Feedback Manager

This object manages the distribution of the feedback requests made by the abstract widget behaviour. It is therefore necessary for it to know about which output modules are being used, so it maintains a table of the different module mappers being used. The addition and removal of an output module is implemented by adding and removing the appropriate module mapper to/from this table. When a new output module is added, a new instance of a module mapper is instantiated and added to the table. Because the feedback manager has references to all the module mappers it is also able to inform the module mappers of any output module-specific information. This information is given to the widget via the `MComponent` interface (Section 7.3.1)

which is implemented by the `MWidget` object. The `MWidget` passes the information on to the feedback manager which in turn informs the appropriate module mapper.

Consider, for example, a button which uses two output modules: module1 and module2. When the button is instantiated, the two modules are added to it in turn, using the `addModule` method of `MComponent`. This method has one parameter: the name of the output module. When the feedback manager receives the name it creates a new instance of module mapper and stores it in the table, keyed by its name. If, subsequently, an output module-specific option is set, the widget is informed using the `setPreference` method of `MComponent`. This method has three parameters: the name of the output module, the name of the option being set and the value for the option. If, for example, module1 uses a parameter called size if the size is set to 50 the `setPreference` method would receive parameters of "module1", "size" and 50. When the feedback manager receives this information, it can use the name to retrieve a reference to the appropriate module mapper and pass it the name and value of the option to be set.

Module Mapper

The module mapper object is used to store the options to be used for a particular output mechanism for a particular widget. When a request for feedback is received from the feedback manager the module mapper embellishes the event with the currently set options and passes the event on to the rendering manager. The current options for the module mapper are stored in a table and are set or cleared when the feedback manager calls the appropriate method. To continue the example given in the previous section, the feedback manager passes the name of the option and its new value to the module mapper The new value is stored in the table, keyed by the name to enable the value to be retrieved when appropriate.

Feedback Requests

The toolkit's widgets make requests for presentation in an abstract, presentation modality independent way. To enable this, a simple feedback request object - or `GELEvent` - was implemented. Section 6.3.6 specifies the information these requests should contain: the widget type, the widget's current state and the event that caused the widget to be in the current state. These pieces of information are of a fixed size and type, so are defined as variables which can be accessed with simple get and set methods. Section 6.3.6 also states that further optional information may be included in a request for feedback: widget state parameter(s) and output module-specific parameter(s). Examples of widget state parameters are the current, minimum and maximum values for a progress indicator. Examples of output module-specific parameters are "Jazz", "Rock" and "Pop" for an audio output module. These parameters are all stored in a single table, with each parameter keyed by its name and able to be any type required. Again, the different parameters can be accessed with get and set methods, this time with the parameter name as an additional argument to the method.

Consider, for example, a button which has just had the mouse moved over it. When the button's state changes it generates a new request for feedback, or `GELEvent`. This request contains the information shown in Figure 7.8.

```
Widget: Button
State: MouseOver
Event: MouseEnter
```

Figure 7.8 - The information held by a request for feedback when it is generated by the abstract widget behaviour of a button.

If the button uses two output modules, the button's feedback controller generates two copies of the request. Assuming one of the modules is called "Audio Module" the request passed to the module mapper for "Audio Module" would be as shown in Figure 7.9.

```
Widget: Button
State: MouseOver
Event: MouseEnter
Module: Audio Module
```

Figure 7.9 - The information held by the request for feedback when it is passed to the module mapper for the "Audio Module" output module.

The module mapper would then add any relevant pieces of output module specific information. In this case, assume the button requests a "Volume" of 50 and a "Style" of Jazz. Volume and Style are options which the output module has stated it understands. The request, which is now passed on to the rendering manager, would contain the information as shown in .

```
Widget: Button
State: MouseOver
Event: MouseEnter
Module: Audio Module
Parameters: Volume, 50
            Style, Jazz
```

Figure 7.10 - The request as it would be given to the rendering manager.

When it receives the request, the rendering manager would make any changes appropriate to the current context. Assuming that no changes are necessary the request would be passed onto the appropriate output module. The rendering manager knows which output module to pass the request to by querying the Module parameter.

7.4 Global Toolkit Objects

The toolkit contains two global components that allow widgets' feedback to be modified to ensure that the best use is made of available presentation resources. The *rendering manager* controls feedback requests made by individual widgets and the *control system* manages changes made to the widget presentation whether by the rendering manager, external sensors or another agent (whether it is the user or another piece of software). This section describes the implementation of these two components.

7.4.1 The Rendering Manager

The rendering manager's main responsibility is to manage the distribution of feedback requests to the appropriate output mechanism, adjusting these requests as appropriate. These adjustments may be made if there is insufficient resource available for the requests, the requests are unsuitable for the current context or if the requests being made will interfere with each other. By modifying the requests directly rather than forcing the source (i.e. the widgets) to change the requests being made has the advantage that the rendering manager does not need to store the preferred options of the widgets so that they can be restored when the reason for the adjustment no longer exists. It has the disadvantage, however, that the rendering manager has to calculate changes continuously. Currently, all the output modules available to the toolkit are stored in a specific directory and loaded into memory when the system starts up although they could not be loaded dynamically as required. References to the output modules are stored in a table in the rendering manager.

Thus far, one rule has been implemented for the rendering manager. This rule manages the audio feedback for multiple progress indicators. This rule was implemented because it attempts to resolve one of the issues that arose with the sonically-enhanced progress indicator described in Chapter 5. If a single progress indicator is requesting audio feedback then no change is required. If two progress indicators are requesting audio feedback, then the requests for audio feedback are modified to minimise their interference with each other. Finally, if three or more progress indicators are requesting audio feedback the only sound played is the sound indicating progress completion. The rule assumes that the audio feedback is being generated by a specific output mechanism. This is a realistic assumption as it is unlikely that different output mechanisms will conform to the same limitations. In this case, the output mechanism generates audio feedback using earcons which are played on a MIDI synthesiser. This output mechanism allows for three levels of fidelity for the different sounds generated. These three levels can correspond to either resource availability or resource suitability. Should there be, for example, insufficient MIDI channels available for the sounds being requested, the output mechanism will notify the control system of this resource shortage and consequently the widgets will be notified to request lower fidelity sounds. Similarly, if the sounds being requested interfere with each other the rendering manager lowers the requested fidelity for the sounds. For a progress indicator, a reduction in the fidelity of the sounds changes the number of sounds played. At the maximum fidelity, all the sounds described in Chapter 5 are played. At the medium fidelity the percentage completion and completion sounds are the only sounds played. Finally, at the lowest fidelity only the completion sound is played.

The implementation of this rule highlights some issues about how these rules should be handled. To allow the rule to be implemented it is necessary to ensure that the rendering manager is aware of how many progress bars are currently requesting audio feedback. The rendering manager therefore maintains a table of progress bars requesting audio feedback. The addition of a progress bar to the table can easily be achieved by checking if an event received by the rendering manager is for a progress indicator and uses the appropriate output mechanism. If this is the case all that is required is to check that the progress bar is not already in the table and, if not, to add it. By taking this approach rather than simply checking for a particular event (e.g. initialisation) the implementation allows for cases where the output module(s) being used by a widget are changed after this event has occurred.

Determining when a progress indicator has stopped requesting audio feedback is not as simple to do. One way would be to check for completion and exception events occurring. This, however, does not handle the situation where the output mechanisms being used are changed and the audio output mechanism has been removed from the widget. To allow for this situation, an extra field was added to the `GELEvent` class (Section 7.3.4) which stores all the output mechanisms currently being used by a widget. Thus, if an event is received for a progress indicator which is in the table of progress indicators requesting audio feedback, but the event does not specify that audio feedback should be used, the progress bar can be removed from the table. This, in combination with checking for completion or exception events, is still not sufficient to determine whether all the progress indicators in the table are still requesting audio feedback, however. If a progress indicator is simply destroyed (for example by closing the window containing the progress indicator) there is no indication of this passed through the toolkit. One way to detect this could be to monitor the number of events received for a progress indicator and should there be no events in a specified period of time assume the progress indicator is no longer active. This, however, is not a reliable technique and would not be very dynamic due to the delays inherent in using a timeout in this way. A second possibility would be to have a method in the `MProgressBar` class which notifies the toolkit when the progress indicator is destroyed. This method could either be called by the system (i.e. the Java Virtual Machine) or by the application destroying the progress indicator. The former approach is problematic because although Java does allow for such a method (the `finalize` method) it is not automatically called when the object is destroyed. Rather, it is called before the automatic garbage collection system reclaims the object [53] which occurs at an unspecified time after the object has been dereferenced. Furthermore, it is incumbent on the programmer to ensure that the object is correctly dereferenced when it is no longer required. The second approach is also unsuitable because this would entail changing the API of the progress indicator from the standard, Java API. Although the API has been modified slightly (i.e. from JProgressBar to MProgressBar) the change is a mechanical one and could be accomplished with a simple parser. The addition of a new method to be called when the object is dereferenced is not such a mechanical change and would require an understanding of the application code to identify the correct location for this method call. This problem is a consequence of the decision to use Java as the implementation language. Java does not require the programmer to deallocate variables when they are no longer required, but instead handles such deallocation automatically. Although this is often seen as being beneficial, reducing both the workload of the programmer and the opportunity for error, in this instance it has been problematic. If the toolkit had been implemented in a language, such as C++, where the programmer is forced to manually deallocate variables this problem would not have arisen[18].

As well as the aforementioned problem of determining how many progress indicators are requesting audio feedback at a particular moment in time, the implementation of rules in the rendering manager needs to be developed. Currently, rules are hard-coded into the rendering manager which clearly limits the flexibility of the system. The architecture presented in Section 6.3.3 describes these rules as being provided for the rendering manager by the control system. The control system receives these rules from an external source, whether it is a resource file or perhaps from an output module. The storage of these rules, therefore, is of no concern to the rendering manager. Once they have been retrieved, however, the rendering manager is

---

[18] Assuming, of course, that the programmer deallocates variables in a timely and correct manner!

required to implement them. This implies a mechanism which would allow this implementation to be achieved dynamically. This could be achieved by providing a mapping between the different elements of a rule and data structures that can be generated by the rendering manager. This information required by the rule described above can be summarised as shown in Table 7.1.

Output Module	Standard Earcon Module
Widget	Progress Indicator
Transformation Applied	Number progress indicators -> Fidelity

Table 7.1 - Information used by the rule controlling audio feedback used between multiple progress indicators

There are three components to this information: the output module(s) affected, the widget(s) the rule applies to and the transformation controlling the feedback. In this case, the output module is the "Standard Earcon Module", the widget is the progress indicator and the rule transforms the "fidelity" of the audio feedback according to the number of progress indicators requesting audio feedback. This mapping can be seen in Table 7.2.

Number Progress Indicators	Fidelity
0-1	High
2	Medium
>2	Low

Table 7.2 – The mapping describing the transformation of the fidelity of audio feedback according to the number of progress indicators requesting audio feedback.

It is simple to imagine an XML instance for this rule as shown in Figure 7.11. This rule conforms to the data type definition presented in Figure 6.2

```
<Rule>
  <OutputModule name="Standard Earcon Module">
  <Widget name="Progress Indicator">
  <Test name="Number Requests">
    <Condition>
      <Min value=0>
      <Max value=1>
      <Result>
        <Enumeration name="Fidelity", value="High">
      </Result>
    </Condition>
    <Condition>
      <Min value=2>
      <Max value=2>
      <Result>
        <Enumeration name="Fidelity", value="Medium">
      </Result>
    </Condition>
    <Condition>
      <Min value=3>
      <Max value=999>
      <Result>
        <Enumeration name="Fidelity", value="Low">
      </Result>
```

135

```
      </Condition>
    </Test>
  </Rule>
```

Figure 7.11 – XML instance of a rule controlling audio feedback between multiple progress indicators.

This specification could be transformed into code by the rendering manager with minimal effort due to the wide availability of XML parsers for Java. To ensure that the number of requests were handled correctly a set of known tests could be added to the rendering manager including "number requests". Other such tests could include "uses output module" or "has parameter". The appropriate data structures can be built upon the test required. In this case, a hashtable of progress indicators would be implemented. The case statement could straightforwardly be implemented as a case statement in Java. The case statement could take one of two forms. In the first form, as used in this example, the case statement tests the value against a range. This can be used to test output module parameters which are defined in terms of a range, such as size or volume, as well as for things like the number of requests being made. The second form is when the condition is a series of options such as output module used or a mutually exclusive output module option such as style or colour.

7.4.2 The Control System

The control system can be considered the heart of the toolkit. It manages the communication between all the major components of the system such as the rendering manager, the widgets, output modules and sensors as well as input from the users. It is also responsible for loading into memory any external components such as output modules and sensors. In this section, the implementation of this component is discussed.

As with the rendering manager, the control system object is initialised using a static initialiser and is accessed via a static method which provides a reference to the sole instance of this class. The primary role of initialisation is to restore any previously stored information describing the state of the system, such as the directories to look in for output modules and sensors. These external components are loaded by searching the specified directory and loading every jar file whose main class implements the appropriate interface. Although this means that many, potentially unused, output modules and sensors are loaded into memory, it simplifies toolkit startup. The existing mechanism is capable of loading individual jar files and consequently could be adapted to allow a more flexible and efficient technique to be applied.

Once an output module has been loaded, a reference to it is passed to the rendering manager, allowing the rendering manager to distribute feedback requests correctly. The control system only retains references to the different output modules' names and options, as these are all that is required to change widget settings. The control system does, however, retain a reference to active sensors as the control system is required to manage the use of these objects.

The control system is also required to maintain a list of all the widgets currently using the system. This is achieved by making the widgets register with the control system when they are constructed. When a widget registers with the control system, it uses a unique identifier which allows the widget to be manipulated by the control system via the widget's `MComponent` interface. The generation of this identifier, ensuring that is

both unique and yet readily identifiable by a user, is not trivial. The use of a unique code, for example an incremental number, whilst ensuring the uniqueness of the identifier does not assist the user in identifying which widget this code represents. Similarly, using something which the user can readily identify, for example, the widget's label if it has one, does not guarantee uniqueness. The current implementation employs a combination of these approaches. Every class of widget has a class variable incremented every time a new instance of that class is created. This number, in combination with a string describing the widget type (and the widget's label if it has one), is used to generate a widget's unique identifier. For example, the first button instantiated may have an identifier of "Button 1 – Ok" where "Ok" is the button's label. The second button instantiated may have an identifier of "Button 2 – Cancel" and so on. Progress indicators, which do not have labels would be identified as "Progress Indicator 1", "Progress Indicator 2" and so on. This technique, however, is still not ideal. A user may need to differentiate between, for example, "Button 7 – Ok" and "Button 8 – Ok". Also, although the identifiers are unique, it cannot be guaranteed that the same widgets will use the same identifiers across multiple sessions. This means that the saving and restoring of settings for the different widgets is impossible to achieve reliably. One possible solution to this problem would be to require user interface designers to specify a unique identifier for every widget used. As well as being time consuming and irrelevant to the designers of the user interface, this approach necessitates a change in the API of the toolkit meaning it is no longer using a similar API to the standard Swing widgets. Furthermore, this does not address the problem of dynamically generated widgets which would necessitate the designers to include a mechanism tracking their generation and naming them appropriately. A second, potentially more effective solution, would be to base the unique identifier for a widget upon the position of that widget in the application's widget hierarchy. This would ensure that the widgets used the same identifier in every session but it is unclear whether the information provided by the widget hierarchy could be effectively presented to a user allowing him/her to identify individual widgets appropriately.

The architecture specified in Section 6.3.1 shows a separate control user interface and control system. This control user interface is designed to allow a user (either an end user of an application built with the toolkit or, more likely, the designer of such an application) to specify the forms of presentation used by individual widgets. In the current implementation, however, the control user interface[19] is tightly coupled with the control system but still gives an example of how such a control user interface might work. The main window of the control panel is shown in Figure 7.12.

---

[19] The control ui was implemented by an MSc student as his project in the summer of 2000.
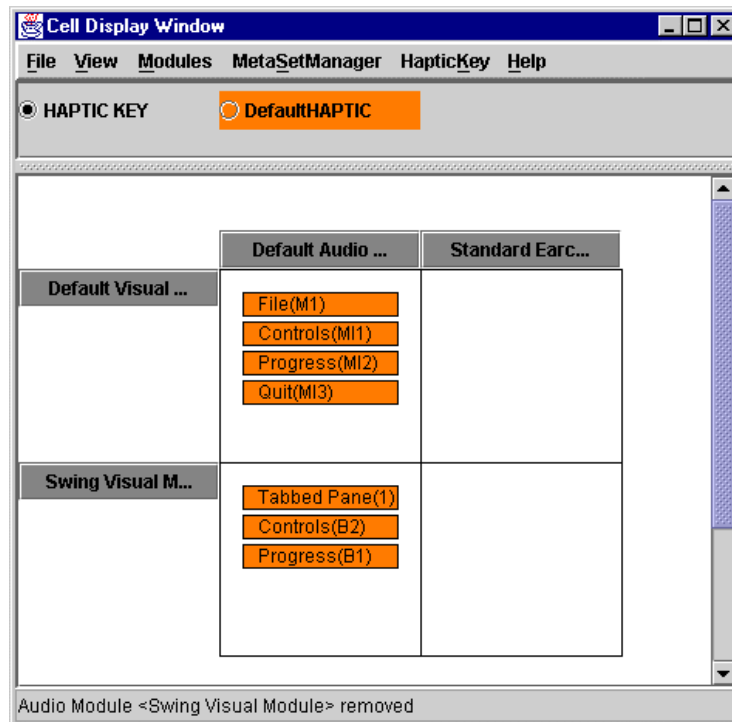
Figure 7.12 – The toolkit control panel. Each column in the table represents a different audio output module and each row a different visual output module. The widgets (represented as rectangles in the table's cells) can be dragged to different cells if the user chooses to use different output modules.

In this window, the output modules used by the different widgets can be changed. The main area of the window displays a table which contains all the widgets (which are shown as small rectangles containing a textual description of the widget). The rows in the table represent different visual output modules and the columns represent different audio output modules. Colour is used to represent the haptic output module used for each widget. The widgets can be grouped together to form sets of widgets which can be manipulated as a single widget would be. Widgets can be manipulated by dragging their on-screen representation into a different cell in the table, thus changing the output module(s) being used. A more detailed view of each cell of the table can be seen by double clicking it, which brings up a new window as shown in Figure 7.13.
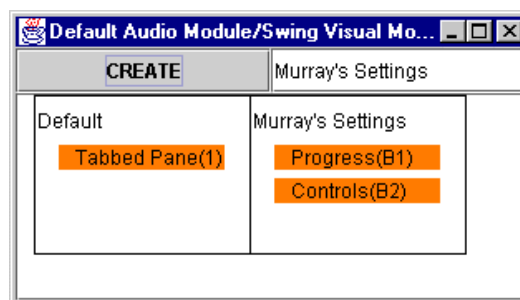
Figure 7.13 – The Cell Zoom window. This window allows the creation of sub-settings for a particular output module pair. Each cell in the table can have different settings for the pair of output modules represented by the table.

The window shown in Figure 7.13 allows widgets which share an output module to use different module-specific settings for that module. The main area in the window contains an array of cells which, as before,

contain widgets which can be dragged between cells. Each cell allows a user to set the options for the output modules differently. This is achieved by double clicking on a cell in the table, popping up a third window (Figure 7.14).
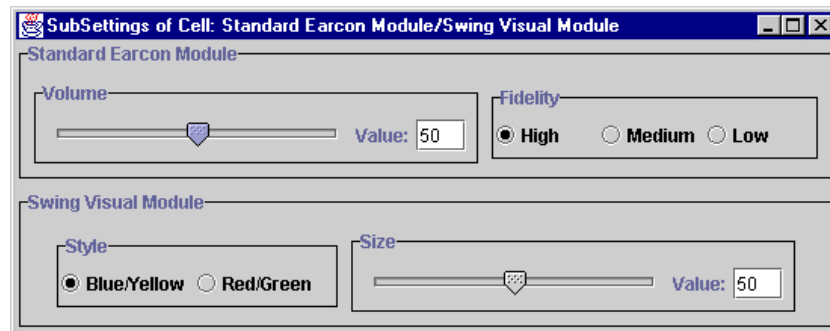
Figure 7.14 – The sub-settings window. This window allows a user to manipulate the settings for a particular pair of output modules.

This window contains two panels which allow a user to set the options for a particular pair of output modules. These panels are generated automatically by the control panel with a slider used to represent an option which is an integer range and a group of radio buttons if the option is a set of mutually exclusive choices.

7.5 External Toolkit Components

The toolkit relies upon different external components to provide concrete feedback for its widgets and information about the environment within which it is operating. This section discusses the implementation of these components: the output modules and sensors.

7.5.1 Output Modules

Because the output modules are external to the toolkit, the toolkit can have no influence over the way they are implemented. What the toolkit does provide, however, is a Java interface to which the output modules must conform. This interface is shown in Figure 7.15.

```
public interface OutputModule
{
        public void releaseResources();
        public void stopListening();
        public void startListening();
        public void processEvent(GELEvent e);
        public int getModality();
        public String getTitle();
        public Hashtable getOptions();
}
```

Figure 7.15 – The interface that all output modules must implement.

The three `get` methods in the interface allow the control system to interrogate the output modules. The `getTitle` method returns the unique name used to identify the output module. The `getModality` method returns the sensory modality the output module employs and the `getOptions` method returns a table of option names mapped to either an array of strings or a default bounded range model as appropriate. These three methods supply enough information to the control system for it to fully manage the use of the different output modules. The `processEvent` method accepts an abstract request for presentation which the output module must translate into concrete feedback. The `releaseResources` method is used to instruct the output module to release any resources it is using for presentation and is called prior to the output module being unloaded from memory.

The role of an output module is to translate requests for feedback from a widget into concrete feedback. How it does this is not specified by the toolkit. The requests received by the output module include the widget type, the current state of the widget, the transition it went through to get into this state. The requests also include output module-specific information such as, for example, the style of feedback or the colour to use. This is sufficient information for the output module to be able to determine the feedback to be generated. Also included in the request for feedback is a reference to the superclass of the widget. This will either be a Swing component which can have its properties (such as colour and size, for example) modified or it will be an extension of the `JPanel` class. In the latter case, the JPanel can be used to present the visual feedback requested by the widget. Clearly, this reference is not required by output modules that do not produce visual feedback. One issue that the designer of an output module will have to face is whether to build a stateless output module, i.e. one that does not build a model of the widgets it is providing feedback for, or a stateful output module. Clearly, the implementation of a stateless module is simpler but imposes some limitations on the way the presentation is managed. Consider, for example, a visual output module which modifies the presentation of the widgets according to their location on the screen. Unless the widgets fire off an event indicating that they have been moved (which none of the widgets implemented currently do) a stateless output module would not be able to provide such feedback. A stateful output module, on the other hand would be able to provide such feedback by monitoring the location of the widgets it has included in its model of the user interface and adjusting their size appropriately when they were moved. The advantages of a stateful output module may be counterbalanced by an increased implementation cost. Not only does a stateful output module have to build a model of the widgets it represents, but it must ensure that this model remains consistent with the states of the widgets.

Currently Implemented Output Modules

Several output modules have been implemented, both audio and visual, but they all work in a similar way. A series of case statements are traversed determining, initially, the type of widget requesting feedback and then the current state of the widget. In the different cases of this second case statement the requests for the feedback are translated into concrete feedback. The output modules store no information about the widgets as they are capable of generating the required feedback based upon the information provided by every request.

Two modules have been built which are used by the toolkit as its default audio and visual modules. These modules do not alter the rendering of the standard Swing widgets and can merely be considered as place

holders. The only exception is the default visual module which is required to provide a rendering for the progress indicator as the toolkit's progress indicator is not based upon the standard Swing widget. This is achieved by adding a standard Swing progress indicator to the blank panel upon (the `MWidgetPanel` object which is the super class of the toolkit progress indicator) which the widget is to be rendered. A second visual module has been created which modifies the standard Swing widgets by changing their colours according to their current state and size according to either user preference or screen size. It also renders the progress indicator differently according the requested size. In Figure 7.16 three examples of how buttons are rendered by this output module are given. In Figure 7.16(a) the mouse is not over any of the buttons and they have been given their default rendering. In Figure 7.16(b) the mouse has been moved over the progress button and so its colouring has been changed to highlight this. In Figure 7.16(c) the mouse button has been pressed inside the progress button and so its colouring has been changed once more. Notice also that the colouring of the controls button has been changed to reflect the fact that the mouse button has been pressed outside of its active area. Although these changes in the rendering of the button may appear to serve no real purpose, they are useful in highlighting the way the behaviour of the widgets have been exposed. Figure 7.17 shows how the rendering of the same progress indicator changes as its size is changed.

(a)
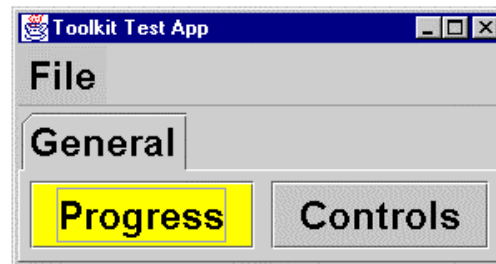
(b)

(c)

Figure 7.16 – Three toolkit windows giving examples of how the rendering of a button changes as the user interacts with it. In (a) the mouse is outside all the buttons and so they are all given their default rendering. In (b) the mouse is over the progress button and so it has been highlighted in yellow. In (c) the progress button has been pressed and its colours have been reversed. Also, the control button has changed colours to appear greyed out in response to the external (to it) mouse press.

141
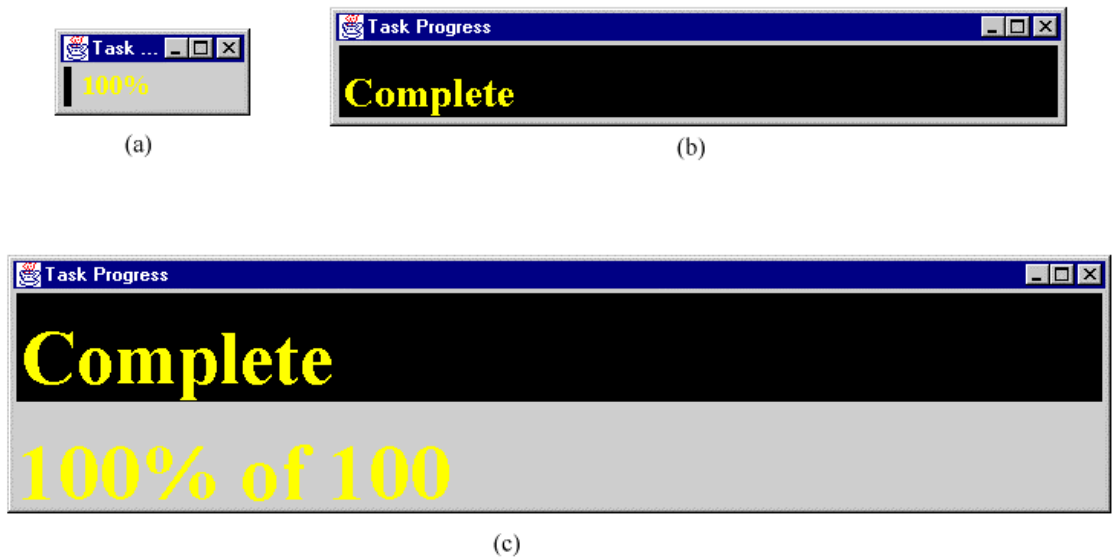
Figure 7.17 – Three different renderings of the same progress indicator as its size increases from the smallest (a) to the largest (a).

A second, audio output module has also been implemented. This module provides audio feedback using earcons designed by Brewster *et al.* (e.g. [14]) as well the sounds described in Chapter 5. This module allows the volume of the audio feedback to be modified and the number of the sounds produced to be changed. The former may be done in response to user preference or perhaps the ambient volume of the surrounding environment. The latter may be done if some of the sounds are interfering with each other (as described in 5.8.1) or if there is a lack of audio resource to produce the sounds.

7.5.2 Sensors

As with output modules, the toolkit does not have any say in the way sensors are implemented but does impose an interface which must be implemented (Figure 7.18).

```
public interface Sensor
{
        public String getSensorName();
        public String getParameterName();
        public void setSensing(boolean sensing);
        public boolean sensing();
}
```

Figure 7.18 – The interface that all sensors must implement.

This interface consists of only four methods. The two methods `getSensorName` and `getParameterName` allow the control system to interrogate sensors about their name (which should be unique) and the parameter they effect. This parameter is one which an output module uses to modify the feedback, for example size or volume. As the models of the presentational modalities are developed, the toolkit will provide a list of supported parameters but currently, they are provided by the output modules. The

`setSensing` method allows the sensor to be switched on and off and the sensing method allows the control system to determine whether a sensor is currently active.

Should a change in the context being monitored by the sensor require that a change is made in the rendering of the toolkit's widgets, the sensor passes a modifier into the control system which is then distributed to all the widgets. This modifier takes the form of a pair: the name of the parameter being modified and the value of the modifier between 0 and 1. This modifier is then included in the request for feedback and when an output module receives this request it modifies its rendering appropriately. Consider, for example, a button which uses an audio output module for presentation. This output module understands how to handle a parameter called volume. A sensor is being used which monitors the ambient volume of the environment, causing the volume of any audio feedback to be modified when appropriate. If the ambient volume is low the server sets the volume modifier to be, for example, 0.1. This value is included with any requests for audio feedback and causes the output module to adjust the volume of the audio feedback accordingly. In this case the volume of the audio feedback is reduced. If the sensor detects an increase in the ambient volume of the surroundings it will increase the modifier value, causing the volume of any audio feedback to be increased.

Currently Implemented Sensors

Thus far, two sensors have been implemented. One, implemented by Brewster *et al.* [26], implemented a volume control sensor which detected the ambient volume of the surrounding environment causing the volume of the audio feedback used by the toolkit's widgets to be adjusted accordingly. This sensor was developed prior to the development of the sensor interface described above but was easily modified to work with the current implementation of the toolkit. By taking audio samples from the environment at regular intervals, the ambient volume of the surroundings can be determined and, if a change is detected, the volume of the audio feedback is changed. A final year undergraduate student is currently developing this work so that it works on a client-server basis for many machines, perhaps operating in a shared environment. The client would be the sensor used by the toolkit, but rather than modifying the audio feedback for a single machine, it would send the volume of its immediate environment to a central server that would then manage the volume on all its client machines. This would be achieved by the server sending the sensor objects a volume modifier value that the sensor would apply to local toolkit objects. The value set by the server could simply be a global value based on the average ambient volume detected by the clients which is applied to all clients, or perhaps the server would employ a more complex algorithm that would take into account the locations of the different clients and their local ambient volume. This could be a useful approach to take in a potentially noisy shared environment as the different machines would no longer be competing with each other for a limited resource (i.e. the audio space), but would share that resource by employing a server to manage its use in a similar way to the way the ENO server [10] manages a virtual audio space.
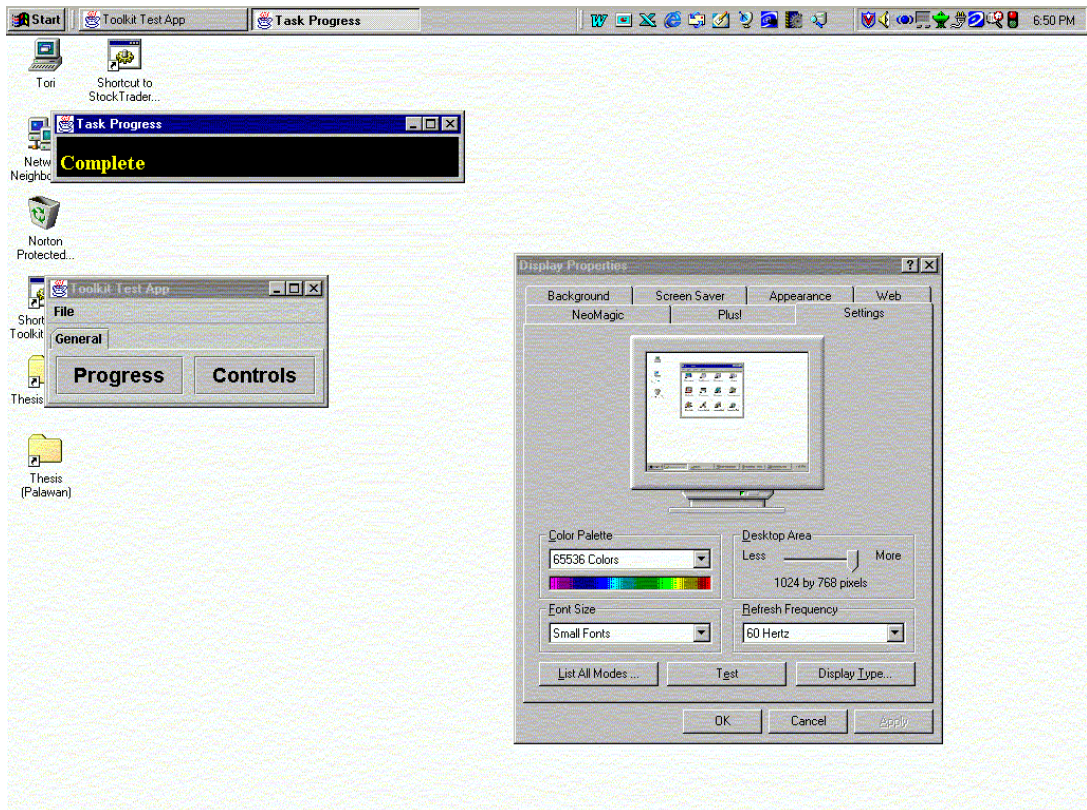
Figure 7.19 – Two toolkit windows with the Windows NT™ display control panel at a screen resolution of 1024x768 pixels.

The second sensor implemented thus far implements the sensor interface and modifies the size of the graphical feedback of the widgets. As with the volume control sensor, this allows a limited resource, the screen real estate, to be shared amongst different, competing widgets. Figure 7.19 shows two windows with toolkit widgets at a screen resolution of 1024x768 pixels. Also shown is the standard Windows NT display control panel.

Figure 7.20 shows the same windows, but this time the screen resolution is 640x480 pixels. The windows using toolkit widgets have remained at about the same proportional size as before, but the display control panel which does not use toolkit widgets now dominates the screen. It should also be noticed that the progress indicator has changed its representation from a horizontal bar containing text to a small vertical bar with text adjacent to utilise the smaller available space more effectively. Although there appears to be wasted space to the right of the text in the window containing the progress indicator, this is an artefact of Java which limits the minimum width of windows.
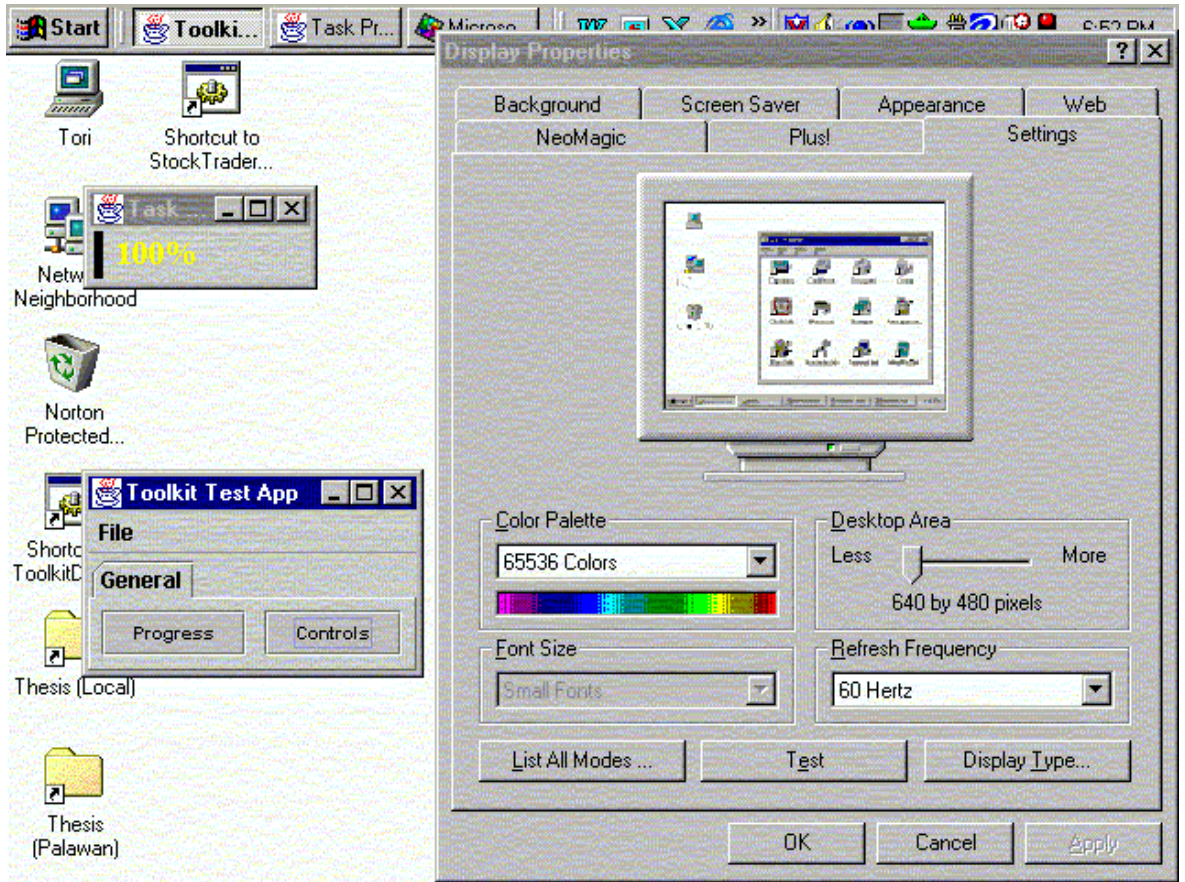
Figure 7.20 – The same windows as in Figure 7.19 but this time at a screen resolution of 640x480 pixels.

7.5.3 Worked Example of a Request Passing Through the Toolkit

In this section a brief worked example is given, showing how the various toolkit components combine. The example is based upon a button that uses two output modules, one for audio feedback and one for visual feedback. The audio output module is capable of handling different volumes and styles for presentation. In this case, the volume for the button is set to 50 and the style to "Jazz". The visual output module is capable of handling different sizes and this is set to 75 for the button. Because the visual output module is only capable of providing one form, or style, of feedback it does not specify a "Style" option and consequently such a parameter is not passed in feedback requests.

In its default state, with the cursor outside the area of the button, the button is drawn as shown. No sounds are played.

The mouse enters the button. This event is passed to the abstract widget behaviour, which is in a state that can accept this event. The event is translated into a request for feedback.
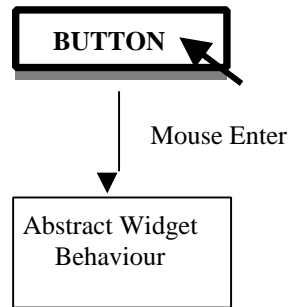
The request is passed to the feedback controller. Two requests are generated, one each for audio feedback and visual feedback. Each request is passed onto the appropriate module mapper.
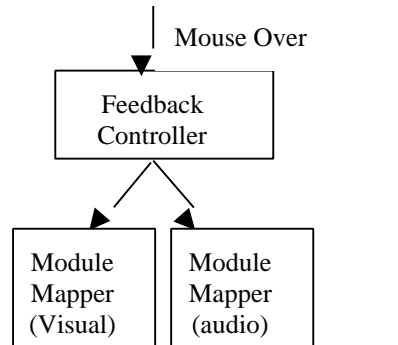
Each modality mapper modifies the event in accordance with user preferences set in the control panel. In this case, a size of 75 is applied to the graphical request and the style 'Jazz' and the volume 50 are applied to the audio request. Each request is passed onto the rendering manager.
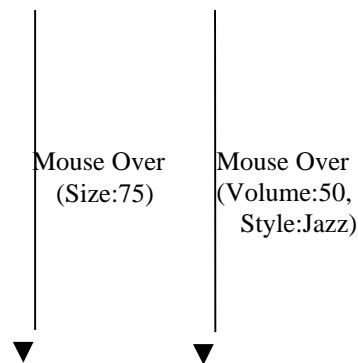
The rendering manager checks for any problems with the suitability and availability of the presentational resources for these requests. In this case there are no problems so the requests are passed onto the appropriate output modules.
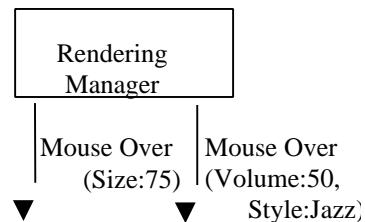
Each output module receives the request and translates the request into concrete output. The visual module draws the button yellow and the audio module plays a persistent tone at a low volume in a Jazz style.
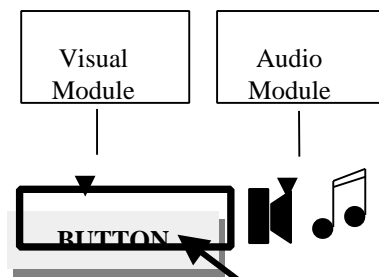
7.5.4 A Recipe for Implementing Widgets

Although every widget is different, the same procedure can be followed to implement every widget. This section discusses this procedure and uses the examples of some of the currently implemented widgets to describe how some of the potential pitfalls can be overcome.

Define the Behaviour of the Widget

The first step in developing a widget is defining its behaviour in terms of a statechart or state table. Although this may seem a trivial step, it is in fact the most important stage if the widget is to be implemented successfully. Indeed, Myers [86] reports the difficulties Cardelli [34] found in developing widgets (or primitives): "The primitives never seem complex in principle, but the programs that implement them are surprisingly intricate". The behaviour of the widget is defined in terms of the method of input to it. This will typically be the mouse for widgets that can be directly manipulated by a user (e.g. buttons) but need not be (progress indicators, for example, only receive data from the tasks whose states they represent). It is important not to limit this behaviour to that which is currently thought of as the typical behaviour for that widget, but rather to make no assumptions about which behaviours are relevant and which are not. For example, the behaviour of a button as described in Section 7.3.3 includes events such as mouse presses external to the widget which, although they affect the behaviour of the widget, are not normally exposed to a programmer. Although this would mean ideally every mouse event would be exposed, this is neither realistic nor worthwhile. Mouse movement, for example, could be considered relevant to a widget. Perhaps, for example, the widget could track the location of the mouse much like the XEyes (e.g. [97]) application. Doing this for every widget, however, could potentially generate many events which could lead to a degradation in the performance of the system with no real benefit gained in return. If, however, such events were required to fully capture the behaviour of a widget, the architecture does support their capture. Another reason for being careful about which events to capture is caused by a limitation in the existing toolkit implementation. Because the toolkit uses the standard Swing widgets as its basis, not all events are exposed to the toolkit. The behaviour of a slider, for example, may include the concept of the current location of the mouse with respect to the thumb wheel. The standard Java widget, JSlider, does not expose such events with the `mouseOver` event including no locational information other than the low level co-ordinates. If it was felt that these locational mouse over events were relevant, the slider widget could not be built using the current architecture but would require this architecture to be extended. A possible extension to the toolkit architecture which would enable such a widget to be built is discussed in Section 9.4.2, but, for the purposes of this thesis, it was decided that the prototype of the toolkit being implemented would not require this extension. The set of widgets that it was possible to build without this extension would be sufficient to determine the feasibility of the approach being taken.

Implement the Behaviour of the Widget

Once the behaviour of the widget has been satisfactorily designed, it should be encoded into the statechart of the widget. The widget is based upon the instantiation of two classes: the `MWidget` class and the `WidgetStateChart` class. The `MWidget` class is the main class for the widget, being the reference object used by the application programmer. This class extends either the standard Swing widget or the

MWidgetPanel. If the widget is based upon the corresponding Swing widget (to take advantage of the input event handling, for example) then it should extend the Swing widget. This means that the toolkit's widget will inherit all the standard widget's methods, thus minimising the implementation required, although if necessary some of these methods can be overridden. If the widget does not require the input handling mechanism provided by Swing it can extend MWidgetPanel. This class extends the standard Java JPanel class but includes methods that allow primitive painting actions to be performed. By providing a panel which accepts painting actions (implemented as objects which encapsulate a single paint action) and, crucially, stores these objects so they can be repainted when necessary, the toolkit provides a mechanism which enables output modules to generate a persistent graphical rendering for a widget without any need to store information about this rendering itself, thus enabling the output module to be stateless. Should the widget need to be repainted after being maximised, for example, the output module does not need to handle this redrawing; it is all managed by the MWidgetPanel. Because the MWidgetPanel extends a standard Swing widget itself (the JPanel) it can be used within Java programs as any other Swing widget. The mechanisms for adding and removing a widget from an interface, for example, are therefore the same for a toolkit widget as any standard Swing widget.

The WidgetStateChart object defines the behaviour of the widget by translating the input to the widget into abstract requests for feedback. It is built using generic StateNode objects which filter incoming events and generate the new requests. These requests are passed onto the feedback manager and subsequently to the rest of the toolkit. The MWidget class for all the widgets are built using a similar template (Figure 7.21).

```
// Constructors for Widget
// Initialise Widget
// MComponent Methods
// Mouse handling inner class
// Override appropriate super methods
```

Figure 7.21 – Template for the construction of a new widget. All widgets should implement these different components.

All the constructors for the standard Swing widget must either be overridden (if the MWidget extends the standard Swing widget) or implemented (if the MWidget implements the MWidgetPanel). If the constructors are being overriden the new constructors call the appropriate constructor from the superclass and then the initialise widget method. Otherwise, the constructor must initialise the widget's state before the calling the initialise widget method. The constructors for the progress indicator, for example, extract information about the range over which the task is to be monitored and store these values in instance variables. The initialise widget method is required to perform several tasks. It instantiates all the widget's internal objects, such as the statechart and feedback manager, and registers the widget with the control system. It is also responsible for initiating the listening mechanisms used by the widget. There are four groups of listeners associated with each widget. One listener is associated with the actual Swing widget. This listener is typically defined as an inner class to the widget. A second listener, a key listener, listens for a key press combination which is associated with the control panel. When this key chord is detected the control panel is displayed. This is necessary because it is unlikely that applications modified from standard Swing applications to use the toolkit will have a button or menu option to allow the user to display the control user

interface. In addition to these two listeners, each widget has two lists of additional listeners: private and public listeners. Private listeners are internal toolkit listeners (either the statechart or the external event listener) whilst the public listeners are listeners added to the widget by external applications. When a mouse event is received by the listener associated with the Swing widget, the listener distributes the event first to the private listeners and thereafter to the public listeners. The reason these listeners are not directly associated with the Swing widget is twofold. Firstly, it ensures that the private listeners can be guaranteed to be given the event before the public listeners, allowing the feedback to maintain pace with the user interaction. Secondly, it allows management of the widget behaviour. Buttons, for example, are typically defined to behave such that they are only selected when the mouse is released, not when it is pressed. If, for whatever reason, it was decided to change this behaviour so that the button is selected when the mouse is pressed, this could easily be achieved by changing when the action event is passed to the appropriate listener. Events such as `ActionEvents` are not atomic events, but rather are high level events occurring after a sequence of atomic events. In this case, to change the behaviour of the button, the `ActionEvent` is no longer generated after a mouse release, but after a mouse press.

The final components of a widget that need to be implemented are any methods of the superclass that need to be overriden. Such methods include methods that allow the addition or removal of mouse listeners or methods that affect the state of the widget as defined by the widget's statechart. The add/remove listener methods need to be overriden because any listeners added to the widget are added to the list of listeners stored by the MWidget rather than listening directly to the Swing widget itself. An example of a method that affects the state of the widget, and consequently needs to be overriden is the `setEnabled` method which allows the widget to be enabled and disabled.

Implementing the Widget's Presentation

The presentation of the widget is defined in the output modules used by the toolkit. If a new widget is defined, this addition will have to be reflected either by creating a new output module or adapting an existing output module. In either case, the changes required are simple. The output module receives requests for feedback and by querying this request determines the widget it is for and either the current state of the widget, the event that caused the state to change or perhaps both. Once the output module has ascertained this information it must determine the rendering of the presentation, perhaps using information about the current state of the widget such as, for example, the current value of a progress indicator.

Some Possible Variations on this Recipe

Whilst this standard approach can serve as a good basis for making most widgets, there will be instances where this recipe needs to be altered. One such possibility is where the widget requires the definition of a new event. A radio button, for example, behaves differently if it is a member of a button group. If it is a member of a group, then, once it has been selected, it can only be deselected when a different button in the group is selected, if it is not in a group, it can be deselected simply by being "selected" again. One way to implement this would be to define two separate behaviours for radio buttons, one for buttons in a group and one for buttons not in a group. A neater solution, however, would be to define a single behaviour

encapsulating both these forms of behaviour. The relevant portion of this behaviour can be seen in Figure 7.22.

State Descriptions

State	Mouse Location	Mouse Pressed	Mouse Press Location	Button State
A	Over Button	Yes	Inside Button	Selected
B	Over Button	No	-	Unselected
C	Over Button	No	-	Selected
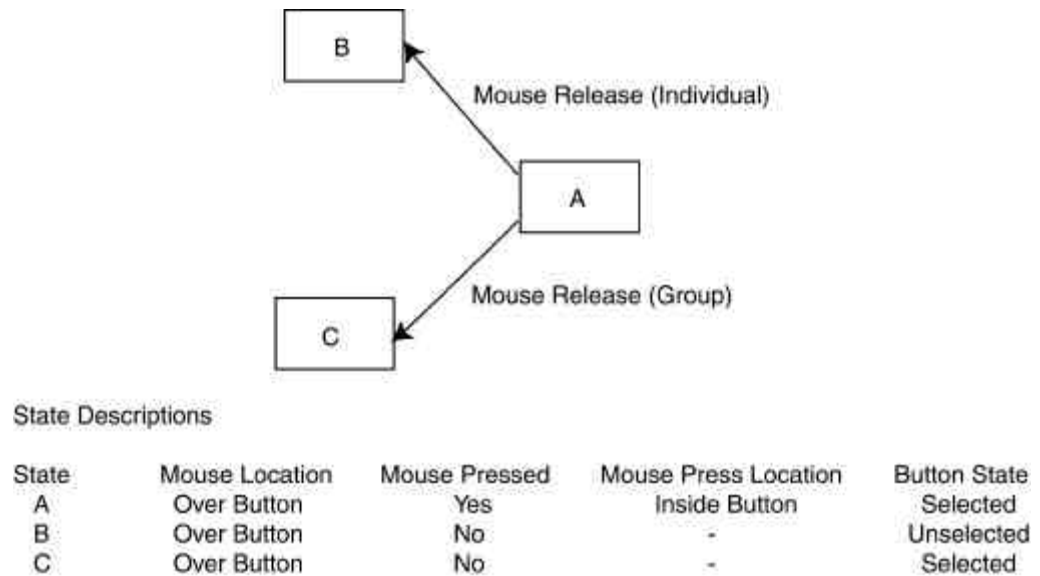
Figure 7.22 – Partial statechart for a radio button showing how the behaviour for a radio button differs dependent upon whether the buttons is a member of a button group or not.

This statechart captures the differences in behaviour between buttons depending upon membership in a button group. State A represents the state of a selected radio button that has been pressed, but not released. When the mouse button is released, the next state is dependent upon whether the button is in a group or not. If it is, the next state will be state C, otherwise the next state will be state B. State node A determines which is the appropriate new state based upon the release event it receives. This, therefore, implies that the object supplying the events must be able to supply the appropriate event. This object is the `MRadioButton` which generates the appropriate event when it receives a standard release event. The new event (for example, group button selection) needs to be defined in the toolkit and correctly handled by the widget object. This serves to highlight that although the behaviour of the widget is primarily defined by the statechart object, this object is driven by events provided by the widget object. Although in many cases the widget merely passes on the events it receives to the statechart, in some instances it must decide the appropriate event to pass on. Another example of how the behaviour of a widget is shared between the widget object and its statechart can be found in the generation of `ActionEvents` by buttons. Although it would be possible to change the behaviour of a button so that it is selected when it is pressed rather than released, in reality the button would still logically be selected upon the release unless the actual widget object was changed to generate an `ActionEvent` upon the mouse press rather than the release.

A second possible reason for changing the way a widget is implemented is that the widget is in fact constructed from several sub-widgets. Such widgets include menus and groups of radio buttons. In this case, there is a logical grouping widget (the menu or the button group) and its member sub-widgets (the menu items or buttons). This leads to several issues regarding the implementation of the widget. The first of these is the communication between the sub-widgets. As events occurring to one sub-widget may well effect the state

of other sub-widget(s) it is important that they can communicate with each other. For example when a radio button in a group is selected it is necessary for this event to be transmitted to all the other buttons in the group so that if another one is currently selected it can deselect itself. A second issue is the control of the feedback for the different sub-widgets. Should it be possible, for example, to change the presentation of a single menu item, leaving it inconsistent with the rest of the menu items in that menu, or should the changes in presentation be controlled at the group level via the menu widget? Either of these possibilities can easily be handled by the toolkit. If the individual sub-widgets are to have their presentation modified then they should be registered with the control system. If the changes are to be made at the group level then only the group widget should be registered with the control system and any changes made should then be propagated through the entire group by the group widget.

Chapter 8: An Evaluation of How Well the Toolkit Meets its Requirements

In Chapter 6 a design for a toolkit of multimodal, resource-sensitive widgets was described. In Chapter 7 an implementation of the toolkit was described. This chapter discusses how well this implementation satisfies the requirements of the design. The requirements for the toolkit are listed in Appendix D.2.

8.1 The Use of Multiple Modalities

In Section 6.2.1 the first requirement, describing how the toolkit should use additional modalities, was discussed. This requirement stated that the toolkit should expose the full behaviour of its widgets allowing a designer to add new forms of feedback to a widget to complement the existing feedback or perhaps replace the existing feedback. This section discusses how well the implementation of the toolkit fulfils this requirement.

The extent to which the full behaviour of the widgets is exposed can easily be demonstrated by building a simple output module whose only form of presentation is displaying a string enumerating the current state of a widget. Such an output module was built for a button and provides the output shown in Figure 8.1. Figure 7.6 shows the statechart specifying the behaviour of a button. This behaviour has 8 states, 7 of which are shown in Figure 7.6 (the 8th state, State H – disabled which is programmatically generated as opposed to user generated, is not shown). As can be seen in Figure 8.1 the full behaviour of the widget (which is described in terms of the mouse location, whether the mouse button is pressed or not and the location of the mouse press if appropriate) is fully exposed.

```
C:\Program Files\Metrowerks\CodeWarrior\(Helper Apps)\runjava.exe
State A.     Mouse: Outside button    Pressed: No       Mouse Press Location: N/A
State D.     Mouse: Outside button    Pressed: Yes  Mouse Press Location: Outside button
State E.     Mouse: Inside button     Pressed: Yes  Mouse Press Location: Outside button
State B.     Mouse: Inside button     Pressed: No       Mouse Press Location: N/A
State C.     Mouse: Inside button     Pressed: Yes  Mouse Press Location: Inside button
State F.     Mouse: Outside button    Pressed: Yes  Mouse Press Location: Inside button
State C.     Mouse: Inside button     Pressed: Yes  Mouse Press Location: Inside button
State G.     Button Selected
State B.     Mouse: Inside button     Pressed: No       Mouse Press Location: N/A
State A.     Mouse: Outside button    Pressed: No       Mouse Press Location: N/A
```

Figure 8.1 The presentation generated by an output module which simply sends a string describing the current state of the widgets to standard output.

The second requirement was that it should be possible to change the presentation of the widgets, either by supplementing the existing feedback with an additional modality or replacing the existing feedback with an alternative form. Figure 8.2 shows a progress indicator widget presented using two different visual output modules. In Figure 8.2(a) the progress indicator is presented using the standard JProgressBar widget. In Figure 8.2(b) the progress indicator is presented using Java graphics primitives to produce a vertical progress indicator with textual feedback adjacent. The current implementation of the toolkit does impose some restrictions on the presentation of most widgets due to their reliance on Swing mechanisms. Section 9.4.2 suggests how this reliance may be removed, allowing the presentation to be modified more freely.

(a)   (b)

Figure 8.2 The same widget using two different forms of visual presentation. In (a) the widget is presented using the standard Swing widget. In (b) the widget is presented using Java graphics primitives .

It is also possible to supplement the existing feedback with new forms of feedback. Output modules which use earcons to provide audio feedback have been built and can be added to the visual feedback shown above. Furthermore, the feedback shown in Figure 8.1 was presented in addition to the graphical presentation of the widget. The simple examples described above show that the requirement to expose the full behaviour of the widgets has been fulfilled and that this does allow the presentation of the widgets to be freely modified.

8.2 Controlling Intra-Modality Clashes of Feedback

Section 6.2.2 discusses two further requirements of the toolkit: that it should be able to handle any potential clashes of feedback between widgets using the same output modality and that it should be able to monitor the presentation of a widget over time and adjust it if necessary. A third requirement which arises as a consequence of these requirements is that because the forms of feedback being used are not fixed, it should be possible to add to or change the rules that manage the these potential clashes. This section describes how well the implementation of the toolkit fulfils these requirements.

The first requirement to be met was that the toolkit should manage any potential interference between widgets using the same output modality. An example of such a clash is the possibility of the feedback from two audio progress indicators interfering with each other. To counter this, a rule which tracks the number of progress indicators requesting audio feedback was defined. As the number of progress indicators increases the rule alters the feedback requested to ensure that there is no interference between the different widgets (for a full description of the implementation of the rule see Section 7.4.1). Thus, the first requirement described above has been met by the toolkit.

The second requirement stated that the toolkit should be able to monitor the presentation of the widgets over time, adjusting the presentation as appropriate. Although this requirement was not explicitly implemented, the mechanism used was shown to be effective as described above. The rendering manager, as a component that receives all the requests for feedback is able to build a model of the requests being made and consequently could monitor the requests made by different widgets over time and, using rules as described above, modify these requests as appropriate.

The third requirement stated that it should be possible to add to, or amend, the set of rules that control these potential clashes. The rule described above, however, was hard-coded into the system and no means of adding new rules is currently implemented. The toolkit, therefore does not meet this requirement. Section 7.4.2 does, however, outline how this requirement could be met by the toolkit. This change does not require an architectural change to the toolkit, merely an enhancement of the control system to read in the rules from a resource file and an enhancement of the rendering manager so that it can use the rules provided by the control system instead of requiring them to be hard-coded. Although not entirely implemented, this shows that the requirement to be able to control feedback from different widgets using the same modality is supported by the toolkit.

8.3 Controlling the Use of a Modality

Section 6.2.3 highlighted two requirements of the toolkit with regard to the control of the use of different modalities: the toolkit should be able to adapt the presentation of the widgets according to the availability of

the presentational resources used to generate the feedback and the suitability of the modality given the current context. An example of a presentation resource that could have limited availability is the visual display on a mobile device. Clearly, such a device is not going to have as much of this form of presentational resource as a desktop machine with a large monitor. An example of an output modality that may vary in suitability is audio feedback. In a very loud environment audio feedback is clearly unsuitable; similarly in a very quiet environment loud audio feedback will probably be unsuitable. Thus, the suitability of a modality does not merely determine whether this modality should be used, but also gives an indication about how this modality should be used. As a consequence of these requirements, a third requirement presented was that the toolkit should be able to sense the current context to determine the availability or suitability of a resource. It should be freely possible to add new sensors to the toolkit dependent upon its expected use.

The way the toolkit manages the availability of a resource has been demonstrated by the implementation of a simple sensor which detects the current screen size. If the screen size is reduced, the sensor detects this change and informs the toolkit so that the visual presentation of the widgets can be changed appropriately. Examples of this working can be seen in Section 7.5.2. Figure 7.19 shows two toolkit windows and the display control panel for Windows NT at a screen size of 1024x768 pixels. Figure 7.20 shows the same three windows at a screen size of 640x480 pixels. The toolkit windows have remained at approximately the same proportional size whereas the display control panel's proportional size has greatly increased at the lower resolution. Thus, the toolkit has been shown to meet the first requirement.

The way the toolkit manages the suitability of a resource has been demonstrated in a similar fashion. A sensor which detects the ambient volume of the environment was implemented by Brewster *et al.* [26]. When this sensor detected a change in the ambient volume, the audio feedback of the toolkit was adjusted appropriately. If the ambient volume increased, the volume of the audio feedback was increased to ensure that it would remain discernible. Similarly, if the ambient volume decreased, the volume of the audio feedback was decreased to prevent it becoming intrusive. Thus, the toolkit has been shown to meet the second requirement.

Section 7.4.2 describes how the control system loads the sensors being used by the toolkit from a specified directory when the toolkit is initialised. This is achieved using the Java `URLClassLoader` class which allows code to be loaded from a specified location. In this case, every jar file which has a class conforming to the `Sensor` interface (specified in Section 7.5.2) and is located in a specified directory is loaded when the toolkit is initialised. The toolkit, therefore, fulfils the third requirement described above.

8.4 End-User Control of Feedback

Section 6.2.4 discusses the requirement that end-users of applications built and being built with the toolkit should have control over the presentation of the widgets. A second requirement extracted from this initial one states that, because different groups of users may have different needs, the user interface that allows this control should be easily changed. Section 7.4.2 describes the implementation of a control panel which allows

users control over the presentation of the toolkit's widgets. This control includes the ability to choose which output modules individual widgets use for their presentation as well as control over the sub-settings of these output modules for individual widgets. The toolkit therefore allows end-users to control the presentation of the toolkit's widgets. The control panel, however, is tightly coupled to the control system meaning that the second requirement above has not been fulfilled. Section 7.4.2 discusses how this may be achieved without making any architectural changes to the toolkit, indicating that the modularisation necessitated by the requirement would be possible.

8.5 Software Engineering Considerations

Sections 6.2.5 and 6.2.6 discussed two software engineering requirements of the toolkit: it should be possible to group widgets together and the toolkit's API should conform to an existing API minimising the cost to engineers who use the toolkit to build interfaces. The first requirement can be considered in three distinct ways. One reason for grouping widgets together is to ensure conformity in their presentation. For example, it may well be desirable to give a group of radio buttons a common colour scheme. The toolkit supports this by allowing the creation of a meta-widget[20] which can be passed to the control system but does not generate any requests for feedback. Because it is added to the control system, the meta-widget can be given the information regarding the output modules and output module options that the group's widgets are to use. As the meta-widget will have references to all the group's widgets it will be able to pass this information on using the `MComponent` interface methods. A second reason is to allow widgets in the group to provide information about their relative location. For example, the feedback given for a menu item may well depend upon its location within the menu. As before, the meta-widget can inform the sub-widgets of this information using the `MComponent` interface, setting, for example, the menu location as a parameter using the `setPreference` method of `MComponent`. This can be extended to allow general state information of the different sub-widgets to be shared when necessary. For example, a group of radio-buttons can be considered as a meta-widget consisting of multiple individual radio-buttons. The grouping is necessary here to allow communication to take place between the different sub-widgets to ensure the mutual exclusivity of the individual radio-buttons. The use of statecharts to drive the behaviour of the widgets means that all the widgets in a group can be driven concurrently. In this case, for example, when a radio-button in the group is selected the `groupSelection` event is passed to all the individual radio-buttons in the group ensuring the mutual-exclusivity of the group. These examples show that the toolkit meets the first requirement described above.

The second requirement described above can be demonstrated to have been met by the toolkit. The toolkit's widgets conform exactly to the API of the standard Swing widget's API. The only difference in the API is the name of the constructors: the toolkit's start with 'M' whereas the standard Swing widgets start with 'J'. Clearly, the use of the toolkit necessitates the import of an additional package, the MM_Toolkit package, as

---

[20] Meta-widget was the name given to a logical widget which enabled a group of widgets which shared some state to be grouped together.

well. These two differences are straightforward and therefore do not greatly add to the cost of developing an interface with the toolkit. Furthermore, because the changes required to include toolkit widgets can be described in mechanical terms - import the MM_Toolkit package and replace `JWidget` constructors with `MWidget` constructors - it is possible to automate the conversion of existing source code so that it uses the toolkit's widgets. This clearly reduces the cost of developing applications using the toolkit's widgets even further.

An implementation[21] of such a source code converter has been successfully completed proving the feasibility of such an approach. This approach could be extended to the conversion of byte-code, meaning that it would be possible to modify existing, compiled applications so that they use the toolkit's widgets.

8.6 A Widget Designer's Experience of Using the Toolkit

In the period between October 2000 and March 2001 Scott Greig, a final year computing science undergraduate, built more widgets for the toolkit as his final year project [64]. Scott implemented radio buttons, which could be used singly or in a group, and a slider. At the end of his project Scott was interviewed to gauge how he found working with the toolkit.

The initial impression Scott had of the toolkit was of its complexity and he therefore found it very hard to get started. This was largely due to the absence of documentation clearly specifying the architecture of the toolkit with regard to the steps required to build new widgets. This has now been rectified with the recipe for making widgets presented in Section 7.5.3. Once this initial hurdle was overcome, however, he found the toolkit to be very straightforward to use, with the "engine" of the toolkit (the generic components of the toolkit that form the basis for the statecharts and handle the communication of the requests from the widget to the output module) proving to be robust.

The implementation of the behaviour for the widgets, although the mechanism was simple, proved to be quite complex. The specification of the widgets' behaviour was quite hard to build due to their complexity. This confirms Cardelli's observation that "The primitives never seem complex in principle, but the programs that implement them are surprisingly intricate" [34]. Furthermore, the implementation of this behaviour, although mechanical in nature, proved to be error prone. This difficulty could be eased by providing a tool which allows the direct manipulation of a visual representation of the statechart specifying the widget's behaviour. As well as providing an aid to the creation of the toolkit, it would enable the automatic generation of the implementation of the statechart.

---

[21] The source code converter was implemented by an undergraduate student employed over the summer of 2000.

The definition of the presentation of the widgets, however, was found to be straightforward due to the separation of presentation and behaviour. Information provided by the widget's requests for feedback was found to be sufficient to provide the appropriate presentation for the different widgets, although it was envisaged that if a situation arose where the same state could be arrived at due to the same event but from different previous states then the information provided would not be sufficient to produce two distinct forms of presentation. One solution to this would be to include the previous state in the request for presentation but this limited solution may not fulfil all requirements. A stateful output module could maintain a history of events and states if required but a stateless output module would require this history to be passed in with request for feedback, perhaps impacting on the performance of the system. Thus, if such information is required to provide appropriate feedback it is probably best maintained by a stateful output module.

One other issue that arose with the toolkit was the difficulty in debugging that was experienced. This arose because of the difficulty in tracking the events through the toolkit. It was sometimes hard to determine the current state of the widget making it difficult to track down the source of the error. A solution to this problem would be to create a tool which could highlight the current state of the widget and display the events received by the widget. This tool could be extended to show the path of the requests for feedback as they are passed through the different components of the toolkit.

Overall, the toolkit was found to be effective in allowing the design of new widgets with the presentation of the widgets, especially, simple to implement. The behaviour of the widgets, however, was harder to implement. This was partly due to the inherent complexity of this behaviour and partly due to the laborious and error-prone implementation of this behaviour. The development of a tool to aid this transformation would probably reduce the errors produced here considerably as well as aiding the development of the behaviour. The other main area of difficulty was found to be debugging errors in the behaviour of a widget. This was due to difficulties in tracking events through the toolkit and determining the current state of the widget. Again, this could be alleviated by building a tool which allows the current node of the statechart defining the widget's behaviour to be highlighted and by including the means to track events through the toolkit architecture.

In the summer of 2001 two further students were employed to extend the implementation of the toolkit and determine its effectiveness in providing multimodal feedback. Lumsden *et al.* implemented a textfield widget which not only enhanced the standard visual feedback with earcons as had been done for previous widgets but also with a third modality: speech [72]. This work demonstrated that the toolkit is not limited to using just two output modalities. Lumsden *et al.* also investigated how the use of the toolkit's widgets in place of standard Swing widgets in an existing application [73]. This work showed that the toolkit could be used to enhance the interface of an existing application (the PARAGLIDE application [56]) relatively straightforwardly and that audio feedback could effectively be used in combination across an interface. In particular, it was shown that by prioritising the sounds played, audio feedback can be effective when used in combination across an interface.

8.7 Summary

In this chapter, the requirements for the toolkit presented in Chapter 6 were compared with the implementation of the toolkit presented in Chapter 7. This chapter showed that the implementation of the toolkit did meet most of the specified requirements. For those requirements the implementation did not meet, it was shown that they could be supported by the implemented architecture at minimal programming cost. The experiences of a software engineer who used the toolkit to build multimodal widgets were discussed. It was found that the development of the behaviour of the widgets was hard to do, in part because of their inherent complexity. It was also found, however, that the implementation of the presentation for these widgets was extremely simple due to the decoupling of the presentation from the widget's behaviour.

Chapter 9: Conclusions

9.1 Introduction

This chapter summarises the work described in this thesis and the results achieved. It discusses some of the limitations of the work and how they could be overcome. It suggests areas for future investigation of sound in the interface and concludes by assessing the contribution of the thesis to the area of multimodal user interfaces.

9.2 Summary of Research Carried Out

A summary of the work carried out in this thesis is given below. This research is described in terms of the two questions this thesis addresses: how best to use of sound at the human-computer interface and the design of a toolkit of multimodal, resource-sensitive widgets.

9.2.1 How Best to Use Sound at the Human-Computer Interface?

Chapter 3 reviews the use of sound at the human-computer interface. This review discussed the advantages and disadvantages of some of the different forms of sound that can be used. It then went on to describe much of the existing work done to provide sound at the human-computer interface. By examining this existing work some guidelines about the way user interfaces should support the incorporation of sound were revealed. It was found, however, that one area, the use of sound to provide persistent background information had never been formally evaluated.

In Chapter 5 a detailed description of the design and evaluation of a sonically-enhanced progress indicator was given. The importance of progress indicators and a description of the information they should provide a user with were discussed. A review of existing sonically-enhanced progress indicators showed that although never formally evaluated, there was anecdotal evidence to indicate that the use of sound in this way could be successful. Four sounds were designed to provide some of the information required by a user and an initial experiment was run to determine the correctness of the design. The results indicated that the sounds were effective in allowing a user to determine whether a background task had completed. Although no formal evidence was gathered indicating that the users were able to monitor the background task using the sounds,

anecdotal evidence indicated that this was the case so a second experiment was designed to determine this. This experiment, which required the participants to monitor a background task, showed that they could do this using the sounds. The results of this experiment demonstrated that the participants were able to monitor a background task with audio feedback as effectively as with visual feedback, with the added benefit of being able to perform other, visually intensive tasks and notice the completion of the background task sooner. This is significant because there was no visual feedback given to the participants in the audio condition of the experiment so it was shown that sound could be used as a viable alternative to visual feedback. Further guidelines on the way user interfaces should support the inclusion of sound were extracted from these experiments.

By examining both the work done in Chapter 5 and the existing systems reviewed in Chapter 3, some insight into how audio feedback can be used as both a supplement and replacement for visual feedback was found. The sonically-enhanced progress indicator, for example, was shown to be capable of replacing visual feedback with only audio feedback. Similarly, the sonically-enhanced buttons described in Section 3.4.1 were shown to be equally effective as larger graphical buttons in situations where there is a limited amount of screen space available. All the work previously done on the use of sound at the human-computer interface did not take into account environmental factors. The evaluations of the sonically enhanced buttons, for example, played the sounds at a low volume to minimise their annoyance but to ensure that the sounds were discernible the ambient volume of the environment was always kept low. This would not necessarily be the case in a real-life situation. Similarly the sounds used for the sonically-enhanced progress indicator were not found to be annoying for the duration of the relatively short tasks in the experimental evaluation but again this may not be the case in a real-life scenario where background tasks may take longer to complete. It was found, therefore that the use of different presentational modalities should vary according to both the availability and suitability of the resources required to generate the presentation.

The work presented in Chapter 3 and Chapter 5 meets the first set of aims for this research described in Section 1.2. These aims were to review the way sound has been used at the human-computer interface, capturing requirements for the toolkit and identifying areas of research that would need to be undertaken prior to the design and implementation of a toolkit of multimodal widgets. It was found that an evaluation of a sonically-enhanced progress indicator had not been done. Because this is a different form of feedback to that provided by other sonically-enhanced widgets that had been evaluated – persistent feedback as opposed to transient feedback generated by a background task as opposed to being generated as a direct result of a user action – it was necessary to design and evaluate such a widget. The design and evaluation of the sonically-enhanced progress indicator was undertaken, completing the research into the use of the sound at the human-computer interface and providing more requirements for the toolkit.

9.2.2 Design and Implementation of a Toolkit of Multimodal Widgets

In Chapter 4 a detailed review of some common user interface architectures and systems was presented. This review discussed how the different architectures and systems support multimodality and resource sensitivity.

Some design guidelines were extracted from these systems and, in combination with the requirements gathered in Chapter 3, were used as the basis for the design given in Chapter 5. The design presented uses a client-server style architecture, much like X-Windows, but in this case the clients are the widgets and the servers are output modules which translate abstract feedback requests into concrete feedback. The use of the client-server style gives the architecture several advantages. Multiple modalities are easily handled by allowing the widgets (or clients) to communicate with several output modules (or servers). Resource sensitivity is also handled by the design through the use of global components. A middle tier has been added to the client-server architecture allowing the requests made by the clients to be managed to ensure that the different requests do not interfere with each other. The use of sensors to detect environmental conditions enables the toolkit to ensure that the most appropriate form(s) of feedback are used for the current context. Finally, a control panel allows the users of the system to indicate their preferences for the widget's presentation.

Chapter 6 describes an implementation of the design discussed above. The implementation of the different components of the toolkit are described individually and in combination. Any issues that arose during the implementation of the toolkit are described along with any consequent changes that had to be made to the design. The process required to add a new widget to the toolkit is described in some detail with advice on how to avoid any potential difficulties that may arise. Chapter 8 revisited the requirements for the toolkit outlined in Chapter 6 demonstrating that the implementation met all the requirements of the design.

The work presented in Chapter 4, Chapter 6, Chapter 7 and Chapter 8 meets the second set of requirements described in Section 1.2: the design and implementation of a toolkit of multimodal widgets. A review of existing user interface architectures and systems revealed how the use of multiple output modalities are currently handled, providing a set of guidelines for the design of the toolkit. Based on these guidelines a design for a toolkit of multimodal widgets was presented in Chapter 6 and in Chapter 7 an implementation of this design was presented. Chapter 8 showed that the implementation of the toolkit met all the requirements it was designed to fulfil.

9.3 Limitations of This Research

Although the work described above was successful in meeting its goals, there are some limitations that should be considered. The limitations for the two areas of research covered by this thesis are covered below.

9.3.1 The Use of Sound at the Human-Computer Interface

The main piece of research done in this area was the design and evaluation of a sonically-enhanced progress indicator. Although this design was shown to be effective there remain a number of unresolved issues in the use of audio in this way. The evaluation used only one progress indicator at any given time meaning the issue of the audio feedback from multiple progress indicators clashing never arose. It is unclear whether the use of

multiple progress indicators using this style of audio would be effective[22]. Similarly, no work has been done to determine how the persistent sounds generated by the progress indicator would combine with other sounds generated by other widgets. Walker *et al.* [112] have done some work investigating the spatialisation of audio as a potential solution to the former problem. One of the reasons investigations into such problems have not been undertaken is the difficulty in producing experimental interfaces which include audio. This problem will be eased by the toolkit of multimodal widgets described in Chapter 6.

The background tasks used in the experimental evaluations of the sonically-enhanced progress indicator were all less than two minutes long. Although the participants did not find the sounds to be annoying, this may not be the case for background tasks that take considerably longer. A further evaluation would be required to determine if this is the case. A potential solution to this problem would be to fade the sounds out over time if the task is progressing at a steady rate, only playing them if a significant change took place.

The scope sound that was investigated in Section 5.7 had a relatively low rate of correct categorisation (about 54%). This was in part because the number of notes played in the sound was shown to have too wide a range. This has been included as a guideline for the design of such sounds. The actual categorisations used by the users to indicate the number of notes they thought were played were not as important, however, as the users' ability to recognise that a potential problem exists, in this case the size of the task being too great. In this way, the scope sound could be considered similar to the item slip sound used to indicate a potential error in the selection of a menu item [25]. The item slip sound was often played incorrectly but alerted the users to a potential problem, enabling them to check if an error had indeed been made. This meant that the sound was not found to be annoying despite it often being played incorrectly. The scope sound could well act in a similar way, highlighting a task that is potentially too lengthy to continue, enabling the user to check if this is indeed the case.

9.3.2 The Design and Implementation of a Toolkit of Multimodal Widgets

Although an implementation of the toolkit was successfully built and shown to fulfil the requirements specified in Chapter 6 there is further work to be done in this area. The set of widgets built for the toolkit thus far is still quite limited, although more widgets are being created[23]. Sufficient different widgets have been built, with two of the most interesting widgets - the button which is perhaps the most fundamental widget and the progress indicator which is unique given the persistent nature of its feedback - implemented, implying that the implementation of the full set of widgets is feasible.

An area requiring more extensive investigation is the control panel. The current implementation is tightly coupled with the control system, whereas the design specified that the control panel should access the control

---

[22] Although no formal work was undertaken to determine whether the use of several such sonically-enhanced progress indicators is feasible, informal trials clearly indicated that their use in such a way would be limited.

[23] Radio buttons, which can be grouped together, and a slider have now both been implemented by a student under my supervision. A second student has implemented a text field that uses a speech output module.

system via an API. This would allow any changes in the implementation of the control panel to be made without any disruption to the toolkit and it would also allow other pieces of software (applications built using the toolkit, for example) to access the control system. This would enable the designers of these applications to ensure that the interface of their application was presented appropriately, and that the user reaped the benefit of the designers' knowledge without restricting the inherent flexibility of the system.

Although an example of a rule controlling feedback across multiple resources was presented, more work needs to be done in the implementation of these rules. Currently, the implemented rule is hard coded into the rendering manager whereas in a future implementation the rules would be read in from a resource file and managed by a rules engine. This implies two strands of work: the format of the rules and the implementation of the rules engine. These are described more fully in Section 9.4. An associated problem is the creation of the rules themselves. To build a comprehensive set of rules, comprehensive models of presentational resources and environmental pressures on these resources will have to be built.

9.4 Contributions of this Thesis

In this section, the contributions of this thesis are summarised. In Section 9.4.1 the contributions of the work already done are summarised. In Section 9.4.2 some areas of future research which can be undertaken on the basis of the research presented in this thesis are presented.

9.4.1 Work Done

The work presented in this thesis makes five contributions to research in the use of sound at the human-computer interface and user interface architectures:

- The guidelines on the way sound should be used at the human-computer interface provide a basis for designing audio feedback.
- The design and evaluation of a sonically-enhanced progress indicator was the first evaluation on the use of sound to monitor background tasks.
- The audio progress-indicator was shown to be effective when the audio feedback *replaced* the standard graphical feedback rather than simply as an *addition* to the graphical feedback.
- The design and implementation of a toolkit of multimodal widgets is one of the first user-interface architectures primarily designed to allow a user interface to be used across different platforms with greatly differing presentational resources.
- The toolkit means the overhead of designing and evaluating new forms of feedback is greatly reduced.

The guidelines for the use of audio (summarised in Appendix D.1) and the design and evaluation of a sonically-enhanced progress indicator both contribute to research in the area of the use of sound at the human-computer interface. The guidelines give an indication of what sounds should (or should not) be used to provide a user feedback in a particular situation as well as giving some high-level indication about the design of the sounds. This complements existing guidelines [30] which describe how best to use different

characteristics of sound for different purposes. The sonically-enhanced progress indicator was the first formal evaluation of audio feedback which was used to monitor the state of a task over time. Although many sonically-enhanced widgets have been evaluated the feedback was all produced as a direct result of a user action (e.g. pressing a button) whereas the sonically-enhanced progress indicator produces feedback as a result of a change in state of a background task (e.g. a file transfer). Whilst sonically-enhanced widgets such as progress indicators have been developed in the past no formal evaluation of their effectiveness has ever been undertaken.

The audio progress indicator was shown to be effective not only as an enhancement to a visual progress indicator, but also as a replacement for a visual progress indicator. That is, it was effective as an audio only widget without any visual feedback at all. Previously evaluated sonically-enhanced widgets have been shown to be effective only as enhancements to the standard visual widgets rather than being effective as audio only widgets. Whilst this is partially due to the nature of the widget – presenting background information – it does show that it is possible to build an effective widget which uses only audio feedback to present information to user.

The design and implementation of a toolkit of multimodal widgets is important as not only is it one of the first architectures designed to allow an interface to be used on multiple different platforms with different presentational resources but it is the only architecture designed to allowed the presentation to be adapted according to the suitability of the available presentational resources. By separating the behaviour of the toolkit's widgets from their presentation and providing global components that can manage the widget's requests for presentation, the toolkit is able to modify these requests to suit the availability and suitability of presentational resources. Thus, the widgets can take advantage of different presentational resources available on different platforms.

By making the components that map the toolkit's widget's requests for feedback to concrete feedback external to the toolkit and thus interchangeable, in addition to fully exposing the behaviour of the widgets, it is possible to implement and evaluate new designs for feedback without the additional cost of re-implementing the behaviour of the widget. Although it is common to allow the presentation of a widget to be changed, the new form of presentation must typically conform to the partial behaviour of the widget that has been exposed by the designer of the widget, thus limiting the scope for the design of new forms of feedback without the overhead of completely re-implementing the widget.

9.4.2 Future Work

Although the toolkit described in this thesis has been implemented (Chapter 7) and shown to be effective in allowing the use of multiple modalities at the human-computer interface (Chapter 8), there are several interesting areas of research that follow on from this work. As a result of the work done, it has become clear that there are several areas of the implementation of the toolkit that need to be extended. The toolkit currently handles only output and should be able to handle input equally flexibly. High level tools could be built around the toolkit allowing it to be incorporated into applications more easily. Additionally, the ease with which the toolkit allows applications which use novel forms of feedback to be built enables research into new

forms of feedback to be undertaken more easily. This section discusses these topics and suggests some avenues of investigation.

Handling Input

One of the areas that requires further investigation is the handling of input by the toolkit. For the purposes of this research, the majority of input event handling was delegated to the existing AWT event handling system. Although this meant that the graphical presentation of the widgets was restricted to that of the standard Swing widgets, it had the advantage minimising the implementation cost for the toolkit. For the toolkit to be truly useful in enabling interfaces which are effective across multiple devices with different capabilities and resources, it is necessary to afford the same flexibility for input as has been afforded for output. This section describes some of the problems that may arise when developing the input side of the toolkit and proposes some possible solutions to these problems.

Handling Multiple Input Mechanisms

The toolkit will need to be able to handle multiple input mechanisms to its widgets just as multiple output mechanisms are currently handled. Currently, widgets could employ for example, both a visual and audio output mechanism but are limited to input via a mouse. In the future, this could be extended so that the widgets could take advantage of all the available input devices, regardless of the mechanism they employ. The problems that arise from this, however, are different from those encountered when handling multiple output mechanisms. Multiple output mechanisms require the ability to generate multiple requests from a single event and to ensure that the items of concrete feedback generated from these multiple requests were appropriate for the current context. Multiple input mechanisms, on the other hand, require the ability to merge multiple pieces of input into *one* event which is consequently translated into a request for feedback. This implies two things: the architecture must allow multiple streams of input to a single widget and the widget itself must be independent of the input mechanism.

The MATIS system [92] (Multimodal Airline Travel Information System) describes a mechanism which allows multiple streams of input to be 'fused' into coherent data. For example, a user could request information about flights by saying "show me the flights from Boston to this city" whilst selecting "Denver" with the mouse. The MATIS system would recognise these parallel inputs and combine them into a request for information about flights from Boston to Denver. In this way, MATIS avoids the imposition of a dominant input modality but ensures all input modalities are treated equally, with support for switching between them. MATIS handles input at the application level, whereas the toolkit handles input at a lower, widget level and consequently does not require data fusion in the same way. The problem the toolkit must overcome is at a lower level and only involves individual widgets and so perhaps is easier to solve. If, for example, a button widget employs two input mechanisms, mouse and keyboard, it is entirely feasible the user could start to select the button using the mouse (perhaps moving the mouse over the button and pressing the mouse button down) before selecting the button with a keyboard shortcut. It is the job of the toolkit, and the individual input mechanisms in particular, to manage this parallel input and consequently decide the states the different input mechanisms should revert to. A possible scenario would be to return the mouse input mechanism to its prior state of partial selection meaning if the user was to subsequently release the mouse

button without moving the mouse, the button would be selected again. The alternative to this would be to reset the interaction, forcing the user to release the mouse button and reselect the button performing the complete selection again. The former method would ensure that the methods of interaction used were consistent in their behaviour, but the latter would avoid potentially accidental multiple selections of the button. The toolkit should support either approach, allowing the input mechanism's designer to specify the appropriate behaviour.

The toolkit, as described in Chapter 7, enables widgets to use multiple output modalities by ensuring the widget behaviour is independent of the presentation used. This allows the behaviour of the widget to make abstract requests for presentation which can be translated into concrete feedback. The behaviour of the widgets, however, is entirely based upon the use of a mouse or similar pointing device as input. To allow the use of multiple input mechanisms for the same widget, it will be necessary to abstract the widget behaviour away from a specific input mechanism in a similar fashion to the way output is abstracted. Mouse specific behaviour that is currently located in the widget would be replaced by an abstract, or generic, behaviour with the mouse specific behaviour now located in an input mechanism, separate from the widget. There could be many such input mechanisms feeding events to the abstract widget which then generates the requests for feedback.

This implies the need for the specification of a generic behaviour for a button to which all input mechanisms must adhere. A possible generic behaviour for a button, or selector, is shown in Figure 9.1.
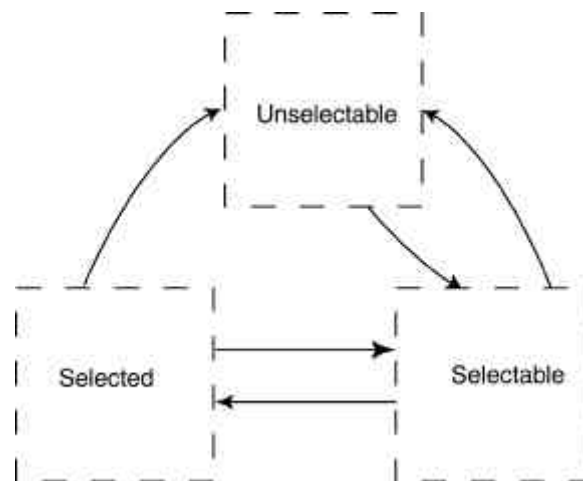
Figure 9.1 – The abstract behaviour for a selector widget. No input mechanism specific behaviour is specified here.

In Figure 9.1 it can be seen there are three potential states for a selector. The selectable state occurs when it is possible to select the widget but it has not yet been selected. The selected state occurs when the widget has been selected and the unselectable state occurs when it is not possible to select the widget if, for example, it has been disabled. These three states are independent of, not only any presentation information, but also of any notion of how the states change. That is, none of the sub-states required to allow the selection of the widget using a mouse, for example, are specified. These sub-states are specified in the different input mechanisms. The mouse input mechanism behaviour of a selector is shown in the statechart in Figure 9.2.

Figure 9.2 – The mouse input mechanism specific behaviour of a selector. The abstract behaviour states are shown as dotted line boxes with the mechanism specific states are shown as solid line boxes.

In Figure 9.2 the abstract states of Figure 9.1 are shown as boxes drawn with dotted lines and the mechanism specific states are shown as boxes drawn with solid lines. In this way, it can be seen how the mechanism specific behaviour relates to the abstract behaviour. If the widget has several input mechanisms associated with it there would need to be a process to ensure that the different input mechanisms remain consistent. That is, if the abstract state of the widget is changed due to one input mechanism, all other input mechanisms

would need to be informed so that all the mechanism specific states remain consistent. This is similar to the MVC model where the model component is required to ensure that multiple views remain consistent.

The change in the way input is being handled would also have an effect on the way output is produced. Currently, requests for feedback are based on the state of the widget with regard to mouse events, such as `mouseEnter` and `mousePress`. If multiple input mechanisms were being used, this would not necessarily be the case. Requests made for feedback would need to include two levels of information: the abstract state of the widget and the input mechanism specific state. Similarly, the output modules would need to be able to provide feedback on the same two levels. If the output module could not handle the input mechanism specific information it should still be able to provide some feedback about the abstract state of the widget. Thus, a minimum level of feedback is guaranteed regardless of the input mechanisms and output modules being used.

Ensuring Input and Output Remain Consistent

Because the widget no longer has any notion of the way it is being presented, nor how its input is being handled, there is a danger that the input and output for a widget could become inconsistent. A button, for example, has an input area that is dependent upon the presentation of the button. If the way the button is presented changes, then the input area may change. For example, if the user was to request that all buttons were to be circular in shape where previously they had been rectangular, the input mechanism would need to know about this change as it would effect the input area of the widget. Furthermore, each widget may have several input "areas" depending upon the output module used. As well as having a graphical input area, a button may have a location in a 3D audio space which can be selected by gesture or a description in an abstract space which can be selected with a keyboard shortcut or speech input. This section discusses two issues that need to be investigated to ensure that input and output remain consistent: interaction spaces and communication between input and output mechanisms.

An interaction space can be thought of as a medium through which a human and computer can communicate. Different interaction spaces will have different sets of associated parameters. A screen, for example, allows a computer to communicate with a user by providing an area within which graphical and textual objects can be rendered. A user can communicate with the computer via the screen using a pointing device such as a mouse, or perhaps a finger if the screen is touch sensitive. These two forms of communication are bonded by the use of a common set of parameters, the screen co-ordinates. Thus, the computer draws a widget at a specific location and with a specific size on the screen. The user points at a specific location on the screen. If the two locations match, then the button has been selected. Similarly, audio feedback which has been spatialised can be considered to use a 3D interaction space surrounding the user. The user can interact with this interaction space by, for example, gesture. If he/she points at the source of the sound, then perhaps that widget will be selected. It is not necessary, of course, for this communication to take place using the same interaction space. A button, represented visually on a screen may also have a keyboard shortcut. This shortcut is not represented visually, but can be accessed via a different, abstract interaction space. This interaction space can be considered as a table which maps labels (e.g. keyboard shortcuts or speech input) to interaction objects. Thus, when the user presses the shortcut, the button is selected and may well change its visual representation. The mechanism to support this use of multiple input modalities was described in the previous section.

Communication between input and output mechanisms is necessary to ensure consistency between these two components. If, for example, the graphical rendering of a widget changes, it is necessary for the input mechanism to know so that it can ensure that input events are handled appropriately. Only the input and output mechanisms that share an interaction area need to communicate with each other. This communication could be facilitated by a presentation model object. Each widget would have a presentation model object for each interaction area it currently uses. This object would define the rendering of that widget in terms of the interaction area's parameters. The presentation object for a button drawn on a screen, for example, would contain the co-ordinates and size of the graphical rendering of the button. This object would be updated by the output module when the rendering of the button changed and queried by the input mechanism when an input event was received (Figure 9.3).
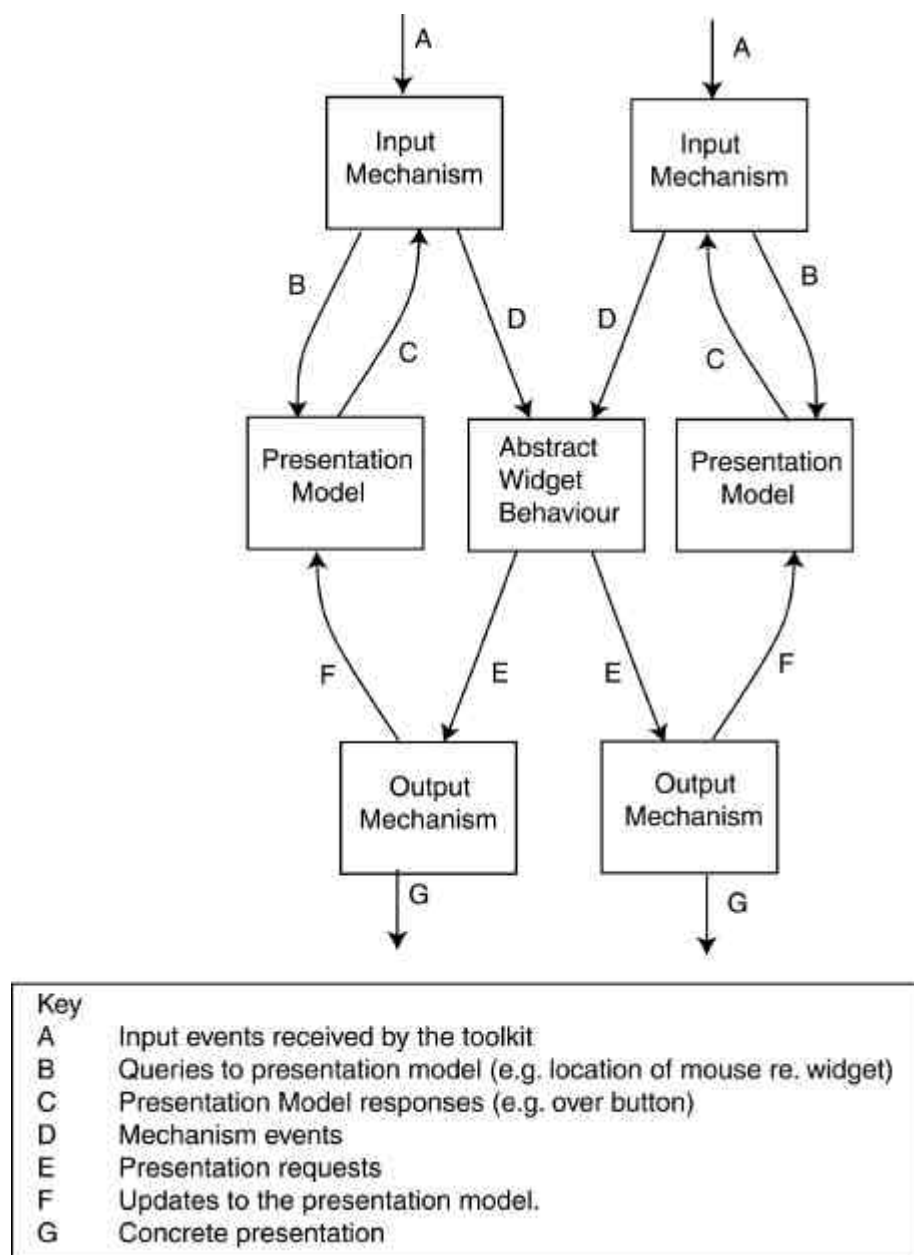
Figure 9.3 – The proposed architecture to enable communication between input and output for a widget. Every widget has a presentation model for each interaction space it uses. The input mechanism queries this object when an event is received and the output module in that interaction space updates it if the rendering of the widget changes.

170

An exception to this architecture is the presentation object for the abstract interaction space described above. This interaction space maps abstract objects such as labels and spoken commands which have no rendering to a widget. This presentation object, therefore, would not be updated by an output module, but would be maintained by the widget itself.

Expanding the Concept of Resource-Sensitive

This thesis describes a toolkit of resource-sensitive widgets. That is, the widgets are capable of adapting their presentation according to the availability and suitability of the necessary resources. The implementation of the toolkit does not consider the suitability of a resource with respect to the user or the task the user is performing. Through the control user interface, users are able to change the presentation to suit themselves or the task they are performing, but there is no scope for automating this process. To enable this process to be automated, models of users and their tasks would need to be built. Once built these models would need to be incorporated into the toolkit. This incorporation could take one of two forms: the models could be used to generate rules which are then incorporated into the toolkit; or the models could be made available to the toolkit's rules. In the former case the rules would require some form of external input, or sensors, to allow them to determine the correct approach to take. These sensors could be very simple - for example simply taking the current user's login and applying known settings to the rules - or may be complicated - for example, determining the current task by monitoring key strokes and mouse presses. In the latter case, the rules would query the current models to determine the correct approach to take. If the models were to be updated dynamically they would need to collect similar information to the sensors described above.

High Level Tools For the Toolkit

Although it is simple to include the toolkit's widgets into an application, there some tools that could be built to make the use of the toolkit easier. One such tool would allow the automatic conversion of an application so that it used the toolkit's widgets rather than the standard Swing widgets. An initial implementation of such a tool could operate on source code but could be developed further to operate on compiled code, or byte-code. Such an automatic conversion is possible because the API followed by the toolkit varies in standard ways from the Swing API. This variance is typically limited to changing the name of the class used for the widget. This mapping could be easily stored in a resource file. Because the toolkit widgets extend a standard Swing widget – either the Swing version of the widget or an extension of the `JPanel` class - they are compatible with any existing methods which require a Swing widget as a parameter or return value.

A second tool could be built which would allow new applications to be built using the toolkit widgets. This User Interface Development Environment (UIDE) would allow a designer to build an interface using a direct manipulation style interface. This would allow the creation of the interface with a minimum of coding. The tool could also allow the designer to specify a user interface to the control system that would allow an end-user to modify the presentation of the interface's widgets. In this way, the designer could specify the amount of flexibility in the presentation of the widgets, thus minimising the risk of an end-user reducing the effectiveness of the interface. This control user interface could, for example, afford an end-user no control over the presentation of the widgets if the application runs in a safety-critical environment, whilst a user interface in a less critical environment may afford much greater control.

Future Research Into Novel Forms of Feedback

By enabling applications that use new forms of presentation to be built relatively quickly and easily, the toolkit allows new and interesting areas of research into new forms of feedback to investigated more quickly than was previously possible. Previous investigations into the effectiveness of sound at the human-computer interface, for example, have been time consuming due to the difficulties in incorporating the sounds into widgets. Thus, the only evaluations completed thus far have been investigating the effectiveness of a single sonically-enhanced widget in a single user environment. Using the toolkit it should be simple to build applications which will allow evaluations into, for example, the use of different sonically-enhanced widgets in combination or perhaps the use of sounds in a multi-user environment.

9.5 Conclusions and Contributions of the Thesis

This thesis makes a major contribution in two areas of related research. The guidelines on the use of sound at the human-computer interface complement existing guidelines which describe how to use the physical characteristics of sounds appropriately [30] by describing when it is appropriate to use different sounds. The design and evaluation of a sonically-enhanced progress indicator was the first evaluation on the use of sound to monitor background tasks. Previous work in this area had not been formally evaluated although the indications were that this was a worthwhile area of research to pursue. Furthermore, previous widgets were evaluated to see if the *addition* of sound was an improvement over the standard graphical feedback on its own. The sonically-enhanced progress indicator was shown to be effective with sound used as a *replacement* for the standard graphical feedback. Results showed that participants were able to monitor the progress of a background task using audio only feedback as effectively as when visual feedback was used. The audio feedback, however, allowed the participants to perform other, visually intensive tasks at the same time as monitoring the background task with the audio. The results also showed that the audio feedback was more effective at informing the participants when the task was completed than the visual feedback.

The design and evaluation of the sonically-enhanced progress indicator meant that a complete set of sonically-enhanced widgets had been evaluated, opening the way for a toolkit of such widgets to be built. The design and implementation of such a toolkit is an important contribution to the two areas of research. It is an important contribution in the area of user interface architectures as it is one of the first architectures primarily designed to allow a user interface to be used across different platforms with greatly differing presentational resources. Previously, such interfaces would have to be implemented several times, once for each of a number of classes of device. Additionally, the toolkit provides the framework to allow the interface to adapt to its current context meaning it provides the most suitable rendering for the interface within the constraints specified by the available presentation resources. Previously, users would have to make do with a static interface at best or manually adjust the rendering of the interface to suit the current context. In addition to the advances in the field of user interface architectures, the implementation of this toolkit provides a significant contribution in the area of auditory interface design. Previously, auditory interfaces had to be

hand-crafted, often by adapting existing visual interfaces. This often proved to be a difficult process meaning the programming overhead for the development of such interfaces was great. The toolkit greatly reduces this overhead, with the only additional work required to produce an audio interface being the implementation of the sounds themselves. The toolkit presented in this thesis therefore provides significant contributions in two distinct areas of research. The inclusion of sound at the human-computer interface has been made significantly easier making both the design and evaluation of such interfaces and their use in everyday situations possible with significantly less cost to the designer of the interface. Furthermore, the framework of the toolkit allows the widgets to be sensitive to their context unlike any other toolkit of widgets currently available.

Appendix A: References

[1]   Adcore Pyramid AB, "The Official Bluetooth SIG Website", http://www.bluetooth.com/, as on 3/09/2001.

[2]   M. C. Albers and E. Bergman, "The Audible Web: Auditory Enhancements for Mosaic," in *Proceedings of ACM CHI'95 Conference on Human Factors in Computing Systems*, vol. 2, *Short Papers: Web Browsing*, 1995, pp. 318-319.

[3]   F. K. Aldrich and A. J. Parkin, "Listening at Speed," *British Journal of Visual Impairment and Blindness*, vol. 7(1), pp. 16-18, 1989.

[4]   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, Sweden, 1991.

[5]   J. L. Alty, "Multimedia -- What is It and How Do We Exploit It?," in *Proceedings of the HCI'91 Conference on People and Computers VI*, *Invited Papers*, 1991, pp. 31-44.

[6]   B. Arons, "SpeechSkimmer: A System for Interactively Skimming Recorded Speech," *ACM Transactions on Computer-Human Interaction*, vol. 4(1), pp. 3-38, 1997.

[7]   W. Barfield, C. Rosenberg, and G. Levasseur, "The Use of Icons, Earcons and Commands in the Design of an Online Hierarchical Menu," *IEEE Transactions on Professional Communication*, vol. 34,101-108, 1991.

[8]   P. G. Barker and K. A. Manji, "Pictorial Dialogue Methods," *International Journal of Man-Machine Studies*, vol. 31(3), pp. 323-347, 1989.

[9]   M. Beaudouin-Lafon and S. Conversy, "Auditory Illusions for Audio Feedback," in *ACM CHI'96 Conference Companion*, Reading, MA: ACM Press, Addison-Wesley, 1996, pp. 299-300.

[10]   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.

[11]   N. Bevan and M. Macleod, "Usability Measurement in Context," *Behaviour and Information Technology*, vol. 13(1,2), pp. 132-145, 1994.

[12]   M. M. Blattner, D. A. Sumikawa, and R. M. Greenberg, "Earcons and Icons: Their Structure and Common Design Principles," *Human-Computer Interaction*, vol. 4(1), pp. 11-44, 1989.

[13]   A. S. Bregman, *Auditory Scene Analysis*. Cambridge, Massachusetts: MIT Press, 1990.

[14]   S. Brewster, "The Design Of Sonically-Enhanced Widgets," *Interacting With Computers*, vol. 11(2), pp. 211-235, 1998.

[15]   S. Brewster, "Using Earcons to Improve the Usability of a Graphics Package," in *Proceedings of BCS HCI'98*, Sheffield, UK: Springer-Verlag, 1998, pp. 287-302.

[16]   S. Brewster, "Using Earcons To Improve The Usability of Tool Palettes," in *Summary Proceedings of ACM CHI'98*, Los Angeles, CA: ACM Press, Addison-Wesley, 1998, pp. 297-298.

[17]   S. Brewster, "Sound In The Interface To A Mobile Computer," in *Proceedings of HCI International'99*, Munich: Lawrence Erlbaum Associates,NJ, 1999, pp. 43-47.

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

[19]   S. Brewster and P. Cryer, "Maximising Screen-Space on Mobile Computing Devices," in *Summary Proceedings of ACM CHI'99*, Pittsburgh, PA: ACM Press, Addison-Wesley, 1999, pp. 224-225.

[20]   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.

[21]   S. Brewster, P. C. Wright, and A. D. N. Edwards, "Parallel Earcons: Reducing the Length of Audio Messages," *International Journal of Human-Computer Studies*, vol. 43(2), pp. 153-175, 1995.

[22]   S. A. Brewster, *Providing a Structured Method for Integrating Non-Speech Audio into Human-Computer Interfaces.* Department of Computer Science PhD. University of York, 1994

[23]   S. A. Brewster, "Sonically-Enhanced Drag and Drop," in *Proceedings of ICAD'98*, Glasgow, UK: British Computer Society, 1998.

[24]   S. A. Brewster and C. V. Clarke, "The Design and Evaluation of a Sonically-Enhanced Tool Palette," in *Proceedings of ICAD'97*, Xerox Parc, USA: Xerox, 1997, pp. 119-123.

[25]   S. A. Brewster and M. G. Crease, "Correcting Menu Usability Problems With Sound," *Behaviour & Information Technology*, vol. 18(3), pp. 165-177, 1999.

[26]   S. A. Brewster, A. Crossan, and M. Crease, "Automatic Volume Control for Auditory Interfaces," in *Proceedings of BCS HCI 2000*, vol. Volume II, Sunderland, UK: IFIP- Edinburgh Press, 2000, pp. 209-212.

[27]   S. A. Brewster, P. C. Wright, A. Dix, and A. D. N. Edwards, "A Detailed Investigation Into the Effectiveness of Earcons," in *Auditory Display, Sonification, Audification and Auditory Interfaces. The Proceedings of the First International Conference on Auditory Display*, Santa Fe Institute, Santa Fe: Addison-Wesley, 1992, pp. 471-498.

[28]   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.

[29]   S. A. Brewster, P. C. Wright, and A. D. N. Edwards, "The Design and Evaluation of an Auditory-Enhanced ScrollBar," in *Proceedings of ACM CHI'94 Conference on Human Factors in Computing Systems*, vol. 1, *Auditory Information Interfaces*, 1994, pp. 173-179.

[30]   S. A. Brewster, P. C. Wright, and A. D. N. Edwards, "Experimentally derived guidelines for the creation of earcons," in *Adjunct Proceedings of HCI'95*, Huddersfield, UK, 1995.

[31]   S. Burbeck, "Application Programming in Smalltalk-80: How to use Model-View-Controller (MVC)", http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html, as on 19/08/2000.

[32]   W. Buxton, "Introduction to this Special Issue on Nonspeech Audio," *Human-Computer Interaction*, vol. 4(1), pp. 1-9, 1989.

[33]   M. Campione and K. Walrath, *The Java Tutorial. Object-Oriented Programming for the Internet*, vol. 1. Reading, MA: Addison-Wesley, 1996.

[34]   L. Cardelli and R. Pike, "Squeak: A Language for Communicating with Mice," in *Computer Graphics, Proceedings of SIGGRAPH'85*, San Francisco, CA, 1985, pp. 199-204.

[35]   J. Cohen, "Monitoring Background Activities," in *Proceedings of the First International Conference on Auditory Display, ICAD'92*, Santa Fe: Addison-Wesley, 1992.

[36]   J. Cohen, ""Kirk Here:" Using Genre Sounds to Monitor Background Activity," in *Proceedings of ACM INTERCHI'93 Conference on Human Factors in Computing Systems -- Adjunct Proceedings*, *Short Papers (Talks): Multi-Modal User Interfaces*, 1993, pp. 63-64.

[37]   J. Cohen, "Out to Lunch: Further Adventures Monitoring Background Activities," in *Proceedings of the Second International Conference on Auditory Display (ICAD'94)*, Santa Fe, NM: Addison-Wesley, 1994.

[38]   A. P. Conn, "Time Affordances: The Time Factor in Diagnostic Usability Heuristics," in *Proceedings of ACM CHI'95 Conference on Human Factors in Computing Systems*, vol. 1, *Papers: Learning from Design Experiences*, 1995, pp. 186-193.

[39]   S. Conversy, "Ad-hoc Synthesis of Auditory Icons," in *Proceedings of ICAD'98*, Glasgow, Scotland: BCS, 1998.

[40]   J. Coutaz, "PAC, an Object-Oriented Model for Dialog Design," in *Proceedings of IFIP INTERACT'87: Human-Computer Interaction, 2. Design and Evaluation Methods: 2.5 Dialogue Design and Evaluation*, 1987, pp. 431-436.

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

[42]   J. Coutaz, L. Nigay, and D. Salber, "Multimodality from the User and System Perspectives," in *ERCIM'95 Workshop on Multimedia Multimodal User Interfaces*, Heraklion, Crete, 1995.

[43]   J. Coutaz, L. Nigay, D. Salber, and A. Blandford, "Four Easy Pieces for Assessing the Usability of Multimodal Interaction: The CARE Properties," in *INTERACT'95: Proceedings of the Fifth IFIP Conference on Human-Computer Interaction*: Chapman-Hall, 1995, pp. 115-120.

[44]   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.

[45]   M. Crease and S. Brewster, "Scope For Progress: Monitoring Background Tasks With Sound," in *Human-Computer Interaction, Interact'99*, vol. II, Edinburgh, UK: IFIP, 1999, pp. 19-20.

[46]   M. Crease, S. Brewster, and P. Gray, "Caring, Sharing Widgets: A Toolkit of Sensitive Widgets," in *People And Computers XIV - Usability or Else. Proceedings of HCI 2000*, Sunderland, UK: Springer, 2000, pp. 257-270.

[47]   M. Crease, P. Gray, and S. Brewster, "Resource Sensitive Multimodal Widgets," in *Proceedings of Interact'99*, vol. II, Edinburgh, Scotland: BCS, 1999, pp. 21-22.

[48]   M. Crease, P. Gray, and S. Brewster, "A Toolkit of Mechanism and Context Independent Widgets," in *7th International Workshop , Design, Specification and Verification of Interactive Systems*, *Lecture Notes in Computer Science*, Limerick Ireland: Springer-Verlag, 2000, pp. 121-143.

[49]   A. Dix and S. A. Brewster, "Causing Trouble With Buttons," in *Ancillary Proceedings of BCS HCI'94*, Glasgow, UK: Cambridge University Press, 1994.

[50]   A. Dix, J. Finlay, G. Abowd, and R. Beale, *Human-Computer Interaction*: Prentice Hall, 1993.

[51]   W. K. Edwards and E. D. Mynatt, "An Architecture for Transforming Graphical Interfaces," in *Proceedings of ACM UIST'94: Symposium on User Interface Software and Technology*, 1994, pp. 39-47.

[52]   J. Edworthy, "Does sound help us to work better with machines? A commentary on Rauterberg's paper ' About the importance of auditory alarms during the operation of a plant simulator'," *Interacting with Computers*10), pp. 401-409, 1998.

[53]   D. Flanagan, *Java In A Nutshell. A Desktop Reference*, 3 ed. Sebastopol, CA.: O'Reilly & Associates, Inc., 1999.

[54]   J. D. Foley, "The Art of Natural Graphic Man-Machine Conversation," *Proceedings of the IEEE*, vol. 62(4), pp. 462-471, 1974.

[55]   A. Fowler, "A Swing Architecture Overview*.* The Inside Story on JFC Component Design", http://java.sun.com/products/jfc/tsc/articles/architecture/index.html, as on 22/01/2001.

[56]   M. Gardener, M. Sage, P. D. Gray, and C. W. Johnson, "Data Capture fro Clinical Anaesthesia on a Pen-Based PDA: Is it a Viable Alternative to Paper?," in *Proceedings of IHM-HCI'01*, Lille, 2001.

[57]   W. Gaver, T. Moran, A. MacLean, L. Lovstrand, P. Dourish, K. Carter, and W. Buxton, "Realizing a Video Environment: EuroPARC's RAVE System," in *Proceedings of ACM CHI'92 Conference on Human Factors in Computing Systems*, 1992, pp. 27-35.

[58]   W. W. Gaver, "Auditory Icons: Using Sound in Computer Interfaces," *Human-Computer Interaction*, vol. 2(2), pp. 167-177, 1986.

[59]   W. W. Gaver, "The Sonic Finder: An Interface that Uses Auditory Icons," *Human-Computer Interaction*, vol. 4(1), pp. 67-94, 1989.

[60]   W. W. Gaver, "What in the World Do We Hear? An Ecological Approach To Auditory Event Perception," *Ecological Psychology*, vol. 5(1), pp. 1-29, 1993.

[61]   W. W. Gaver and R. B. Smith, "Auditory Icons in Large-Scale Collaborative Environments," *ACM SIGCHI Bulletin*, vol. 23(1), pp. 96, 1991.

[62]   W. W. Gaver, R. B. Smith, and T. O'Shea, "Effective Sounds in Complex Systems: The ARKola Simulation," in *Proceedings of ACM CHI'91 Conference on Human Factors in Computing Systems*, 1991, pp. 85-90.

[63]   J. Gettys, P. L. Karlton, and S. McGregor, "The X Window System, Version 11," *Software Practice and Experience*, vol. 10(S2), pp. 35-67, 1991.

[64]   S. Greig, *A Toolkit of Context and Resource Sensitive Widgets.* Department of Computing Science Final Year Undergraduate Project. University of Glasgow, 2001

[65]   D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, vol. 8(1), pp. 231-274, 1987.

[66]   S. Hart and L. Staveland, "Development of NASA_TLX (Task Load Index): Results of Empirical and Theoretical Research," in *Human Mental Workload*, Amsterdam, Holland, 1988, pp. 139-183.

[67]   R. D. Hill, "The Abstraction-Link-View Paradigm: Using Constraints to Connect User Interfaces to Applications," in *Proceedings of ACM CHI'92 Conference on Human Factors in Computing Systems*, 1992, pp. 335-342.

[68]   id-Software, "id Software - Quake 2", http://www.idsoftware.com/quake2/index.html, as on 17/01/2001.

[69]   A. I. Karshmer, P. Brawner, and G. Reiswig, "An Experimental Sound-Based Hierarchical Menu Navigation System for Visually Handicapped Use of Graphical User Interfaces," in *First Annual ACM Conference on Assistive Technologies*, 1994, pp. 123-128.

[70]   S. Kawai, H. Aida, and T. Saito, "Designing Interface Toolkit with Dynamic Selectable Modality," in *Second Annual ACM Conference on Assistive Technologies*, *The User Interface -- II*, 1996, pp. 72-79.

[71]   M. A. Linton, P. R. Calder, and J. M. Vlissides, "The Design and Implementation of InterViews," in *Proceedings of the Usenix C++ Workshop*, Santa Fe, New Mexico, 1987.

[72]   J. Lumsden, J. Williamson, and S. Brewster, "Enhancing Textfield Interaction With the Use of Sound," University of Glasgow, Glasgow, Dept of Computing Science Technical Report TR-2001-99, October 2001 2001.

[73]   J. Lumsden, A. Wu, and S. Brewster, "Evaluating the Combined Use of Audio Toolkit Widgets," University of Glasgow, Glasgow, Dept of Computing Science Technical Report TR-2001-101, October 2001.

[74]   L. H. Miller, "A Study In Man-Machine Interaction," in *Proceedings of the National Computer Conference*, vol. 46: AFIPS Press, 1977, pp. 409-421.

[75]   R. B. Miller, "Response Time in Man-Computer Conversational Transactions," in *Proceedings Fall Joint Computer Conference*, vol. 33(part 1): AFIPS Press, 1968, pp. 267-277.

[76]   E. N. Mitsopoulos and A. D. N. Edwards, "Auditory Scene Analysis as the Basis for Designing Auditory Widgets," in *Proceedings of ICAD'97*, Palo Alto, CA, 1997.

[77]   E. N. Mitsopoulos and A. D. N. Edwards, "A Principled Methodology for the Specification and Design of Non-Visual Widgets," in *Proceedings of ICAD'98*, Glasgow, Scotland, 1998.

[78]   B. Myers, "The Garnet User Interface Development Environment," in *Proceedings of ACM CHI'91 Conference on Human Factors in Computing Systems*, 1991, pp. 486.

[79]   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(11), pp. 71-85, 1990.

[80]   B. Myers, S. E. Hudson, and R. Pausch, "Past, Present and Future of User Interface Software Tools," *ACM Transactions on Computer Human Interaction*, vol. 7(1), pp. 3-28, 2000.

[81]   B. A. Myers, "The Importance of Percent-Done Progress Indicators for Computer-Human Interfaces," in *Proceedings of ACM CHI'85 Conference on Human Factors in Computing Systems*, 1985, pp. 11-17.

[82]   B. A. Myers, "A New Model for Handling Input," *ACM Transactions on Information Systems*, vol. 8(3), pp. 289-320, 1990.

[83]   B. A. Myers, "The Garnet User Interface Development Environment," in *Proceedings of ACM INTERCHI'93 Conference on Human Factors in Computing Systems -- Adjunct Proceedings*, 1993, pp. 223.

[84]   B. A. Myers, "The Garnet User Interface Development Environment," in *Proceedings of ACM CHI'94 Conference on Human Factors in Computing Systems*, vol. 2, 1994, pp. 25-26.

[85]   B. A. Myers, "The Garnet and Amulet User Interface Development Environments," in *Proceedings of ACM CHI'95 Conference on Human Factors in Computing Systems*, vol. 2, 1995, pp. 334.

[86]   B. A. Myers, "User Interface Software Tools," *ACM Transactions on Computer-Human Interaction*, vol. 2(1), pp. 64-103, 1995.

[87]   B. A. Myers, R. C. Miller, R. McDaniel, and A. Ferrency, "Easily Adding Animations to Interfaces Using Constraints," in *Proceedings of the ACM Symposium on User Interface Software and Technology*, *Papers: Tools*, 1996, pp. 119-128.

[88]   B. A. Myers and K. A. Strickland, "Easily Adding Sound Output To Interfaces", http://www.cs.cmu.edu/afs/cs.cmu.edu/user/bam/www/resume.html#publications, as of 19/01/2001.

[89]   E. D. Mynatt, M. Back, R. Want, M. Baer, and J. B. Ellis, "Designing Audio Aura," in *Proceedings of CHI'98*, Los Angeles: ACM, Addison-Wesley, 1998, pp. 566-573.

[90]   E. D. Mynatt, M. Back, R. Want, and R. Frederick, "Audio Aura: Light-Weight Audio Augmented Reality," in *Proceedings of UIST'97*, Banff, Canada: ACM Press, Addison-Wesley, 1997, pp. 211-212.

[91]   E. D. Mynatt and W. K. Edwards, "Mapping GUIs to Auditory Interfaces," in *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, 1992, pp. 61-70.

[92]   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, 1995, pp. 98-105.

[93]   I. Oakley, M. McGee, S. Brewster, and P. Gray, "Putting The Feel Into Look And Feel," in *Proceedings of ACM CHI'2000*, The Hague, Netherlands: ACM Press, Addison-Wesley, 2000.

[94]   R. D. Patterson, "Guidelines for Auditory Warning Systems on Civil Aircraft," Civil Aviation Authority, London CAA Paper No. 82017, 1982.

[95]   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*, 1991, pp. 117-125.

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

[97]   R. B. Rodger, "The XEyes Client", http://www.strath.ac.uk/CC/Courses/oldXC/subsection3_6_5.html, as on 24/02/01.

[98]   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, ACM Press, Addison-Wesley, 1995, pp. 106-113.

[99]   A. Savidis and C. Stephanidis, "The HOMER UIMS for Dual User Interface Development: Fusing Visual and Non-Visual Interactions," *Interacting With Computers*, vol. 11,173-209, 1998.

[100]   R. W. Scheifler and J. Gettys, "The X Window System," *ACM Transactions on Graphics*, vol. 5(2), pp. 79-109, 1986.

[101]   E. Schlungbaum, "Individual User Interfaces and Model-based User Interface Software Tools," in *Proceedings of the 1997 International Conference on Intelligent User Interfaces*, Orlando, Florida: ACM-Press, 1997, pp. 229-232.

[102]   P. A. Scholes, *The Oxford Companion to Music*, 10th ed. Oxford: Oxford University Press, 1970.

[103]   B. Shneiderman, "Direct Manipulation for Comprehensible, Predictable and Controllable User Interfaces," in *Proceedings of the 1997 International Conference on Intelligent User Interfaces*, Orlando, Fl: ACM-Press, 1997, pp. 33-39.

[104]   L. M. Slowiaczek and H. C. Nusbaum, "Effects of Speech Rate and Pitch Contour on the Perception of Synthetic Speech," *Human Factors*, vol. 27(6), pp. 701-712, 1985.

[105]   Sun Microsystems, "java.sun.com - The Source For Java (TM) Technology", http://java.sun.com, as on 13/11/2000.

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

[107]   Sun Microsystems, "Coming Swing API Changes for Java 2 SDK, Standard Edition, v. 1.4", http://java.sun.com/products/jfc/tsc/articles/merlin/index.html, as on 06/03/2001.

[108]   The UIMS Tool Developers' Workshop, "A Metamodel for the Runtime Architecture of an Interactive System," *ACM SIGCHI Bulletin*, vol. 24(1), pp. 32-37, 1992.

[109]   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.

[110]   R. Valdes, "A Virtual Toolkit for Windows and the Mac," *Byte*, vol. 14(3), pp. 209 - 216, 1989.

[111]   J. Vanderdonckt, "Knowledge-Based Systems for Automated User Interface Generation: The TRIDENT Experience," Fac. Univ. de Notre-Dame de la Paix, Inst. d'Informatique, Namur, Technical Report RP-95-010, 1995.

[112]   A. Walker and S. Brewster, "Spatial Audio In Small Display Screen Devices," *Personal Technologies*, vol. 42(2), pp. 144-154, 2000.

[113]   A. Walker and S. Brewster, "Informing Soundtracks," Department of Computing Science, University of Glasgow, Glasgow, Scotland, Technical Report TR-2001-75, February 2001.

[114]   W. Yu, R. Ramloll, and S. A. Brewster, "Haptic Graphs For Blind Computer Users," in *Haptic Human-Computer Interaction*, Springer LNCS vol. 2028, 2001, pp. 41-58.

Appendix B: Glossary of Audio Terms

This appendix summarises some of the terminology used in this thesis. This terminology is presented in two sections: first, terms associated with audio feedback, and second, terms associated with the toolkit and its architecture.

B.1: Audio Feedback Terminology

Term	Definition
Auditory Icon	Audio feedback which uses everyday sounds to present information.
Avoidable	Used to describe feedback which can be missed by the user.
Compound Earcon	Earcons that are combined in such a way that the motives are played one after the other.
Demanding	Used to describe feedback that cannot easily be avoided or habituated
Earcon	Abstract, synthetic sounds used in structured combinations whereby the musical qualities of the sounds hold the information.
Habituation	The sub-conscious process by which a user becomes sufficiently immune to a sound that the sound is not demanding or attention-grabbing and instead constitutes background noise – i.e. the user lets the sound fade into the background.
Parallel Earcon	Earcons that are combined in such a way that the motives are played concurrently.
Pitch	The relative height or depth of a sound – the quality which distinguishes the sound of different notes played on the same instrument.
Rhythm	Regular recurrence of a pattern of notes.
Serial Earcon	Earcons that are played together and use different spatial location for differentiation.
Shepard-Risset Tones	A progression of notes which generates the auditory-illusion of a never-ending rising or falling sequence of notes.
Spatial Location	The position of a sound source within a 1-, 2- or 3-dimensional space.
Spatialisation	The deliberate position of sound within a 3-dimensional space.
Sustained Sound	A continuous sound that lasts an indefinite length of time.
Timbre	Quality of sound; the sound made by each musical instrument represents a different timbre.
Threshold	The ambient sound level in the immediate environment.
Transient Sound	A sound that lasts for a discrete length of time.

B.2: Toolkit Terminology

Term	Definition
Abstract Widget Behaviour	Toolkit component that exposes the behaviour of the toolkit's widgets. For each widget this component: defines the behaviour of the widget, accepts events that occur to the widget, and translates the events into requests for presentation.
Control User Interface	Interface by which the developer of a system and the end-users of a system can control the presentation of Toolkit widgets used within a user-interface.
Control System	Toolkit component which maintains references to all other components of the Toolkit and thereby acts as the 'glue' that holds the system together and manages the communication between all the major components of the Toolkit.
Feedback Manager	The toolkit component which controls the distribution of feedback requests to the different module mappers. When it receives a request from an abstract widget behaviour component it passes a duplicate request onto all module mappers which then embellish the request with output module specific information.
Feedback Request	An abstract, modality independent request for presentation made by the Toolkit's widgets.
GEL Event	Generic Event Language event. The protocol used by the toolkit for feedback requests.
Media	The substance or agency by which information is conveyed to the user and vice-versa.
Mode	See Representational System.
Modality	The pair consisting of the representational system and media used to present information to a user.
Module Mapper	Each Toolkit widget has a module mapper for every output module used. Module mappers provide a link between the abstract requests made by the abstract widget behaviour and the concrete feedback generated by the different output modules.
Multimodal	The use of different modalities to present information to a user.
Presentational Resource	The means by which it is possible to present information to a user.
Output Module	A component, external to the toolkit, which translates abstract requests for feedback into concrete feedback.
Rendering Manager	A global toolkit component which receives and manages the requests for feedback from module mappers before they are passed on to output modules.
Resource Availability	The ability of a system to produce output in a particular modality that uses the resource.
Resource Sensitive	Aware and sensitive to the availability and/or suitability of presentational resources.
Resource Suitability	The suitability of the use of a particular resource for presentation with regard to users' comprehension of the feedback being presented.
Representational System	The style or nature of the interaction between the user and the computer incorporating the appropriate control actions both sides of the interaction may take. Also known as Mode.
Toolkit	A collection of widgets supported by a runtime architectural framework which manages their runtime operation.
Widget	A user interface object which defines specific interaction behaviour and a model of information presented to the user.
Widget Statechart	The widget component which defines the behaviour of the widget by translating input to the widget into abstract requests for feedback.

Appendix C: Raw Data For The Progress Bar Experiments

C.1: Introduction

This appendix contains the raw data from the two progress bar experiments referred to in Chapter 5. It also contains the instructions, descriptions of the workload ratings and questionnaires given to the participants.

Mental Demand

L o w   H i g h

Physical Demand

L o w   H i g h

Time Pressure

L o w   H i g h

Effort Expended

L o w   H i g h

Performance Level Achieved

L o w   H i g h

Frustration Experienced

L o w   H i g h

Annoyance Experienced

L o w   H i g h
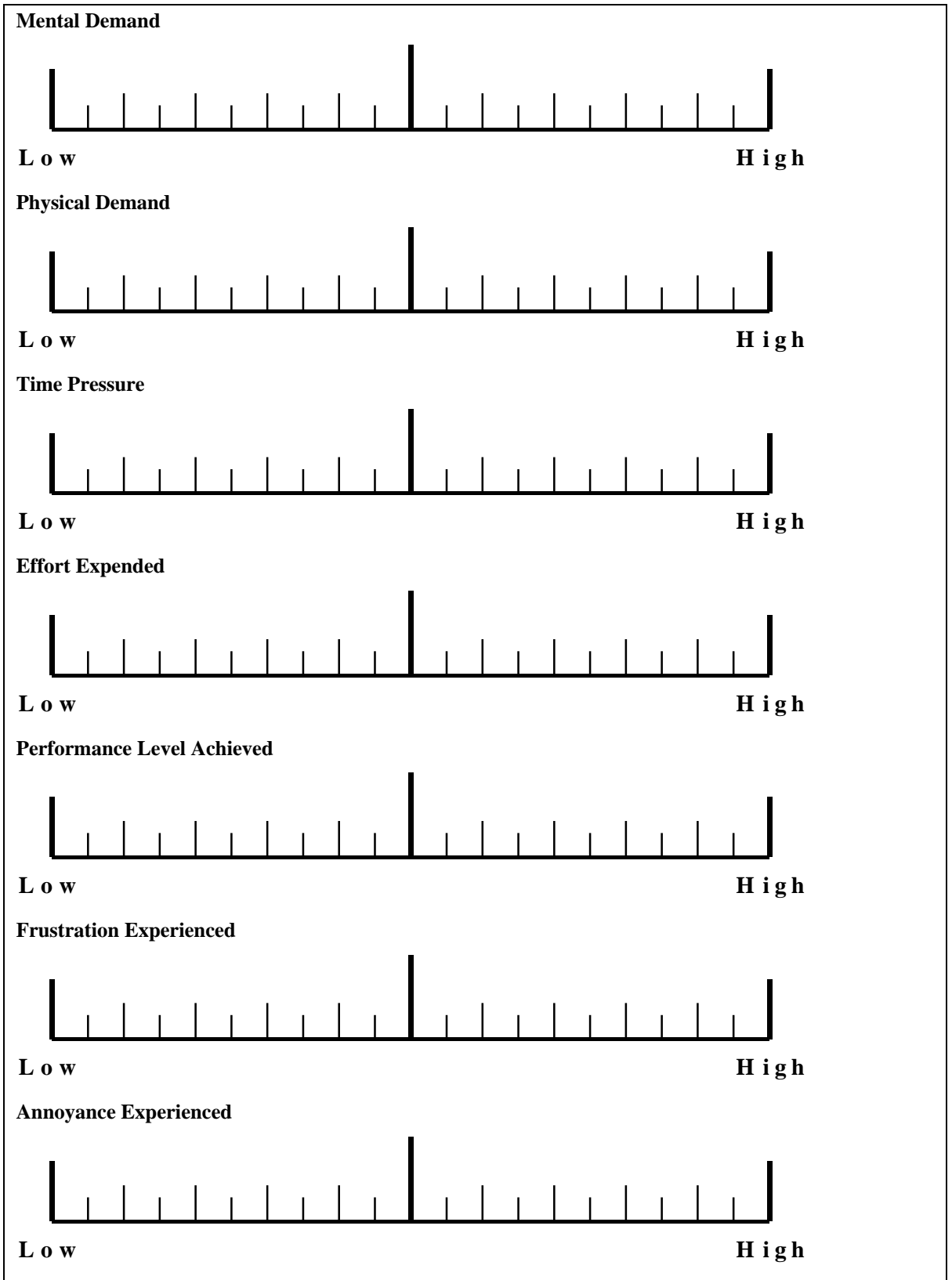
Appendix C Table 1 – Workload Charts used in the progress indicator experiments

Questionnaire

Could you now please spend two minutes answering the questions below. If you have any comments please feel free to write them in the space provided. Also please feel free to ask any further questions you may have.

Overall Preference

L o w　　　　　　　　　　　　　　　　　　　　　　　　**H i g h**
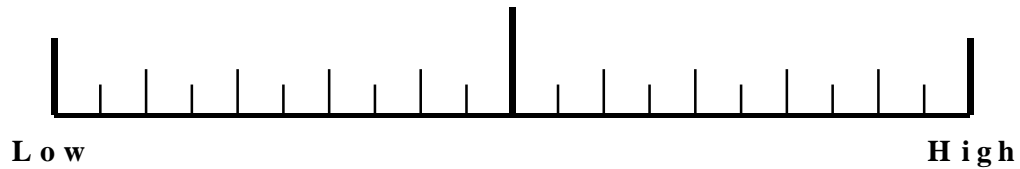
Typing Ability

Are you a touch typist? Please tick the appropriate box below.

Yes　　　　　　　　　　No

Comments

Please write any comments you have on the experiment or on the effectiveness of the sounds used with the progress bar

Appendix C Table 2 – Questionnaire used in progress indicator experiments.

Rating Scale Definitions		
Title	**Endpoints**	**Description**
Mental Demand	*Low/High*	How much mental, visual and auditory activity was required? (e.g. thinking, deciding, calculating, looking, listening, scanning, searching)
Physical Demand	*Low/High*	How much physical activity was required? (e.g. pushing, pulling, turning, controlling)
Time Demand	*Low/High*	How much time pressure did you feel because of the rate at which things occurred? (e.g. slowly, leisurely, rapid, frantic)
Effort Expended	*Low/High*	How hard did you work (mentally and physically) to accomplish your level of performance?
Performance Level Achieved	*Poor/Good*	How successful do you think you were in doing the task set by the experimenter? How satisfied were you with your performance? Don't just think of your success in terms of typing, but how you felt you performed.
Frustration Experienced	*Low/High*	How much frustration did you experience? (e.g. were you relaxed, content, stressed, irritated, discouraged?)
Annoyance Experienced	*Low/High*	How annoying did you find using the progress bars in the two experiments?
Overall Preference	*Low/High*	Rate your preference for the two experiments. Which one did you find the easiest? The one with sounds or the one without.

Appendix C Table 3 – Workload descriptions given to the participants when filling in the workload charts. The descriptions are based upon those used in [14].

C.2: First Progress Indicator Experiment Raw Data

This section contains the raw data for the first progress indicator experiment. First there is the raw workload data and this is followed by physical data.

Participant	Mental	Physical	Time	Effort	Performance	Frustration	Annoyance	Overall
S1	5	15	8	8	19	5	4	17
S2	7	5	5	6	14	5	6	16
S3	16	4	13	14	10	13	14	12
S4	9	13	13	9	15	5	6	13
S5	5	2	7	5	3	4	1	15
S6	3	5	7	7	10	7	4	12
S7	11	5	8	6	4	17	18	3
S8	4	2	10	1	11	2	6	14
S9	13	17	8	16	17	3	3	19
S10	11	8	12	7	10	10	10	14
S11	5	2	12	9	13	4	3	16
S12	10	3	9	6	16	4	7	14
S13	3	6	6	8	14	6	6	14
S14	10	11	12	10	14	7	7	14
S15	8	5	13	15	13	8	8	7
S16	15	13	12	17	18	5	9	14
Average	8.44	7.25	9.69	9.00	12.56	6.56	7.00	13.38
Total	135	116	155	144	201	105	112	214
Standard Deviation	4.13	4.96	2.75	4.41	4.49	3.90	4.27	3.79

Appendix C Table 4 - Workload Data (out of 20) for the audio condition in the first progress indicator experiment.

Participant	Mental	Physical	Time	Effort	Performance	Frustration	Annoyance	Overall
S1	11	17	8	12	18	9	10	11
S2	16	16	14	18	14	11	14	4
S3	15	4	10	14	8	8	8	10
S4	11	13	13	13	13	8	8	7
S5	8	2	8	10	3	7	5	7
S6	5	7	9	9	10	9	7	7
S7	13	10	8	12	6	19	17	4
S8	7	2	12	4	8	2	2	12
S9	18	18	15	19	17	10	10	12
S10	13	10	13	10	8	14	14	5
S11	14	14	7	16	12	8	6	11
S12	10	3	10	8	15	4	8	8
S13	6	9	12	12	16	7	7	14
S14	12	14	12	12	11	10	9	7
S15	12	9	8	16	9	3	4	12
S16	13	11	11	16	16	5	7	11
Average	11.50	9.94	10.63	12.56	11.50	8.38	8.50	8.88
Total	184	159	170	201	184	134	136	142
Standard Deviation	3.61	5.26	2.47	3.91	4.34	4.18	3.88	3.14

Appendix C Table 5 – Workload Data (out of 20) for the visual condition in the first progress indicator experiment.

Participant	Touch Typist	Total Time (msec)	Ave Time (msec)	Timeouts	Words	Words/Min	Glances
S1	Yes	693260	3149	0	292	25.27	16
S2	No	677178	2164	0	432	38.28	11
S3	No	679063	2296	0	352	31.10	44
S4	Yes	670917	1789	0	432	38.63	18
S5	No	702816	3768	0	377	32.18	17
S6	No	694690	3243	0	259	22.37	19
S7	No	704989	3924	0	212	18.04	22
S8	No	689473	2927	0	420	36.55	36
S9	No	670686	1769	0	258	23.08	44
S10	No	693354	3188	0	286	24.75	17
S11	No	695250	3286	0	342	29.51	18
S12	No	676382	2115	0	460	40.81	17
S13	No	676125	2087	0	299	26.53	16
S14	Yes	715680	4560	0	376	31.52	29
S15	No	690627	3019	0	216	18.77	19
S16	Yes	670609	1735	0	487	43.57	58
Average		687568.69	2813.69	0.00	343.75	30.06	25.06
Total		11001099	45019	0	5500	480.97	401
Standard Deviation		13665	853	0	87.12	7.89	13.36

Appendix C Table 6 – Physical Data for the audio condition in the first progress indicator experiment.

Visual Condition - Timings Etc.							
Participant	Touch Typist	Total Time (msec)	Ave Time (msec)	Timeouts	Words	Words/Min	Glances
S1	Yes	735715	5817	0	448	36.54	89
S2	No	766830	7770	0	313	24.49	48
S3	No	715973	4594	0	417	34.95	67
S4	Yes	775268	8305	0	381	29.49	54
S5	No	763750	7586	0	404	31.74	58
S6	No	724197	5114	0	214	17.73	77
S7	No	723240	5058	0	234	19.41	62
S8	No	706656	4007	0	335	28.44	114
S9	No*	698167	3484	0	264	22.69	87
S10	No	757256	7191	0	246	19.49	67
S11	No	716143	4594	0	403	33.76	86
S12	No	697871	3481	0	346	29.75	114
S13	No	691327	3066	0	376	32.63	85
S14	Yes	723325	5036	0	288	23.89	130
S15	No	761967	7482	0	288	22.68	77
S16	Yes	679454	2304	0	390	34.44	128
Average		727321.19	5305.56	0.00	334.19	27.63	83.94
Total		11637139	84889	0	5347	442.11	1343
Standard Deviation		29867	1868.43	0	72.35	6.17	25.72

Appendix C Table 7 - Physical Data for the visual condition in the first progress indicator experiment.

Participant		Download 1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S1	a	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	v	8	10	1	7	2	9	6	6	1	7	1	6	6	7	5	7
S2	a	1	1	1	2	1	1	1	0	0	2	0	0	1	0	0	0
	v	5	4	1	6	1	5	4	4	1	3	1	3	3	3	1	3
S3	a	4	2	1	4	2	3	2	2	2	4	1	2	3	6	4	2
	v	6	5	1	10	1	4	4	4	1	4	2	6	5	4	4	6
S4	a	0	2	1	4	0	0	2	1	1	1	1	0	1	1	1	2
	v	7	4	1	5	1	3	3	3	1	5	1	3	5	3	2	7
S5	a	1	1	1	1	1	1	2	1	1	1	1	1	1	1	1	1
	v	9	3	1	8	1	2	5	5	1	4	1	4	3	3	2	6
S6	a	1	2	1	2	1	1	1	2	1	1	1	1	1	1	1	1
	v	8	4	1	9	1	7	6	6	1	5	1	6	7	4	5	6
S7	a	1	1	1	1	1	2	2	1	1	1	2	1	1	3	1	2
	v	3	5	5	6	1	3	4	3	1	4	1	5	5	4	4	8
S8	a	2	2	1	3	2	4	2	1	1	3	2	1	3	2	3	4
	v	14	9	3	16	1	8	6	7	1	11	3	8	6	5	6	10
S9	a	5	3	2	5	1	2	4	5	1	3	3	2	2	4	1	1
	v	8	3	1	11	1	5	5	7	1	14	1	8	9	4	5	4
S10	a	2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	v	5	4	1	9	4	3	4	3	2	6	1	5	5	6	3	6
S11	a	1	1	1	1	1	1	2	1	2	1	1	1	1	1	1	1
	v	6	7	3	9	3	6	5	6	1	8	1	6	5	5	6	9
S12	a	1	1	1	1	1	1	1	1	1	1	1	2	1	1	1	1
	v	8	8	4	15	4	10	5	6	2	8	2	8	8	9	5	12
S13	a	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	v	10	5	1	11	5	7	5	3	1	8	1	5	5	4	5	9
S14	a	4	2	2	3	1	2	2	2	1	2	1	1	1	2	1	2
	v	11	10	2	15	2	12	7	7	1	14	2	11	6	10	10	10
S15	a	1	1	1	1	1	1	1	1	1	1	2	1	2	2	1	1
	v	7	5	2	6	2	4	3	6	1	9	1	9	5	7	4	6
S16	a	7	5	1	8	1	4	3	5	1	4	1	4	3	2	3	6
	v	17	13	4	16	2	10	4	9	1	13	2	9	7	8	5	8

Appendix C Table 8 – Number of glances at each download made by each participant in both conditions.

Participant		Download															
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S1	a	2251	3080	2592	2696	2789	3958	3545	3357	2845	3175	2781	4372	3027	3424	2980	3134
	v	2962	8121	6875	876	13769	2110	6492	8544	2614	5399	5640	3474	3521	4384	15991	2305
S2	a	2152	2897	2224	3531	2529	2292	1735	1700	1795	2019	1727	2036	2347	1817	1926	1906
	v	1086	3626	20096	5595	15407	8208	9431	5651	18137	4443	1155	4243	2710	17762	5147	1632
S3	a	1901	2160	4138	2878	1714	2180	2767	2087	1440	1678	2433	1988	2507	1965	3213	1695
	v	3946	1493	11606	4142	1706	4917	5246	5239	2373	2610	2478	5113	2046	2198	11678	6721
S4	a	1999	2110	1745	1734	1782	1665	2042	1703	974	1521	1517	1858	1575	2101	2562	1737
	v	737	1361	3220	7010	15530	1858	11510	8255	32465	651	17816	12271	7948	1774	7102	3381
S5	a	2759	2717	4408	3120	3514	5398	4486	3931	3471	3435	3960	4346	3871	3719	3600	3553
	v	917	2526	14727	4052	19613	7363	2464	2852	16414	874	16019	2953	2708	10224	16578	1097
S6	a	2653	2964	2659	2916	3369	2647	3451	3176	2744	3566	2640	5232	3294	3372	3820	3388
	v	2468	3183	6019	5907	15024	2946	5075	2810	3926	5314	12361	2997	4352	2815	2652	3977
S7	a	2965	5782	3218	2558	3425	4732	5312	3822	3267	3895	3478	4583	4128	4751	3148	3725
	v	11745	3861	5104	4298	1420	8483	4192	8480	1218	2360	2911	4979	3571	2817	9342	6155
S8	a	2781	2245	2451	2313	2675	2470	2238	2871	2444	2139	2927	8879	2452	2868	2235	2847
	v	2364	3164	4294	1709	15209	3060	2019	3433	3370	2772	2205	6388	1972	5033	4575	2560
S9	a	1613	1691	2076	1865	1895	2068	1598	1326	680	1941	2001	2334	2049	1392	1693	2091
	v	1740	3251	6502	3967	2702	2567	3414	2762	1114	2849	8305	2323	2069	1814	9244	1130
S10	a	2956	2715	3077	3762	4360	3001	2824	3013	3262	3152	3850	3033	3225	2849	2829	3107
	v	6213	4307	16351	4113	8072	2309	5024	6794	2418	3479	6737	10698	4940	17084	12112	4412
S11	a	2730	3083	2552	3039	3209	2820	3406	3365	4101	3286	3601	3169	4548	3610	3187	2881
	v	2130	5875	2380	3031	3044	3707	2375	4505	3259	3714	14202	2296	7692	5337	7287	2674
S12	a	2648	1820	2515	2289	2044	1636	1952	2711	1897	1751	2553	2194	1721	2190	1780	2144
	v	6376	3100	3230	1877	5330	2335	6588	2029	4377	789	1843	6075	3186	4345	2934	1295
S13	a	3133	2895	2093	2432	2227	1752	1639	1958	1583	1721	1549	1789	1623	1398	1598	4015
	v	155	1153	5885	2130	2562	1068	2035	2272	12023	2581	1082	3724	1601	2541	6325	1023

S14	a	3009	4058	3615	3983	4589	3109	4512	5686	5092	8440	4459	3660	3669	3980	7419	3691
	v	3970	4087	4467	3425	3159	5814	5428	11425	7025	5683	2721	4896	5852	3811	4703	4125
S15	a	3296	3021	2804	2402	3212	3030	2701	2780	2749	3138	2886	2738	3694	3462	3398	3007
	v	3600	2147	3688	8528	8107	8372	5720	7645	9696	7200	7334	10716	4806	4675	21402	6078
S16	a	1882	2134	2182	707	2109	2043	1955	1488	1958	1248	579	1721	1089	2759	2320	1595
	v	733	2004	1709	3839	1912	1666	1804	3259	5101	688	2111	3386	3031	1994	2921	708

Appendix C Table 9 – Time taken (in msec) to press button after each download completed.

C.3: Second Progress Indicator Experiment Raw Data

This section contains the raw data for the second progress indicator experiment. First there is the raw workload data and this is followed by physical data.

Participant	Mental	Physical	Time	Effort	Performance	Frustration	Annoyance	Overall
S1	14	3	15	12	9	12	16	17
S2	12	3	8	13	6	11	10	12
S3	12	11	7	10	9	12	11	16
S4	15	3	12	10	13	6	7	17
S5	11	5	10	11	11	12	12	16
S6	20	11	20	15	1	16	1	15
S7	7	7	13	9	14	3	6	15
S8	1	5	5	10	11	6	6	15
S9	9	4	6	7	12	6	5	15
S10	13	6	13	8	15	4	6	16
S11	11	6	5	9	16	6	4	13
S12	12	2	10	10	15	10	7	13
S13	11	9	10	12	16	9	8	16
S14	8	5	11	9	9	13	12	11
Total	156	80	145	145	157	126	111	207
Avg	11.14	5.71	10.36	10.36	11.21	9.00	7.93	14.79
Standard Deviation	4.31	2.89	4.16	2.10	4.23	3.86	3.89	1.85

Appendix C Table 10 – Workload data (out of 20) for the audio condition in the second progress indicator experiment.

Participant	Mental	Physical	Time	Effort	Performance	Frustration	Annoyance	Overall
S1	19	3	16	17	5	18	17	11
S2	14	3	6	15	4	6	4	17
S3	14	11	9	14	9	7	8	13
S4	18	4	14	16	12	16	13	7
S5	15	5	13	15	7	13	9	14
S6	15	19	20	12	11	11	11	10
S7	15	15	11	13	10	8	11	9
S8	5	8	5	14	12	6	8	10
S9	13	6	11	9	10	7	9	13
S10	15	15	11	13	12	12	12	11
S11	15	9	11	16	12	13	16	7
S12	14	2	7	12	13	12	12	9
S13	16	13	10	16	13	14	13	14
S14	10	5	9	11	10	10	10	11
Total	198	118	153	193	140	153	153	156
Avg	14.14	8.43	10.93	13.79	10.00	10.93	10.93	11.14
Standard Deviation	3.37	5.37	3.97	2.26	2.85	3.77	3.36	2.82

Appendix C Table 11 – Workload data (out of 20) for the visual condition in the second progress indicator experiment.

Part	No.	Comp	Stalls	Stop	Total Time	Time(C)	Time(S)	Stop Rate	Stop %	Stop S	Stop M	Stop F	Total	WPM	Bytes/sec
S1	20	13	6	1	705063	2332	5062	6000	1	0	0	1	357	30.4	30079
S2	22	13	5	4	806829	2719	3867	26375	72	1	3	0	301	22.4	25688
S3	20	14	5	1	750847	2379	5411	5000	29	1	0	0	379	30.3	30328
S4	22	14	6	2	786446	2314	6788	33000	41	1	0	1	307	23.4	27539
S5	20	14	5	1	763305	2495	7382	4000	29	1	0	0	506	39.8	29811
S6	21	14	5	2	758397	2029	3898	32500	57	2	0	0	334	26.4	27422
S7	20	14	6	0	754971	2344	4597			0	0	0	440	35	30167
S8	20	14	6	0	796558	2475	11279			0	0	0	290	21.8	28575
S9	20	13	6	1	747103	1991	10989	42500	37	1	0	0	346	27.8	27738
S10	20	13	5	2	693250	1807	5722	4500	14	1	0	1	354	30.6	30600
S11	23	13	4	6	847920	3626	6150	24750	61	3	2	1	574	40.6	28270
S12	20	13	5	2	698453	2282	4692	3000	14	1	0	1	288	24.7	30381
S13	20	14	6	0	771724	2598	6746			0	0	0	373	29	29501
S14	21	14	5	2	793034	2536	8760	31000	63	1	1	0	349	26.4	26211
Total	289	190	75	24	10673900	33927	91343	212625	418	13	6	5	5198	409	402310
Avg	20.6	13.57	5.36	1.71	762421.43	2423.36	6524.50	19329.55	38.00	0.93	0.43	0.36	371.2	29.19	28736.43
S.D.	1.0	0.51	0.63	1.63	43550	425.68	2379.2	14889.6	23.16	0.83	0.94	0.50	83.0	5.89	1626.03

Appendix C Table 12 – Physical data for the audio condition in the second progress indicator experiment. Time(C): time to press the button after a download completed. Time(S): time to press the button after a download stalled. Stop S: Number of downloads stopped when the download was slowing. Stop M: Number of downloads stopped when the download at a constant rate. Stop F: Number of downloads stopped when the download was speeding up. WPM: Words per minute.

Part	No	Comp	Stalls	Stop	Total Time	Time(C)	Time(S)	Stop Rate	Stop %	Stop S	Stop M	Stop F	Total	WPM	Bytes/sec
S1	20	14	5	1	762013	2985	6497	8000	29	1	0	0	242	19.1	29850
S2	20	14	5	1	803532	4054	16056	30000	19	1	0	0	401	29.9	28326
S3	20	14	6	0	783164	2592	8902			0	0	0	251	19.2	29049
S4	20	13	5	2	704324	2844	6440	12000	14	1	0	1	373	31.8	30122
S5	20	14	5	1	783779	3605	9297	9000	28	1	0	0	329	25.2	29049
S6	20	14	5	1	799134	4215	9459	3000	29	1	0	0	502	37.7	28468
S7	20	14	5	1	773878	4095	6116	9000	28	1	0	0	284	22	29425
S8	20	14	6	0	788047	3868	6587			0	0	0	339	25.8	28865
S9	20	14	6	0	822228	4376	11305			0	0	0	261	19	27671
S10	20	14	5	1	771368	4552	7985	28000	20	1	0	0	502	39	29501
S11	20	13	5	2	747181	3466	9865	6500	16	1	0	1	327	26.3	28388
S12	20	14	6	0	807449	4072	9507			0	0	0	415	30.8	28185
S13	20	13	4	3	679174	3358	4932	21666	32	2	1	0	202	17.8	32549
S14	20	14	6	0	776988	2758	7479			0	0	0	437	33.7	29311
Total	280	193	74	13	10802259	50840	120427	127166	215	10	1	2	4865	377	408759
Avg	20.0	13.79	5.29	0.93	771589.93	3631.43	8601.93	14129.56	23.89	0.71	0.07	0.14	347.5	26.96	29197.07
S.D.	0	0.43	0.61	0.92	39193.58	642.71	2788	9860.7	6.62	0.61	0.27	0.36	95.2	7.08	1180.8

Appendix C Table 13 – Physical data for the visual condition in the second progress indicator experiment. Time(C): time to press the button after a download completed. Time(S): time to press the button after a download stalled. Stop S: Number of downloads stopped when the download was slowing. Stop M: Number of downloads stopped when the download at a constant rate. Stop F: Number of downloads stopped when the download was speeding up. WPM: Words per minute.

S1	No. Notes	4	4	11	6	11	12	9	8	12	10	3	11	8	10	9
	Cat	S	S	XL	M	L	XL	L	M	XL	L	S	L	M	L	L
	No. Notes	3	3	4	10	5	6	7	12	9	5	7	8	7	5	6
	Cat	S	S	S	L	M	M	L	XL	L	S	M	M	M	S	M
S2	No. Notes	4	4	11	6	11	12	9	8	12	10	3	11	8	10	9
	Cat	S	S	L	M	XL	XL	L	M	XL	XL	M	XL	L	L	L
	No. Notes	3	3	4	10	5	6	7	12	9	5	7	8	7	5	6
	Cat	S	S	M	XL	L	L	L	XL	L	M	M	L	L	M	M
S3	No. Notes	4	4	11	6	11	12	9	8	12	10	3	11	8	10	9
	Cat	S	S	XL	M	XL	XL	L	L	XL	XL	S	XL	L	XL	L
	No. Notes	3	3	4	10	5	6	7	12	9	5	7	8	7	5	6
	Cat	S	S	S	XL	M	M	L	XL	XL	M	L	L	L	M	M
S4	No. Notes	4	4	11	6	11	12	9	8	12	10	3	11	8	10	9
	Cat	S	S	L	M	L	L	L	M	L	L	S	L	M	M	M
	No. Notes	3	3	4	10	5	6	7	12	9	5	7	8	7	5	6
	Cat	S	S	S	L	S	M	M	L	L	S	M	M	M	S	S
S5	No. Notes	4	4	11	6	11	12	9	8	12	10	3	11	8	10	9
	Cat	S	S	L	M	L	XL	L	M	L	M	S	L	M	L	M
	No. Notes	3	3	4	10	5	6	7	12	9	5	7	8	7	5	6
	Cat	S	S	S	XL	M	M	M	XL	L	S	M	L	M	S	S
S6	No. Notes	4	4	11	6	11	12	9	8	12	10	3	11	8	10	9
	Cat	S	S	L	M	L	XL	L	L	XL	XL	S	XL	M	L	L
	No. Notes	3	3	4	10	5	6	7	12	9	5	7	8	7	5	6
	Cat	S	S	M	XL	M	M	L	XL	XL	M	L	L	L	M	M
S7	No. Notes	4	4	11	6	11	12	9	8	12	10	3	11	8	10	9
	Cat	S	S	XL	M	XL	XL	L	L	XL	XL	S	L	L	XL	XL
	No. Notes	3	3	4	10	5	6	7	12	9	5	7	8	7	5	6
	Cat	S	S	M	L	M	L	L	XL	L	M	M	L	L	M	L
S8	No. Notes	4	4	11	6	11	12	9	8	12	10	3	11	8	10	9
	Cat	S	S	XL	S	L	L	M	S	XL	L	S	XL	M	L	L
	No. Notes	3	3	4	10	5	6	7	12	9	5	7	8	7	5	6
	Cat	S	S	S	L	S	S	M	XL	L	S	S	M	S	S	S
S9	No. Notes	4	4	11	6	11	12	9	8	12	10	3	11	8	10	9
	Cat	S	S	XL	M	XL	XL	M	M	XL	M	S	L	M	M	M
	No. Notes	3	3	4	10	5	6	7	12	9	5	7	8	7	5	6
	Cat	S	S	S	M	S	M	M	L	L	M	M	M	M	S	S
S10	No. Notes	4	4	11	6	11	12	9	8	12	10	3	11	8	10	9
	Cat	S	S	L	M	L	XL	M	M	L	L	S	L	M	L	L
	No. Notes	3	3	4	10	5	6	7	12	9	5	7	8	7	5	6
	Cat	S	S	M	L	M	M	M	XL	L	M	L	L	L	M	M
S11	No. Notes	4	4	11	6	11	12	9	8	12	10	3	11	8	10	9
	Cat	S	S	L	M	XL	XL	L	M	XL	L	S	L	L	L	M
	No. Notes	3	3	4	10	5	6	7	12	9	5	7	8	7	5	6
	Cat	S	S	M	L	M	L	L	XL	L	M	L	L	L	M	M
S12	No. Notes	4	4	11	6	11	12	9	8	12	10	3	11	8	10	9
	Cat	S	S	XL	M	XL	XL	L	M	XL	L	S	XL	M	L	L
	No. Notes	3	3	4	10	5	6	7	12	9	5	7	8	7	5	6
	Cat	S	S	S	XL	S	M	M	XL	L	M	M	L	L	S	M
S13	No. Notes	4	4	11	6	11	12	9	8	12	10	3	11	8	10	9
	Cat	S	M	XL	M	L	XL	L	M	L	L	S	XL	L	XL	L
	No. Notes	3	3	4	10	5	6	7	12	9	5	7	8	7	5	6
	Cat	S	S	S	L	M	M	L	XL	XL	M	L	L	M	M	M
S14	No. Notes	4	4	11	6	11	12	9	8	12	10	3	11	8	10	9
	Cat	S	S	XL	S	L	XL	L	M	L	L	S	L	M	M	M
	No. Notes	3	3	4	10	5	6	7	12	9	5	7	8	7	5	6
	Cat	S	S	S	L	M	M	M	L	L	M	M	M	M	S	S

Appendix C - Table 14 – Raw data for the scope sound evaluation. The categorisations (Cat) are small (S), Medium (M), Large (L) and Extra Large (XL).

Appendix D: Guidelines and Requirements

This appendix provides a quick reference to the guidelines and requirements that are gathered throughout this thesis.

D.1: Guidelines for the Use of Audio

Guideline 1	If several background sounds are playing simultaneously the absence of one of these sounds may not be sufficient feedback to alert a user to a problem.

Guideline 2	The use of sound can provide relevant information by being mapped to the data model as well as the interaction technique.

Guideline 3	The absence of a sound where one is expected is demanding feedback if the sound would have been as a result of a user interaction, not as a piece of background information.

Guideline 4	If possible, the different sounds used to represent an interaction should be ranked in order of importance. In this way, it is possible to determine which sounds should be played if it is not possible to play all the desired sounds.

Guideline 5	Limit the number of different sounds required by analysing the way a user interacts with an object rather than naïvely mapping a different sound to every different event.

Guideline 6	To prevent annoyance, audio feedback should be limited to that which provides useful information to users. If the information provided to the user is either irrelevant or provided by another source the sounds could become annoying.

Guideline 7	Blending sounds together rather than playing them in isolation can ensure that the audio feedback is less intrusive.

Guideline 8	When structuring complex earcons consider using instruments and rhythm in an analogous way to the structure used in music.

Guideline 9	When using frequency as a means to convey information avoid using more than 6 notes per second as users can find more than 6 notes difficult to differentiate.

D.2: Requirements for a Toolkit of Multimodal Widgets

Requirement 1	The full behaviour of the toolkit's widgets should be exposed allowing suitable forms of presentation to be generated.

Requirement 2	The toolkit should be able to determine if it is not possible to play all the requested sounds and, if so, modify the feedback appropriately.

Requirement 3	It must be possible to group widgets together to allow their feedback to be co-ordinated appropriately.

Requirement 4	A toolkit of multimodal widgets must allow users to control the form of presentation used for its widgets.

Requirement 5	The toolkit must be able to manage any interference between different pieces of feedback.

Requirement 6	The toolkit must be able to monitor the presentation of a widget over time and, if appropriate, adjust the presentation.

Requirement 7	The rules which manage the presentation of the toolkit's widgets must be modifiable to ensure that the flexibility afforded by the support for different output modalities is matched.

Requirement 8	The toolkit should be able to modify the use of different presentation modalities according to their suitability.

Requirement 9	The sensors which are used to detect the context of the toolkit's widgets should be external to the toolkit so that they can be modified as required.

Requirement 10	It should be possible to develop different tools for the toolkit, each of which allow the presentation of the interface to be modified in different ways.

Requirement 11	The toolkit should conform to an existing API to minimise the cost to engineers who build new applications and/or modify existing applications so they can use the toolkit's widgets.

D.3: Design Guidelines for a Toolkit of Multimodal Widgets

Design Guideline 1	Separating the application model from the presentation enables the presentation to be changed.

Design Guideline 2	Describing the required presentation in terms of abstract, implementation-independent objects allows the separation of the presentation from the application model meaning the concrete presentation can be freely changed or supplemented.

Design Guideline 3	Separating the output from the input makes it easier to modify the presentation of a widget (albeit with the overhead of communication between the input and output components).

Design Guideline 4	A client server architecture facilitates multimodality by enabling the client to use multiple servers to provide a service at the cost to the client of determining which server to use.

Design Guideline 5	By providing a component which can intercept and modify requests for presentation global control over the resources used can be obtained.

Design Guideline 6	Providing hooks onto different aspects of the behaviour of widgets means that it is possible to change the presentation of the widgets without the explicit knowledge of the behaviour.

Design Guideline 7	Although information about the availability of presentational resources and the suitability of presentational resources will often come from different sources they can both be considered as contextual information and therefore can be managed using similar mechanisms.

Appendix E: Summary of Intra-Toolkit Communication

Summary of intra-toolkit communications. For each form of communication, its form and where its used is given.

Communication	Form	Where Used
Feedback Requests	<Widget Type> <State> <Event> <State Info> <Module Parameters>

<State Info> ::= <Name> <Value> \| <State Info> <Name> <Value>
Module Parameters ::= <Name> <Value> \| <State Info> <Name> <Value> | Abstract Widget Feedback Manager Module Mapper Rendering Manager Output Module |
| **Transformation Flags** | <Module> <Widget> <Flag Type> | Rendering Manager Output Module |
| **Options or Abilities** | <Options>

<Options> ::= <Name> <Value> \| <Name> <Options> \| <Options> <Options> | Control System Output Module Sensor |
| **Commands** | <Widget> <Module> <Command> <Parameter List>

<Parameter List> ::= <Name> <Value> \| <Parameter List> <Name> <Value> | Rendering Manager Control System Sensor Control Panel |