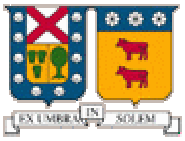
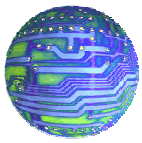


UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE ELECTRÓNICA
Programación de Sistemas



ANEXO

"JAVA MEDIA FRAMEWORK"

Integrantes	Christian Nievas G.
	Sergio Catalán O.
Fecha	**29 Octubre 2003**

Apéndice A : "Arquitectura de Alto Nivel de JMF"

Los elementos principales (clases e interfaces) de la arquitectura de alto nivel que presenta JMF son las siguientes:

- ***Time Model.***

  JMF lleva el tiempo con precisión de nanosegundos. Un punto determinado de tiempo es representado típicamente por un objeto de la clase Time, aunque algunas clases también utilizan la especificación del tiempo en nanosegundos.

  Las clases que utilizan el modelo del tiempo de JMF, implementan la interfaz Clock para llevar el control del tiempo para una secuencia determinada de medios. La interfaz Clock define las operaciones básicas de sincronización que son necesarias para controlar la presentación de los medios.

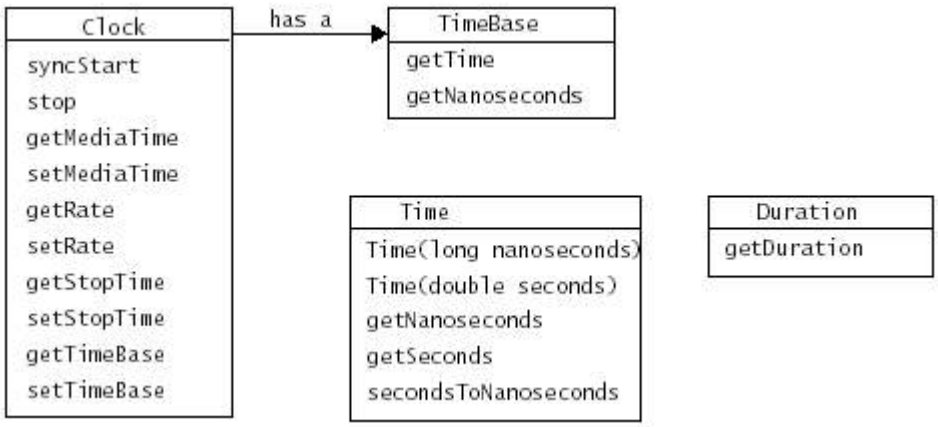


Figura 1.
JMF Time Model.

Para funcionar, Clock utiliza:

TimeBase: Esta interfaz es la encargada de dar los ticks, la fuente del compás del tiempo. Da el tiempo actual (time base).

Duration: La interfaz Duration proporciona una manera de determinar la duración de los elementos de media que son reproducidos por un objeto de Media. Los objetos de Media que exponen una duración implementan esta interfaz.

Time: Clase del paquete javax.media que lleva el tiempo con precisión de nanosegundos.

- ***Managers.***

El JMF API consiste principalmente en interfaces que definen el comportamiento e interacción de los objetos usados para capturar, procesar y presentar los medios basados en el tiempo. Las implementaciones de estas interfaces funcionan dentro de la estructura del JMF. Usando los objetos intermedios llamados managers, JMF hace fácil integrar nuevas implementaciones de interfaces claves que se pueden utilizar junto con las clases existentes.

JMF utiliza cuatro managers:

1) Manager → maneja la construcción de Players, Processors, DataSources y DataSinks. Este nivel de indirección permite que las nuevas implementaciones sean perfectamente integradas con JMF. De la perspectiva del cliente, estos objetos siempre se crean la misma manera si el objeto solicitado está construido de una implementación por default o de encargo.

2) PackageManager → mantiene un registro de paquetes que contiene clases de JMF, tales como Players, Processors, DataSources y DataSinks.

3) CaptureDeviceManager → mantiene un registro de dispositivos de captura disponibles.

4) PlugInManager → mantiene un registro de componentes de procesamiento de JMF plug-in disponibles, tales como Multiplexers, Demultiplexers, Codecs, Effects y Renderers.

- ***Event Model.***

JMF utiliza un mecanismo de reporte de eventos estructurado para mantener los programas de JMF informados sobre el estado actual del sistema de media y permitirles responder a las condiciones de error. Siempre que un objeto de JMF necesite reportar sobre las condiciones actuales, envía un MediaEvent. MediaEvent tiene subclases para identificar muchos tipos determinados de eventos. Estos objetos siguen los modelos establecidos de Java-Beans para los eventos. Para cada tipo de objeto de JMF que pueda fijar MediaEvents, JMF define el listener interface correspondiente.

Los objetos de Controller (tales como Players y Processors) y ciertos objetos de Control tales como GainControl fijan eventos de media.

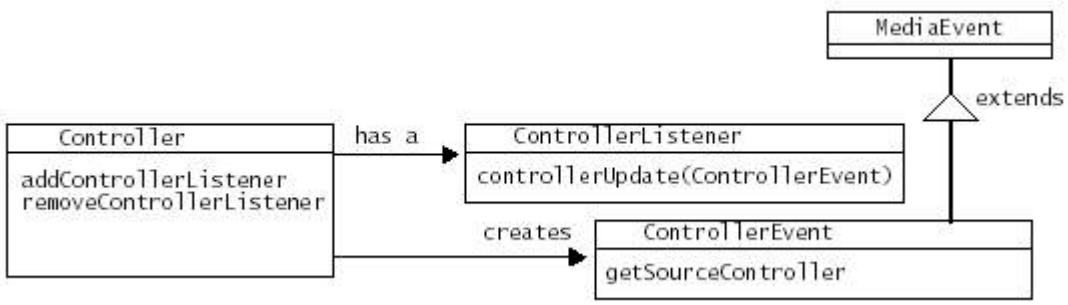


Figura 2.
JMF Event Model.

- ***Data Model.***

Los reproductores de medios de JMF utilizan generalmente DataSources para manejar la transferencia del contenido de medios. Un DataSource encapsula tanto la localización de los medios y el protocolo y el software usado para entregarlo. Una vez obtenido, el DataSource no se puede reutilizar para entregar otros medios. Un DataSource es identificado por un JMF MediaLocator o un URL.

Un DataSource maneja un conjunto de objetos de SourceStream. Una fuente de datos estándar utiliza un arreglo de bytes como unidad de la transferencia. Una fuente de datos de buffer utiliza un objeto de clase Buffer como su unidad de la transferencia.

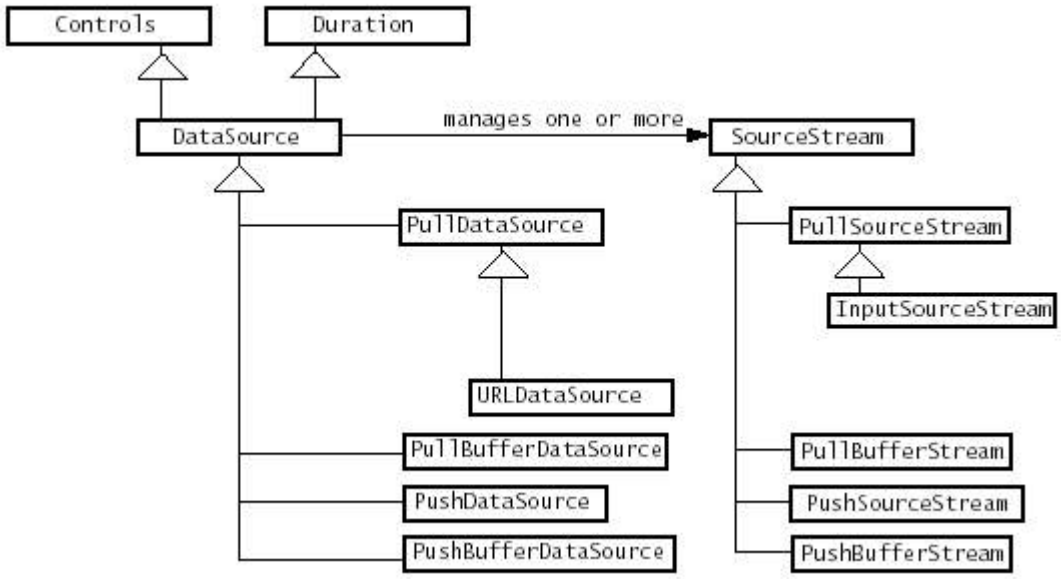


Figura 3.
JMF Data Model.

El formato exacto de un objeto de media es representado por un objeto de la clase Format. El formato en sí mismo no lleva ningún parámetro específico de codificación o información de sincronización, describe el nombre de la codificación del formato y el tipo de datos que el formato requiere.

JMF extiende Format para definir formatos de audio y video específicos.

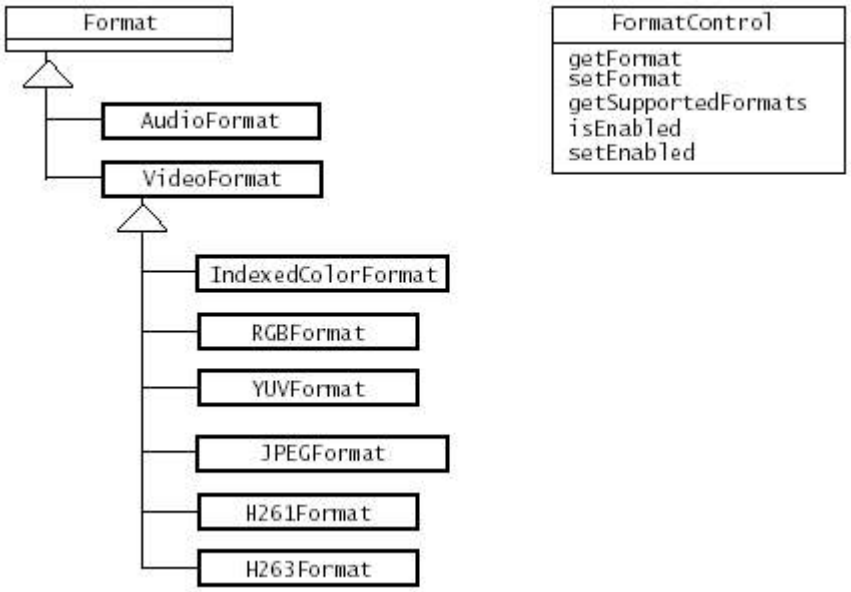


Figura 4.
JMF Media Formats.

- ***Controls.***

El Control de JMF proporciona un mecanismo para asignar y obtener los atributos de un objeto. Un Control proporciona a menudo acceso a un componente correspondiente de interfaz de usuario que habilita el control del usuario sobre los atributos de un objeto. Muchos objetos de JMF exponen Controls, incluyendo objetos de Controller, DataSource, DataSink y JMF plug-ins.

Cualquier objeto de JMF que desee proporcionar acceso a sus objetos correspondientes de Control, puede implementar la interfaz Controls. Controls define los métodos para extraer objetos asociados de Control. DataSource y PlugIn utilizan Controls para proporcionar al acceso a sus objetos de Control.

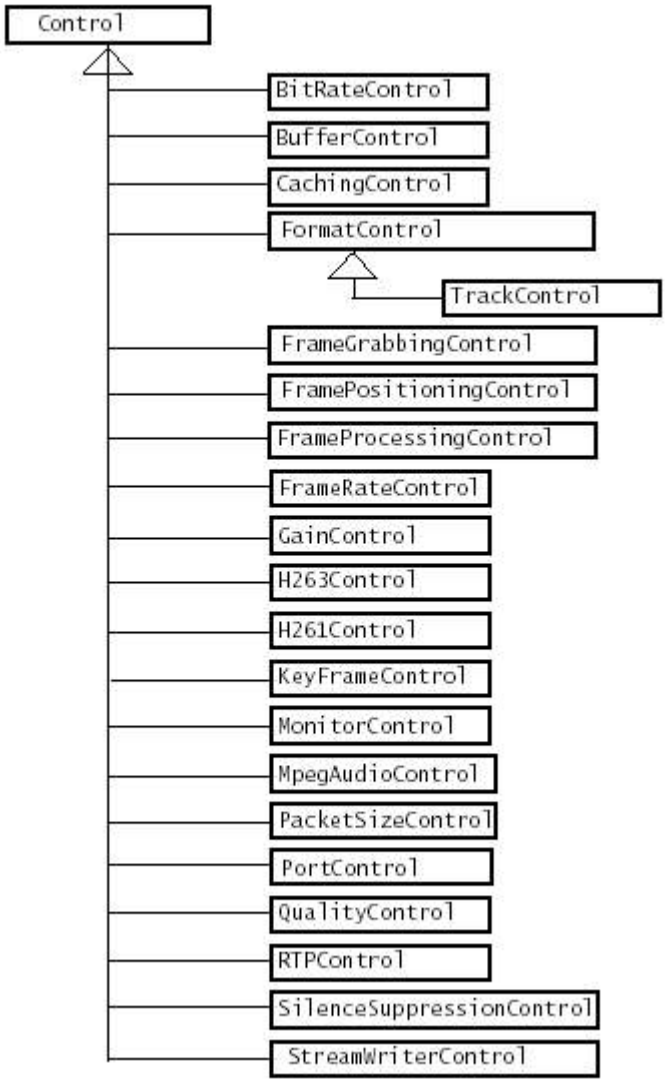


Figura 5.
JMF Controls.

Entre estas interfaces, destacan:

CachingControl: Permite que el progreso de la transferencia sea vigilado y visualizado. Si un Player o Processor puede reportar el progreso de su transferencia, implementa esta interfaz para que el usuario pueda visualizar una barra de progreso.

GainControl: Permite ajustes del volumen del audio tales como fijar el nivel y apagar la salida de un Player o Processor. También utiliza un mecanismo de escucha para los cambios de volumen.

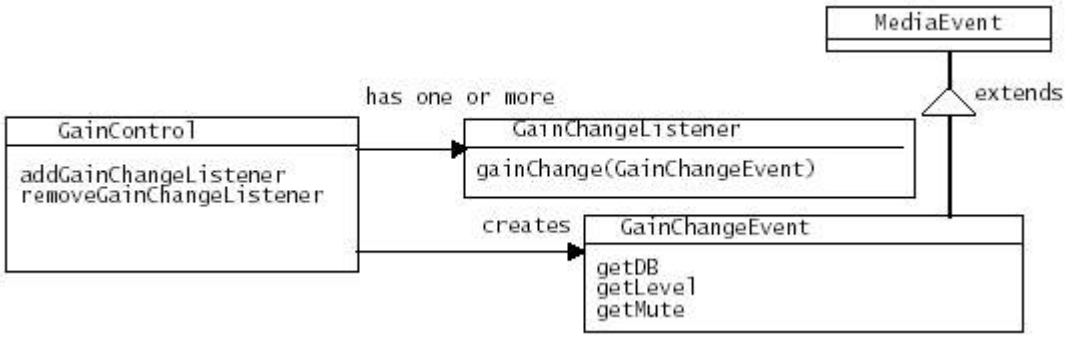


Figura 6.
JMF Gain Control.

Apéndice B: "Ejemplos de Presentación y Captura en JMF".

Ejemplo 1: Reproduciendo un clip MPG en un applet

Invocando PlayerApplet

```
<APPLET CODE=ExampleMedia.PlayerApplet
WIDTH=320 HEIGHT=300>
<PARAM NAME=FILE VALUE="sample2.mpg">
</APPLET>
```

PlayerApplet

```
import java.applet.*;
import java.awt.*;
import java.net.*;
import javax.media.*;
public class PlayerApplet extends Applet implements ControllerListener {
Player player = null;
public void init() {
setLayout(new BorderLayout());
String mediaFile = getParameter("FILE");
try {
URL mediaURL = new URL(getDocumentBase(), mediaFile);
player = Manager.createPlayer(mediaURL);
player.addControllerListener(this);
}
catch (Exception e) {
System.err.println("Got exception "+e);
}
}
public void start() {
player.start();
}
public void stop() {
player.stop();
player.deallocate();
}
public void destroy() {
```

```
player.close();
}
public synchronized void controllerUpdate(ControllerEvent event) {
if (event instanceof RealizeCompleteEvent) {
Component comp;
if ((comp = player.getVisualComponent()) != null)
add ("Center", comp);
if ((comp = player.getControlPanelComponent()) != null)
add ("South", comp);
validate();
}
}
}
```

Inicializando PlayerApplet

```
public void init() {
setLayout(new BorderLayout());
// 1. Get the FILE parameter.
String mediaFile = getParameter(“FILE”);
try {
// 2. Create a URL from the FILE parameter. The URL
// class is defined in java.net.
URL mediaURL = new URL(getDocumentBase(), mediaFile);
// 3. Create a player with the URL object.
player = Manager.createPlayer(mediaURL);
// 4. Add PlayerApplet as a listener on the new player.
player.addControllerListener(this);
}
catch (Exception e) {
System.err.println("Got exception "+e);
}
}
```

Iniciando el reproductor en PlayerApplet

```
public void start() {
player.start();
```

```
}
```

Deteniendo el reproductor en PlayerApplet

```
public void stop() {
player.stop();
player.deallocate();
}
```

Destruyendo el reproductor en PlayerApplet

```
public void destroy() {
player.close();
}
```

Respondiendo a eventos de medios

```
public synchronized void controllerUpdate(ControllerEvent event)
{
if (event instanceof RealizeCompleteEvent) {
Component comp;
if ((comp = player.getVisualComponent()) != null)
add ("Center", comp);
if ((comp = player.getControlPanelComponent()) != null)
add ("South", comp);
validate();
}
}
```

Ejemplo2: Capturando y reproduciendo datos de audio en vivo

Capturando y reproduciendo audio desde un micrófono

```
// Get the CaptureDeviceInfo for the live audio capture device
Vector deviceList = CaptureDeviceManager.getDeviceList(new
AudioFormat("linear", 44100, 16, 2));
if (deviceList.size() > 0)
di = (CaptureDeviceInfo)deviceList.firstElement();
else
// Exit if we can't find a device that does linear, 44100Hz, 16 bit,
// stereo audio.
System.exit(-1);
// Create a Player for the capture device:
try{
Player p = Manager.createPlayer(di.getLocator());
} catch (IOException e) {
} catch (NoPlayerException e) {}
```

Ejemplo3: Escribiendo el audio capturado en un archivo

Escribiendo el audio capturado en un archivo con un DataSink

```
CaptureDeviceInfo di = null;
Processor p = null;
StateHelper sh = null;
Vector deviceList = CaptureDeviceManager.getDeviceList(new
AudioFormat(AudioFormat.LINEAR, 44100, 16, 2));
if (deviceList.size() > 0)
di = (CaptureDeviceInfo)deviceList.firstElement();
else
// Exit if we can't find a device that does linear,
// 44100Hz, 16 bit,
// stereo audio.
System.exit(-1);
try {
p = Manager.createProcessor(di.getLocator());
sh = new StateHelper(p);
} catch (IOException e) {
```

```
System.exit(-1);
} catch (NoProcessorException e) {
System.exit(-1);
}
// Configure the processor
if (!sh.configure(10000))
System.exit(-1);
// Set the output content type and realize the processor
p.setContentDescriptor(new
FileTypeDescriptor(FileTypeDescriptor.WAVE));
if (!sh.realize(10000))
System.exit(-1);
// get the output of the processor
DataSource source = p.getDataOutput();
// create a File protocol MediaLocator with the location of the
// file to which the data is to be written
MediaLocator dest = new MediaLocator("file://foo.wav");
// create a datasink to do the file writing & open the sink to
// make sure we can write to it.
DataSink filewriter = null;
try {
filewriter = Manager.createDataSink(source, dest);
filewriter.open();
} catch (NoDataSinkException e) {
System.exit(-1);
} catch (IOException e) {
System.exit(-1);
} catch (SecurityException e) {
System.exit(-1);
}
// if the Processor implements StreamWriterControl, we can
// call setStreamSizeLimit
// to set a limit on the size of the file that is written.
StreamWriterControl swc = (StreamWriterControl)
p.getControl("javax.media.control.StreamWriterControl");
//set limit to 5MB
if (swc != null)
swc.setStreamSizeLimit(5000000);
// now start the filewriter and processor
try {
filewriter.start();
} catch (IOException e) {
System.exit(-1);
}
```

```
// Capture for 5 seconds
sh.playToEndOfMedia(5000);
sh.close();
// Wait for an EndOfStream from the DataSink and close it...
filewriter.close();
```

Ejemplo4: Codificando el audio capturado

Codificando el audio capturado

```
// Configure the processor
if (!sh.configure(10000))
System.exit(-1);
// Set the output content type
p.setContentDescriptor(new
FileTypeDescriptor(FileTypeDescriptor.WAVE));
// Get the track control objects
TrackControl track[] = p.getTrackControls();
boolean encodingPossible = false;
// Go through the tracks and try to program one of them
// to output ima4 data.
for (int i = 0; i < track.length; i++) {
try {
track[i].setFormat(new AudioFormat(AudioFormat.IMA4_MS));
encodingPossible = true;
} catch (Exception e) {
// cannot convert to ima4
track[i].setEnabled(false);
}
}
if (!encodingPossible) {
sh.close();
System.exit(-1);
}
// Realize the processor
if (!sh.realize(10000))
System.exit(-1);
```

Ejemplo5: Capturando y almacenando datos de audio y video

Creando un Processor de captura con ProcessorModel

```
Format formats[] = new Format[2];
formats[0] = new AudioFormat(AudioFormat.IMA4);
formats[1] = new VideoFormat(VideoFormat.CINEPAK);
FileTypeDescriptor outputType =
new FileTypeDescriptor(FileTypeDescriptor.QUICKTIME);
Processor p = null;
try {
p = Manager.createRealizedProcessor(new ProcessorModel(formats,
outputType));
} catch (IOException e) {
System.exit(-1);
} catch (NoProcessorException e) {
System.exit(-1);
} catch (CannotRealizeException e) {
System.exit(-1);
}
// get the output of the processor
DataSource source = p.getDataOutput();
// create a File protocol MediaLocator with the location
// of the file to
// which bits are to be written
MediaLocator dest = new MediaLocator("file://foo.mov");
// create a datasink to do the file writing & open the
// sink to make sure
// we can write to it.
DataSink filewriter = null;
try {
filewriter = Manager.createDataSink(source, dest);
filewriter.open();
} catch (NoDataSinkException e) {
System.exit(-1);
} catch (IOException e) {
System.exit(-1);
} catch (SecurityException e) {
System.exit(-1);
}
// now start the filewriter and processor
try {
```

```
filewriter.start();
} catch (IOException e) {
System.exit(-1);
}
p.start();
// stop and close the processor when done capturing...
// close the datasink when EndOfStream event is received...
```