An Abstract Java Class for Device-Independent Speech Output

Thomas E. O'Neil
Computer Science Department
University of North Dakota
oneil@cs.und.edu

Abstract

The Audio Interface Project at the University of North Dakota is dedicated to the development of a library of software components that facilitate integration of synthesized speech and other audio effects into software application interfaces. Ideally, the software components are device independent. This paper describes an abstract class for Java called AbstractVoice that provides a set of methods to be used for interaction with common text-to-speech devices. The AbstractVoice methods are device independent. Speech applications can be written using the methods of AbstractVoice, and concrete Java classes can be written to implement AbstractVoice for various text-to-speech hardware and software products.

AbstractVoice can be used as a tool in developing audio interfaces for general applications. It also serves as an instructional tool for courses in user interface design to illustrate the use of audio components as an alternative to strictly graphical user interfaces.

Introduction

While graphical presentation of information still dominates the user interfaces for general computing applications, research is continuing on the use of speech and other audio effects. The standard application interface of the future will probably be a mixed-mode interface – one that integrates graphical, audio, and tactile components. The audio components may be a mixture of recorded speech, synthesized speech, music, and various sound effects. This paper describes a tool that will facilitate the incorporation of synthesized speech into interfaces. Specialized hardware devices that synthesize human speech have been available for years, and with the rapid improvement in speed and storage capacity of microcomputers, speech synthesis can now be implemented in software. Speech synthesis devices typically support text-to-speech translation, so that the user simply sends text to the device as if it were a printer. The AbstractVoice component described here is a device-independent speech synthesis object. It provides methods for the use and control of a voice in application interfaces. The collection of methods is based on the common characteristics of text-to-speech devices and the general characteristics of audio devices.

The AbstractVoice object is implemented in Java, as a component of Java's Abstract Windowing Toolkit (AWT). The object-oriented paradigm presented by Java is well suited for development of device independent objects realized as abstract classes. The abstract classes define methods and fields inherited by all concrete classes derived from the abstract class. The implementations of the abstract class methods are found in the concrete classes, not the parent abstract class. Application code can be based on the specification of the abstract class. Thus the AbstractVoice class can be used in the development of voice applications. A concrete implementation of AbstractVoice provides the link between the application software and some specific text-to-speech device. When the application software is ported from one system to another, a new implementation of AbstractVoice may be required, but no changes are required in the application software.

AbstractVoice provides a tool for software development, research, and instruction. It relates to research literature on auditory interfaces as an example of a synthesized speech server in the Mercator architecture described by Elizabeth Mynatt (1994). Mercator is a prototype system for rendering graphical interfaces in audio. The system architecture includes a Sound Manager that communicates with three servers: a non-speech audio server, a spatialized sound server, and a synthesized speech server.

T. V. Raman (1997) developed an auditory computing environment called Emacspeak based on the Emacs text-editor. This system includes a number of techniques for implementing an audio desktop including speech-enabled widgets and arrow-key directional navigational commands. These are combined with facilities that are a standard part of the editor. Audio components in Emacspeak are integrated into a single application, and do not provide a toolkit that can be reused in developing other applications. The device-specific layer is designed for the Dectalk family of synthesizers.

O'Neil (2000) has previously developed auditory output tools that provide general-purpose audio menu systems. The user controls an application by listening to the audio output and entering keystrokes in response. The menu components contain device-specific procedures that would have to be rewritten for different synthesizers. The menu components will eventually be rewritten to employ the device-independent AbstractVoice.

The AbstractVoice class described here is not designed for any particular application or for any specific device. Its functionality is derived from the common characteristics of various speech devices, such as Doubletalk, Accent, and Dectalk. While any concrete implementation of the abstract class can add methods based on the unique characteristics of a particular device, developers can write device-independent voice applications by employing only those methods defined in the abstract class.

The AbstractVoice Class

Since a voice is a particular kind of audio object, the AbstractVoice class is a special case of an AbstractAudioDevice. There are a number of control functions that are common to all audio devices, such as volume and tone settings. For this reason AbstractVoice is defined to be an extension of the AbstractAudioDevice class. Both of these classes use a

```
abstract class AbstractAudioDevice extends Class
{
    private int volumelevel;  private int volumeDefault;
    private int echolevel;    private int echoDefault;
    private int basslevel;    private int bassDefault;
    private int treblelevel;  private int trebleDefault;

    // Methods for setting device parameters
    public abstract void setVolume(int level);
    public abstract void setVolume(float percentage);
    public abstract void increaseVolume();
    public abstract void increaseVolume(float percentage);
    public abstract void decreaseVolume();
    public abstract void decreaseVolume(float percentage);

    //  .
    //  .  Similar methods for setting Echo, Bass, and Treble
    //  .

    // Methods for default settings
    public abstract void setDefaultVolume(int level);
    public abstract void setDefaultVolume(float percentage);
    public abstract void resetVolume();

    //  .
    //  .  Similar methods for Echo, Bass, and Treble defaults
    //  .
```

Figure 1. A partial listing of the AbstractAudioDevice class.

```
abstract class AbstractVoice extends AbstractAudioDevice
{
    static boolean initialized = false;
    private int speedlevel;          private int speedDefault;
    private int pitchlevel;          private int pitch Default;
    private int timbrelevel;         private int timbreDefault;
    private int intonationlevel;     private int intonationDefault;
    private LinkedList wordlist;

    public AbstractVoice()
    {
      if (!initialized)
      {
          initDevice();
          initialized = true;
      }
    }
    // Methods for initializing, interrupting, and restarting
    public abstract void initDevice();
    public abstract void reset();
    public abstract void flush();
    public abstract void pause();
    public abstract void continue();

    //  .
    //  .  Methods for setting speed, pitch, timbre, intonation
    //  .     similar to those in AbstractAudioDevice

    // Methods for sending data to be spoken to the device
    public abstract int say(String phrase);
    public abstract int emphasize(String phrase);
    public abstract int spell(String phrase);
    public abstract int proofRead(String phrase);
    public abstract int pronounce(String phonemes);

    // Methods used to control audio interactions
    public abstract boolean isSpoken(int textID);
    public abstract String lastWordSpoken();
}
```

Figure 2. A partial listing of the AbstractVoice class.

supporting class called DiscreteRange that provides values for settings in any discrete range that is a subrange of the integers. AbstractAudioDevice, AbstractVoice, and DiscreteRange are illustrated in Figures 1, 2, and 3 respectively.

The class AbstractVoice has methods for general device management such as initialization and resetting, methods for setting control parameters such as volume and pitch, methods for sending data to the device, and methods for determining what has or has not yet been spoken by the device.

The AbstractVoice class is designed to allow multiple distinct voices to share the same physical device. Initialization of the device takes place when the first voice is created. A class-wide boolean variable is used to suppress initialization when more voices are

created. Initialization sets any system-level parameters necessary for communication with the device, flushes the device buffer, and sets all control parameters to their factory defaults.

The abstract class provides methods for control of several parameters. Settings for volume, echo (reverberation), bass, and treble are common to most audio devices, whether they produce music, sound effects, or speech. For that reason, the fields and methods related to these parameters are defined in the AbstractAudioDevice class. Additional settings for parameters specific to speech synthesis are defined in the AbstractVoice class. These include speed, pitch, timbre, and intonation. Controlling the speed of the voice is essential for efficiency and intelligibility. Variations in pitch can be used to define distinct voices and to implement emphasis for certain words. Timbre refers to the general quality of the voice. Speech synthesizers give varying levels of control over voice quality. Dectalk has several device parameters dedicated to this, while Doubletalk has only one. Intonation refers to the amount of tone variation that is used in speaking sentences. Text-to-speech devices follow some rules of prosody when speaking sentences. A setting of zero for intonation would result in monotone speech. A high setting might result in exaggerated prosody.

Each control parameter *X* has a similar set of methods: *setX(int level)*, *setX(float percentage), increaseX(int steps), increaseX(float percentage), decreaseX(int steps), decreaseX(float percentage), setDefaultX(int level), setDefaultX(float percentage),* and *resetX( )*. The *set*, *increase*, and *decrease* methods take an integer value or a floating point percentage as a parameter. With the *set* methods, the integer parameter is a fixed value in the control range of *X*, and the float parameter is a percentage of the range. With the *increase* and *decrease* methods, the integer parameter is the number of steps to increment or decrement the setting using the units of the range of *X*, while the float parameter is an increment or decrement expressed as a percentage of the range. A current default value is kept for each control parameter. This may or may not match the factory default. The *resetX( )* method restores control parameter *X* to its current default.

The *reset( )* method for the voice object is similar to the *initDevice( )* method, except that it restores the control parameters to the locally defined defaults, not the factory defaults, and it does not reset the system communication parameters. Also, it is invoked as an instance method, not as a class-wide method.

The *flush( )* method stops speech and empties the device buffer and the local buffer of the voice object. The local buffer is implemented as a linked list of word records. Each word record includes a field to indicate whether or not the word has been spoken by the synthesizer.

The *pause( )* method stops speech and empties the device buffer, but it does not empty the local buffer. The *resume( )* method resets the control parameters to the local defaults and refreshes the device buffer from the local buffer. These methods make it possible to interleave the speech of several voices that share the same synthesizer.

```
class DiscreteRange
{
    int minimum;
    int maximum;
    public DiscreteRange(int min, int max)
    {
      minimum = min;
      maximum = max;
    }
    public int getMinimum()
    {
      return minimum;
    }
    public int getMaximum()
    {
      return maximum;
    }
    public int getMiddle()
    {
      return (minimum+maximum)/2;
    }
    public int getPercentage(float percent)
    {
      return Math.round((float)(maximum - minimum) * percent);
    }
}
```

Figure 3. The DiscreteRange Class

There are five methods that transmit data to the synthesizer. All of these take a single string parameter and return an identifier to the client application. The *say(phrase)* method causes the device to speak the phrase using all the current control settings. The *emphasize(phrase)* method causes the synthesizer to speak the phrase emphatically, that is, with higher pitch and volume. The *spell(phrase)* method causes the synthesizer to spell all the words in the phrase letter-by-letter. Synthesizers typically have a "spell" mode that can be used for this purpose. The *proofRead(phrase)* method speaks the words of the phrase in monotone and explicitly speaks each punctuation mark. Normally, punctuation is used to affect intonation, but it is not explicitly spoken. With the *pronounce(phonemes)* method, the parameter is a string representation of a sequence of phonemes, not a string of text. Most synthesizers provide a mode in which the input to the synthesizer is assumed to be a representation of phonemes. This representation is not necessarily standard.

Finally, AbstractVoice provides two methods that allow client programs to control an audio interaction. Writing a speech application involves some real-time programming. It takes much longer for the synthesizer to speak a buffer-full of data than it takes to fill the buffer. Some applications may require the user to press keys in response to hearing spoken words. AbstractVoice has to provide some way for a client program to know when words are spoken. The *isSpoken(id)* method can be used by the client program to check whether a phrase previously given to the voice has actually been spoken. The parameter to this method is an integer text identifier that was returned by the previously invoked data transmission method. The *lastWordSpoken( )* method can be invoked by a

client program to get a string representation of the word that was most recently spoken by the synthesizer.

Some Implementation Notes for the Doubletalk LT Synthesizer

Much of the work of implementing AbstractVoice is defining the ranges for the control parameters and translating the method calls to control sequences for the physical device. To set the speed for a Doubletalk synthesizer (RC Systems, 1995), it is necessary to send it the character sequence "*n*S", where *n* is a digit in 0..9. The implementing class would contain a statement "speedRange = new DiscreteRange(0, 9)" in its constructor. Client programs would use the methods of DiscreteRange to create parameters for calls to the *setSpeed* methods, e.g. "setSpeed(speedRange.getMaximum( ))". The *setSpeed* method would produce the appropriate string, in this case "9S", and transmit it to the synthesizer.

The Doubletalk LT model is an external synthesizer that connects to a microcomputer through a serial port. The initialization routine has to configure the port appropriately for the synthesizer, since the system defaults for serial ports are intended for external monitors, not synthesizers. Handshaking is always a critical setting for successful communication. On Unix systems, the initialization routine uses the *stty* command to configure the port.

The most challenging task is the implementation of the *isSpoken(id)* and *lastWordSpoken( )* methods. Doubletalk has a mechanism whereby integer markers in the text stream are sent back to the host computer when they are removed from the device buffer. These markers must be transmitted to the synthesizer between words and must be read from the serial port to detect when the surrounding words have been spoken. The implementation must generate identifiers and markers for words and phrases and store them in a local buffer as they are transmitted to the synthesizer. When the implementation class finds a marker coming back from the synthesizer, it updates the local buffer to indicate that the corresponding word or phrase has been spoken.

A full implementation of the Doubletalk synthesizer would require the addition of methods not found in AbstractVoice. Doubletalk, for instance, has a dictionary of words and corresponding phoneme representations that can be modified by the user of the device. AbstractVoice does not have methods that allow modification of a dictionary, but the concrete implementation of AbstractVoice for Doubletalk could include additional methods for this purpose.

Conclusion

The AbstractVoice class cannot hope to encapsulate all the functionality found in all speech synthesizers. It can, however, provide a set of common methods for development of device-independent audio applications. The abstract class described here will continue to evolve as it is used for application development with different devices. As defined

above, is robust enough to facilitate the development of a large collection of audio components that will enhance the use of audio in the design of application interfaces. In the instructional setting, it opens doors to experimentation with multi-modal interfaces in courses on GUI design and human/computer interaction.

References

Mynatt, E. (1994). Auditory presentation of graphical user interfaces. In Kramer, G. (ed.), *Auditory display: sonification, audification, and auditory interfaces* (pp. 533-555). Reading, MA: Addison-Wesley.

O'Neil, T. E. (2000). Adding Some Audio to the Visual Component Library. *Proceedings of the Midwest Instruction and Computing Symposium (MICS '00)*. St. Paul, MN.

RC Systems, Inc. (1995). *Doubletalk PC/LT User's Manual*. Everett, WA.

Raman, T. (1997). *Auditory user interfaces: toward the speaking computer*. Boston: Kluwer Academic Publishers.