31st October 2003

Greg Howell
16 Hume Rd
High Wycombe WA 6057

Professor Syed Islam
School of Electrical and Computer Engineering
Curtin University of Technology
Kent St
Bentley WA 6102

Dear Sir,

Re: Final Year Project

As part of the requirement for Bachelor of Engineering (Computer Systems Engineering), I hereby submit this final year project thesis entitled "Development of a Screen Reader for Mac OS X" for your perusal.

Yours Sincerely,

Greg Howell

SYNOPSIS

This thesis outlines the design, development and testing of a screen-reader application for the Macintosh operating system, Mac OS X. Included is a review of existing screen-reader products, for both the Macintosh and other platforms, and an examination of assistive architectures in general. The product is a screen-reader application that will enable the vision-impaired to use the graphical user interface of the Mac OS X operating system. A screen-reader converts the graphical information that comprises the user interface into textual information. This textual information can then be conveyed to the vision-impaired user using speech synthesis or other technologies.

ACKNOWLEDGEMENTS

The author would like to express his thanks and gratitude to the following organizations or people:

- Apple University Consortium (AUC), for providing a software and hardware grant that allowed for the development and testing of this project.

- Iain Murray, for his guidance and advice throughout the course of the project.

- Andrew Brownell, for his assistance in proofreading this document.

TABLE OF CONTENTS

TABLE OF FIGURES

TABLE OF TABLES

1. INTRODUCTION

In the last decade or so, the personal computer has moved from being a useful item for business to being an essential part of everyday life. The modern personal computer is employed extensively in the education and training environment, is an essential communications tool and is used increasingly as a form of entertainment and recreation.

With the expansion and enhancement of the Internet, more and more information is being made available in a digital format. In some areas, the Internet is not only replicating but is replacing traditional media forms. Computer literacy and access is no longer desirable but it is essential. In an increasingly large number of fields of employment, the ability to use a computer is assumed.

For these reasons it is essential that all people in society can make use of computers. This includes those with disabilities. In recent years, guidelines and laws have been introduced that ensure that people with disabilities can access the wealth of information made available in digital format via the computer.

Disabled people face many difficulties when using a computer. Those with physical disabilities may find using a keyboard or mouse in an effective manner impossible. Those with sensory disabilities such as vision impairment or deafness may experience difficulty receiving some forms of feedback that the computer may provide. Those with learning difficulties may find the format of information provided to them via the computer difficult to interpret. All of these disabilities limit or prevent effective computer access. From an engineering viewpoint, making the computer accessible to the disabled is a multi-disciplined effort, involving research and development in both hardware and software areas. Making the computer accessible also involves the field of psychology as usability and visualisation issues arise.

Vision impairment is one disability that makes accessing computers difficult. In Western Australia, it is estimated that 1.36% of people (approximately 24,900 people) are blind or vision-impaired (Association for the Blind of Western Australia, 2001). Around two-thirds of the vision-impaired population of Western Australia are over 65 years of age. The number of blind or vision-impaired people is expected to increase over the next fifteen years by around 50%. ABWA states that the increase is linked to age related vision conditions that cannot be corrected. With an ageing population, providing computer access to the vision impaired is an issue that will increase in significance with time. Murray (2001) states that there is a "demographic trend toward a growing elderly population (particularly as the "baby boom" generation ages)". This trend suggests a significant sector of the population will be elderly and financially independent. These people will require assistive technologies such as screen readers to use computers and will make up a significant proportion of the market for such technologies.

Those with vision impairments face many difficulties when using computers. For those with partial blindness, the text on the screen may prove too small to be read.

This can be overcome with various screen magnification utilities that enlarge parts of the screen for users. Other vision-impaired users may find software that inverts or alters the colour scheme displayed on the screen useful in interpreting the visual feedback that the computer provides. For those people with total blindness, accessing a computer without accessibility aids is impossible.

A screen reader is an accessibility aid that allows a vision-impaired or blind person to use a computer. A screen reader is a piece of software that monitors the properties of the graphical user interface. The screen reader then interprets what is happening and converts the visual information into textual information that is then conveyed to the user. This information may be conveyed in an audible format via speech synthesis or conveyed in a braille format via a refreshable braille display device.

This report focuses on the design, development and testing of a screen reader for the Macintosh OS X operating system. An overview of screen reader theory is included and a review of existing products for other platforms is provided. A discussion of the technologies required by a screen reader is also included and a brief overview of the development environment used during implementation is provided. The screen reader makes use of Apple's speech synthesis technologies to provide feedback to the vision-impaired user as they navigate the graphical user interface.

2. SCREEN READER THEORY

2.1. Overview of Screen Reader Theory

This section focuses on the theory of the technologies behind screen readers and reviews a number of existing screen reader products. Alternatives to screen readers are also discussed. Accessibility APIs, a feature of modern operating systems that significantly improve the accuracy and aid the development of screen readers, are also examined and compared.

2.2. Basics of Screen Readers

A basic screen reader consists of three main components:

- An engine that interrogates the graphical user interface and produces information concerning what is happening on the screen
- An interpretive engine that processes the information, builds a model of what is happening on the screen, converts the model into a textual format and parses the textual model to extract pertinent information
- A speech synthesis engine that takes the textual information and presents it to the user in an aural format

A screen reader can be viewed as software that converts information from one format (graphical information that is displayed on the screen) to another format (textual information that can be spoken to the user). A basic overview of the major components of a screen reader is shown in the figure below. This illustrates the three primary components of a basic screen reader described above.

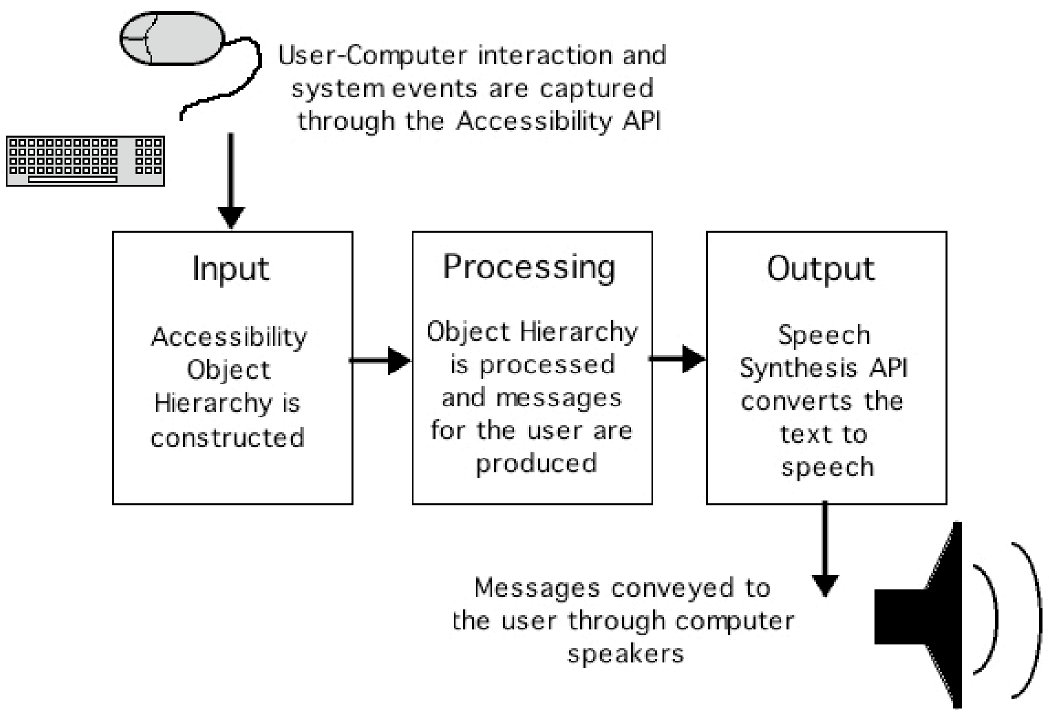

User-Computer interaction and system events are captured through the Accessibility API
Input
Accessibility Object Hierarchy is constructed
Processing
Object Hierarchy is processed and messages for the user are produced
Output
Speech Synthesis API converts the text to speech
Messages conveyed to the user through computer speakers

Figure 1: Overview of Screen Reader

There are two general approaches to interrogating the graphical user interface produced by the operating system. The first is by way of examining the video signals that are being sent to the monitor. This is usually achieved by examining the video memory. These are processed and interpreted in order to discover what is happening on the screen. This approach is both computationally intensive and difficult to develop. It relies on a large amount of information being hard coded into the screen reader, making it difficult to update as graphical user interfaces are updated. The second approach involves linking the screen reader into an accessibility API (Application Programming Interface) that is usually provided by the vendor of the operating system or operating system interface. This API provides information in a textual format about the components that make up the graphical user interface. This approach allows the screen reader to have greater accuracy as it can interpret the user interface as it is should be interpreted. This interpretation is defined by the accessibility API. Making use of the accessibility API allows the screen reader to be modified for new interface designs and elements in a straightforward manner. This is because all information used by the screen reader is obtained from the accessibility API, and this API will be updated as the user interface is updated.

The interpretive component of the screen reader is responsible for taking the information provided by the video interpretation or accessibility API and extracting some meaning from it. Ideally, it needs to be able to have some understanding of what the user is trying to do with the computer if it is going to have any success in conveying this information to the user. The interpretive component provides whatever intelligence (artificial as it is) the screen reader possesses. The richer the interpretive engine of the screen reader the easier it is to use.

The speech synthesis component of the screen reader is generally provided as a feature of the operating system. The textual information produced by the interpretive engine is spoken to the user through the sound output system via the speech synthesis capabilities of the operating system. In cases where there is no speech synthesis support, this component must be developed separately.

2.3. Approaches to Building Screen Readers

There are two distinct approaches to developing screen reader applications:

- Using an accessibility API
- Interpreting video signals

Using an accessibility API is the desired approach as it provides greater accuracy and usually allows the screen reader to extract information in a significantly more efficient manner when compared to interpreting video signals. Using an accessibility API does have limitations in that you are limited to whatever information the API knows about. Accessibility APIs generally rely on the applications that they are interrogating being coded in a manner that is aware of the accessibility API. On modern operating systems, there are many ways to construct graphical user interfaces, and not all of these methods provide information to the accessibility API. As a result, there are always applications and features on operating systems that cannot be accessed using the accessibility API. These areas may or may not be accessible via the approach of interpreting video signals.

Despite this limitation, building a screen reader application using an accessibility API is by far the most desirable approach. Using the accessibility API approach allows screen readers to be adapted to new versions of the operating system and applications in a more structured and straightforward manner.

2.4. Accessibility Architectures

2.4.1. Overview

This section examines a range of accessibility architectures that are available for modern operating systems. Architectures from three popular platforms (Macintosh, Linux and Windows) are examined and compared.

2.4.2. Apple's Accessibility API

The accessibility API implemented in Mac OS X consists of a hierarchy of accessibility objects. Accessibility objects may be buttons, windows, menu items or any other element that makes up the graphical user interface. These accessibility objects are arranged in a hierarchy, with objects down the hierarchy as children and objects up the hierarchy as parents. For example, a "menu item" accessibility object is the child of a "menu" accessibility object, and the "menu" accessibility object the parent of the "menu item" accessibility object.

Accessibility objects have the following attributes (Apple Computer Inc., 2002a):

- Role
- Role description
- Value (eg. the text string contained in a text field accessibility object)
- A list of the accessibility objects that are children of the object
- A localized help string

Accessibility objects also contain information pertaining to the action that they perform. Examples of these are “press” for a button, “pick” for a checkbox and “increment” for a slider.

The following figure illustrates two example accessibility object hierarchies. The numbering system illustrated on the diagram shows the parent-child relationships that exist in each hierarchy. For instance, the menu item accessibility object has a number of 4, and is the child of the menu accessibility object, which has a number of 3. The hierarchy can be viewed in manner such that higher objects (with smaller numbers) contain lower objects (with higher numbers). Actual instances of the examples in the following figure are shown later in this section.

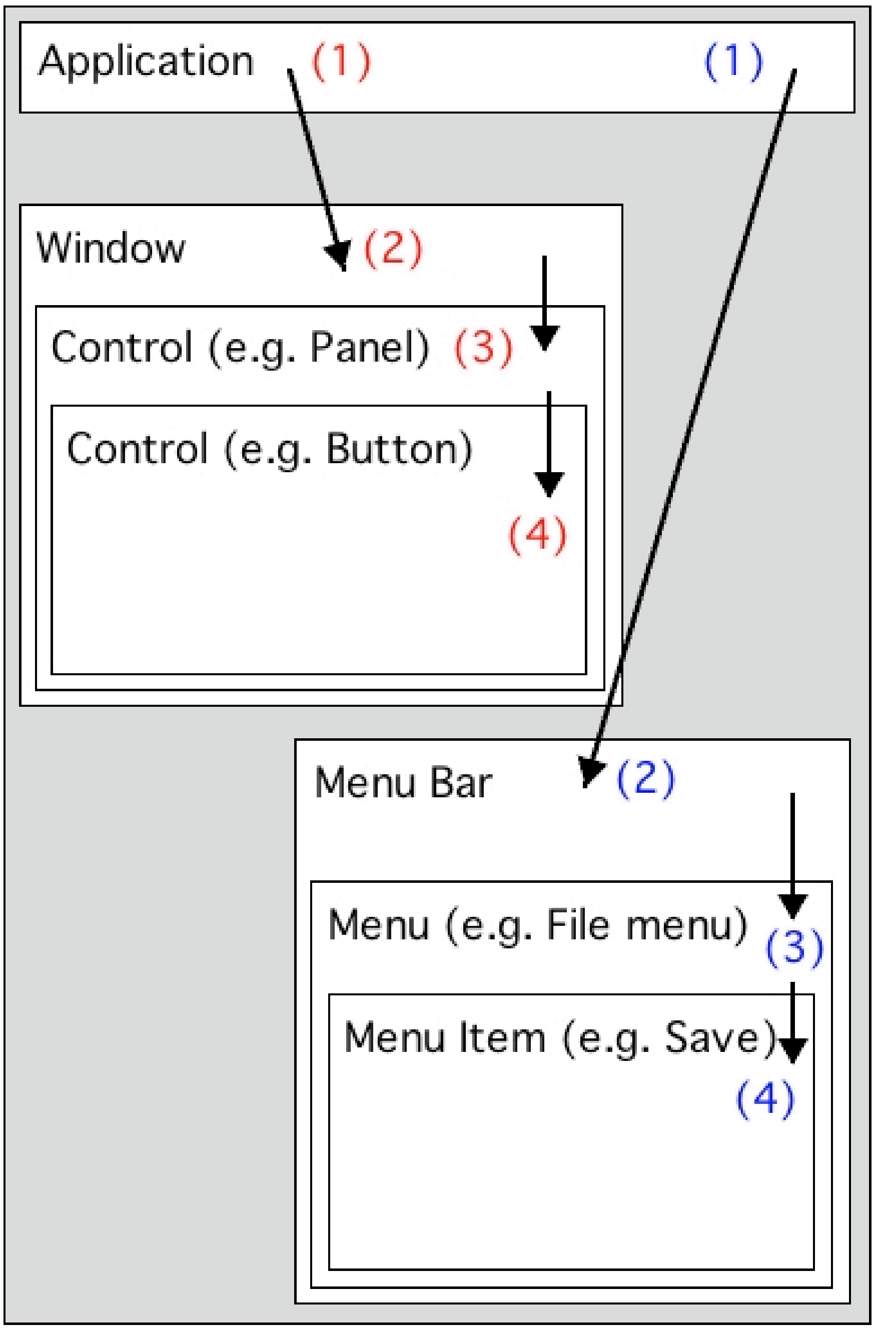
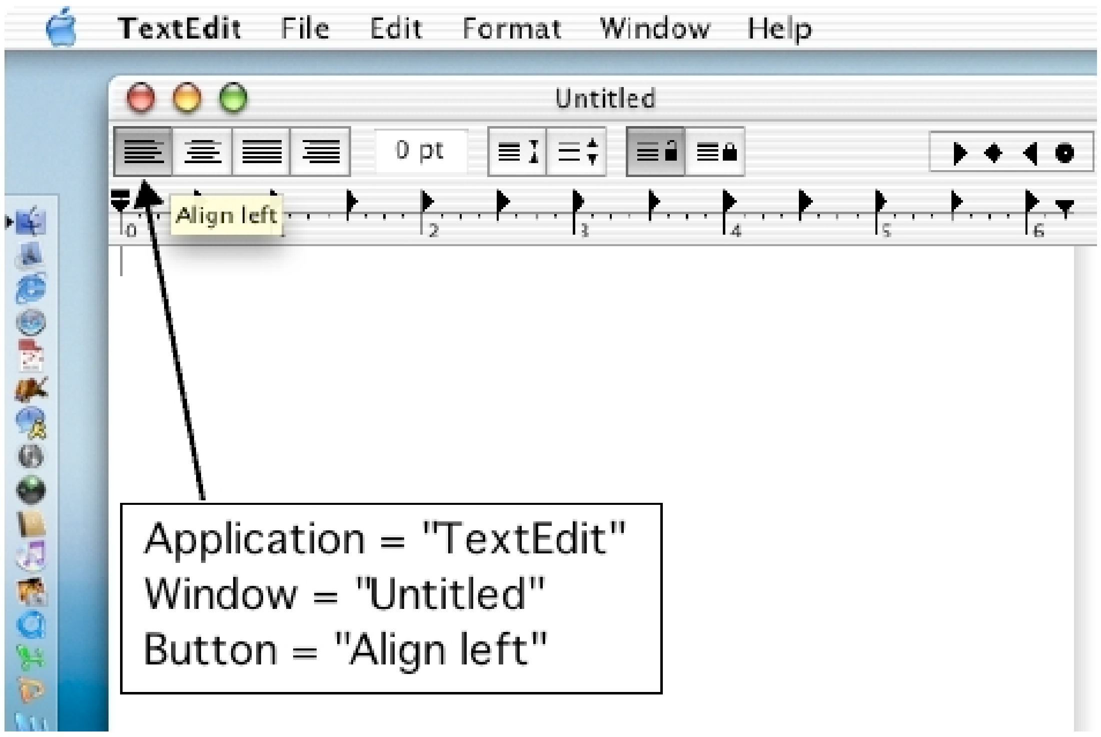


Figure 2: Accessibility Hierarchy Example (Apple Accessibility Model)

The following figure shows the basic Accessibility Object Hierarchy that is constructed for a button. The button is considered a component contained within a window (in this case entitled "Untitled"), which in turn is contained within the application ("TextEdit"). The button itself has the name property of "Align left".



Application = "TextEdit"
Window = "Untitled"
Button = "Align left"

Figure 3: Button Example (TextEdit)

The following figure is a second example of an Accessibility Object Hierarchy. In this case, menu structures are examined. The "New" menu item is contained in the "File" menu, which itself is contained in the "TextEdit" menu bar. It is worth noting that both the name of the menu item ("New") and the shortcut key combination ("Command-N") can be extracted from the hierarchy. In Mac OS X, (and earlier versions of the Macintosh operating system) there is a common menu bar. This differs from Microsoft Windows significantly, where each application can have their own menu bar. This means that as the user switches between applications, the contents of the menu bar (and the name of the menu bar itself) changes. It is therefore important that the name of the application that currently "owns" the menu bar is obtained.

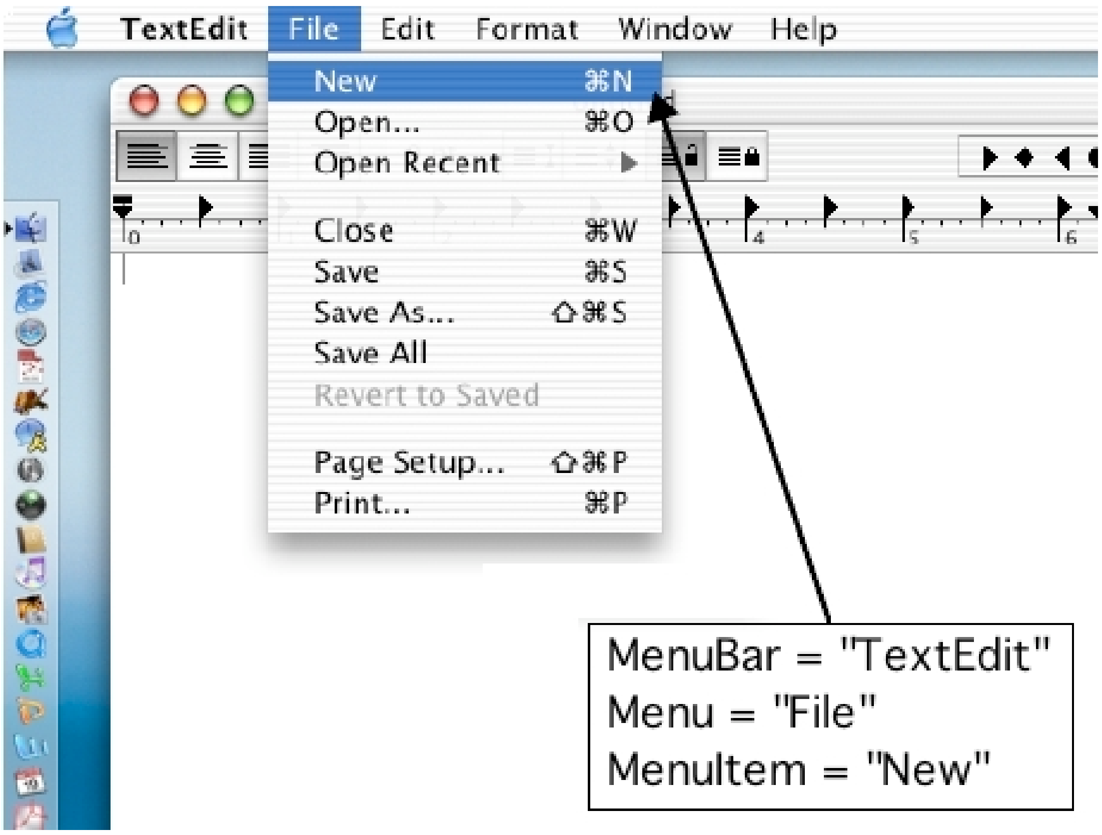


Figure 4: Menu Example (TextEdit)

It is worth noting that not all applications that execute on the Mac OS X operating system are accessible. Applications that execute in the Mac OS 9 based “Classic” environment do not interact with the Accessibility API. As Apple is no longer installing the “Classic” environment on new Macintosh computers, lack of accessibility for this environment is not a significant issue. Carbon-based applications are also generally not supported, unless the developer has implemented appropriate functionality into the Carbon user interface elements.

2.4.3. GNOME Accessibility Project

A popular graphical user interface for the Linux platform (and other Unix based platforms) is the GNOME Desktop user interface. The GNOME Desktop user interface is based around windows, menus and buttons in a similar manner to Mac OS X or Microsoft Windows.

Associated with the development of the GNOME Desktop environment is the GNOME Accessibility Project (GAP). The aim of this project is to produce an accessibility architecture for applications that are available for use with the GNOME Desktop. This architecture will then allow access to the user interface for various assistive technologies including screen readers, on screen keyboards, magnifiers and braille devices.

The accessibility architecture constructed by the GAP is based on the GTK+ widget set. This widget set is large and includes most of the widgets that are required to construct standard user interfaces. The widget set includes:

- Buttons
- Labels
- Frames
- Dialogs
- Radio Buttons
- Images
- Scrollbars
- Progress Bars

The architecture itself is loosely based on the Java Accessibility API (GNOME Project, 2003). Whenever a programmer uses a GTK+ widget from the standard widget set the accessibility features of the widget are automatically enabled. Little programmer involvement is required to create an accessible application using standard GTK+ widgets.

There are a number of issues with the accessibility architecture produced by the GAP. Firstly, non-standard or custom widgets used or created by developers must be modified in order to function correctly with the accessibility architecture. Secondly, not all GNOME based applications are created using GTK+ widgets. This means that on a standard install of the GNOME Desktop environment there may be a significant number of inaccessible applications.

2.4.4. Windows Accessibility

Microsoft Active Accessibility is the collection of APIs that are used on the Windows operating system to enable assistive technologies. Documentation for Microsoft Active Accessibility can be found at the Microsoft Developer Network website (http://msdn.microsoft.com/, search for "Active Accessibility"). Most versions of the Microsoft Windows operating system are supported to some degree, with the exception of versions earlier than Windows 95.

Active Accessibility is built on the Component Object Model (COM) that has been developed by Microsoft. This technology is supported by most Microsoft applications and by many developers of third-party software for Windows. Through the dominance of Windows, COM technology has become the de-facto industry standard communication between applications and the operating system on the Windows platform. As a consequence of this, COM is a well supported and documented technology.

The extraction of information from user interface elements in Active Accessibility is achieved via the IAccessible COM interface. This COM interface also allows the user interface element to be manipulated directly. This allows for greater flexibility and increased functionality of assistive software that takes advantage of this COM interface. Using a screen reader as an example, this would allow the user to not only determine the functionality of the button they have selected but also "click" the button using an interface provided by the screen reader.

Blenkhorn and Evans, (2002), describe two methods for making Windows applications accessible. If the application provides an Active Accessibility interface, the application becomes a server for information about the application. This means assistive software can extract information from the application in a straightforward manner. If the application does not provide an Active Accessibility interface, an assistive application can extract some information from it assuming it is constructed using standard user interface elements. This can be achieved by employing the Object Linking and Embedding (OLE) technologies provided in many Windows applications. Blenkhorn and Evans describe how applications that contain an OLE interface expose their object model to other applications. This technology allows an assistive application to not only extract information from OLE enabled applications but to control their behaviour as well.

Active Accessibility depends on the developer using standard COM-based objects when they construct their applications. Other non-standard objects can be modified to interact with the Active Accessibility system but this process involves recoding sections of the code for the particular object. Limiting accessibility to applications built using COM-based objects limits the number and range of accessible applications. There are many methods for building applications on the Windows platform and only a few of these methods will result in Active Accessibility compliant software products.

2.4.5. Summary

This section has provided an overview of the accessibility features that exist in the popular modern operating systems. All three accessibility architectures discussed are satisfactory in that they allow assistive applications to be constructed that will be compatible with a reasonable amount of available software. All three follow the same approach by enabling elements of their user interface to return information regarding properties and functionality.

The common weakness with all accessibility architectures discussed is that unless the user interface of applications is constructed using the standard elements or widgets provided, adding accessibility to an application requires significant effort on the part of the developer. Legacy applications that execute on modern operating systems are common and are unlikely to be redeveloped to enable accessibility.

The Accessibility API provided by Apple on Mac OS X is an excellent solution in that it combines a feature rich accessibility framework with a stable operating system and a relatively consistent user interface. With the majority of new applications for Mac OS X being developed using Cocoa or Carbon technologies for their graphical user interface, the number and range of accessible applications available will increase significantly.

The Gnome Accessibility Project is a viable solution for Linux and Unix-based platforms but is limited in that it only supports certain graphical user interface construction toolkits. It has the advantage of being an open project in terms of design and implementation, which should result in the project being both feature rich and stable.

The Microsoft Active Accessibility framework is the only viable option for assistive applications on Windows because it is compatible with the large amount of Microsoft products available (in particular the Office suite of applications and Microsoft's Internet applications such as Outlook and Internet Explorer). As is the issue with the GNOME Accessibility Project, the number of non-accessible methods to use to construct applications under Windows is large, meaning that the majority of applications available may not be supported by an assistive application.

2.5. Alternatives

2.5.1. Overview

This section outlines some alternatives solutions available that enable the vision-impaired to use a computer. Software for most major operating systems is considered and the advantages and disadvantages of each is outlined. The add-on software for the emacs text editor, Emacspeak, was considered for porting to Mac OS X and as such is covered in depth.

2.5.2. Emacspeak

Emacspeak is a speech interface add-on for the emacs text editor. The emacs text editor is a Unix application that has been ported to most other operating systems, including Mac OS X, Linux and Windows. It is described fully in T.V. Raman's article for the Dr. Dobb's Journal (Raman, 1997). Emacspeak is independent of emacs in terms of code base and is structured in a layered manner. Only the lowest layers of Emacspeak are device dependent. Emacspeak also functions with other applications that are Unix based.

Emacspeak was considered a viable option for Mac OS X due to the fact that Mac OS X has BSD Unix underpinnings. The process of porting Unix based applications to Mac OS X is described by Apple Computer Inc. (2002d). This would conceivably make the process of speech-enabling Unix software that is ported to Mac OS X straightforward.

The process of porting Emacspeak to Mac OS X was stopped as it became difficult to re-route the textual output to Apple's speech synthesis API. Emacspeak includes drivers for specific speech synthesizers and does not support speech output through the speakers of the computer. Hardware speech synthesizers are an expensive option and would prove difficult to configure with modern Macintosh computers. This is due to the lack of drivers and serial ports on the Macintosh platform.

2.5.3. Macintosh Based Alternatives

There are currently no other screen readers available for Mac OS X. Outspoken is a screen reader application available for earlier versions of the Macintosh operating system (Mac OS 8 and 9). Support for these versions of the Mac OS operating system, however, does not exist on modern Apple hardware. The latest version of Outspoken for the Macintosh (version 9.2) supports Mac OS 9 and development has ceased at this revision (Lakes, 2003).

2.5.4. Windows Based Alternatives

The most popular screen reader application for the Windows platform is JAWS. JAWS has reasonable reviews, and functions correctly with a significant amount of Windows software. Prices for JAWS range from $895 to $1095 (USD), making it an expensive option. The price of JAWS approximately doubles the cost of a personal computer for a vision-impaired user. This extra cost will prove prohibitive in most cases as the majority of vision-impaired people are dependent on social security benefits.

3. SCREEN READER DESIGN

3.1. Overview

This section proposes the design for the screen reader. The components of the screen reader are considered individually and the design for each is presented. The overall design of the structure of the screen reader software is then considered.

Considered in the design of the screen reader are forms of user input, methods for accessing the accessibility API, methods for processing the accessibility object hierarchy, requirements of the speech synthesis system and the design of a user preference system. Included also is a discussion of general screen reader design criteria and an overall program structure of the proposed screen reader.

3.2. Input from the User

For the vision impaired, using the keyboard for input is preferable to using the mouse. In order to use a mouse effectively, users must be able to interpret the feedback it provides. This feedback is usually graphical in the form of a mouse cursor that appears on the screen and is able to be moved.

The Mac OS X operating system is primarily mouse driven. Keyboard shortcuts are available that execute commands from menu structures but in order to navigate the menus the mouse must be used. The same comments apply for the Dock application and other regions of the interface. This is in contrast to the Windows operating system, which allows full access to menus via the ALT key and arrow keys.

The regions of the graphical user interface of Mac OS X are illustrated on the following diagram.

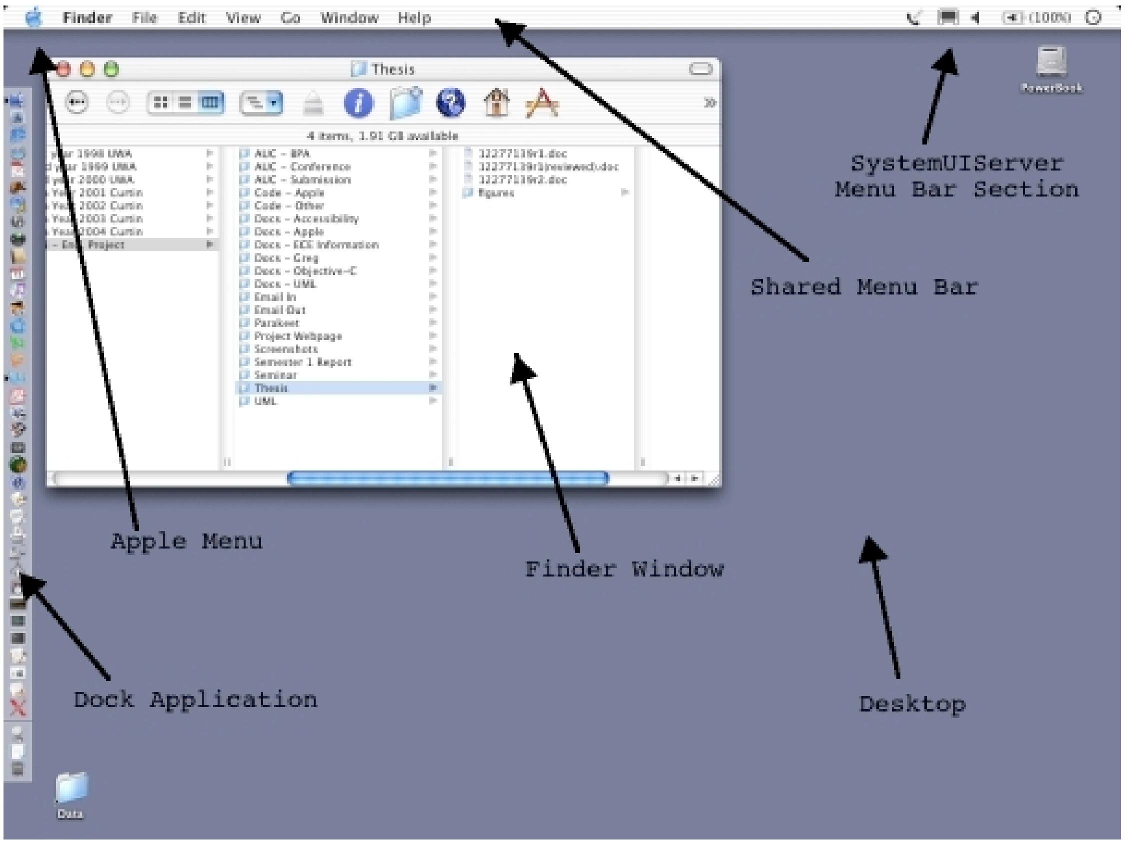

Finder File Edit View Go Window Help
Thesis
SystemUIServer Menu Bar Section
Shared Menu Bar
Apple Menu
Finder Window
Dock Application
Desktop

Figure 5: Basic Mac OS X Graphical User Interface

The Dock application, shown on the left hand side of the screen shot figure above, can be positioned at the left, right or bottom edges of the screen. It functions as a launcher for applications and documents and provides a convenient way to switch between running applications. The Dock is driven entirely by mouse.

There are two menu bars at the top of the screen. The one to the left is controlled by whatever application is running in the foreground (with the exception of the Apple menu). As the user switches between applications, the content of this menu bar changes. The menu bar located at the top right of the screen is managed by the SystemUIServer application. It contains utilities such as the clock, battery level and sound volume. The menu structures are primarily driven by mouse however shortcut keys do exist for some of the menu functions.

Finder windows, one of which is shown in the preceding figure, list the drives, files and directories (or folders as they are referred to on the Macintosh) associated with the system. These windows can be driven by the keyboard. This is achieved by using a combination of the arrow keys to select items and the keyboard shortcuts in the Finder application to manipulate them.

For these reasons, a screen reader for Mac OS X must be at least partially mouse driven. In initial development, the screen reader for Mac OS X will be mouse driven and provide feedback to the user regarding the position and movement of the mouse cursor. Further developments of the screen reader will seek to remove the dependency on the mouse by providing keyboard access to the user interface.

The following figure illustrates the basic algorithm that will be used to allow the screen reader to be mouse driven. The algorithm is driven by a timer and requires the mouse coordinates to be stored after the timer expires. When the timer next expires, the stored coordinates (X and Y locations) are compared with the current mouse coordinates to determine if the user has moved the mouse in the last timer period. The "Process" step in the flowchart is where the screen reader extracts and processes the information from the user interface and produces the descriptive strings for speech synthesis.

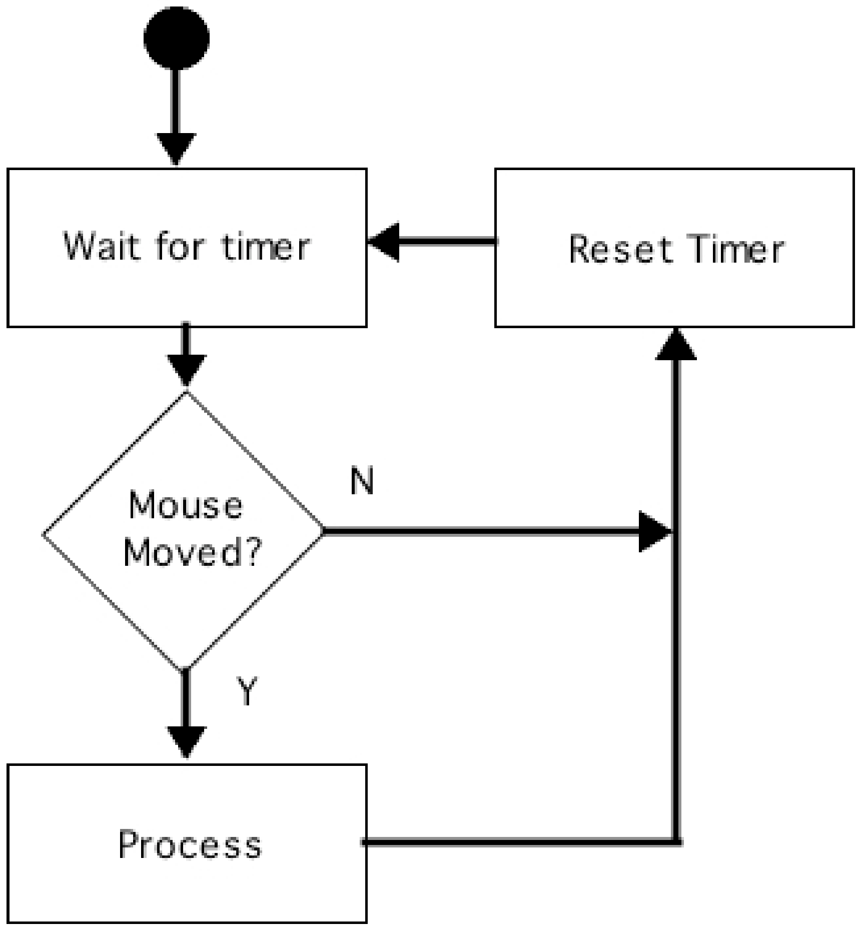
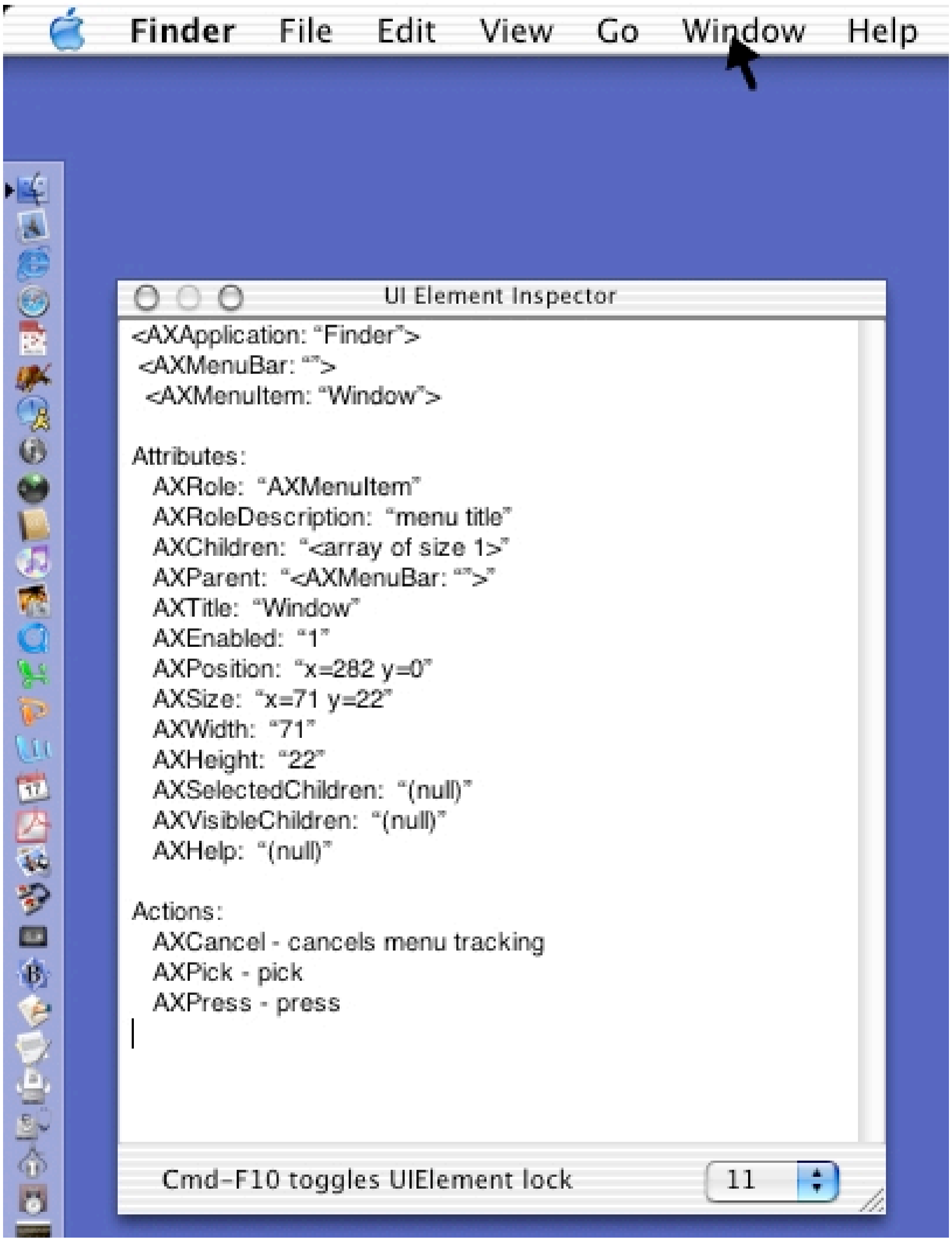


Figure 6: Timer-driven Update Flowchart

3.3. Accessing the Accessibility Architecture

The methods of accessing the accessibility architecture are illustrated well by a free utility produced by Apple Computer called “UI Element Inspector”. This utility extracts the accessibility information from the element of the user interface that is directly below the mouse cursor. It presents this information in a structured manner showing the accessibility object hierarchy, attributes and actions separately. A screen shot of the application running (with the mouse cursor over the “Window” menu) is shown below.



```
<AXApplication: "Finder">
 <AXMenuBar: "">
  <AXMenuItem: "Window">

Attributes:
   AXRole:  "AXMenuItem"
   AXRoleDescription:  "menu title"
   AXChildren:  "<array of size 1>"
   AXParent:  "<AXMenuBar: "">"
   AXTitle:  "Window"
   AXEnabled:  "1"
   AXPosition:  "x=282 y=0"
   AXSize:  "x=71 y=22"
   AXWidth:  "71"
   AXHeight:  "22"
   AXSelectedChildren:  "(null)"
   AXVisibleChildren:  "(null)"
   AXHelp:  "(null)"

Actions:
   AXCancel - cancels menu tracking
   AXPick - pick
   AXPress - press
```

Figure 7: UI Element Inspector

The source code for "UI Element Inspector" is freely available from Apple's developer website in the sample code section (http://developer.apple.com/). The code is an excellent example of how to extract accessibility information from the user interface and will be used as a starting point in the construction of the screen reader.

3.4. Processing the Accessibility Object Hierarchy

Once the accessibility object hierarchy for the user interface area that we are interested in has been obtained, the process of extracting information from it and processing this information to produce descriptive strings is considered.

The first step in the process it to extract the type of object that is at the base of the accessibility object hierarchy that has been obtained. This can be achieved by processing the hierarchy tree component of the object model. An example hierarchy tree is shown below.

```
<AXApplication: “Finder”>
 <AXMenuBar: “”>
  <AXMenuItem: “Window”>
   <AXMenu: “Window”>
    <AXMenuItem: “Bring All to Front”>
```

Figure 8: Sample Hierarchy Tree

The regular formatting and structure of this sample will allow generic methods to be developed that allow the value (eg. “Finder”) to be extracted for a key (eg. “AXApplication”). The object at the root of the hierarchy tree in the preceding sample is an AXMenuItem object with value “Bring All to Front”. All other supported user interface elements are represented in the same manner. A looping algorithm that processes the hierarchy tree on a line-by-line basis would allow the root object to be discovered. This algorithm would also provide information pertaining to the context in which the object is found. Such information includes the parent application of the object and the region of the user interface the object is found. This step is important as it allows contextual information about the interface to be obtained. Using the above sample hierarchy, the contextual information that can be obtained is:

- The user is in the “Finder” application
- They are accessing the “Window” menu
- They have selected the “Bring All to Front” menu option

The screen reader can now construct a textual message that describes to the user the area of the user interface with which they are interacting. Using the above example, this message may be “Finder, Window menu, Bring All to Front menu item”.

The following table shows the format of messages for different objects. The [name] parameter refers to the text value of the object. In the previous example, this would be “Bring All to Front”. The [application] parameter represents the name of the foreground application. In the previous example, this would be “Finder”.

Root Object	Message Format
AXApplication	[name] or [name] [application]
AWWindow	[name] window
AXButton	[name] button
AXMenuBar	[application] menubar
AXMenu	[name] menu
AXMenuItem	[name] menu item
AXTextField	[name] text field, contains {contents of text field}
AXTextArea	[name] text area, contains {contents of text area}

Table 1: Descriptive Message Formats

The above table shows a basic set of user interface objects and the general structure of the messages that should be generated for them. It should be noted that for specific applications this format might be inappropriate. An example of this is the "Dock" application, where the icons to launch applications or documents are of the AXButton type. In this case a message of the form "Dock, launch Safari" when the user has selected the "Safari" icon in the "Dock" application region.

3.5. Speech Synthesis

3.5.1. Overview

This section outlines the design requirements of the speech synthesis component of the screen reader. The speech produced is the only output provided by the screen reader. Speech parameters including speed and voice are described and issues relating to the properties of the speech synthesis system provided by Apple are discussed.

3.5.2. Speech Speed

Speech synthesis speed (measure in words per minute) is the most important property of the speech synthesis system in the context of screen readers. The more proficient the vision-impaired user becomes at using the operating system and screen reader combination, the faster they wish to hear the feedback provided.

The Dectalk hardware speech synthesizer is capable of speaking at speeds between 75 and 650 words per minute (Mates, 2000). To be of use, the developed screen reader must be capable of speech synthesis speeds in this range. Basic tests involving the TextEdit application (which is speech enabled in Mac OS X) and a few sample paragraphs of text were conducted. The results are shown in the following table. These paragraphs were made up of text from actual documents to ensure an appropriate average word length and complexity. Punctuation was also included.

Test Word Count	Elapsed Time (seconds)	Calculated WPM
593	175	203
85	22	232
43	10	258
12	3	240

Table 2: Speech Synthesis Benchmarking

(Note: The above tests were conducted using a PowerBook G3 400Mhz with 192 MB of RAM and running Mac OS X 10.2.6).

The basic tests conducted show that speech synthesis speeds of around 250 words per minute are possible for strings with low numbers of words. As the strings to be synthesized in a screen reader are generally short, expecting word rates of around 250 words per minute is reasonable. In the above tests, TextEdit sets up a new speech channel every time the speech synthesis functionality is requested. In the screen reader, the speech channel will be set up once at launch and used until the program terminates. For this reason, high speech rates are expected from the screen reader.

It is also important for the speech synthesis speed to be adjustable. This allows the screen reader to cater to the needs of a wide range of different vision-impaired users and to scale well as users become more familiar with using the software.

3.5.3. Speech Voice

The voice used by the screen reader for speech synthesis defines the accent and punctuation behaviour of the output. For that reason, the optimal voice for the screen reader is a personal preference of the user. The ability for the user to pick the voice used is therefore a requirement of any screen reader. This area of design is covered in section 3.6.

The default installation of Mac OS X version 10.2 includes 22 different voices. There are, however, only a few that suit the requirements of a screen reader. Candidate voices should be able to be heard clearly at high speeds and should match the accent of the user as closely as possible.

It is worth noting that in order to customize the screen reader for languages other than English, voices that speak those languages must be created. Currently all voices that are installed with Mac OS X are English American speaking voices. These voices do not pronounce other languages correctly.

3.5.4. Other Properties

It is important that any speech synthesis system considered can be interrupted when it is part of the way through speaking a string. This means that when the mouse is over one particular area of the screen, the screen reader is describing that area, and the mouse moves to another area, the speech synthesis regarding the first area is stopped and the screen reader begins to describe the new area immediately. The speech synthesis libraries provided by Apple in Mac OS X support this feature.

The speech synthesis libraries under Mac OS X are also non-blocking in that once a user application initiates speech synthesis the user application is free to continue executing (Apple Computer Inc., 2002c). This is an important property as it allows the screen reader to keep up with the actions of the user.

3.5.5. Conclusions

The speech synthesis functionality and libraries provided by Apple in Mac OS X satisfy the requirements of a screen reader application. The provided voices are sufficient for a screen reader although it is anticipated that third-party voices will be constructed or used as required by users. These can be added to the operating system and enabled independently of the screen reader. While initial testing of the speech synthesis libraries indicate that speeds achieved by Dectalk devices and the like are unattainable, the speed achieved is adequate for initial development. Methods of speeding up the speech synthesis will have to be investigated if the screen reader is to be used by all members of the vision-impaired community.

3.6. User Preference System

3.6.1. Overview of User Preference System

A useful feature of a screen reader is the ability to store sets of preferences for different users. This would allow multiple users who share a common computer to have individual sets of preferences. Sets of preferences could be loaded by default or selected from a list when the screen reader is started.

3.6.2. Preferences to Store

The following screen reader properties can be stored as user specific preferences:

- Speech synthesis voice
- Speech synthesis speed
- Speech synthesis volume
- A user experience indicator that corresponds to the verbosity of the output
- A user specific dictionary of words and pronunciations

A default set of preferences should be available and should correspond to the system settings on the computer.

3.6.3. Using XML to Store User Preferences

A suitable format for storing user preferences is the Extensible Markup Language (XML). Methods for defining XML documents are discussed by Graham and Quin in their book "XML Specification Guide" (Graham & Quin, 1999). A sample user preference defined in an XML format is shown below.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<user_preference>
    <pref_name>TTS_Speed</pref_name>
    <pref_value>80</pref_value>
</user_preference>
```

Figure 9: Sample XML Preference Storage

The Mac OS X operating system includes libraries that allow XML files to be loaded, interpreted and saved in a straightforward manner. Preference files under Mac OS X (with a .plist extensions) are standard XML files and the Cocoa programming environment provides functions to access and manipulate them. Being text files in terms of file format, XML files are also editable by any user familiar with an ASCII text editor.

3.7. Screen Reader Design Criteria

The most important property of screen reader is the degree in which it is compatible with the operating system and applications that the user utilizes. Unless the screen reader can extract a useful amount of information from the accessibility API about the components that make up the graphical user interface the screen reader will be ineffective. This is dependent largely on the accessibility API provided by the operating system.

The speed at which the screen reader operates is important in terms of how usable it is. If there is a significant lag between the event occurring on the screen and the user learning about it through the output of speech synthesis the screen reader will prove difficult to use. In most instances, this is dependent on the processor load.

As the synthesized speech produced by the screen reader is it's only form of output, speech quality is an important design criterion. The voice used for speech synthesis needs to be intelligible at high rates. The speed and voice used by the speech synthesis engine needs to be customisable.

An important aspect of an effective screen reader is the ability to customize its properties to suit the needs of particular users. This would be implemented via a user preference system, as discussed in section 3.6.

The manner in which the screen reader handles errors is also of importance. The stability of the screen reader is an important issue as the only sign the user will get when the screen reader exits prematurely is that the speech synthesis will stop. Without speech synthesis running there is no way to report to the user any errors or problems. A screen reader with excellent functionality and compatibility but poor stability becomes an system administration issue as generally the vision-impaired user will be unable to restart the screen reader (or indeed reboot the operating system).

3.8. Program Structure

The overall structure of the screen reader consists of three components. The first is the main application shell that is responsible for initialising the application, extracting the accessibility information from the user interface and managing the speech synthesis functionality. This component is also responsible for maintaining the timer-based update system. It is labelled "Appshell" on the following diagram. The second component is responsible for the basic graphical user interface that the screen reader has. It is labelled "InspectorWindow" on the following diagram. The third component provides the processing function of the screen reader and is responsible for searching and interpreting the accessibility information that has been extracted from the user interface. It is labelled "AXStringProcessor" on the following diagram. Note that the "AX" is short hand for accessibility.

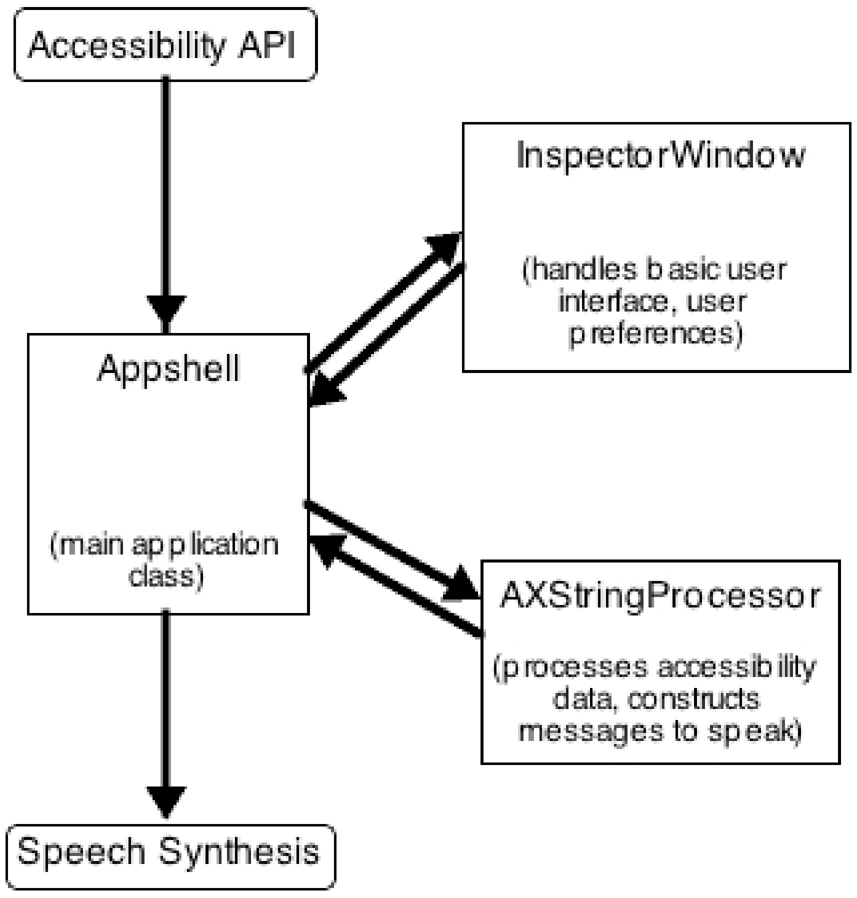

Accessibility API
InspectorWindow
(handles basic user interface, user preferences)
Appshell
(main application class)
AXStringProcessor
(processes accessibility data, constructs messages to speak)
Speech Synthesis

Figure 10: Program Structure

Below is a use case diagram for the screen reader that has been designed. The actors in the system are the user, the screen reader itself, the speech synthesis subsystem and the accessibility API subsystem. The latter two are considered actors as the screen reader simply uses services that are provided by them. The diagram illustrates the interactions between the actors that are required to provide the necessary functionality of the screen reader.

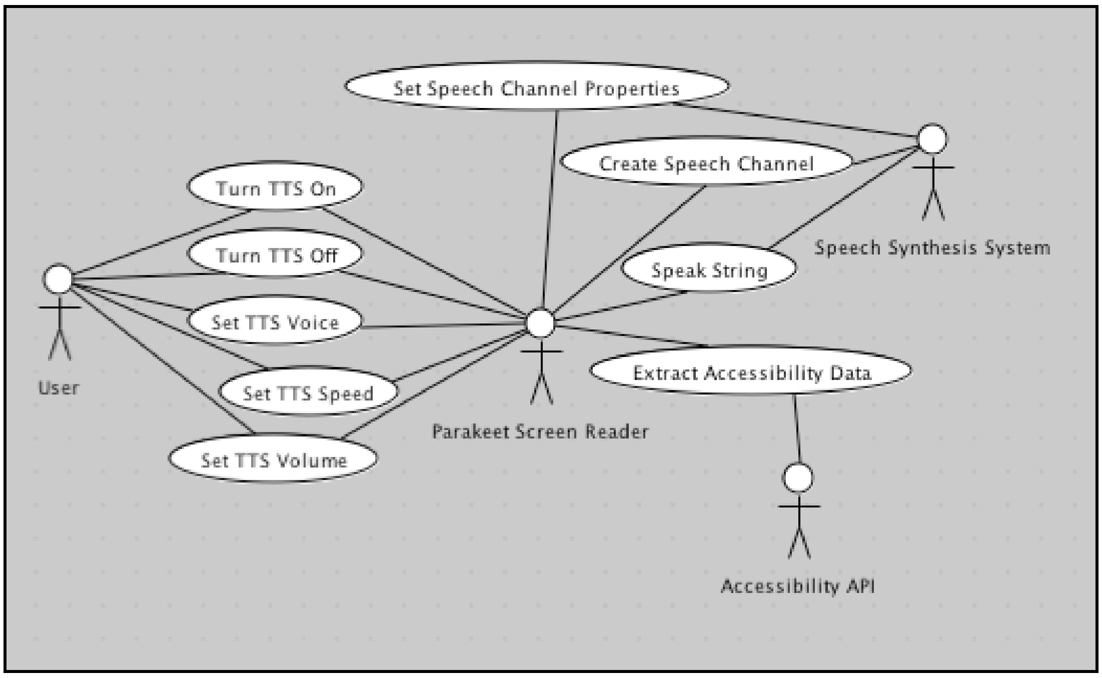


Figure 11: Use Case Diagram

In the current design, the output of the screen reader is text that is sent to the speech synthesis system for conversion and speech. It is important for this to be designed in a manner that allows other feedback systems to be used (for example a refreshable braille display) in a straightforward manner. For this reason the component responsible for speech synthesis and the component responsible for constructing the text to be synthesized to speech are separate and modular.

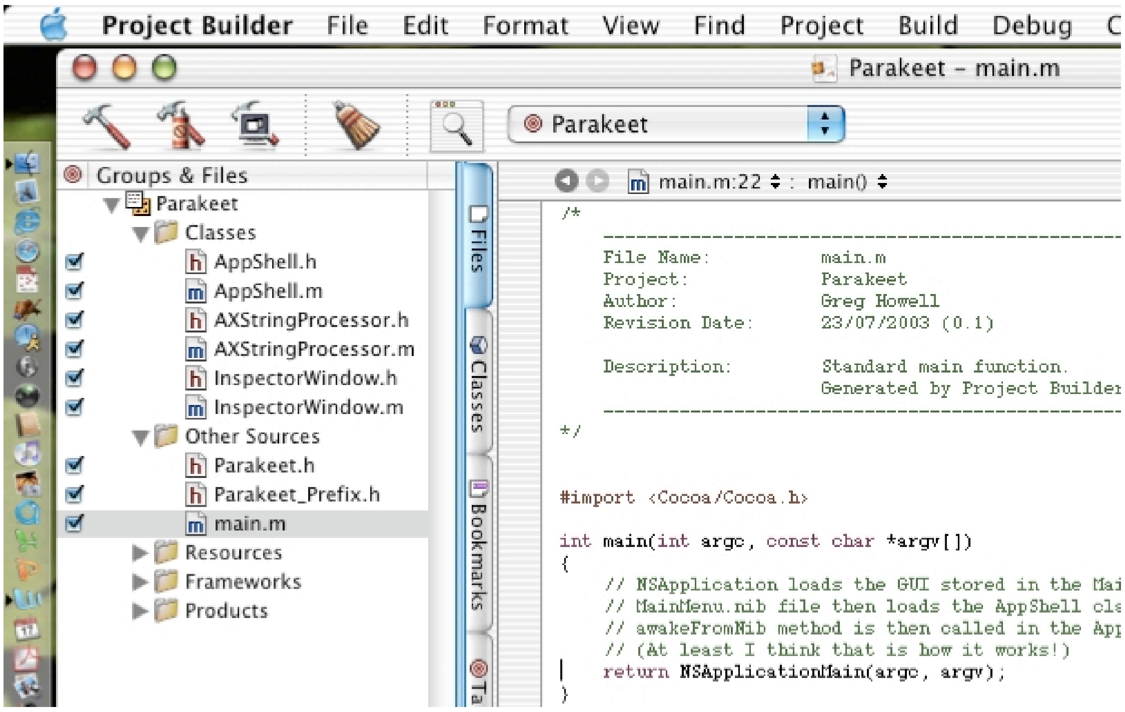
4. SCREEN READER IMPLEMENTATION

4.1. Implementation Overview

The design of a Mac OS X screen reader described in earlier sections was implemented using Apple's Project Builder and Interface Builder software. The program developed, named Parakeet, was coded in Objective-C. Parakeet is Cocoa based, compatible with Mac OS X version 10.2 and above, and requires no additional hardware or software to function.

4.2. Project Builder

Project Builder is the integrated development environment (IDE) provided by Apple with their operating system (Mac OS X). It is the hub application of Apple's developer tools (Davidson, 2002). It provides a graphical front-end to the GNU C Compiler (gcc) and the GNU Debugger (gdb) as well as allowing source code and other resources to be collected together to form a project. It supports development in many programming languages (including Objective-C, C, C++ and others) and has good integration with the compiler and debugger. A screen shot of the Project Builder IDE is shown below.



```
/*
    ------------------------------------------------
    File Name:          main.m
    Project:            Parakeet
    Author:             Greg Howell
    Revision Date:      23/07/2003 (0.1)

    Description:        Standard main function.
                        Generated by Project Builder
    ------------------------------------------------
*/


#import <Cocoa/Cocoa.h>

int main(int argc, const char *argv[])
{
    // NSApplication loads the GUI stored in the Mai
    // MainMenu.nib file then loads the AppShell cl
    // awakeFromNib method is then called in the App
    // (At least I think that is how it works!)
    return NSApplicationMain(argc, argv);
}
```

Figure 12: Project Builder IDE

The Project Builder IDE allowed all of the source files, resource files and the application icon to be collected together in a single project and accessed quickly and easily via the IDE interface. Compilation settings, revision numbering and reference manuals are also easily accessible through the Project Builder interface. Refer to Appendix A for the version numbers of the components of the Project Builder IDE used in the development of Parakeet.

4.3. Cocoa and Carbon

There are two programming APIs that have been used in the development of the Parakeet screen reader: Cocoa and Carbon. Carbon is a set of procedural APIs that are based on legacy toolbox APIs that have been modified to function in Mac OS X. The Carbon APIs allow Mac OS 9 based applications to be ported to Mac OS X in a well-defined manner. Cocoa is a set of object oriented APIs that are based on the NeXT operating system programming environment. Most modern Mac OS X applications are Cocoa-based (Davidson, 2002).

In Parakeet, the speech synthesis features used are Carbon-based (Apple Computer Inc., 2002c). The graphical user interface of Parakeet is constructed using Cocoa widgets. The Accessibility API is accessible by both Carbon and Cocoa based applications (Apple Computer Inc., 2002a).

4.4. Objective-C

The Objective-C programming language is a superset of the ANSI C programming language. Included in Objective-C is some syntax and runtime extensions that enable object oriented programming (Davidson, 2002, Pinson, 1991 and Apple Computer Inc. 2002b).

Objective-C was chosen as the primary language for development of the screen reader for a number of reasons. Firstly, it is the predominant language for Cocoa development on the Macintosh platform. This means that a substantial amount of sample source code and documentation is available to aid development. Secondly, Objective-C allows easier integration with the user interface components that exist under Mac OS X. This is important for a screen reader as it allows straightforward access to these components.

Being an object-oriented programming language, Objective-C offers advantages over languages such as C in the form of code reuse through structured object-oriented design. Other object-oriented features of Objective-C are not specifically exploited in Parakeet.

Objective-C code differs from regular C code in a few specific areas. Firstly, header files in Objective-C (with the .h file extension) can be either regular C style header files or files that describe the accessible interface to an object. For instance, “Queue.m” will contain the implementation of the queue class and “Queue.h” will contain the interface to the queue class. Secondly, the syntax for calling methods on objects is unique is a significant departure from ANSI C syntax. An example is shown below.

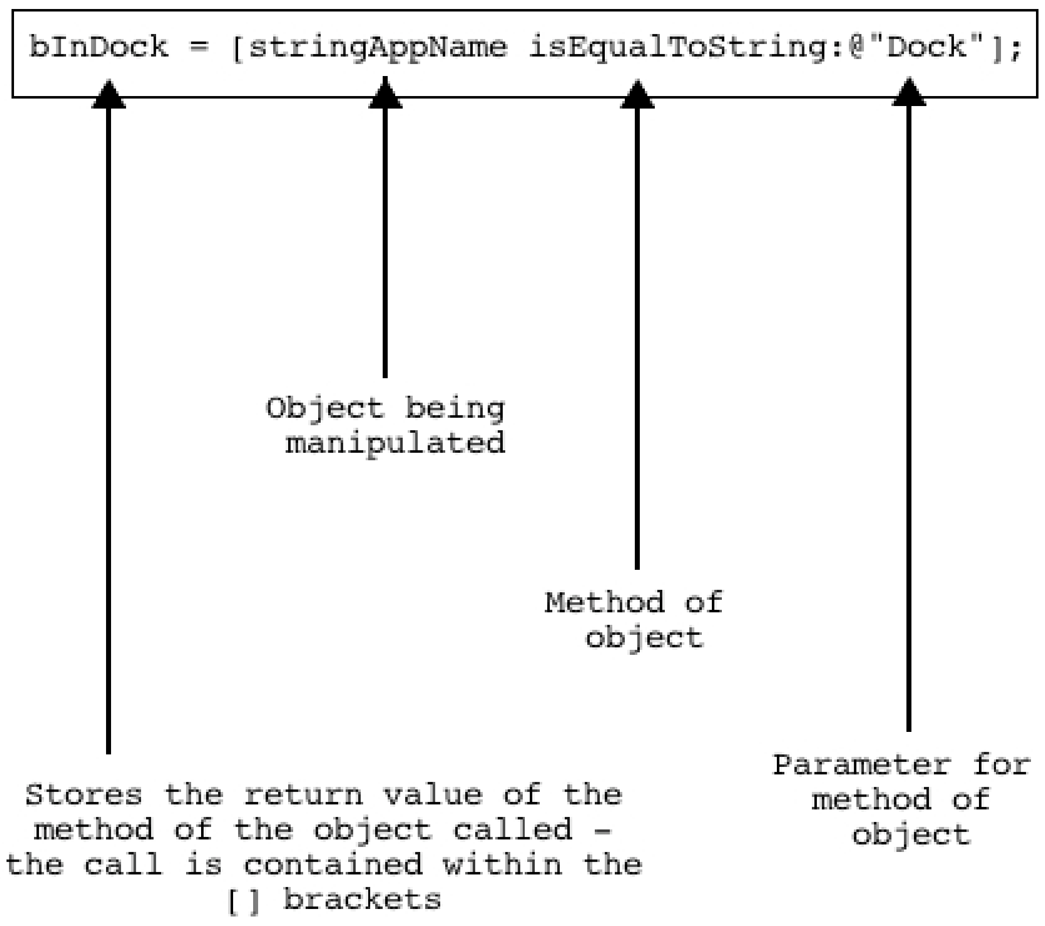

```
bInDock = [stringAppName isEqualToString:@"Dock"];
```
Object being manipulated
Method of object
Stores the return value of the method of the object called - the call is contained within the [] brackets
Parameter for method of object

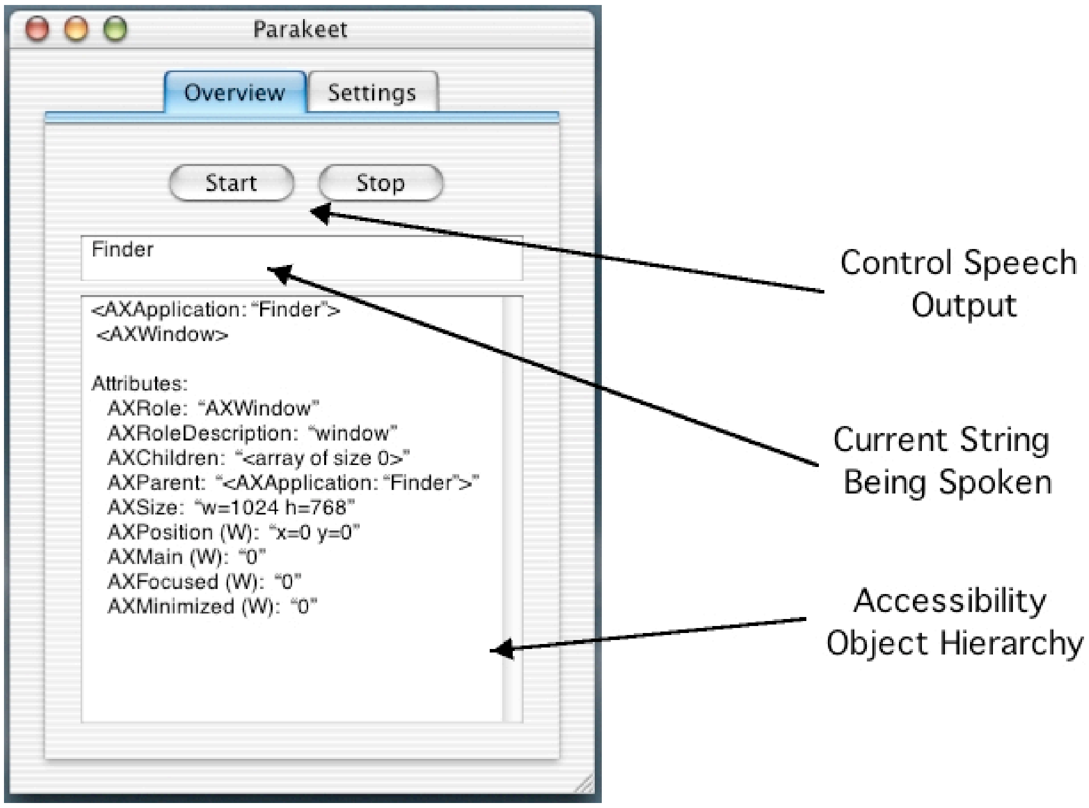
Figure 13: Objective-C object method call syntax

In the above example, the “bInDock” Boolean variable stores the result of executing the “isEqualToString” method of the NSString object (“stringAppName”) with the parameter “Dock”. If the contents of the “stringAppName” object equal “Dock”, bInDock will be set to true, otherwise it will be set to false.

4.5. Parakeet Description

4.5.1. General

The screen reader developed, named Parakeet, is a single window application that can be launched by the user. Automatic launching by the operating system is possible. It is designed to run in a minimized state to avoid window clutter. The user interface is simple and consists of two tabbed panels. The first, entitled "Overview" contains buttons to start and stop speech synthesis, a text field showing the current information being conveyed to the user and a text field to show the Accessibility Object Hierarchy that has been extracted from the graphical user interface. The second, entitled "Settings" will contain controls that allow the user to change the various parameters of the screen reader. This has not been implemented. The following screen shot illustrates Parakeet during execution. Highlighted are the various regions of the interface described.

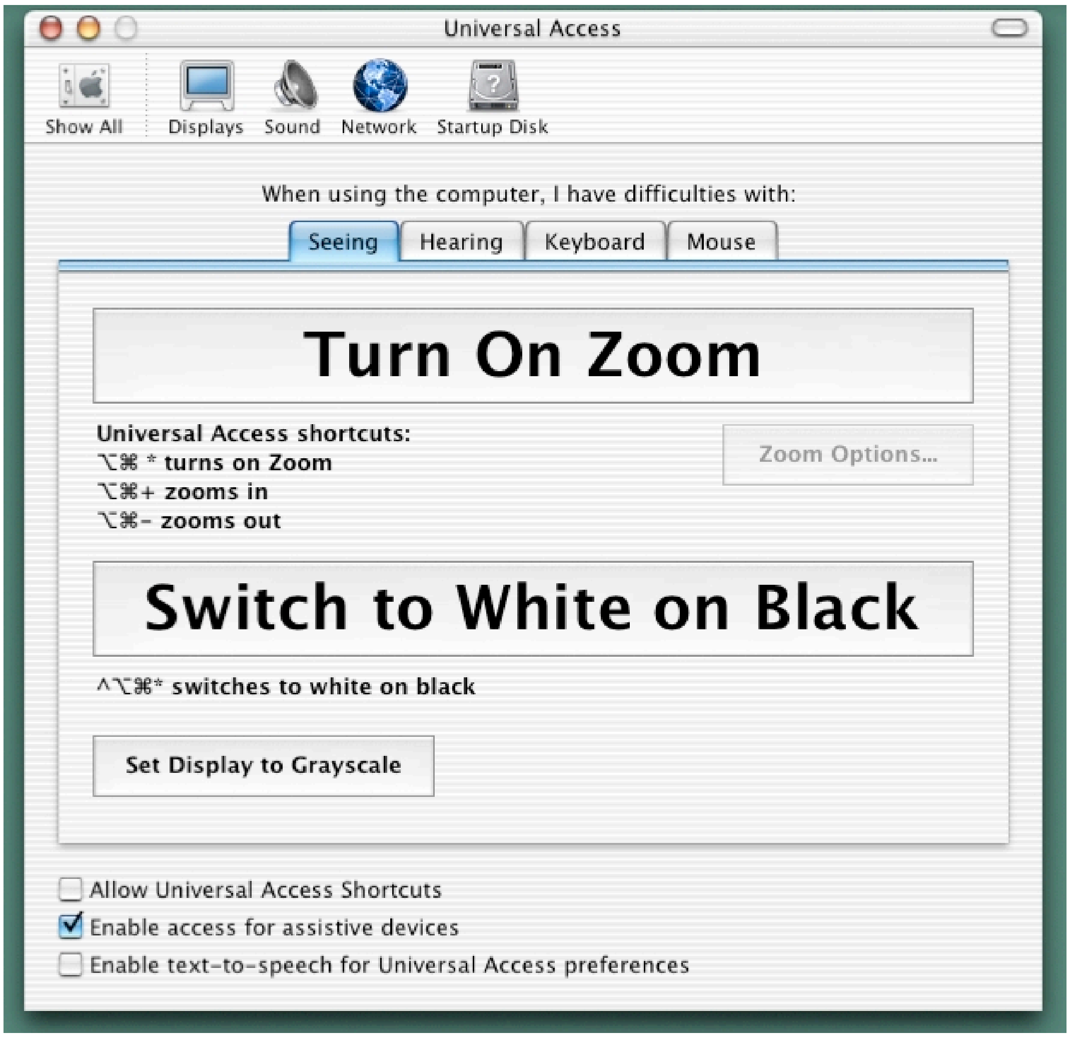


```
<AXApplication: "Finder">
 <AXWindow>

Attributes:
   AXRole:  "AXWindow"
   AXRoleDescription:  "window"
   AXChildren:  "<array of size 0>"
   AXParent:  "<AXApplication: "Finder">"
   AXSize:  "w=1024 h=768"
   AXPosition (W):  "x=0 y=0"
   AXMain (W):  "0"
   AXFocused (W):  "0"
   AXMinimized (W):  "0"
```

Figure 14: Screenshot of Parakeet during execution

In order to run Parakeet, some settings in the System Preferences panels under Mac OS X need to be changed. To allow Parakeet to access the accessibility APIs the "Enable Access for Assistive Devices" option in the Universal Access system preferences panel must be checked. This option can be found under the "Seeing" tab towards the bottom of the panel and is illustrated in the following figure. This tab contains other options that aid access to the computer by vision impaired users.



When using the computer, I have difficulties with:

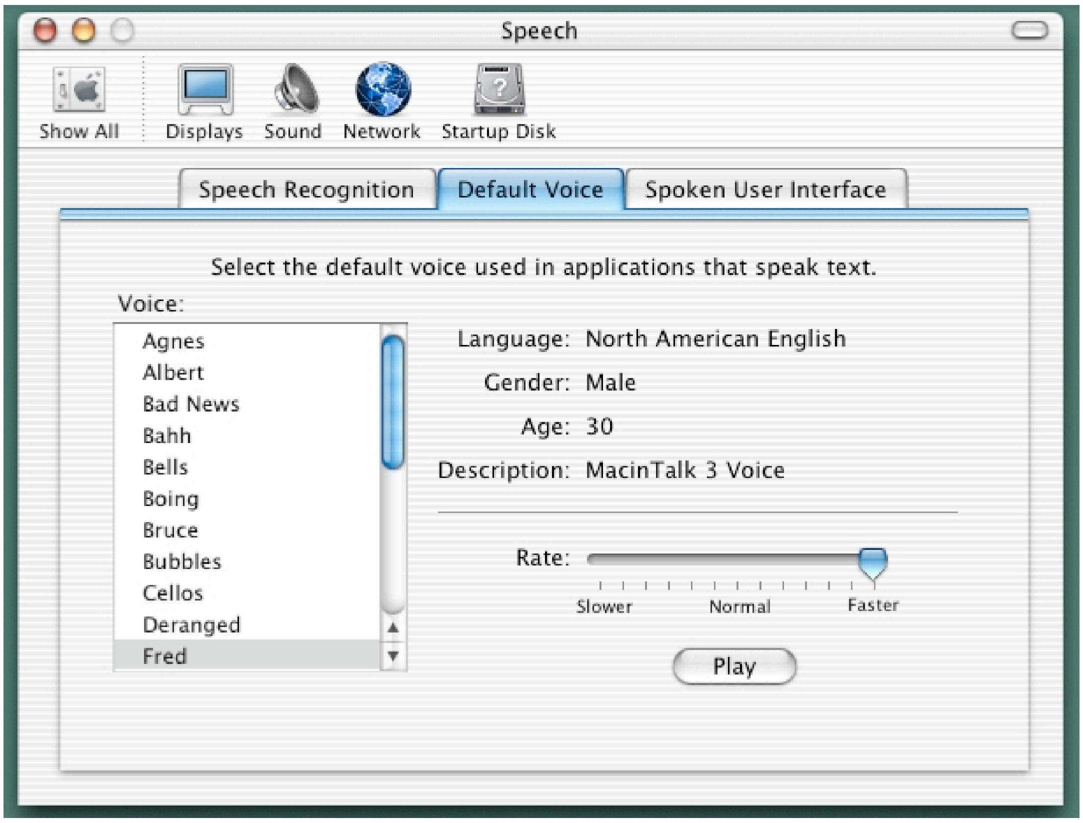
Universal Access shortcuts:
⌥⌘ * turns on Zoom
⌥⌘+ zooms in
⌥⌘- zooms out

^⌥⌘* switches to white on black

☐ Allow Universal Access Shortcuts
☑ Enable access for assistive devices
☐ Enable text-to-speech for Universal Access preferences

Figure 15: Universal Access Preference Panel Settings

In the current implementation, the speech synthesis settings for Parakeet are the same as the system speech synthesis settings. These settings can be accessed via the System Preferences under the "Speech" panel. The "Default Voice" tab contains the options that can be adjusted to change the speech synthesis properties of Parakeet. The only changes that can be made are the voice used and the speed at which text is spoken. These settings are shown in the screenshot figure that follows.



Select the default voice used in applications that speak text.

Voice: Agnes, Albert, Bad News, Bahh, Bells, Boing, Bruce, Bubbles, Cellos, Deranged, Fred

Language: North American English

Gender: Male

Age: 30

Description: MacinTalk 3 Voice

Rate: Slower – Normal – Faster

Figure 16: Speech Synthesis Settings

4.5.2. Speech Synthesis Implementation

The document entitled "Speech Synthesis Manager Reference" by Apple Computer Inc., 2002 (available from the website http://developer.apple.com/ as a PDF file) outlines the speech synthesis facilities available under Mac OS X. Described in this reference is the SpeakString function that allows a string to be passed directly to the speech synthesis engine for immediate speech. The function makes use of the default speech properties of the system outlined in section 4.5.1. The SpeakString function uses the speech channel that is provided internally for the Speech Synthesis Manager. This speech channel is also used by the spoken user interface features of Mac OS X so after synthesizing the string passed by the SpeakString function, the speech synthesis manager continued to synthesize speech as part of the spoken user interface utility. This is not satisfactory for a screen reader as there is no way to prevent extra and unnecessary speech being generated with each SpeakString call.

The solution to this problem was to create a dedicated speech channel and use the SpeakText function. The SpeakString function accepts a speech channel as a parameter and allows greater control of speech synthesis. A speech channel was created in the following manner:

```
SpeechChannel fCurSpeechChannel;     // in AppShell.h

[self createNewSpeechChannel];       // in awakeFromNib in AppShell.m
```

The speech channel has been defined in the header file but instantiated in the awakeFromNib method of the class implementation. The awakeFromNib method is executed when the application is loaded and performs other initialisation functions. The createNewSpeechChannel method of the AppShell class was developed, and uses the default speech synthesis settings as described earlier. The code for the createNewSpeechChannel method is shown below:

```
- (void)createNewSpeechChannel
{
    OSErr theErr = noErr;

    // Dispose of the current one, if it exists
    if (fCurSpeechChannel) {
        theErr = DisposeSpeechChannel(fCurSpeechChannel);
        if (theErr != noErr)
           NSRunAlertPanel(@"DisposeSpeechChannel", [NSString
stringWithFormat:@"Error #%d returned.", theErr], @"Ok", NULL,
NULL);

        fCurSpeechChannel = NULL;
    }
    // Create a speech channel
    if (theErr == noErr) {
        theErr = NewSpeechChannel(NULL, &fCurSpeechChannel);
        if (theErr != noErr)
            NSRunAlertPanel(@"NewSpeechChannel", [NSString
stringWithFormat:@"Error #%d returned.", theErr], @"Ok", NULL,
NULL);
        }

        // Setup our refcon to the document controller object so we
have access within our Speech callbacks
        if (theErr == noErr) {
            theErr = SetSpeechInfo(fCurSpeechChannel, soRefCon,
(Ptr)self);
            if (theErr != noErr)
                NSRunAlertPanel(@"SetSpeechInfo(soRefCon)",
[NSString stringWithFormat:@"Error #%d returned.", theErr], @"Ok",
NULL, NULL);
        }

    if(DEBUG_PARAKEET){
        printf("AppShell:createNewSpeechChannel\n");
    }
}
```

This code checks that the speech channel that is being created has not already been created and if not proceeds with instantiation. If the speech channel already exists, it is closed and a new speech channel created. The next section of the code allows Parakeet to be aware of when individual strings have been synthesized. This is achieved using a call back function. The call back function, OurSpeechDoneCallBackPrac, is a Pascal function that returns once the particular piece of text we have sent to be spoken has been synthesized.

```
pascal void OurSpeechDoneCallBackProc(SpeechChannel inSpeechChannel,
long inRefCon)
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    // calls the AppShell method "speechIsDone" when TTS has
completed
    [(AppShell *)inRefCon
performSelectorOnMainThread:@selector(speechIsDone) withObject:NULL
waitUntilDone:false];
    [pool release];
}
```

The speechIsDone method in the AppShell object is called when the call back function is called. Currently this function does nothing apart from report that speech synthesis has completed. It is worth noting that unless a given string is completely synthesized the speechIsDone method is not called. Any speech synthesis of strings that is interrupted by another speech synthesis request does not return via the call back function.

The function that initiates speech synthesis in the AppShell object is called startSpeakingText, and is shown below:

```
- (void)startSpeakingText
{
    OSErr theErr = noErr; // store any errors that we might
encounter

    // check if we can speak by accessing the start/stop TTS state
stored
    // in the InspectorWindow object
    if ([_inspectorWindow getStartStopState] == TRUE){
        // set up the callback (TTS has finished)
        // this is done via the SetSpeechInfo method
        if (theErr == noErr){
            theErr = SetSpeechInfo(fCurSpeechChannel,
soSpeechDoneCallBack, OurSpeechDoneCallBackProc);
            if (theErr != noErr)

NSRunAlertPanel(@"SetSpeechInfo(soSpeechDoneCallBack)",[NSString
stringWithFormat:@"Error #%d returned.", theErr], @"Ok", NULL,
NULL);
        }

        // speak the string
        // strlen only works on "C" strings
        theErr =
SpeakText(fCurSpeechChannel,theTTSString,strlen(theTTSString));
    }
}
```

This method checks that the screen reader is allowed to “speak” and sets the call back function information via the SetSpeechInfo function. The SpeakText function is then used to speak the string via speech synthesis.

4.5.3. Accessing the User Interface

The approach used in Parakeet to extract the accessibility information from the user interface is based largely on the UIElementInspector utility. Parakeet is driven off a timer that is set to expire every tenth of a second. This parameter should eventually be user configurable. When the timer expires the new location for the mouse cursor is checked against the old location of the mouse cursor.

If these two cursor locations differ, the accessibility API is interrogated and the user interface element that is at the new cursor location is copied into an off screen storage element. The accessibility object hierarchy for this element is then constructed and the attributes extracted. This information is stored in a string.

This string is then processed by the AXStringProcessor object.

4.5.4. Processing the Accessibility Object Hierarchy

A class was developed, named AXStringProcessor, to process the accessibility object hierarchy string produced by the accessibility API. The first step in processing involves converting the string into an array delimited by the new line character. The following code shows this conversion.

```
NSArray * AXDataArray = [(NSString *)passedString
componentsSeparatedByString:@"\n"];
```

The resulting array is then searched entry by entry for particular object tags. Once a tag corresponding to an accessibility object has been found (AXApplication for instance) the value is extracted (eg. Finder) and the information is added to the string that is being built. The presentation format of these strings is discussed in section 3.3. This string is the descriptive string that will be synthesized to speech.

5. SCREEN READER TESTING

5.1. Overview of Testing Methods

There are two approaches for testing the effectiveness of a screen reader. The first involves testing with sighted users and requires that the screen of the computer is covered or turned off during testing. The second involves testing with vision-impaired users. A secondary testing issue is the level of computer proficiency that the user has.

The second method of testing is favoured for a number of reasons. Firstly, vision-impaired users are the target users of this application and optimising the screen reader using feedback from this form of testing is essential. Secondly, a sighted user that has used the graphical user interface of the Macintosh has a mental picture of what the interface looks like. This mental picture is different to the one constructed by a vision-impaired user as they navigate the user interface. A vision-impaired user will rely entirely on the feedback provided by the screen reader to use the computer, whereas a sighted person will rely partially on what they know of the interface from previous experience. A better gauge of the performance and effectiveness of the screen reader will be obtained through testing using vision-impaired users.

The level of experience that the tester has with computers in general is also an important factor in interpreting the results of any tests. The screen reader may function in an effective manner for experienced computer users but may be inappropriate for users with limited computer experience. A testing process that includes both experienced and novice computer users will provide the most effective feedback data in that it will highlight all weaknesses and issues with the software.

5.2. Results of Testing

A very limited amount of testing was conducted with Parakeet. This was due mainly to time constraints and the fact that testing and improvement of the software will be the focus of a later project. The testing that was conducted focussed on checking and documenting the compatibility of Parakeet with the "Finder" application and the testing of various applications for both compatibility with Parakeet and level of interaction with the Mac OS X Accessibility API. The results of these tests are documented in the following section.

5.3. Effectiveness of the Screen Reader

The following table shows the regions of the "Finder" application (the primary desktop application in Mac OS X) that Parakeet has been found to function correctly with. Of note is the fact that no information can be extracted from the Accessibility API when the user is interacting with the desktop in the "Finder". Finder windows are also inaccessible.

Region	Functional	Comments
Dock	Yes	Contextual Dock menu not supported
Menu Bar (left)	Yes	Contains Apple Menu and other menus
Menu Bar (right)	No	Contains clock, battery level etc.
Finder window	No	No information can be extracted from the API
Desktop	No	No information regarding the files etc. on the desktop can be extracted from the API

Table 3: Functionality of the Finder in OS X with Parakeet

The following table lists commonly used applications on Mac OS X. These applications have been tested with software (UIElementInspector) that attempts to interrogate the accessibility API for information about the user interface. The level of information that is extracted by UIElementInspector will indicate whether Parakeet will be compatible with the application. UIElementInspector has been used as Parakeet has not been developed to the point where it handles all user interface elements.

Application	Functional	Comments
Microsoft Office X (Word, Excel, PowerPoint, Entourage)	No	No Microsoft applications are accessible (Office X is carbonised but uses non-standard user interface elements)
Internet Explorer 5.2.1	No	See above
Safari 1.0 (web browser)	Limited	Buttons, menus and bookmarks are supported but no HTML pages
Mail (client email application)	Yes	Buttons, menus, email message browsing and reading are all supported
AppleWorks (office suite)	Partially	Some UI elements appear to be custom
iChat (chat client)	Yes	

Table 4: Functionality of Applications with Parakeet

Of considerable concern is the limited accessibility of the Safari web browser. Given that Microsoft's Internet Explorer web browser is inaccessible, it was hoped that Safari would be fully accessible. The HTML based web pages that are rendered by Safari are not compatible with the accessibility API, however the rest of the application appears to be fully compliant. It is anticipated that access to the rendered HTML pages (via the WebKit object architecture) will be available in future revisions of the Mac OS X operating system. Discussions with Apple software engineers confirm that this is a priority for Apple.

6. CONCLUSIONS

6.1. Summary

This project has shown that with a suitable speech synthesis engine and a comprehensive accessibility API, a simple screen reader application can be designed and constructed with relative ease. With the accessibility interrogation and speech synthesis components requiring little development, the developer can focus on ensuring that the screen reader is compatible with as many user interface elements (and combinations of elements) as possible and produces messages for the user that are as informative and appropriate as required.

Any screen reader that relies on an accessibility API to garner information about the graphical user interface is inherently limited by the quality and compatibility of the API. The success of the developed screen reader largely depends on the level of compatibility it has with both the Mac OS X operating system and the third party applications that are required by the user. Currently, not all of the Mac OS X Finder application is accessible by the screen reader. It is hoped and expected that these issues will be resolved in future versions of the operating system. Depending on the methods used in development, most third party applications are accessible to some degree. As complete compatibility in these applications will generally require extra effort on the part of the programmer, it is unclear whether software developers will support Apple's Accessibility API. It is hoped that all applications developed and supported by Apple will be completely compatible with the Accessibility API. With a basic office application in AppleWorks, a web browser in the form of Safari and chat and email client, the Apple software suite contains most applications required by users. If all of these are accessible by the screen reader, a complete computer experience can be provided to the vision-impaired user.

It is worth noting that the U.S. Government's Section 508 standard will eventually force software and hardware developers to make their products accessible to the disabled. This will mean future software products from Apple and third-party developers should have an increased in level of compliance with the Accessibility API.

6.2. Future Work

With the basic design and program structure complete, future work on this project will involve coding extra functionality into the AXStringProcessor class to allow interaction with a wider range of user interface elements. This involves extending the processing algorithm to search for other user interface elements and constructing meaningful messages to convey to the user.

The user preference system needs to be developed and a convenient method of changing settings implemented. This needs to be convenient for all users of the screen reader. The screen reader also needs to be able to detect key presses (and combinations of key presses) and map them to the keyboard shortcuts available in the current active application.

The capturing of system events also needs to be implemented, possibly via kernel extensions. This would allow the screen reader to be aware of messages boxes that appear rather than relying on the user to discover them by chance. The speech synthesis function of the screen reader also needs to be examined and methods of extracting greater speech rates investigated.

A significant component of making Mac OS X accessible to the vision-impaired is removing the dependency on the mouse for user input. Developing an add-on for Mac OS X that allows the user to drive the graphical user interface using the keyboard would improve accessibility significantly. This may be possible to implement using a kernel extension.

The Parakeet screen reader requires an extensive program of testing and feedback if it is to be developed into a useful product. This would involve testing Parakeet with vision-impaired users of varying computer literacy, collecting feedback information and observing usage patterns. This information would then need to be analysed and the necessary changes made to the software.

REFERENCES

Apple Computer Inc. (2002). Making Your Carbon Application Accessible to Users with Disabilities, Apple Computer Inc. Downloaded from http://developer.apple.com/techpubs/macosx/Carbon/HumanInterfaceToolbox/Accessibility/accessibility.html in 2003.

Apple Computer Inc. (2002). Inside Mac OS X: The Objective-C Programming Language, Apple Computer Inc. Downloaded from http://developer.apple.com/ in 2003.

Apple Computer Inc. (2002). Inside Mac OS X: Speech Synthesis Manager Reference, Apple Computer Inc. Downloaded from http://developer.apple.com/ in 2003.

Apple Computer Inc. (2002). UNIX Porting Guide: An Overview of How to Bring UNIX Applications to Mac OS X, Apple Computer Inc. Downloaded from http://developer.apple.com/ in 2002.

Association for the Blind of W.A. (Inc), (2001). Understanding Blindness, Association for the Blind of W.A. Downloaded from http://www.abwa.asn.au/resources.html in 2003.

Davidson, J. D. & Apple Computer Inc. (2002). Learning Cocoa with Objective-C, O'Reilly.

Evans, G. & Blenkhorn, P. (2002). Architectures of assistive software applications for Windows-based computers, Journal of Network and Computer Applications (online).

The GNOME Accessibility Project, The GNOME Project. Downloaded from http://developer.gnome.org/projects/gap/ in 2003.

Graham, I. & Quin, L. (1999). XML Specification Guide, Wiley Computer Publishing.

Lake, L. (2003). ALVA to Cease Development of outSPOKEN for Macintosh and inLARGE for Macintosh, ALVA Access Group. Downloaded from http://www.aagi.com/news/news.asp?44 in 2003.

Mates, B. (2000). Adaptive Technology for the Internet: Making Electronic Resources Accessible to All, American Library Association.

Murray, I. (2000). Making IT Accessible, School of Electrical and Computer Engineering, Curtin University of Technology. Downloaded from http://www.ece.curtin.edu.au/~iain/webaccess/ in 2003.

Pinson, L.. & Wiener, R. (1991). Objective-C: Object-Oriented Programming Techniques, Addison-Wesley Publishing Company.

Raman, T.V. (1997). Emacspeak: A Speech-Enabling Interface, Dr. Dobb's Journal, September 1997 (available online from http://www.ddj.com/).

A. APPENDIX A – OPERATING SYSTEM AND COMPILER VERSIONS

The software was developed concurrently on an Apple eMac (supplied as part of the AUC grant) and an Apple Macintosh PowerBook G3. Both were running the same version of the Mac OS X operating system, namely Mac OS X 10.2.6.

The software was developed and compiled using Apple's Project Builder integrated development environment. The version used was Project Builder 2.1 (December 2002 Developer Tools Edition). The components of the IDE are recorded in the following table.

Component	Version
PB IDE	Revision 114.0
PB CODE	Revision 112.0
ToolSupport	Revision 110.0

Table 5: IDE Component Versions

The following table lists the versions of the compiler and debugger used by the Project Builder IDE. It is worth noting that this software was developed using GCC 3.1, and has not been tested with the newer version of GCC (version 3.3) that is included with the Mac OS X 10.3 (Panther) operating system.

Tool	Version
gcc (GNU C Compiler)	GCC version 1175, based on gcc version 3.1 20020420 (prerelease)
gdb (GNU Debugger)	GNU gdb 5.3-20021014 (Apple version gdb-250)

Table 6: Compiler and Debugger Versions

B. APPENDIX B – SOURCE CODE

B.1. AppShell.h

```
/*
    -----------------------------------------------------------------------
-----------------------
    File Name:              AppShell.h
    Project:                Parakeet
    Author:                 Greg Howell (with some help from Apple sample
code)
    Revision Date:          22/08/2003 (0.2)

    Description:  AppShell object interface.
    -----------------------------------------------------------------------
-----------------------
*/

#import <Cocoa/Cocoa.h>
#import <Appkit/NSAccessibility.h>
#import "InspectorWindow.h"
#import "AXStringProcessor.h"

/*
    -----------------------------------------------------------------------
-----------------------
    AppShell interface
    -----------------------------------------------------------------------
-----------------------
*/
@interface AppShell : NSObject {
    // reference to the InspectorWindow object that we have
    // InspectorWindow is the interface window of Parakeet
    IBOutlet InspectorWindow * _inspectorWindow;

    // stores information for the current UI element selected
    AXUIElementRef _currentUIElementRef;
    AXUIElementRef _systemWideElement;

    // used to determine if the mouse has moved since the last timer-based
update
    Point _lastMousePoint;

    // speech channel used by TTS code
    SpeechChannel fCurSpeechChannel;
}

// class methods
+ (void)updateCurrentUIElement;
+ (NSString *)descriptionOfValue:(CFTypeRef)theValue
beingVerbose:(BOOL)beVerbose;
+ (NSString *)descriptionForUIElement:(AXUIElementRef)uiElement
attribute:(NSString *)name beingVerbose:(BOOL)beVerbose;
+ (NSString *)stringDescriptionOfAXValue:(CFTypeRef)valueRef
beingVerbose:(BOOL)beVerbose;
+ (id)valueOfExistingAttribute:(CFStringRef)attribute
ofUIElement:(AXUIElementRef)element;
+ (void)processAXString:(NSMutableString *)theAXString;

// instance methods
```

```
- (void)setCurrentUIElement:(AXUIElementRef)uiElement;
- (AXUIElementRef)currentUIElement;
- (void)performTimerBasedUpdate;
- (void)updateCurrentUIElement;
- (void)createNewSpeechChannel;
- (void)startSpeakingText;
- (void)speechIsDone;
- (void)setTTSString:(NSMutableString *)theTTSString;

@end
```

B.2. AppShell.m

```
/*
    ---------------------------------------------------------------------------
-----------------------
    File Name:              AppShell.m
    Project:                Parakeet
    Author:                 Greg Howell (with some help from Apple sample
code)
    Revision Date:          22/08/2003 (0.2)

    Description:            AppShell object implementation.
                            Interrogates the Accessibility API and produces
strings to be spoken via the TTS API.
    ---------------------------------------------------------------------------
-----------------------
*/

#import <Cocoa/Cocoa.h>
#import <AppKit/NSAccessibility.h>
#import <Carbon/Carbon.h>
#import "AppShell.h"
#import "Parakeet.h"

AppShell * gAppShell = NULL;                    // an AppShell object called
gAppShell, initialised to NULL
AXStringProcessor * _AXStringProcessor = NULL;

NSMutableString * lastTTSString;    // state variable that stores the last
spoken string
NSMutableString * currentTTSString; // variable that stores the string to be
spoken
char * theTTSString;                            // string used to call the TTS
functions (a "C" string)

const UInt32 kShutupHotKeyIdentifier = 'sUIk';
const UInt32 kShutupHotKey = 109;   // corresponds to F10 (and command key)

EventHotKeyRef gMyHotKeyRef;                    // hot key related object
EventHotKeyID gMyHotKeyID;          // hot key related object
EventHandlerUPP gAppHotKeyFunction; // hot key related object

// prototype of the callback process that lets us know when the TTS has
finished
static pascal void OurSpeechDoneCallBackProc(SpeechChannel inSpeechChannel,
long inRefCon);

// prototype for the shutup hot-key handler
pascal OSStatus ShutupHotKeyHandler(EventHandlerCallRef nextHandler,
EventRef theEvent, void *userData);

@implementation AppShell
```

```
/*
    ------------------------------------------------------------------------
-----------------------
    updateCurrentUIElement
    ------------------------------------------------------------------------
-----------------------
*/
+ (void)updateCurrentUIElement
{
    [gAppShell updateCurrentUIElement];
}
/*
    ------------------------------------------------------------------------
-----------------------
    descriptionOfValue

    Deals with element title (eg. Mail) and role (eg. Application)
    ------------------------------------------------------------------------
-----------------------
*/
+ (NSString *)descriptionOfValue:(CFTypeRef)theValue
beingVerbose:(BOOL)beVerbose
{
    NSString * theValueDescString = NULL;

    if (theValue) {

        if (AXValueGetType(theValue) != kAXValueIllegalType) {
            theValueDescString = [AppShell
stringDescriptionOfAXValue:theValue beingVerbose:beVerbose];
        }
        else if (CFGetTypeID(theValue) == CFArrayGetTypeID()) {
            theValueDescString = [NSString stringWithFormat:@"<array of size
%d>", [(NSArray *)theValue count]];
        }
        else if (CFGetTypeID(theValue) == AXUIElementGetTypeID()) {

            NSString *     uiElementRole     = NULL;

            if (AXUIElementCopyAttributeValue( (AXUIElementRef)theValue,
kAXRoleAttribute, (CFTypeRef *)&uiElementRole ) == kAXErrorSuccess) {
                NSString * uiElementTitle  = NULL;

                uiElementTitle = [AppShell
valueOfExistingAttribute:kAXTitleAttribute
ofUIElement:(AXUIElementRef)theValue];

                #if 0
                // hack to work around cocoa app objects not having titles
yet
                if (uiElementTitle == nil && [uiElementRole
isEqualToString:(NSString *)kAXApplicationRole]) {
                    pid_t theAppPID = 0;
                    ProcessSerialNumber      theAppPSN = {0,0};
                    NSString * theAppName = NULL;

                    if (AXUIElementGetPid( (AXUIElementRef)theValue,
&theAppPID ) == kAXErrorSuccess
                        && GetProcessForPID( theAppPID, &theAppPSN ) ==
noErr
                        && CopyProcessName( &theAppPSN, (CFStringRef
*)&theAppName ) == noErr ) {
                        uiElementTitle = theAppName;
                    }
                }
                #endif
                if (uiElementTitle != nil) {
                    theValueDescString = [NSString stringWithFormat:@"<%@:
“%@”>", uiElementRole, uiElementTitle];
                }
                else {
                    theValueDescString = [NSString stringWithFormat:@"<%@>",
uiElementRole];
                }
                [uiElementRole release];
            }
            else {
                theValueDescString = [(id)theValue description];
            }
        }
        else {
            theValueDescString = [(id)theValue description];
        }
    }
    return theValueDescString;
}

/*
    ------------------------------------------------------------------------
-----------------------
    descriptionForUIElement
    ------------------------------------------------------------------------
-----------------------
*/
+ (NSString *)descriptionForUIElement:(AXUIElementRef)uiElement
attribute:(NSString *)name beingVerbose:(BOOL)beVerbose
{
    NSString *    theValueDescString         = NULL;
    CFTypeRef     theValue;
    CFIndex       count;
    if (([name isEqualToString:NSAccessibilityChildrenAttribute]
            ||
         [name isEqualToString:NSAccessibilityRowsAttribute]
        )
            &&
        AXUIElementGetAttributeValueCount(uiElement, (CFStringRef)name,
&count) == kAXErrorSuccess) {
        // No need to get the value of large arrays - we just display their
size.
        // We don't want to do this with every attribute because
AXUIElementGetAttributeValueCount
        // on non-array valued attributes will cause debug spewage.
        theValueDescString = [NSString stringWithFormat:@"<array of size
%d>", count];
    } else if (AXUIElementCopyAttributeValue ( uiElement, (CFStringRef)name,
&theValue ) == kAXErrorSuccess && theValue) {
        theValueDescString = [self descriptionOfValue:theValue
beingVerbose:beVerbose];
    }
    return theValueDescString;
}
/*
    ------------------------------------------------------------------------
-----------------------
    valueOfExistingAttribute
    ------------------------------------------------------------------------
-----------------------
*/
+ (id)valueOfExistingAttribute:(CFStringRef)attribute
ofUIElement:(AXUIElementRef)element
{
    id result = nil;
    NSArray *attrNames;

    if (AXUIElementCopyAttributeNames(element, (CFArrayRef *)&attrNames) ==
kAXErrorSuccess) {
        if ( [attrNames indexOfObject:(NSString *)attribute] != NSNotFound
                &&
         AXUIElementCopyAttributeValue(element, attribute, (CFTypeRef
*)&result) == kAXErrorSuccess
        ) {
            [result autorelease];
        }
        [attrNames release];
    }
    return result;
}
/*
    ------------------------------------------------------------------------
-----------------------
    stringDescriptionOFAXValue
    ------------------------------------------------------------------------
-----------------------
*/
+ (NSString *)stringDescriptionOfAXValue:(CFTypeRef)valueRef
beingVerbose:(BOOL)beVerbose
{
    NSString *result = @"AXValue???";

    switch (AXValueGetType(valueRef)) {
        case kAXValueCGPointType: {
            CGPoint point;
            if (AXValueGetValue(valueRef, kAXValueCGPointType, &point)) {
                if (beVerbose)
                    result = [NSString stringWithFormat:@"<AXPointValue x=%g
y=%g>", point.x, point.y];
                else
                    result = [NSString stringWithFormat:@"x=%g y=%g",
point.x, point.y];
            }
            break;
        }
        case kAXValueCGSizeType: {
            CGSize size;
            if (AXValueGetValue(valueRef, kAXValueCGSizeType, &size)) {
                if (beVerbose)
                    result = [NSString stringWithFormat:@"<AXSizeValue w=%g
h=%g>", size.width, size.height];
                else
                    result = [NSString stringWithFormat:@"w=%g h=%g",
size.width, size.height];
            }
            break;
        }
        case kAXValueCGRectType: {
            CGRect rect;
            if (AXValueGetValue(valueRef, kAXValueCGRectType, &rect)) {
                if (beVerbose)
                    result = [NSString stringWithFormat:@"<AXRectValue  x=%g
y=%g w=%g h=%g>", rect.origin.x, rect.origin.y, rect.size.width,
rect.size.height];
                else
                    result = [NSString stringWithFormat:@"x=%g y=%g w=%g
h=%g", rect.origin.x, rect.origin.y, rect.size.width, rect.size.height];
            }
            break;
        }
        case kAXValueCFRangeType: {
            CFRange range;
            if (AXValueGetValue(valueRef, kAXValueCFRangeType, &range)) {
                if (beVerbose)
                    result = [NSString stringWithFormat:@"<AXRangeValue 
pos=%ld len=%ld>", range.location, range.length];
                else
                    result = [NSString stringWithFormat:@"pos=%ld len=%ld",
range.location, range.length];
            }
            break;
        }
        default:
            break;
    }
    return result;
}
/*
    --------------------------------------------------------------------------
-----------------------
    processAXString
    
    - Takes the output of the InspectorWindow displayInfoForUIElement 
method.
    - Uses the AXStringProcessor object to process the string.
    - Sets the global currentTTSString with the string that 
AXStringProcessor produces.
    
    - Coded by Greg Howell 01/09/2003, revised and added to 07/09/2003.
    - Revised by Greg Howell 21/09/2003 to use AXStringProcessor object.
    --------------------------------------------------------------------------
-----------------------
*/
+ (void)processAXString:(NSMutableString *)theAXString
{
    if(DEBUG_PARAKEET){printf("AppShell:processAXString\n");}
    
    [_AXStringProcessor setAXString:theAXString];
    [_AXStringProcessor processAXString];
    NSString * strToSpeak = [_AXStringProcessor returnAXString];
    
    if(DEBUG_PARAKEET){printf("AppShell:processAXString produced: 
%s\n",(char *)[strToSpeak cString]);}
    currentTTSString = (NSMutableString *)strToSpeak;
}

/*
    --------------------------------------------------------------------------
-----------------------
    awakeFromNib
    
    - Executes when the application is loaded (nib file loaded).
    - Checks that the Accessibility APIs are turned on.
    - Sets up the application (key listening, TTS initialisation).
    --------------------------------------------------------------------------
-----------------------
*/
- (void)awakeFromNib
{
    EventTypeSpec eventType;
    
    // We first have to check if the Accessibility APIs are turned on.  If 
not, we have to tell
    // the user to do it (they'll need to authenticate to do it).  If you 
are an accessibility app
    // (i.e., if you are getting info about UI elements in other apps), the 
APIs won't work unless
    // the APIs are turned on.
    if (!AXAPIEnabled ())
    {
        NSRunAlertPanel(@"Error",@"Please check the 'Enable access for
assistive devices' checkbox in the Universal Access System Preferences and
relaunch Parakeet.",@"OK",NULL,NULL);
         // quit the application here as we cannot keep going from here
        [NSApp terminate:nil];
    }

    gAppShell = self;

    if(DEBUG_PARAKEET){printf("AppShell:awakeFromNib (DEBUGGING
ENABLED)\n");}

    _AXStringProcessor = [[AXStringProcessor alloc] init];
    [_AXStringProcessor reset];

    _systemWideElement = AXUIElementCreateSystemWide();

    gAppHotKeyFunction = NewEventHandlerUPP(ShutupHotKeyHandler);
    eventType.eventClass = kEventClassKeyboard;
    eventType.eventKind = kEventHotKeyPressed;
    InstallApplicationEventHandler(gAppHotKeyFunction, 1, &eventType, NULL,
NULL);
    gMyHotKeyID.signature = kShutupHotKeyIdentifier;
    gMyHotKeyID.id = 1;
    // unfortunately we cannot simply remove the "cmdKey" parameter from
this call
    // to make just F10 the key we are listening for
    RegisterEventHotKey(kShutupHotKey, cmdKey, gMyHotKeyID,
GetApplicationEventTarget(), 0, &gMyHotKeyRef);

    // set the lastTTString variable to an initial value so comparisons
further on don't fail
    // we need to "cast" our initialisation string as an NSMutableString
    // (this avoids a compiler warning but either way the code works fine?)
    // I have initialised with a dash character as this is unlikely to
conflict with any generated
    // strings when it comes to performing string comparisons
    // Greg Howell 15/08/2003
    lastTTSString = (NSMutableString *)@"-";

    // create a new speech channel to use
    [self createNewSpeechChannel];
    [self performTimerBasedUpdate];

    // initialise the start/stop state so that we start speaking as soon
    // as the application is launched
    [_inspectorWindow setStartStopState:TRUE];
}
/*
    -----------------------------------------------------------------------
-----------------------
    performTimerBasedUpdate

    - application uses timer based updates to check what is going on with
the user and the system
    -----------------------------------------------------------------------
-----------------------
*/
- (void)performTimerBasedUpdate
{
    [gAppShell updateCurrentUIElement];

    [NSTimer scheduledTimerWithTimeInterval:0.1 target:self
selector:@selector(performTimerBasedUpdate) userInfo:nil repeats:NO];
}
/*
```

```
    ------------------------------------------------------------------------
-----------------------
    setCurrentUIElement
    ------------------------------------------------------------------------
-----------------------
*/
- (void)setCurrentUIElement:(AXUIElementRef)uiElement
{
    if (uiElement)
        CFRetain( uiElement );

    if (_currentUIElementRef)
        CFRelease( _currentUIElementRef );

         _currentUIElementRef = uiElement;
}
/*
    ------------------------------------------------------------------------
-----------------------
    currentUIElement
    ------------------------------------------------------------------------
-----------------------
*/
- (AXUIElementRef)currentUIElement
{
    return _currentUIElementRef;
}
/*
    ------------------------------------------------------------------------
-----------------------
    updateCurrentUIElement
    ------------------------------------------------------------------------
-----------------------
*/
- (void)updateCurrentUIElement
{
    // Point object
    Point pointAsCarbonPoint;

    // The current mouse position with origin at top left.
    GetMouse( &pointAsCarbonPoint );

    // Only ask for the UIElement under the mouse if has moved since the
last check.
    // Prevents unnecessary updating of information.
    if (pointAsCarbonPoint.h != _lastMousePoint.h || pointAsCarbonPoint.v !=
_lastMousePoint.v) {

        CGPoint   pointAsCGPoint;
        AXUIElementRef newElement = NULL;

        pointAsCGPoint.x = pointAsCarbonPoint.h;
        pointAsCGPoint.y = pointAsCarbonPoint.v;

        // Ask Accessibility API for UI Element under the mouse
        // And update the display if a different UIElement
        if (AXUIElementCopyElementAtPosition( _systemWideElement,
pointAsCGPoint.x, pointAsCGPoint.y, &newElement ) == kAXErrorSuccess &&
newElement && ([self currentUIElement] == NULL || ! CFEqual( [self
currentUIElement], newElement ))) {

            [self setCurrentUIElement:newElement];
            [_inspectorWindow displayInfoForUIElement:newElement];

            // display the current application name (GH)
            [_inspectorWindow setCurrentAppString:currentTTSString];
            // set the TTS string (GH)
            [self setTTSString:currentTTSString];
        }
        // Update _lastMousePoint
        _lastMousePoint = pointAsCarbonPoint;
    }
}

/*
    ---------------------------------------------------------------------------
-----------------------
    createNewSpeechChannel

    - Create a new speech channel for the given voice spec.  A nil voice
spec pointer
      causes the speech channel to use the default voice.  Any existing
speech channel
      for this window is closed first.

    - Borrowed from Apple's Cocoa Speech Synthesis Example (24/07/2003)
    - Modified by Greg Howell (31/07/2003)
    ---------------------------------------------------------------------------
-----------------------
*/
- (void)createNewSpeechChannel
{
    OSErr theErr = noErr;

    // Dispose of the current one, if it exists
    if (fCurSpeechChannel) {
        theErr = DisposeSpeechChannel(fCurSpeechChannel);
        if (theErr != noErr)
           NSRunAlertPanel(@"DisposeSpeechChannel", [NSString
stringWithFormat:@"Error #%d returned.", theErr], @"Ok", NULL, NULL);

        fCurSpeechChannel = NULL;
    }

    // Create a speech channel
    if (theErr == noErr) {
         theErr = NewSpeechChannel(NULL, &fCurSpeechChannel);
        if (theErr != noErr)
            NSRunAlertPanel(@"NewSpeechChannel", [NSString
stringWithFormat:@"Error #%d returned.", theErr], @"Ok", NULL, NULL);
         }

         // Setup our refcon to the document controller object so we have
access within our Speech callbacks
         if (theErr == noErr) {
            theErr = SetSpeechInfo(fCurSpeechChannel, soRefCon, (Ptr)self);
            if (theErr != noErr)
                NSRunAlertPanel(@"SetSpeechInfo(soRefCon)", [NSString
stringWithFormat:@"Error #%d returned.", theErr], @"Ok", NULL, NULL);
         }

    if(DEBUG_PARAKEET){
        printf("AppShell:createNewSpeechChannel\n");
    }
}

/*
    ---------------------------------------------------------------------------
-----------------------
    startSpeakingText

    - Speaks the text.

    - Modified code from Apple's Cocoa Speech Synthesis Example (24/07/2003)
    - Added ability to check start/stop TTS state information (20/08/2003)
    ------------------------------------------------------------------------
-----------------------
*/
- (void)startSpeakingText
{
    OSErr theErr = noErr; // store any errors that we might encounter

    // check if we can speak by accessing the start/stop TTS state stored
    // in the InspectorWindow object
    if ([_inspectorWindow getStartStopState] == TRUE){
        // set up the callback (TTS has finished)
        // this is done via the SetSpeechInfo method
        if (theErr == noErr){
            theErr = SetSpeechInfo(fCurSpeechChannel, soSpeechDoneCallBack,
OurSpeechDoneCallBackProc);
            if (theErr != noErr)

NSRunAlertPanel(@"SetSpeechInfo(soSpeechDoneCallBack)",[NSString
stringWithFormat:@"Error #%d returned.", theErr], @"Ok", NULL, NULL);
        }

        // speak the string
        // strlen only works on "C" strings
        theErr =
SpeakText(fCurSpeechChannel,theTTSString,strlen(theTTSString));
    }
}

/*
    ------------------------------------------------------------------------
-----------------------
    speechIsDone

    - This method is called by the OurSpeechDoneCallBackProc process.
    ------------------------------------------------------------------------
-----------------------
*/
- (void)speechIsDone
{
    // do nothing for the time being
    if(DEBUG_PARAKEET){
        printf("AppShell:speechIsDone\n");
    }
}

/*
    ------------------------------------------------------------------------
-----------------------
    setTTSString

    - Method to set the TTS string (calls the TTS routine if required)
    - Coded by Greg Howell (13/08/2003)
    ------------------------------------------------------------------------
-----------------------
*/
- (void)setTTSString:(NSMutableString *)stringToSpeak
{
    // declare local NSMutableString to store the last string spoken
    NSMutableString *tempLast = [[NSString alloc]
initWithString:lastTTSString];
    // declare local NSMutableString to store the next string to be spoken
    NSMutableString *tempNext = [[NSString alloc]
initWithString:stringToSpeak];

    // perform a comparison on the two strings
    BOOL bCompare = [tempLast isEqualToString:tempNext];
    // if they are the same we don't speak the string
    // this stops us repeating ourselves (as that gets a tad annoying)

    if (bCompare != TRUE){
        // convert the NSString provided into a "C" string
        // you need to call the cString method AND cast the result
        // as a (char *)
        theTTSString = (char *)[tempNext cString];

        // set the last string variable to the string we are about to say
        // could go in the SpeechIsDone method but we will do it here to
make sure
        // it happens
        lastTTSString = tempNext;

        // call startSpeakingText
        [self startSpeakingText];
    }


}

/*
    -------------------------------------------------------------------------
-----------------------
    toggleShutup

    - Called when the user activates our hot-key
    - Coded by Greg Howell (20/08/2003)
    -------------------------------------------------------------------------
-----------------------
*/
- (void)toggleShutup:(id)sender
{
    if ([_inspectorWindow getStartStopState] == TRUE)
    {
        [_inspectorWindow setStartStopState:FALSE];   // set to FALSE (shut
up)
    }
    else
    {
        [_inspectorWindow setStartStopState:TRUE];    // set to TRUE (stop
being shut up)
    }
}

@end

/*
    -------------------------------------------------------------------------
-----------------------
    OurSpeechDoneCallBackProc

    - Called by speech channel when all speech has been generated.
    - Modified code from Apple's Cocoa Speech Synthesis Example (24/07/2003)
    -------------------------------------------------------------------------
-----------------------
*/
pascal void OurSpeechDoneCallBackProc(SpeechChannel inSpeechChannel, long
inRefCon)
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    // calls the AppShell method "speechIsDone" when TTS has completed
    [(AppShell *)inRefCon
performSelectorOnMainThread:@selector(speechIsDone) withObject:NULL
waitUntilDone:false];
```

```
    [pool release];
}
/*
    ------------------------------------------------------------------------
-----------------------
    ShutupHotKeyHandler

    - Called when the user activates our hot-key
    - Modified code from Apple's UIElementInspector Example (20/08/2003)
    ------------------------------------------------------------------------
-----------------------
*/
pascal OSStatus ShutupHotKeyHandler(EventHandlerCallRef nextHandler,
EventRef theEvent, void *userData){

    [NSTimer scheduledTimerWithTimeInterval:0.1 target:gAppShell
selector:@selector(toggleShutup:) userInfo:nil repeats:NO];

    return noErr;
}
```

B.3. AXStringProcessor.h

```
/*
    ------------------------------------------------------------------------
-----------------------
    File Name:              AXStringProcessor.h
    Project:                Parakeet
    Author:                 Greg Howell
    Revision Date:          21/09/2003 (0.2)

    Description:            AXStringProcessor object interface.
    ------------------------------------------------------------------------
-----------------------
*/

#import <Foundation/Foundation.h>

@interface AXStringProcessor : NSObject {
    NSMutableString * stringToProcess;                          // current
string to process
    NSString * currentStringProduced;                           // current
string produced by object
    NSString * lastStringProduced;                     // last string
produced by object
}
- (void)setAXString:(NSMutableString *)newString;     // set the AXString to
be processed
- (void)processAXString;                               // process the
AXString
- (NSString *)returnAXString;                                   // return
processed result
- (void)reset;                                                  // reset the
internal variables of the object
- (NSString *)extractAXValue:(NSString *)sourceString;          // extract
the AX value from a string
- (BOOL)lineIsEmpty:(NSString *)lineToTest;            // utility method to
test for empty string
- (void)processAXHierarchy:(NSString *)passedString;  // process the
hierarchy part of the AX info
- (void)processAXAttributes:(NSString *)passedString; // process the
attributes part of the AX info
- (void)processAXRemaining:(NSString *)passedString;  // process the
```

```
remaining part of the AX info

@end
```

B.4. AXStringProcessor.m

```
/*
    ---------------------------------------------------------------------------
-----------------------
    File Name:              AXStringProcessor.m
    Project:                Parakeet
    Author:                 Greg Howell
    Revision Date:          21/09/2003 (0.2)

    Description:            AXStringProcessor object implementation.
    ---------------------------------------------------------------------------
-----------------------
*/

#import "AXStringProcessor.h"
#import "Parakeet.h"

NSString * smartQuoteLeft = @"\322";         // Note that \322 is the code
for the opening smart
NSString * smartQuoteRight = @"\323";        // quote and \323 the code for
the closing smart quote

@implementation AXStringProcessor

/*
    ---------------------------------------------------------------------------
-----------------------
    Method Name:  setAXString
    Parameters:             NSMutableString object
    Returns:                None

    - Instance method for setting the string to process.
    ---------------------------------------------------------------------------
-----------------------
*/
- (void)setAXString:(NSMutableString *)newString
{
    [newString retain];                      // keep the string we are
passed
    [stringToProcess release];               // release the previous string
    stringToProcess = newString;    // store the passed string
    if(DEBUG_PARAKEET){printf("AXStringProcessor:setAXString\n");}
}

/*
    ---------------------------------------------------------------------------
-----------------------
    Method Name:            processAXString
    Parameters:             None
    Returns:                None

    - Instance method that produces the string to be returned.
    ---------------------------------------------------------------------------
-----------------------
*/
- (void)processAXString
{
    if(DEBUG_PARAKEET){printf("AXStringProcessor:processAXString\n");}
    [self processAXHierarchy:stringToProcess];
}

/*
    ------------------------------------------------------------------------
-----------------------
    Method Name:            returnAXString
    Parameters:             None
    Return Value:           NSString object

    - Instance method that returns the produced string.
    ------------------------------------------------------------------------
-----------------------
*/
- (NSString *)returnAXString
{
    if(DEBUG_PARAKEET){printf("AXStringProcessor:returnAXString\n");}
    return currentStringProduced;
}

/*
    ------------------------------------------------------------------------
-----------------------
    Method Name:            reset
    Parameters:             None
    Returns:                None

    - Instance method that resets the object.
    ------------------------------------------------------------------------
-----------------------
*/
- (void)reset
{
    [currentStringProduced release];                            // reset the
currentStringProduced object
    currentStringProduced = @"";

    [stringToProcess release];                                  // reset the
stringToProcess object
    stringToProcess = (NSMutableString *)@"";

    [lastStringProduced release];                       // reset the
lastStringProduced object
    lastStringProduced = @"";

    if(DEBUG_PARAKEET){printf("AXStringProcessor:reset\n");}
}

/*
    ------------------------------------------------------------------------
-----------------------
    Method Name:            extractAXValue
    Parameters:             NSString object
    Returns:                NSString object

    - Instance method that extracts a key out of an AX string.
    - Passed an NSString object.
    - String between the smart quotes in the passed string is extracted and
returned.

    - Coded by Greg Howell 22/09/2003
    ------------------------------------------------------------------------
-----------------------
*/
- (NSString *)extractAXValue:(NSString *)sourceString
{
    NSRange startRange;                         // used to locate the starting
```

```
quote
    NSRange endRange;                        // used to locate the ending
quote
    NSRange outputRange;            // used to select the AX key value
    NSString * stringToExtractFrom; // stores the string that is passed to
the method
    NSString * valueToReturn;                // stores the string that the
method is going to return
    int lengthOfInput;                       // length of the passed string

    stringToExtractFrom = sourceString;                       // store
passed string
    lengthOfInput = 0;                                        // initialize
length
    lengthOfInput = [stringToExtractFrom length];     // store length of
passed input

    if (lengthOfInput > 0) {                                  // check that
the passed string is of valid length
        startRange = [stringToExtractFrom rangeOfString:smartQuoteLeft]; //
get start quote location
        endRange = [stringToExtractFrom rangeOfString:smartQuoteRight]; //
get end quote location

        if((startRange.length < 0) && (endRange.length < 0)){
            valueToReturn = @"";
        }
        else{
            outputRange.location = startRange.location + 1; // compute value
location
            outputRange.length = endRange.location - startRange.location -
1;       // compute value length

            if (outputRange.length > 0){     // ensure that we are not going
to return garbage
                valueToReturn = [stringToExtractFrom
substringWithRange:outputRange];
            }
            else{
                valueToReturn = @"";         // return empty string if
result is nothing
            }
        }
    }
    else{
        valueToReturn = @"";                 // return empty string if input
is empty string
    }

    return valueToReturn;           // return string to calling method
}

/*
    ------------------------------------------------------------------------
-----------------------
    Method Name:            lineIsEmpty
    Parameters:             NSString object
    Returns:                BOOL object

    - Instance method tests if a string object (line in this application) is
empty.
    - Passed an NSString object to test.

    - Coded by Greg Howell 23/09/2003
    ------------------------------------------------------------------------
-----------------------
*/
```

```
- (BOOL)lineIsEmpty:(NSString *)lineToTest
{
    NSString * stringPassed;                          // string to store
passed object
    stringPassed  = lineToTest;                       // store what we are
passed

    if ([stringPassed isEqualToString:@""]){ // test for equality to the
empty string object
        return TRUE;                                  // return true if
equality returns true
    }
    else if([stringPassed isEqualToString:@" "]){
        return TRUE;
    }
    else{
        return FALSE;                                 // return false
otherwise
    }
}


/*
    ------------------------------------------------------------------------
-----------------------
    Method Name:           processAXHierarchy
    Parameters:            NSString object
    Returns:               None

       -  Processes  the  "hierarchy"  section  of  the  data  returned  by  the
Accessibility API.
    - Breaks the data into an array.
    - Loops through the array until it reaches an empty row.
      -  Checks  for  various  tags  and  constructs  the  string  to  speak  to  the
user.
    - Provides the "intelligence" (dim as it may be) for Parakeet.

    - Coded by Greg Howell 23/09/2003, revised 14/10/2003
    ------------------------------------------------------------------------
-----------------------
*/
- (void)processAXHierarchy:(NSString *)passedString
{
    int  rowBeingProcessed;                                   //  the  line
of the array we are processing
             NSString   *   lineToProcess;                            //
the contents of the line of the array we are processing
    BOOL  keepGoing;                                                  //
flag used in the while loop
                NSRange   searchRange;                        //   utility
search range
                 NSMutableString    *    stringToReturn @""; //    string (NSMutab
that we are producing
    // (note the strange way of initialising an NSMutableString - looks very
dodgy - GH 24/09/2003)

             NSString   *   stringAppName;                            //
application name
    BOOL  bInDock;                                            //  flag  to
indicate we are in the Dock application
    BOOL  bInFinder;                                                  //
flag to indicate we are in the Finder application
    BOOL  bInMenuBar;                                                 //
flag to indicate we are in the Menu Bar area

      NSMutableString  *  menuInfo  =  (NSMutableString  *) @""; //  string  to
store info about the child menu object
```

```
              keepGoing   =   TRUE;                        //
initialisation section
    rowBeingProcessed = 0;
                 NSArray   *   AXDataArray   =   [(NSString   *)passe
componentsSeparatedByString:@"\n"];
    bInDock = FALSE;
    bInMenuBar = FALSE;

    if(DEBUG_PARAKEET){printf("AXStringProcessor:processAXHieracrhy (%i rows
of text to process)\n", [AXDataArray count]);}

    while(keepGoing){
        lineToProcess = [AXDataArray objectAtIndex:rowBeingProcessed];

        if([self lineIsEmpty:lineToProcess]){
            keepGoing = FALSE;
        }
        else{
            // --------------------------------------------------------------
----------------- AXApplication
            searchRange = [lineToProcess rangeOfString:@"AXApplication"];
            if(searchRange.length){
                                                                      self
extractAXValue:lineToProcess];

                bInDock = [stringAppName isEqualToString:@"Dock"];
                bInFinder = [stringAppName isEqualToString:@"Finder"];

                                                                           str
*)stringToReturn stringByAppendingString:stringAppName];
                                                                           str
*)stringToReturn stringByAppendingString:@" "];
            }
            // --------------------------------------------------------------
----------------- AXWindow
            searchRange = [lineToProcess rangeOfString:@"AXWindow"];
            if(searchRange.length){
                                                                           str
*)stringToReturn stringByAppendingString:@" "];
                if(!(bInFinder)){
                                                                           str
*)stringToReturn  stringByAppendingString:(NSMutableString  *)[self
extractAXValue:lineToProcess]];
                                                                           str
*)stringToReturn stringByAppendingString:@" window "];
                }
            }
            // --------------------------------------------------------------
----------------- AXMenuBar
            searchRange = [lineToProcess rangeOfString:@"AXMenuBar"];
            if(searchRange.length){
                bInMenuBar = TRUE;
                menuInfo = (NSMutableString *)@"";
                                                  menuInfo   =   (NSMutableString
stringByAppendingString:(NSMutableString    *)[self
extractAXValue:lineToProcess]];
                                                  menuInfo   =   (NSMutableString
stringByAppendingString:@" menu bar "];
            }
            // --------------------------------------------------------------
----------------- AXMenu
            searchRange = [lineToProcess rangeOfString:@"AXMenu"];
            if(searchRange.length){
                bInMenuBar = TRUE;
                menuInfo = (NSMutableString *)@"";
                                                  menuInfo   =   (NSMutableString
stringByAppendingString:(NSMutableString      *)[self
extractAXValue:lineToProcess]];
                                                menuInfo   =   (NSMutableString
stringByAppendingString:@" menu "];
            }
            // ------------------------------------------------------------
----------------- AXMenuItem
            searchRange = [lineToProcess rangeOfString:@"AXMenuItem"];
            if(searchRange.length){
                bInMenuBar = TRUE;
                menuInfo = (NSMutableString *)@"";
                                                menuInfo   =   (NSMutableString
stringByAppendingString:(NSMutableString      *)[self
extractAXValue:lineToProcess]];
                                                menuInfo   =   (NSMutableString
stringByAppendingString:@" menu item "];
            }
            // ------------------------------------------------------------
----------------- AXButton
            searchRange = [lineToProcess rangeOfString:@"AXButton"];
            if(searchRange.length){
                if(bInDock){
                                                                                    str
*)stringToReturn stringByAppendingString:@" launch "];
                                                                                    str
*)stringToReturn  stringByAppendingString:(NSMutableString  *)[self
extractAXValue:lineToProcess]];
                }
                else{
                                                                                    str
*)stringToReturn stringByAppendingString:@" "];
                                                                                    str
*)stringToReturn  stringByAppendingString:(NSMutableString  *)[self
extractAXValue:lineToProcess]];
                                                                                    str
*)stringToReturn stringByAppendingString:@" button "];
                }
            }
            // ------------------------------------------------------------
----------------
            rowBeingProcessed = rowBeingProcessed + 1;
        }
    }

    // add the menu information if we are in the menu bar
    if(bInMenuBar){
        stringToReturn = (NSMutableString *)@"";
                stringToReturn  =  (NSMutableString  *)[(NSString  *)stringToRetu
stringByAppendingString:menuInfo];
    }

    // return the string we have produced
    currentStringProduced = (NSString *)stringToReturn;
}
/*
    ------------------------------------------------------------------------
-----------------------
    Method Name:           processAXAttributes
    Parameters:            NSString object
    Returns:               None

    -
    ------------------------------------------------------------------------
-----------------------
*/
- (void)processAXAttributes:(NSString *)passedString
{
    // not coded yet
}

/*
    --------------------------------------------------------------------------
-----------------------
    Method Name:           processAXRemaining
    Parameters:            NSString
    Returns:               None

    -
    --------------------------------------------------------------------------
-----------------------
*/
- (void)processAXRemaining:(NSString *)passedString
{
    // not coded yet
}

@end
```

B.5. InspectorWindow.h

```
/*
    --------------------------------------------------------------------------
-----------------------
    File Name:             InspectorWindow.h
    Project:               Parakeet
    Author:                Greg Howell (with some help from Apple sample
code)
    Revision Date:         22/08/2003 (0.2)

    Description:  InspectorWindow object interface.
    --------------------------------------------------------------------------
-----------------------
*/

#import <Cocoa/Cocoa.h>
#import <Appkit/NSAccessibility.h>

/*
    --------------------------------------------------------------------------
-----------------------
    InspectorWindow interface
    --------------------------------------------------------------------------
-----------------------
*/

@interface InspectorWindow : NSPanel
{
    IBOutlet NSTextView * _consoleView;      // the large NSTextView
displaying the current element's info
    IBOutlet NSTextView * _currentApp;       // displays the current
application
    BOOL boolAudibleState;           // if TRUE we are speaking, if FALSE we
are not
}

// instance methods
- (void)setCurrentAppString:(NSMutableString *)currentApp;
- (void)setStartStopState:(BOOL)onoff;
- (BOOL)getStartStopState;
- (void)displayInfoForUIElement:(AXUIElementRef)uiElement;
- (NSString *)stringDescriptionOfUIElement:(AXUIElementRef)inElement;
- (NSString *)stringDescriptionOfCFArray:(NSArray *)inArray;

// instance methods for the interface elements
- (IBAction)stopButtonPressed:(id)sender;
- (IBAction)startButtonPressed:(id)sender;

@end
```

B.6. InspectorWindow.m

```
/*
    ------------------------------------------------------------------------
-----------------------
    File Name:              InspectorWindow.m
    Project:                Parakeet
    Author:                 Greg Howell (with some help from Apple sample
code)
    Revision Date:          22/08/2003 (0.2)

    Description:  InspectorWindow object implementation.
    ------------------------------------------------------------------------
-----------------------
*/

#import <Cocoa/Cocoa.h>
#import <Carbon/Carbon.h>
#import "AppShell.h"
#import "InspectorWindow.h"
#import "Parakeet.h"

@implementation InspectorWindow
/*
    ------------------------------------------------------------------------
-----------------------
    awakeFromNib
    ------------------------------------------------------------------------
-----------------------
*/
- (void)awakeFromNib
{
    // We're using Cocoa's mouseMoved: message to trigger updating
    //[self setAcceptsMouseMovedEvents:true];
}
/*
    ------------------------------------------------------------------------
-----------------------
    setCurrentAppString
    ------------------------------------------------------------------------
-----------------------
*/
- (void)setCurrentAppString:(NSMutableString *)currentApp
{
    [_currentApp setString:currentApp];
    [_currentApp display];
}
/*
    ------------------------------------------------------------------------
-----------------------
    setStartStopState
    ------------------------------------------------------------------------
-----------------------
*/
```

```
- (void)setStartStopState:(BOOL)onoff
{
    // set the global boolAudibleState to onoff, the variable
    // passed to the method
    boolAudibleState = onoff;
}
/*
    -----------------------------------------------------------------------
-----------------------
    getStartStopState
    -----------------------------------------------------------------------
-----------------------
*/
- (BOOL)getStartStopState
{
    return boolAudibleState;
}

/*
    -----------------------------------------------------------------------
-----------------------
    mouseMoved
    -----------------------------------------------------------------------
-----------------------
*/
- (void)mouseMoved:(NSEvent *)theEvent
{
    // Tell AppShell that the mouse moved
    // mouseMoved is used to trigger updating
    [AppShell updateCurrentUIElement];
}
/*
    -----------------------------------------------------------------------
-----------------------
    displayInfoForUIElement
    -----------------------------------------------------------------------
-----------------------
*/
- (void)displayInfoForUIElement:(AXUIElementRef)uiElement
{
    NSString *temp = [[NSString alloc] initWithString:[self
stringDescriptionOfUIElement:uiElement]];

    [_consoleView setString:temp];
    [_consoleView display];
    [AppShell processAXString:(NSMutableString *)temp];
}
/*
    -----------------------------------------------------------------------
-----------------------
    lineageOfUIElement
    -----------------------------------------------------------------------
-----------------------
*/
- (NSArray *)lineageOfUIElement:(AXUIElementRef)element{
    NSArray *lineage = [NSArray array];
    NSString *elementDescr = [AppShell descriptionOfValue:element
beingVerbose:NO];
    AXUIElementRef parent = (AXUIElementRef)[AppShell
valueOfExistingAttribute:kAXParentAttribute ofUIElement:element];

    if (parent != NULL) {
        lineage = [self lineageOfUIElement:parent];
    }
    return [lineage arrayByAddingObject:elementDescr];
}
/*
```

```
    ------------------------------------------------------------------------
-----------------------
    lineageDescriptionOfUIElement
    ------------------------------------------------------------------------
-----------------------
*/
- (NSString *)lineageDescriptionOfUIElement:(AXUIElementRef)element {
    NSMutableString *result = [NSMutableString string];
    NSMutableString *indent = [NSMutableString string];
    NSArray *lineage = [self lineageOfUIElement:element];
    NSString *ancestor;
    NSEnumerator *e = [lineage objectEnumerator];
    while (ancestor = [e nextObject]) {
        [result appendFormat:@"%@%@\n", indent, ancestor];
        [indent appendString:@" "];
    }
    return result;
}
/*
    ------------------------------------------------------------------------
-----------------------
    stringDescriptionOfUIElement
    ------------------------------------------------------------------------
-----------------------
*/
- (NSString *)stringDescriptionOfUIElement:(AXUIElementRef)inElement
{
    NSMutableString * theDescriptionStr = [[NSMutableString new]
autorelease];
    NSArray * theNames;
    CFIndex nameIndex;
    CFIndex numOfNames;

    [theDescriptionStr appendFormat:@"%@", [self
lineageDescriptionOfUIElement:inElement]];

    //
    // Display attributes
    //
    AXUIElementCopyAttributeNames( inElement, (CFArrayRef *)&theNames );
    if (theNames) {

        numOfNames = [theNames count];

        if (numOfNames)
            [theDescriptionStr appendString:@"\nAttributes:\n"];

        for( nameIndex = 0; nameIndex < numOfNames; nameIndex++ ) {

            NSString * theName = NULL;
            id theValue = NULL;
            Boolean theSettableFlag = false;

            // Grab name
            theName = [theNames objectAtIndex:nameIndex];

            // Grab settable field
                  AXUIElementIsAttributeSettable( inElement,
(CFStringRef)theName, &theSettableFlag );

            // Add string
            [theDescriptionStr appendFormat:@"   %@%@:  “%@”\n", theName,
(theSettableFlag?@" (W)":@""), [AppShell descriptionForUIElement:inElement
attribute:theName beingVerbose:false]];

            [theValue release];
        }
        [theNames release];
    }

    //
    // Display actions
    //
         AXUIElementCopyActionNames( inElement, (CFArrayRef *)&theNames );
    if (theNames) {

        numOfNames = [theNames count];

        if (numOfNames)
            [theDescriptionStr appendString:@"\nActions:\n"];

        for( nameIndex = 0; nameIndex < numOfNames; nameIndex++ ) {

            NSString * theName = NULL;
            NSString * theDesc = NULL;

            // Grab name
            theName = [theNames objectAtIndex:nameIndex];

            // Grab description
         AXUIElementCopyActionDescription( inElement, (CFStringRef)theName,
(CFStringRef *)&theDesc );

            // Add string
            [theDescriptionStr appendFormat:@"   %@ - %@\n", theName,
theDesc];

            [theDesc release];
        }

        [theNames release];
    }

    return theDescriptionStr;
}
/*
    -------------------------------------------------------------------------
-----------------------
    stringDescriptionOfCFArray
    -------------------------------------------------------------------------
-----------------------
*/
- (NSString *)stringDescriptionOfCFArray:(NSArray *)inArray
{

    NSMutableString * theDescriptionStr = [[NSMutableString new]
autorelease];
    CFIndex theIndex;
    CFIndex numOfElements = [inArray count];

    [theDescriptionStr appendFormat:@"{"];

    for( theIndex = 0; theIndex < numOfElements; theIndex++ ) {

        id theObject = [inArray objectAtIndex:theIndex];

         if (CFGetTypeID(theObject) == CFDictionaryGetTypeID()) {

            if (theIndex == 0)
                [theDescriptionStr appendFormat:@"(<UI Element: %d>)",
theObject];
            else
                [theDescriptionStr appendFormat:@", (<UI Element: %d>)",
theObject];
        }
        else {
            if (theIndex == 0)
                [theDescriptionStr appendFormat:@"%@", [inArray
objectAtIndex:theIndex]];
            else
                [theDescriptionStr appendFormat:@", %@", [inArray
objectAtIndex:theIndex]];
        }
    }

    [theDescriptionStr appendFormat:@"}"];

    return theDescriptionStr;
}
/*
    -------------------------------------------------------------------------
-----------------------
    close
    -------------------------------------------------------------------------
-----------------------
*/
- (void)close
{
    [super close];
    [NSApp terminate:NULL];
}
/*
    -------------------------------------------------------------------------
-----------------------
    stopButton
    -------------------------------------------------------------------------
-----------------------
*/
- (IBAction)stopButtonPressed:(id)sender
{
    [self setStartStopState:FALSE];
    if(DEBUG_PARAKEET){
        printf("InspectorWindow:stopButtonPressed (STOP)\n");
    }
}
/*
    -------------------------------------------------------------------------
-----------------------
    startButton
    -------------------------------------------------------------------------
-----------------------
*/
- (IBAction)startButtonPressed:(id)sender
{
    [self setStartStopState:TRUE];
    //[StartButton setEnabled:true];
    if(DEBUG_PARAKEET){
        printf("InspectorWindow:startButtonPressed (START)\n");
    }
}
@end
```

B.7. Parakeet.h

```
/*
    -------------------------------------------------------------------------
```

```
-----------------------
    File Name:              Parakeet.h
    Project:                Parakeet
    Author:                 Greg Howell
    Revision Date:          22/08/2003 (0.2)

    Description:            Parakeet header file.
                            Includes header information common to the
application as a whole.
    -------------------------------------------------------------------------
-----------------------
*/

// Setting DEBUG_PARAKEET to TRUE allows Parakeet to print debugging
// information to the run window in Project Builder
// Setting DEBUG_PARAKEET to FALSE turns this feature off
// Just remember to turn it off before you compile a public release
version....
#define DEBUG_PARAKEET TRUE
```

B.8. Parakeet_Prefix.h

```
/*
    -------------------------------------------------------------------------
-----------------------
    File Name:              Parakeet_Prefix.h
    Project:                Parakeet
    Author:                 Greg Howell
    Revision Date:          01/08/2003 (0.1)

    Description:  Prefix header for all source files of the 'Parakeet'
target in the
                        'Parakeet' project. Compiled first (compiles the
Cocoa headers).
    -------------------------------------------------------------------------
-----------------------
*/

#ifdef __OBJC__
    #import <Cocoa/Cocoa.h>
#endif
```

B.9. main.m

```
/*
    -------------------------------------------------------------------------
-----------------------
    File Name:              main.m
    Project:                Parakeet
    Author:                 Greg Howell
    Revision Date:          23/07/2003 (0.1)

    Description:            Standard main function.
                            Generated by Project Builder, documented by Greg
Howell
    -------------------------------------------------------------------------
-----------------------
*/
```

```
#import <Cocoa/Cocoa.h>

int main(int argc, const char *argv[])
{
    // NSApplication loads the GUI stored in the MainMenu.nib file.
    // MainMenu.nib file then loads the AppShell class and the program
starts.
    // awakeFromNib method is then called in the AppShell class
    // (At least I think that is how it works!)
    return NSApplicationMain(argc, argv);
}
```

B.10. InfoPlist.strings

```
/* Localized versions of Info.plist keys */

CFBundleName = "Parakeet";
CFBundleShortVersionString = "Parakeet v 0.2";
CFBundleGetInfoString = "Parakeet v 0.2, Copyright 2003 Greg Howell & Iain
Murray.";
NSHumanReadableCopyright = "Copyright 2003 Greg Howell & Iain Murray.";
```

C. APPENDIX C – CONTENTS OF THE CD-ROM

The CD-ROM attached to this report is compatible with Mac OS X systems. The Project Builder software is required to open the project files for Parakeet. Refer to Appendix A for a listing of the software versions used.

The CD-ROM contains:

- An electronic copy of this report
- The source code and project files for the Parakeet screen-reader project
- A compiled Parakeet application including README file