INTRODUCTION

When I first thought about writing a book on the processing of sound, I concluded that for this book to be interesting and useful for people, the code presented had to be interactive. Part of this interactivity, of course, is being able to hear the sound as it is processed. It was a given that the user should be able to hear the processed sound in real time or near real time. What I wasn’t sure about was another aspect of interactivity. That was, could or should the user be able to manipulate the sound processing parameters in real time and immediately hear the result? Or, should the usage model be that all changes be implemented in code, a recompile and a rerun being necessary to hear the result?

I realized that while the recode/recompile/rerun approach would be the easiest to implement, it would go against the concept of interactivity. It is pretty hard to judge subtle differences in processed sound if one must run a different or modified application to hear them. In the end, it was a relatively easy decision to make even though it meant I had a lot more work to do.

The decision to offer users interactive control of the sound processing parameters meant that user interfaces would have to be implemented for all processing devices. The next question was, should these user interfaces be built using standard AWT controls (again the easy way out) or should the user interfaces try to simulate to some degree actual front panels for equivalent hardware devices?

Again, I decided that, to make the interactivity as compelling as possible, it would be necessary to implement simulated front panels for all processing devices.
Nothing beats the listening experience of being able to tweak a knob and immediately hear a difference in the sound produced. Having made that decision, I started to look around for a set of controls and indicators that I could use for building user interfaces for audio devices. At that point, Sun’s Java was still new and few widget sets existed with the capabilities I required. The widget sets I did find were either too limited, too buggy, or just too expensive for my use. So, after much deliberation I decided to explore the idea of building my own sets of controls and indicators for building the simulated front panels. After all, I was interested in the new Java bean technology anyway and it seemed suited for this endeavor. The rest, as they say, is history.

So, what kind of controls and indicators would be required for building these simulated front panels? To determine this, I looked closely at real audio hardware devices, including mixing boards, reverb units, compressors/expanders, special effect units, and other devices and came up with the following lists.

### REQUIRED CONTROLS

1. An assortment of buttons and switches including round and square types with labels and a toggle switch. All types of switches need to operate in either push on/push off or momentary on modes. These buttons must be operated using the mouse.

2. A rotary potentiometer (pot) and a linear slide pot for manipulating parameters in real time using the mouse. These pots need to operate in either linear mode or in pseudo audio taper mode.

### REQUIRED INDICATORS

1. Light emitting diodes or LEDs. These simulated LEDs would need to be available in round and square form factors, be available in various sizes, be available in any color, and have various modes of operation, including on/off solid, blinking at adjustable rates, or single pulse.

2. A seven-segment display capable of displaying the characters 0…9. This display element should be available in any size and with any color and a configurable number of digits.

3. Numerous types of meter indicators, including a simulated analog meter with movable needle, an LED bar graph meter, and a VU-type meter implemented using round LEDs.
Additional requirements placed on the design of these controls and indicators include:

1. The controls must be built using the Java 1.1 event model to provide a uniform and easy-to-use interface to other code.

2. These controls and indicators must be built with a common look and feel so they go together well when used on the simulated front panels.

3. These controls and indicators should provide a three-dimensional presentation to make the simulated front panels appear as real as possible (this is harder to do than you might think).

4. These controls and indicators should be written as Java beans whether they are used that way or not. That way there is maximum flexibility in how they are deployed.

In this section of the book, all of the controls and indicator devices mentioned above will be described and implemented. These devices will be used in Part Two to build simulated front panels for the audio processing devices and also in Part Three for the stand-alone audio applications presented there.

A short discussion is presented at the end of this section detailing how these controls and indicators are used in the construction of simulated front panels.
INTRODUCTION

Before we can delve into how all of the various controls and indicators are implemented, we must first provide some background material that will help it all make sense. The choice was made to use the Java 1.1 event model for all controls and indicators presented in this book because it is much cleaner and easier to use than the Java 1.0 event model method for event propagation. This means this code must be run on a version of Java 1.1 or greater. These devices were first implemented using 1.1.7 and later tested using 1.2 (Java2).

Abstract Window Toolkit (AWT) 1.0 had its problems. To paraphrase Sun’s Java documentation, event processing in version 1.0 of the AWT was based upon inheritance. In order for a program to catch and process GUI events, it had to subclass GUI components and override either action() or the handleEvent()2 methods. Returning “true” from one of these methods consumes the event so it is not processed further; otherwise the event is propagated sequentially up the GUI hierarchy until either it is consumed or the root of the hierarchy is reached. While this technique worked fine for small applets with simple interfaces, it did not scale well for larger Java programs for the following reasons:

1. The requirement to subclass a component in order to make any real use of its functionality is cumbersome to developers; subclassing should be reserved for circumstances where components are being extended in some functional or visual way.

2. The inheritance model does not lend itself well to maintaining a clean separation between the application model and the GUI because applica-
tion code must be integrated directly into the subclassed components at some level.

3. Since all event types are filtered through the same methods, the logic to process the different event types (and possibly the event targets in #2 above) is complex and error-prone. It is not uncommon for programs to have perplexing bugs that are the result of returning an incorrect result (true or false) from the handleEvent() method. This becomes an even greater problem as new event types are added to the AWT; if the logic of existing handleEvent() methods isn’t set up to deal properly with unknown types, programs could potentially break in very unpredictable ways.

4. There is no filtering of events. Events are always delivered to components, regardless of whether the components actually handle them or not. This is a general performance problem, particularly with high-frequency type events, such as mouse moves.

5. For many components, the action() method passes a String parameter which is equivalent to either the label of the component (Button, MenuItem) or the item selected (List, Choice). For programs which use the second approach, this often leads to poor coding and unwieldy string-compare logic that doesn’t localize well.

The 1.1 event model, also referred to as the delegation model, was introduced to fix the problems mentioned above and to provide a more robust framework to support more complex Java programs. To this end, the design goals were as follows:

1. Provide a simple and easy-to-learn event model.
2. Support a clean separation between application and GUI code.
3. Facilitate the creation of robust event-handling code which is less error-prone (strong compile-time checking).
4. Provide a design flexible enough to enable varied application models for event flow and propagation.
5. For visual tool builders, enable run-time discovery of the events that a component generates as well as the events it may observe.
6. Support backward binary compatibility with the old model.

In the delegation model, event types are encapsulated in a class hierarchy rooted at java.util.EventObject. An event is propagated from a “source” object to a “listener” object by invoking a method on the listener and passing in the instance of the event subclass which defines the event type generated.

A listener is an object that implements a specific EventListener interface extended from the generic java.util.EventListener. An EventListener interface defines one or more methods which are to be invoked by the event source in response to each spe-
cific event type handled by the interface.

An event source is an object which originates or “fires” events. The source defines the set of events it emits by providing a set of set<EventType>Listener (for single-cast) and/or add<EventType>Listener (for multi-cast) methods which are used to register specific listeners for those events.

In an AWT program, the event source is typically a GUI component and the listener is commonly an “adapter” object which implements the appropriate listener (or set of listeners) for an application to control the flow/handling of events. The listener object could also be another AWT component which implements one or more listener interfaces for the purpose of hooking GUI objects up to each other.

One final note. All controls and indicators presented in this book were written directly on top of the 1.1 version of AWT. An option would have been to write them on top of Sun’s UI toolkit, Swing, but that was not done as these devices were being developed at the same time Swing was evolving. This of course could still be done if one had good reason to do so.

**AUDIO COMPONENTS AS JAVA BEANS**

When I first began to implement the audio controls and indicators for this book, I thought Java beans were the way to go. I liked the idea of standalone little components that could interact with a visual design environment for building larger applications. The idea of connecting event producers to the event consumers visually (and having the design tool write the simple interface code) seemed compelling. So after analyzing what it meant for a component to be a Java bean, I came up with a short list of requirements including:

1. The component had to use the AWT 1.1 event model.
2. The component had to have a zero argument constructor and a sufficient number of setter methods for configuring the component after it was instantiated.
3. For sizing within the Java bean design environment, the component had to provide a getPreferredSize method to tell its environment what size it wanted to be.
4. The component had to use simple, single parameter get and set methods for the various component properties, using a uniform naming convention.
5. The component had to provide special make files that compiled the component’s code and produced a jar file of the code, any icon files, and a manifest describing the jar files content.
In reality, there are more requirements for industrial strength beans than presented in this list but I was not developing a component library for sale as a product, I was developing a library for my own amusement as I had no notion that I would be writing a book on audio any time soon. A real library of audio component beans would possibly need to provide:

1. Multiple icon gif files per component. One for $16 \times 16$ pixel icons and one for $32 \times 32$ pixel icons depending upon the mode the visual environment wants to operate in.

2. BeanInfo support classes for each component that are used to associate icons with the component for displaying within the design environment and to hide the properties of the component that need not be visible to the component user.

3. Special custom property editors that don’t rely on the editors supplied by the Java bean environment for simple data types.

4. Some better design-time vs. run-time differences in behavior. That is, better design-time presentation that would assist in the layout process when used in a graphical Java bean environment.

Be that as it may, I set off down the Java bean path. I coded all controls and indicators to this simple (non-industrial-strength) list of requirements and that is how they are provided to you today. As part of my testing efforts, all controls and indicators have been placed in Sun’s bean box and have been connected together and tested. They all interacted with the bean box environment and with each other as expected. They have not, however, been tested with any of the new visual development environments as I don’t use these tools yet.

While writing these controls and indicators as Java beans was a good learning experience for me, I found that I was not inclined to use them in a visual development environment for two reasons—first the environments available at the time these components were developed were slow and buggy, and second, any time I found a bug in one of these components, I had to build new jar files for importation into the visual environment. This was time-consuming and, as a result, frustrating.

Instead, I use these components as normal UI component classes within my Java code. In other words, I build my UIs with these components using the traditional approach of using layout managers for component placement. I can then use the setter and getter methods coded into the components to configure the controls or indicators for use after the layout manager has placed and sized them. This is really a testament to the bean architecture that components coded as beans can be used as beans or can be used as standalone Java classes. The standalone Java class approach worked so well I changed the make files to compile these components at the same time all of the other Java code is compiled. If I want to produce jar files containing these components (for use as Java beans), I run make specifying the `beanjarmakefile` makefile.
In keeping with the Java bean tradition, almost every visual aspect of controls and indicators is configurable via simple method calls. Table 1.1 shows the various types of UI controls and indicators provided and the types of attributes that can be manipulated for each. The details of each device’s operation and configuration will be described in later chapters. Note: The non-visible controls like the Blinker and DataGen are not shown here even though they too provide simple methods for their configuration. They are not shown because they are not visible in an application that uses them.

<table>
<thead>
<tr>
<th>Control or Indicator</th>
<th>Configurable Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>Various buttons</td>
<td>Width, height, fonts used for labeling, caption, caption placement (top or bottom), sticky or momentary action mode, initial state, whether component displays a highlight or not, panel color, button color, and text color.</td>
</tr>
<tr>
<td>LED display (7-segment variety)</td>
<td>Width, height, number of digits, initial value, fonts used for labeling, caption, caption placement (top or bottom), panel color, LED segment color, LED segment background color, and text color.</td>
</tr>
<tr>
<td>Various LEDs</td>
<td>Radius (for round) or width and height for rectangular LEDs, LED color, panel color, mode (solid or blinking), rate (fast or slow blink), and initial state.</td>
</tr>
<tr>
<td>Various Meters</td>
<td>Width, height, fonts used for labeling, caption, whether or not the meter has labels and if so where they are to be placed, initial value, the number of sections that make up the meter’s display surface, whether meter displays a highlight or not, panel color, needle color, and text color. Additionally, meters have color zones that can be set.</td>
</tr>
<tr>
<td>Various Potentiometers</td>
<td>Width, height, fonts used for labeling, caption, whether or not the pot has labels and if so where they are to be placed, initial value, the number of discrete values a pot has, whether pot displays a highlight or not, panel color, knob color, text color, tic mark color, and scale color.</td>
</tr>
</tbody>
</table>

From this table you can see that the controls and indicators provided are highly configurable.
Painting Controls and Indicators

Much of the effort that goes into making fancy controls and indicators goes into the paint method that gets called to render the control or indicator in a graphics context. Not only does the paint method have to perform the mundane tasks of coloring and labeling the device, it must also draw the device with some degree of realism. To enhance realism, the current vogue is to draw controls and indicators with a spectral highlight (bright white spot), as if the lighting was coming from above and to the left of the device. This lighting model is further enhanced by the appropriate use of shadows to simulate depth in places that would be dark due to absence of light. This combination of light and dark is used to give the devices a three-dimensional look.

As straightforward as this sounds, it is not always easy to do. To make controls and indicators life-like requires not only good programming skills but also artistic talent. I refer to this as the “3D challenge” because it is a challenge for those of us with limited or no artistic skills. You may have noticed in the table above that the use of a highlight is configurable for the components provided. You can turn highlighting on and off to see if it provides the affect you desire in your audio applications. How highlighting is done for each specific component will be dealt with in subsequent chapters. Note that some of the components provided don’t support highlights at this time even though they may have a method of setting the attribute.

Flickering, which is the result of screen updates that happen too fast for a component to keep up with, reduces realism for components because real components do not flicker. To combat this problem, many of the controls and indicators in this book use double-buffering techniques. That is, the image of the component is drawn onto an offscreen buffer, and when the drawing is complete, it is copied in its entirety into the graphics context. This tends to reduce flickering in that only the portion of a control or indicator that needs to be redrawn is redrawn and the static background will only be updated when needed.

Control and Indicator Events

Table 1.2 summarizes the event types used by the various components described in this section of the book. Also shown are the interfaces these classes of components implement, as required for interoperability within the 1.1 event model and to be usable as Java beans.
Table 1.2 Component Event Types

<table>
<thead>
<tr>
<th>Generic Component Type</th>
<th>Implements Interface(s)</th>
<th>Collects for Distribution of Events</th>
<th>Fires</th>
</tr>
</thead>
<tbody>
<tr>
<td>blinker</td>
<td></td>
<td>PropertyChangeListeners</td>
<td>PropertyChangeEvent</td>
</tr>
<tr>
<td>button(s)</td>
<td></td>
<td>ActionListeners</td>
<td>ActionEvent</td>
</tr>
<tr>
<td>datagen</td>
<td>Adjustable</td>
<td>AdjustmentListeners</td>
<td>AdjustmentEvent</td>
</tr>
<tr>
<td>leddisplay</td>
<td></td>
<td>AdjustmentListener</td>
<td></td>
</tr>
<tr>
<td>leds</td>
<td>PropertyChangeListener and ActionListener</td>
<td></td>
<td></td>
</tr>
<tr>
<td>meters</td>
<td></td>
<td>AdjustmentListener</td>
<td></td>
</tr>
<tr>
<td>pots</td>
<td>Adjustable</td>
<td>AdjustmentListeners</td>
<td>AdjustmentEvent</td>
</tr>
</tbody>
</table>

PropertyChangeEvent Data Range Values

As shown in the table above, only the Blinker component (described shortly) produces (fires) PropertyChangeEvent. The only consumer of these events are the LEDs described in the next chapter.

ActionEvent Data Range Values

Only Button components fire ActionEvents and only the LEDs consume these events. The only values that ActionEvents can have in this book is ON or OFF.

AdjustmentEvent Data Range Values

It is very important to understand that all controls, indicators, and data generators developed in this book that utilize AdjustmentEvents were designed to operate over the integer range of values from 0 to 100. Devices that produce AdjustmentEvents use 0 as the minimum value and 100 as the maximum or full-scale value. Devices that consume these events interpret the data the same way.

This means that a potentiometer at the minimum value position (rotated fully counter clockwise) generates (or fires) an AdjustmentEvent with a value of 0 to all of its listeners. At the maximum value position (fully clockwise), the AdjustmentEvent contains a value of 100. On the consumer side, a value of 0 in an AdjustmentEvent will...
not register on a meter, for example. But a value of 100 in an \textit{AdjustmentEvent} will cause the meter to read full scale. Normalization of the range of values of operation makes the interoperability of \textit{AdjustmentEvent} producers and consumers possible.

This range of values was chosen because it was felt that 101 possible values offered enough granularity for our use here and because this range of values maps well into percentage values.

The first two Java bean components, \textit{Blinker} and \textit{DataGen}, are discussed next. These components have no visual aspects but perform important system functions nevertheless. The visual components like \textit{pots} and \textit{LED}s are discussed in subsequent chapters.

\textbf{Blinker Component}

The \textit{Blinker} is an invisible Java bean component that fires a property change event at a regular, specified interval. A \textit{Blinker} is the data source required for running \textit{LED} indicators. Actually, it drives the state machine that each \textit{LED} runs. A \textit{Blinker} fires a \textit{PropertyChangeEvent} named “blink” that alternatively has the values of \texttt{Boolean.TRUE} and \texttt{Boolean.FALSE}. A \textit{Blinker} can be thought of as a square wave generator with a programmable period. The period can be set in the constructor or by using the \texttt{setInterval} method \textit{Blinker} provides. Once started, a \textit{Blinker} runs forever in its own thread. Because a \textit{Blinker} drives \textit{LED} state machines, the shorter the period, the faster the \textit{LED} blinks if in blink mode and the faster the \textit{LED} responds to changes in state in general.

A \textit{Blinker} has methods for adding \textit{PropertyChangeListeners} called \texttt{addPropertyChangeListener} and for removing \textit{PropertyChangeListeners} called \texttt{removePropertyChangeListener}. The \textit{Blinker} fires a \textit{PropertyChangeEvent} to each registered listener. Therefore one \textit{Blinker} can support any number of \textit{LED}s that might be used in an application. The code below illustrates how a \textit{Blinker} is connected to multiple \textit{LED}s.

\begin{verbatim}
// Start a blinker for the LEDs with a period of 250 milliseconds
Blinker blink = new Blinker(250);

RoundLED firstLED = RoundLED();
blink.addPropertyChangeListener(firstLED);

RoundLED secondLED = RoundLED();
blink.addPropertyChangeListener(secondLED);
\end{verbatim}

Note: The \textit{Blinker} runs the \textit{LED} state machine but does not control the state (off/on) or the mode (solid/blink) of the \textit{LED}. State is controlled through various other means that will be discussed when \textit{LED}s are described later in this section. Suffice it
to say that an LED must always be connected to a Blinker whether or not the LED is programmed to blink.

The blinker is contained in the package craigl.beans.blinker and resides in the file Blinker.java.

**DataGen Component**

The data generator class, DataGen, was developed for testing other beans, specifically those beans that are AdjustmentListeners like the various meters and the seven-segment LED display. It is similar to a Blinker in that it fires events periodically and runs forever in its own thread. But instead of firing a PropertyChangeEvent like the Blinker, it fires an AdjustmentEvent. The time between the AdjustmentEvents can be set in the DataGen constructor or by using the setInterval method it provides.

The DataGen class has methods for adding AdjustmentListeners to its internal list called addAdjustmentListener and for removing listeners from its list called removeAdjustmentListener. DataGen fires an AdjustmentEvent to each registered listener on the list. One DataGen device can support any number of AdjustmentEventListener that might be used in an application. The code snippet below illustrates how a DataGen is connected to an AnalogMeter device.

```java
// Instantiate a DataGen data source for test with a period of 200 milliseconds
DataGen dg1 = new DataGen(200);

// Instantiate an AnalogMeter to receive the events from DataGen
AnalogMeter leftAnalogMeter = new AnalogMeter();

// Make the meter a listener to the events produced by DataGen
dg1.addAdjustmentListener(leftAnalogMeter);

After this code is run and the meter is connected to the DataGen source, each AdjustmentEvent fired by DataGen will result in a new value being displayed on the meter. In the case of the AnalogMeter used here, the needle’s position will deflect appropriately to the new value.

The data contained in the AdjustmentEvent produced by DataGen is random. The only constraints placed on the data is that it must be an integer value in the range 0 to 100 as explained previously for all AdjustmentListener devices. The code that generates the random data inside of DataGen is shown below for reference.
/**
 * Fire an adjustment event containing random data
 */
void fireAdjustmentEvent() {
    // Generate a random data value between 0 and 100
    value = (int) (Math.random() * 100);

    // Synchronously notify the listeners so that they are
    // guaranteed to be up-to-date with the Adjustable before
    // it is mutated again.
    AdjustmentEvent e = new AdjustmentEvent(this,
        AdjustmentEvent.ADJUSTMENT_VALUE_CHANGED,
        AdjustmentEvent.TRACK, value);

    // Send it out if there is a listener
    if (adjustmentListener != null)
        adjustmentListener.adjustmentValueChanged(e);
}

DataGen is contained in the package craigl.beans.datagen and resides in the file DataGen.java.
The visual components are discussed in the chapters to follow.

NOTES .............................................................

1 Java AWT: Delegation Event Model from jdk1.2/docs/guide/awt/designspec/events.html
2 In fact, action() and handleEvent() have been deprecated as of version 1.1 of Java