Tratamiento multimedia en Java con JMF.

Escrito por:
Carlos Prades del Valle.
Versión 1.0.2.
Febrero de 2001.

Historial del documento.

Versión	Autor	Resumen de la modificación.	Fecha
0.1.0.	CPV	JMF: Visión general. Primera versión no completa.	01-2001
1.0.1.	CPV	Repaso para presentación. Añadir gráficos.	02-2001
1.0.1.	CPV	Cambio de formatos.	02-2001

Autores del documento.

CPV: Carlos Prades del Valle. e-mail: cprades@eresmas.com
Sitio web: http://cprades.eresmas.com/

Resumen.

Este documento es una guía para el programador que quiere integrar multimedia en sus programas en JAVA.

Palabras relacionadas.

Java, programación, sonido, procesado, voz, señal, api, JMF, media, framework.

Índice.

Tabla de ilustraciones.

Glosario.

API: *Application Program Interface*. Es la interfaz proporcionada por un sistema al programador para poder acceder a los servicios de ese sistema.

Applets: Aplicaciones de pequeño tamaño para ser incluidas en documentos.

Array: Es un grupo de datos de un tipo determinado puestos uno a continuación de otro. Coincide con los tipos de datos `array` de Java.

IBM: Empresa norteamericana que ha desarrollado JMF junto a Sun.

Java: Lenguaje de programación orientado a objetos.

JDK: *Java Development Kit*. Es el entorno de desarrollo para Java proporcionado por Sun.

JMF: *Java Media Famework*. API para el tratamiento de datos multimedia en Java.

JSAPI: *Java Speech API.* API para el tratamiento de voz y manejo de sintetizadores y reconocedores en Java.

MIDI: Estándar para el almacenamiento y transporte de música para o desde un sintetizador.

Package: Paquete. Agrupamiento especificado por el programador de clases con características comunes.

RTP: *Real-time Transfer Protocol*. Protocolo de transferencia en tiempo real.

Sun: Empresa norteamericana que desarrolló el lenguaje de programación Java.

Thread: Hilo de ejecución.

Ámbito y alcance del documento.

Este documento pretende dar una visión de la potencia de la potencia de la definición de JFM. Sin llegar a ser suficiente para el programador avanzado, el programador de aplicaciones puede ver lo fácil que resulta añadir multimedia a sus creaciones, así como comprender el modo de funcionamiento del API y la problemática asociada.

La versión de JMF que se trata en este escrito es la 2.0 y el entono de desarrollo utilizado es el JDK 1.3.

Este documento presupone que el lector tiene un conocimiento medio sobre el lenguaje Java siendo recomendable un conocimiento básico del API más común. Por otro lado el lector deberá tener un conocimiento mínimo sobre las características de la señal multimedia.

Convenciones del documento.

En este documento se intentará traducir los términos en inglés siempre que sea posible, exceptuando aquellos términos en inglés que, por el uso común en nuestro idioma, no necesitan tal traducción. Estos vocablos están escritos en *cursiva.* Por otro lado los nombres propios de las compañías comerciales o de las marcas de sus productos también son puestos en cursiva.

La parte de código y los ejemplos están escritos con fuente de letra `Curier`. Cuando se habla de una clase o de un objeto se considera un nombre propio por lo que no se traduce, aún así en determinados casos se especifica la traducción de forma aclaratoria. Los nombre de paquetes y ficheros también se consideran nombres propios, pero al no pertenecer exclusivamente al código se muestran con fuente de letra normal en cursiva (como productos software que son).

1 Introducción.

El API JMF es una definición del interfaz utilizado por Java para la utilización de multimedia y su presentación en *Applets* y aplicaciones. Como definición de interfaz, JMF no tiene por qué proporcionar las clases finales que manejan los datos multimedia, pero dice cómo los proveedores de dichas clases deben encapsularlas y registrarlas en el sistema.

Para el tratamiento de multimedia en casos especiales existen además otras APIs complementarias a JMF y más simples o mejor adaptadas a otros problemas. Para el caso del sonido el Java Sound API permite la captura, el tratamiento, el almacenamiento y la presentación de sonido, tanto en formato MIDI como sonido muestreado, aunque no se pueden tratar todos los formatos existentes de almacenamiento y transmisión de sonido. En el caso particular de la voz, JSAPI (Java Speech API) define la interfaz existente para trabajar con sintetizadores y reconocedores de voz.

Históricamente, JMF nació con el objetivo de poder representar datos multimedia en los programas pero, en la última versión (JMF 2.0.) las capacidades se extienden a todo tipo de tratamiento como la adquisición, procesado y almacenaje de datos multimedia, así como la transmisión y recepción a través de la red mediante el protocolo RTP.

JMF 2.0. ha sido desarrollada por Sun e IBM y no está incluida en las especificaciones de Java 2 o de la máquina virtual, por lo que es necesario obtener el paquete adicional que contiene el JMF para la plataforma que se esté utilizando.

Para abordar el problema, el presente documento realiza una doble aproximación. En un primer paso presenta, desde el punto de vista del usuario, los componentes más importantes de la definición de este API. Con esta visión global del sistema, se puede realizar en un segundo paso un estudio más detallado de cada parte, sus interrelaciones y el modo de funcionar de cada una de ellas. Debido a la extensión de JMF esta parte del documento se centra más en un determinado tipo de objeto (el `Player`) que es el corazón del API.

2 Visión general.

Antes de empezar el estudio de cada uno de los componentes del JMF hay que aclarar algunos conceptos relativos a la transmisión multimedia.

En todo tratamiento que se pueda hacer con los datos multimedia siempre existen tres pasos, estos son: la adquisición de datos (captura desde un dispositivo físico, lectura de un fichero o recepción desde la red), procesado (aplicación de efectos como filtrado o realces, compresión y/o descompresión, conversión entre formatos) y la salida de datos (presentación, almacenamiento en fichero o transmisión a través de la red).

Los datos deberán estar en un formato definido como WAV, QuickTime, MPEG, etc. vengan de la fuente que vengan este tipo siempre debe estar definido.

El flujo de datos (o *media stream*) puede contener varios canales (o *tracks*) de datos, cada uno de los canales pueden tener un tipo de datos distinto, así puede haber un *media stream* con video y audio contenido en dos *tracks* distintos. Un *media stream* queda perfectamente definido por su localización y el protocolo necesario (HTTP, FILE, etc.) para acceder a él. Los *media streams* se pueden dividir en tipos *push* y *pull* cuando el servidor o el cliente respectivamente controlan el flujo de datos. Como ejemplos de los primeros pueden estar RTP, VoD o la captura de sonido desde el micrófono, ejemplos del segundo caso son los protocolos HTTP y FILE o la captura de imágenes de una webcam.

Para prevenir interrupciones, parte de los datos se almacenan en un buffer antes de la presentación, por este motivo la presentación no es inmediata existiendo un cierto tiempo de retardo o latencia.

A continuación se presentan las clases e interfaces más comunes en JMF.

2.1 Clase `DataSource`.

Esta clase se encarga de encapsular la localización, el protocolo de transferencia y el software necesarios para la adquisición. Por lo tanto una vez obtenido un `DataSource` no se puede asociar a otra fuente de datos.

Se construye a partir de un `URL` o un `MediaLocator`. Un `MediaLocator` es como un `URL` pero se puede construir aunque no se encuentre instalado en el sistema el *handler* para el protocolo especificado.

Para la construcción de estos objetos hace falta localizar los datos, por ello su creación por parte del programador sería muy complicada. Para facilitar este extremo, lo que se hace es pedir al sistema el `DataSouce` necesario a partir de la `URL` o el `MediaLocator` a través de la clase `Manager` como se verá más adelante en el punto 2.4.

Figura 2.1: Creación de `DataSource`.

2.2 Clases `Player` y `Processor`.

Un `Player` es un objeto encargado de procesar cierto flujo de datos multimedia y presentarlo en su preciso momento. La forma de presentación y los recursos necesarios dependen del tipo de media que contenga el flujo.

Como en el caso anterior, no se utiliza el constructor para crear objetos de esta clase sino que se llama a una función (`createPlayer()`) que busca en el sistema los componentes adecuados y crea el `Player` que se necesita en cada momento.

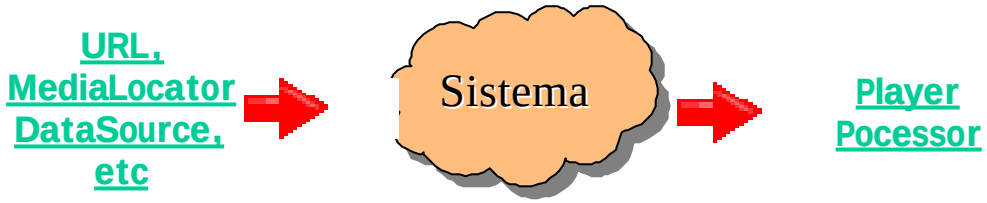

Figura 2.2: Creación de `Player` o `Processor`.

Para obtener el flujo de datos se utiliza un objeto `DataSource` que se "engancha" a la entrada del `Player`. En el caso de que a la función `createPlayer()` le pasemos un `URL` o un `MediaLocator`, esta construye primero el `DataSource` y luego el `Player`.

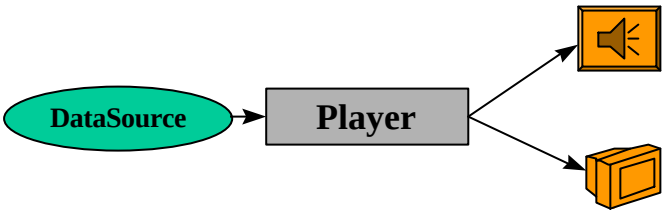

Figura 2.3: Flujo de datos en `Player`.

Como veremos más adelante para utilizar un `Player` es necesario que obtenga recursos del sistema pro lo que pasa por varios estados antes de poder reproducir los datos multimedia, sin embargo, el usuario de JMF no debe preocuparse de esta gestión ya que hay funciones que en un sólo paso crean un `Player` y le asignan recursos. Aquí podemos ver la gran potencia de este interfaz ya que si lo único que queremos es añadir media a un programa basta con crear el objeto a partir de una fuente de datos y reproducir los datos multimedia, sin embargo usuarios más avanzados podrían controlar el proceso de la obtención de recursos, liberándolos cuando no sean necesarios, y, por último, programadores expertos, como veremos más adelante podrán crear los `Player`s a medida.

En cuanto a los objetos `Processor`, estos son hijos de `Player`, por lo que lo contado anteriormente también es aplicable a esta clase de objetos. Las únicas diferencias estriban en que la función que crea el `Processor` es distinta (`createProcessor()`) y que se le añaden nuevas capacidades. Un `Processor` puede presentar los datos multimedia como hace el `Player`, pero lo que lo hace interesante es que la salida puede ser un objeto `DataSource` que proporciona un *media stream* a otros objetos, gracias a esto

se puede cambiar el formato de los datos multimedia o incluso procesarlos dentro del `Processor`. El hecho de que se puedan modificar los datos, y para añadir más funcionalidad, hace que aumente un poco su complejidad al añadir más posibilidades en los estados del objeto, como ya veremos, y, lo más importante, permite modificar las características internas del propio proceso.

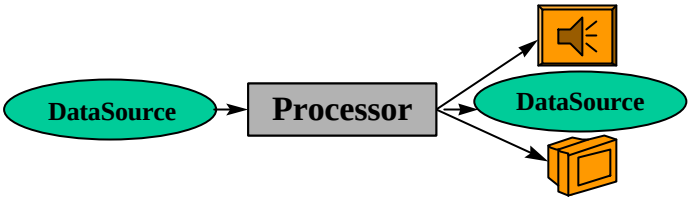

Figura 2.4: Flujo de datos en `Processor`.

2.3 Clase `DataSink`.

El último paso en el proceso de los datos multimedia puede ser el almacenamiento en algún fichero o la transmisión. Para estos casos JMF proporciona la clase `DataSink`. Un objeto de esta clase, como en los casos anteriores, se construye a través de la clase `Manager` usando un `DataSource`. La función de este objeto es obtener el *media stream* y almacenarlo en un fichero local, a través de la red o transmitirlo mediante RTP.

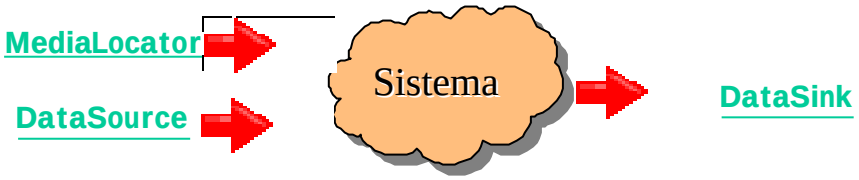

Figura 2.5: Creación de `DataSink`.

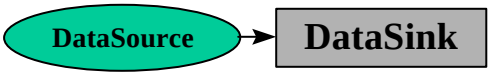

Figura 2.6: Fujo de datos en `DataSink`.

2.4 `Manager` y otras clases de control.

Debido a la complejidad del sistema se presentan algunos problemas como el hecho de que no se puedan utilizar constructores para todos los objetos, o el desconocimiento por parte del programador de las clases existentes en el sistema que heredan de las clases básicas

JMF. Para evitar estos problemas surge la idea de las clases *managers*, que facilitan el uso de los demás tipos de objetos.

La más básica de estas clases es la clase `Manager`, esta es una clase intermedia para facilitar la construcción de los componentes JMF descritos anteriormente (`Player`, `Processor`, `DataSource` y `DataSink`). El tener esta indirección permite crear, desde el punto de vista del usuario, de igual manera componentes ya existentes o nuevas implementaciones. Este *manager* es el único que utiliza el usuario básico pero conviene echar un vistazo a los otros tres *managers* existentes.

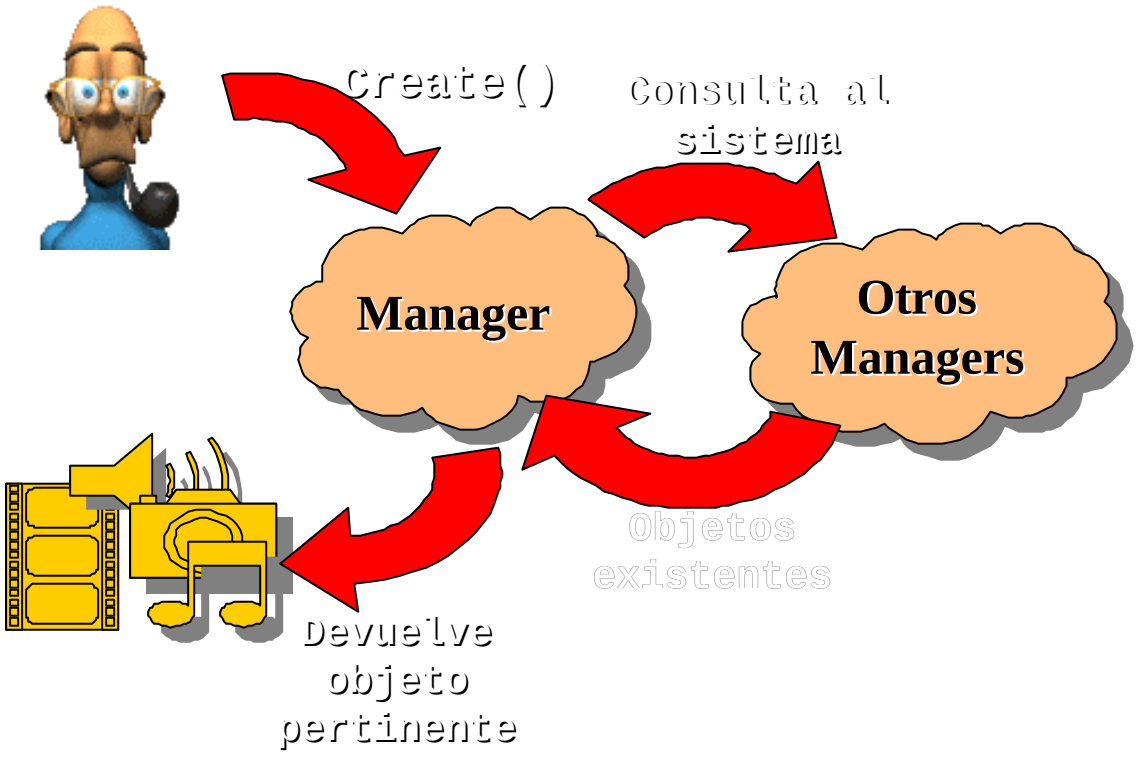

Figura 2.7: Funcionamiento de `Manager`.

Para mantener un registro de los paquetes que pueden tener clases JMF como `Players`, `Processors`, `DataSources` y `DataSinks` se utiliza el `PackageManager`. `CaptureDeviceManager` mantiene registro de los dispositivos de captura de datos multimedia del sistema. Por último existe el `PlugInManager` que lleva la cuenta de los *plug-ins* existentes en el sistema. Un *plug-in* es un componente capaz de procesar los datos, como anticipo se puede decir que son los que componene un Player o un Processor, pueden ser Multiplexer, Demultiplexer, Codec, Effect o Renderer como se verá en el capítulo 3.4.3.

2.5 Otras clases importantes.

Por último que dan por comentar otras clases importantes que, aunque no son las que hacen la parte más importante del sistema son necesarias para definir o controlarlas. Estas clases son, por ejemplo `Format`, `MediaEvent`, `EventListener`, `Controls`, `ProcessorModel`, `MediaLocator`, `MediaError`, `MediaException`, `Control`, `TimeBase`, , etc.

`Format` sirve para definir el tipo de media, existen las clases hijas `AudioFormat` y `VideoFormat` (que a su vez se divide en más) para adaptarse mejor. Estas clases tienen constantes (atributos `final static`) que definen los formatos más comunes. Son necesarias para definir el formato de salida de `Processor`, obtener el formato de un *media stream*, construir un `ProcessorModel`, etc.

`MediaEvent` es la clase padre de todos los eventos lanzados por los componentes JMF, como en otros casos sirven para advertirnos de pequeños errores, anunciar el fin de una actividad (muy útil para comunicar el fin de la presentación de media), comunicar el paso de un estado a otro en un objeto (necesario en `Player` y `Processor`), etc. Estos eventos siguen los patrones *Java Beans* establecidos permitiendo realizar fácilmente una comunicación con otros objetos del sistema (por ejemplo para cerrar la ventana de visualización de un vídeo al terminar). Para los diferentes tipos de eventos existen diferentes `EventListener`.

`ProcessorModel` es una clase capaz de definir internamente un `Processor`, cada uno de los *tracks* y tipos de datos internos, se usa para la creación de un `Processor` con unas características muy definidas o para analizar un `Processor` existente.

Del objeto `MediaLocator` ya se ha hablado anteriormente, define la localización de la fuente de datos y el protocolo a utilizar para obtenerlos, a diferencia de un `URL`, no necesita que el sistema implemente dicho protocolo para la creación del objeto.

`MediaError` y `MediaException` son los padres de todos los errores y las excepciones que los objetos JMF pueden lanzarcuando ocurre un error.

`Control` es una clase diseñada para controlar las características del flujo de datos de un *track* al procesarse, dependiendo del tipo de datos el objeto `Control` puede ofrecer unas u otras características, por ello esta clase se divide en herencia en otras muchas. Hay que tener cuidado y no confundir la clase `Control` con la clase `Controls` que es la definición de los dispositivos que componen un `Player`.

`TimeBase` representa la base de tiempos que tienen todos los objetos `Clock` (como veremos más adelante `Player` y `Processor`) la modificación de la velocidad o el momento de comienzo y final se obtiene respecto a esta base temporal. Para realizar una sincronización entre varios `Player` estos deben compartir sus elementos `TimeBase`.

A parte de estas clases existen otras menos importantes, para un listado completo se puede consultar http://java.sun.com/java-media/jmf/2.1/apidocs/overview-tree.html en la referencia [1].

3 Visión en detalle.

3.1 Jerarquía `DataSource`.

La jerarquía de esta clase se puede ver en la Figura 3.1, no se ha extendido el gráfico de herencia debido a la complejidad que entraña, donde existen más implementaciones de la clase hay en el gráfico flechas discontinuas. Por otra parte, a diferencia de algunos lenguajes como UML, la flecha que indica herencia apunta hacia la clase hija y está rellena (es completamente negra). Además no hay distinciones entre clases e interfaces Java ya que para comprender el sistema es indiferente.

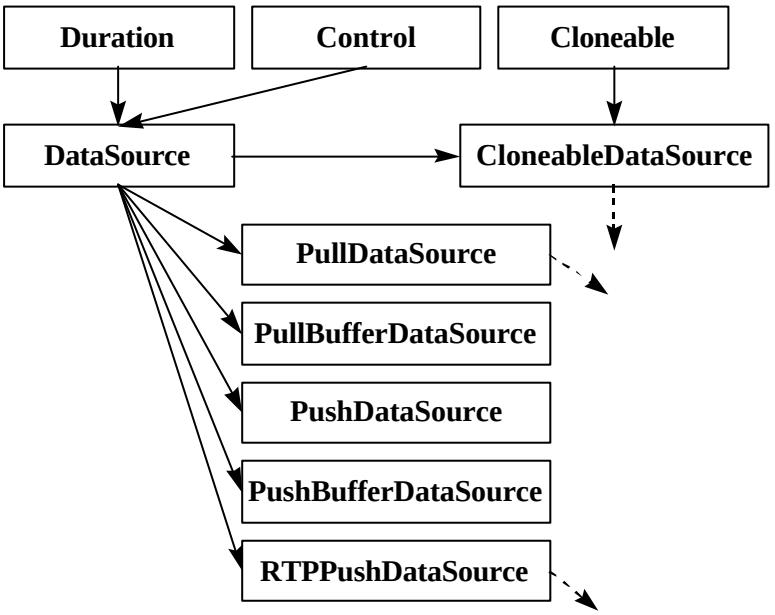

Figura 3.1: Jararquía `DataSource`.

3.1.1 Padres.

El primero de los padres de `DataSource` que observamos es `Controls`, esta interfaz tiene un mecanismo para poder ajustar las características del objeto, `DataSource` usa este interfaz para proporcionar acceso a los objetos `Control` que tenga como atributos.

El otro padre es `Duration`. Un objeto `Duration` tiene una duración definida, a la que se puede acceder con la función `getDuration()`, con esto se indica la duración

total del *media stream*, en algunos casos esta duración puede ser `DURATION_UNKNOWN` (cuando es desconocida) o, en el caso de una transmisión en directo, `DURATION_UNBOUNED`.

3.1.2 Hijos.

En la parte de los descendientes de `DataSource` podemos ver dos partes, por un lado están los que descienden solamente de `DataSource` y por otro `CloneableDataSource` y sus hijos.

En el primero de los casos podemos dividirlos en los de tipo *Push* y los de tipo *Pull* y entre los que tienen *Buffer* y los que no. La diferencia entre *Push* y *Pull* radica en el tipo de media stream que utilizan como se explicó en el capítulo 2. La diferencia entre tener *Buffer* y no tenerlo tiene que ver con el tipo de unidad de datos que se utiliza en al flujo de datos, esta puede ser una unidad simple como es el caso de una imagen o un *buffer* con varias muestras de la señal simple. Hay que señalar la existencia de `RTPPushDataSource` que se utiliza para la transmisión de datos a través de la red utilizando RTP, de esta clase hereda `RPTSocket` que facilita enormemente el uso de este protocolo para la transmisión multimedia en la red.

Respecto a las clases que derivan de `Cloneable` hay que decir que permiten realizar copias del objeto, esto en caso de `DataSource` es de gran utilidad al permitir duplicar el *media stream* para diferentes propósitos. Existen clases derivadas de `CloneableDataSource` equivalentes a las comentadas en el punto anterior.

3.1.3 Implementar nuevos `DataSource`.

Cuando se quiere utilizar nuevos tipos de datos multimedia es necesario implementar nuevas clases `SourceStream` y `DataSource`, esta nueva clase debe heredar de alguna de las ya existentes (`PushDataSource`, `PullDataSource`, `PushBufferDataSource` o `PullBufferDataSource`), si además implementa los interfaces `Seekable` o `Positionable` ofrecerá la posibilidad de saltar a un punto del *media stream* o cambiar de posición en él respectivamente.

Para que el sistema pueda registrarlo y luego pueda ser utilizado por otros objetos, hay cumplir dos condiciones. La primera de ellas es el nombre completo (incluido el *package*) del objeto este debe ser `<prefijo del paquete>.media.protocol.<protocolo>.DataSource`, así para el protocolo *ftp* se puede crear la clase `es.upm.dit.media.protocol.ftp.DataSource`. El otro requisito es registrar el prefijo del paquete en el sistema, para ello basta con obtener un `Vector` con los prefijos de los paquetes utilizando `PackageManager.getProtocolPrefixList()` , añadirle un nuevo `String` con el nombre del prefijo y salvarlo de nuevo.

3.2 Atributos de `DataSource`.

En este caso los atributos de DataSource están representados en la Figura 3.2. Cada línea de las que parten de `DataSource` significa "tiene", el número indica la cantidad de objetos que puede tener de cada tipo.

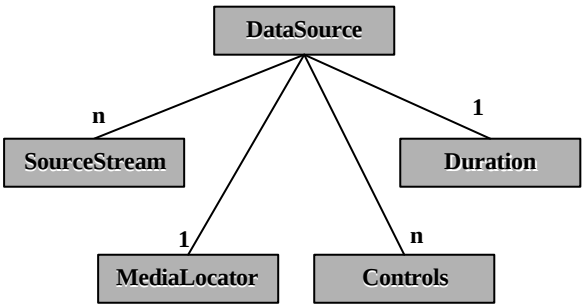

Figura 3.2: Atributos `DataSource`.

Como se puede ver un `DataSource` puede tener varios `SourceStream`, cada uno de ellos es una representación de los propios flujos de datos, dependiendo del tipo de `DataSource` los `SourceStream` pueden ser `PullBufferStream`, `PullSourceStream`, `PushBufferStream` o `PushsourceStream`.

`MediaLocator` ya fue descrito con anterioridad y, como se dijo, indica donde se encuentra la fuente de datos y que protocolo hay que usar para poder obtenerla.

Los objetos Control permiten modificar alguna de las particularidades de la señal, se dividen por herencia en un montón de tipos (20 clases) se puede obtener una lista de los

existentes llamando a la función `getControls()` que implementa todo objeto `Controls` (como los `DataSource`).

Por último de `Duration` ya hemos hablado con anterioridad, encapsula la información sobre la duración del flujo de datos que se esté tratando.

3.3 Jerarquía de `Player`.

La clase `Player` es la más importante de las que se comentan en este documento debido a que es la que trata y/o representa los datos multimedia, por este motivo se presenta su diagrama de herencia antes de tratar los atributos o la parte dinámica del objeto comentando sus posibles estados.

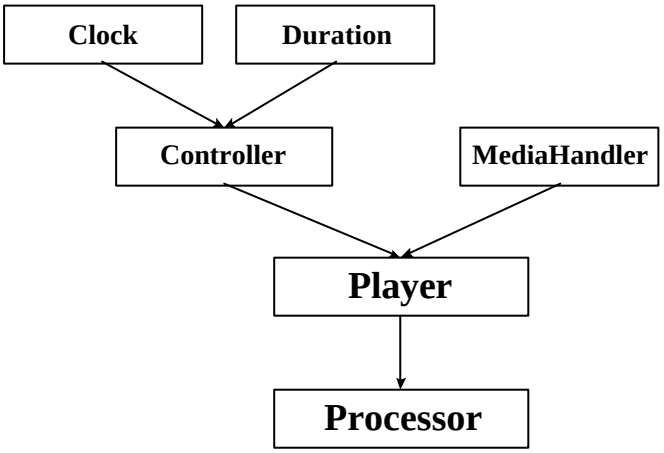

Figura 3.3: Jerarquía `Player` y `Processor`.

La primera clase que se ve en el diagrama de herencia es `Clock` que define las operaciones básicas de temporización y sincronización. Como ya se contó anteriormente esta clase tiene un objeto `TimeBase`, para poder manejarlo se utilizan las funciones `getTimeBase()` y `setTimeBase()` permitiendo la sincronización de varios elementos al obligarles a compartir la base de tiempos a todos ellos. Para controlar la velocidad de reproducción de los datos se usan las funciones `getRate()` y `setRate()`. Finalmente con `getMediaTime()` y `setMediaTime()`, que utilizan objetos `Time`, se puede manejar el tiempo actual de la reproducción.

Como ya se ha comentado el objeto `Duration` encapsula la información temporal del *media stream*. Hay que comentar que `Controller` modifica su funcionalidad y, en lo

referente al tiempo, cuando sincroniza varios `Player` (o `Processor`) se devuelven los datos del flujo más largo.

A continuación `Controller` aúna la funcionalidad de los anteriores, además define el modelo de eventos, estos eventos notificarán cambios en el flujo de datos, transiciones de estado de los objetos o eventos de terminación al acabar la reproducción o producirse un error.

Por otro lado `MediaHandler` define algunas funcionalidades de los elementos que derivan de él como la interfaz con objetos `DataSource` o elementos de creación y destrucción.

Debido a la importancia que tiene la clase `Processor` se comenta más adelante sus características y diferencias con respecto al `Player`.

3.4 Atributos `Player`.

Los elementos de `Player`, como se muestra en Figura 3.4, pueden ser muchos y muy variados debido a complejidad del sistema, a continuación se muestran los más importantes.

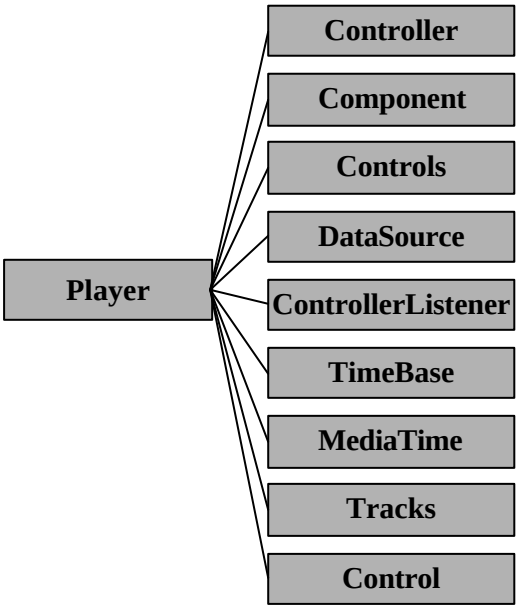

Figura 3.4: Atributos de `Player`.

3.4.1 Otros `Controller`: Sincronización de elementos.

El hecho de poder relacionar un `Player` con otro `Controller` permite que el primero pueda comunicarse con el segundo controlando parte de la funcionalidad, además comparten la base de tiempos permitiendo realizar la sincronización de forma sencilla.

En general para realizar la sincronización hay que compartir la base de tiempos, por ejemplo con una sentencia como `p1.setTimeBase( p2.getTimeBase())`, (cuando veamos el modelo dinámico del objeto veremos que esto sólo se puede realizar en los estados `Realized` o `Prefetched`.) Además tienen que tener un `ControllerListener` que pueda controlar los diversos lanzamientos de eventos y hay que asegurarse de que el `Rate` (velocidad de presentación) es el mismo en todos los `Player`. Para empezar la reproducción hay que llevar a todos los `Controller` al mismo estado del modelo dinámico (por ejemplo a `stop`) y sincronizar las operaciones utilizando las funciones específicas para ello (por ejemplo hay que usar `syncstart()` en lugar de `start()`). Para poder actuar adecuadamente hay que llevar la cuenta del estado en el que se encuentra cada `Controller`.

Usando un `Player` para sincronizar otros `Controller` se simplifica mucho el asunto. Basta con añadir el `Controller` al `Player` con los métodos `addController()` y `removeController()` si deseamos separarlos. A partir de aquí las acciones realizadas en el `Player` pasan a todos sus `Controller`, el `Player` mandará un evento cuando todos los `Controller` que tiene lo han mandado. El tener otros `Controller` hace que alguna característica cambie ligeramente, así la duración o la latencia será la máxima entre la del `Player` y las de los otros `Controller`.

3.4.2 Control temporal: `TimeBase` & `MediaTime`.

Por el hecho de ser de tipo `Clock` todo `Controller`, como los `Player` y los `Processor`, tienen un objeto TimeBase, que se encarga de proporcionar pulsos de reloj a velocidad constante como un cristal de cuarzo en un sistema hardware, y un objeto MediaTime que lleva la cuenta del tiempo de presentación del flujo de datos, de esta forma:

$$MediaTime = MediaStartTime + Rate * (TimeBaseTime - TimeBaseStartTime)$$

3.4.3 Tiempo de presentación: `Duration`.

Como ya hemos visto podemos obtener de `Player` un objeto `Duration` que muestra la duración del flujo de datos multimedia que se procesan.

3.4.4 Tratamiento de la señal: `Controls` y `PlugIn`.

Desde el punto de vista del tratamiento de datos un `Player` está compuesto por flujos de datos que van pasando a través de varios objetos transformándose hasta su consumo en la representación. Desde este punto de vista estos objetos, que son `PlugIn` y `Controls`, son los más importantes del `Player`. En la figura Figura 3.5 se representa la jerarquía de Controls. Aquí trataremos los más importantes.

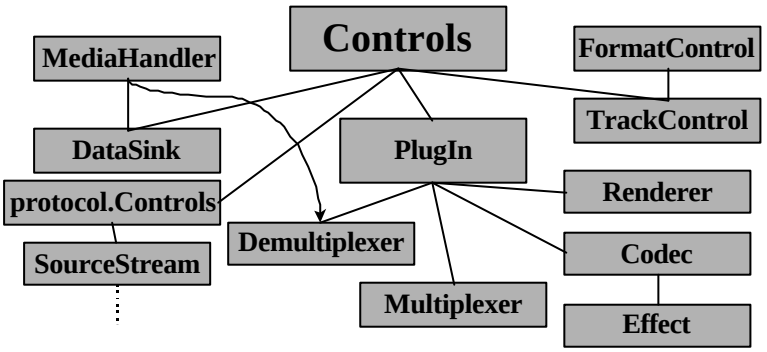

Figura 3.5: Jerarquía `Controls`.

En el estudio del `Player` los `Controls` más importantes son los `PluIn`, estos toman los datos los tratan adecuadamente y los dejan para el siguiente proceso. Entre ellos se encuentran los `Demultiplexer` que toman los datos de un *media stream* y los dividen en diferentes *tracks*, sus principales métodos son `setSource()` donde se le especifica el `DataSource` de donde obtener los datos, `getTracks()` que devuelve los objetos `Track` representantes de los *tracks* extraídos del `DataSource` y `getSupportedInputContendDescriptor()` que devuelve información sobre los formatos de entrada soportados.

Los `Codec` son `PlugIn` que tratan la señal, codifican y/o decodifican cambiando, si procede, su formato. Los `Effect` son un caso específico de los `Codec` que realizan

algún efecto o proceso a la señal. Los métodos de los `Codec` son `getSupportedInputFormats()` y `getSupportedOutputFormats()` que devuelven los formatos soportados como entrada y salida del `Codec` respectivamente, `setInputFormat()` y `setOutputFormat()` sirven para definir el formato de entrada y salida del `Codec`, por último `process()` realiza el proceso en el *track* correspondiente.

En el camino de la señal a través de un `Player` el último paso es un `Renderer`, cada uno de estos objetos es un `PlugIn` que representa el dispositivo físico que muestra el multimedia. Sus métodos son `getSupportedInputFormats()` que devuelve una lista con los formatos de entrada que soporta el objeto, `setInputFormat()` define el formato a presentar y `process()` que presenta los datos del *track* en el que se ha incluido.

Con todos estos objetos el `Player` ya es capaz de presentar los datos multimedia. La disposición de estos objetos es la mostrada en la Figura 3.6.

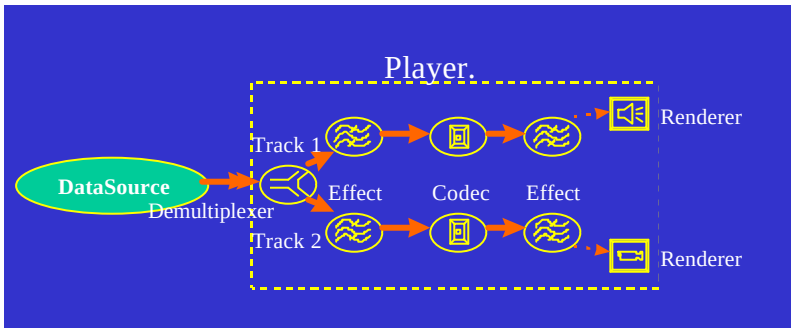

Figura 3.6: Disposición de `Controls` en `Player`.

Además de estos `PlugIns`, en un `Processor` puede haber un cuarto tipo, el `Multiplexer` que, al contrario que el `Demultiplexer`, se encarga, con los datos de varios *tracks*, de obtener el flujo de datos correspondiente que formará el *stream* del `DataSource` a la salida del `Processor`. Los métodos de `Multiplexer` son `getSupportedOutputContentDescriptor()` que advierte de los formatos de salida soportados, `setOutputContentDescriptor()` que selecciona el formato de salida, `process()` convierte los *tracks* individuales en el *stream* de salida. Hay que decir que un `DataSink` también puede realizar la multiplexación por lo que no siempre este elemanto es necesario.

Al añadir esta posibilidad la distribución de los objetos por los que pasan los datos queda como en la Figura 3.7.

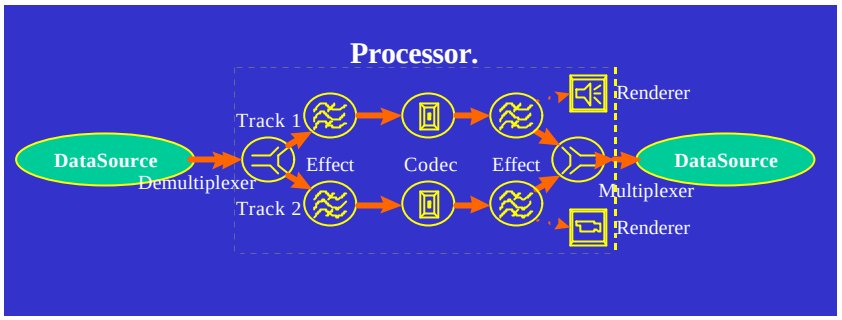

Figura 3.7: Disposición de `Controls` en un `Processor`.

Un `Processor`, además de los `Controls` ya comentados, puede tener uno o varios `TrackControl`, con este objeto se puede configurar cada uno de los caminos internos de la señal. Sus métodos `setFormat()`, que especifica las conversiones en el *track*, `setCodeChain()`, que selecciona el `PlugIn` usado, y `setRenderer()`, que selecciona el "visor", hacen que el usuario pueda configurar totalmente un `Process`. Para obtener el `TrackControl` de cada `Track` sólo hay que invocar al método `getTrackControls()` de `Processor`.

Aunque quedan fuera del `Prayer` y el `Processor`, también hay que señalar que dentro de los `Controls` se encuentran los objetos de clase `DataSink` y `SourceStream`, aunque de estos ya hemos hablado.

3.4.5 Características del proceso: `Control`.

Lo primero que se debe decir es que no hay que confundir la Clase `Controls`, que procesa el flujo de datos, con `Control` que una clase de objetos que cambia o presenta alguna característica del *stream*. Se obtiene del `Player` con `getControls()` que devuelve una lista con los existentes. Estos `Control` pueden formar parte de `DataSources` o `PlugIns`. hay varios tipos de `Control`, los más comunes son `CachingControl` que informa de la cantidad de datos leídos de la fuente (muy útil cuando se obtienen los datos de la red), `GainControl` que puede alterar el volumen del audio, `BitRateControl` permite modificar la velocidad de transmisión, BufferControl muestra el estado del *buffer* y permite realizar operaciones simples en él, etc.

3.4.6 Control visual del proceso: `Component`.

Estos objetos son *AWT* `Component`, lo que significa que se pueden incluir en la ventana de una aplicación o en un `Applet`. Al ser componentes definidos en *AWT* se puede cambiar las propiedades y manipularlos con objetos `LayOut`. Puede haber otros `Component`, incluso los los objetos `Control` pueden tener este tipo de objetos, pero los más corrientes son `ControlPanelComponent` y `VisualComponent`. Para obtener estos objetos se recurre a los métodos del `Player` (o el `Processor`) `getControlPanelComponent()` y `getVisualComponent()`.

`ControlPanelComponet` mostrará al usuario los botones y controles que le sirvan para el manejo de la señal como pueden ser botones de reproducción y pausa, barra de volumen, etc.

`VisualComponent` muestra al usuario la representación de la propia señal o sus características, así en una señal de vídeo este objeto es la ventana donde se visualiza el vídeo, en una seña de audio puede verse la forma de onda, un panel de vúmetros o puede no existir.

3.4.7 Control del objeto: `ControllerListener`.

Como el modelo de eventos sigue los patrones habituales, el `ControllerListener` funciona como en las demás APIs, par ayudar al programador existe la clase `ControllerAdapter` con la declaración trivial de todas las funciones.

Esta interfaz es necesaria para poder llevar el control del estado de un `Player` en el modelo dinámico, esta importancia es incrementada por el hecho de que las funciones que llevan de un estado a otro no son síncronas y los métodos regresan sin haber terminado su función. En el caso de sincronizar "manualmente" varios `Controller` es necesario el control de los eventos que puedan lanzar para tomar las medidas oportunas en el resto de `Controller`. También es importante cuando se almacena un flujo de datos la escucha de los eventos con el fin de cerrar el fichero al terminar. En otra ocasión en que se hace importante esta escucha es cuando se desea liberar recursos si no se están utilizando.

3.4.8 Origen y destino de la señal: `DataSource`.

Todo `MediaHandler` debe tener una entrada con el flujo de datos como ya se vio antes, en un `Player` esto se hace mediante un `DataSource`. En el `Processor`, además, puede haber otro `DataSource` a la salida, para obtener este último se utiliza el método `getDataOutput()`. Al tener `DataSource` un capítulo propio (2.1) y varios apartados (3.1 y 3.2) no se comentará aquí más.

3.5 Modelo dinámico de `Player` y `Processor`.

Aunque es posible utilizar un `Player` y reproducir un fjujo de datos sin conocer el modelo dinámico de este objeto, es conveniente estudiar dicho modelo para poder comprender el funcionamiento del objeto.

Por ser `Clock` un `Player` tiene dos estados `Started` y `Stoped`. Como esto no es suficiente la clase `Controller` ya divide `Stop` en cinco estados que, junto con `Started` son los que tiene `Player`.

Cuando se crea un `Player` con el método `Manager.createPlayer()` este se encuentra en su primer estado: `Unrealized` y lo único que tiene determinado es el `DataSource`. Con el método `Realize()` pasa al estado `Realizing`, en este nuevo estado el `Player` determina que recursos necesita (sólo los determina, no los captura) para la reproducción. Cuando se ha completado con éxito esta acción el `Player` manda un evento (`RealizeCompleteEvent`) pasando al siguiente estado: `Realized`.

En `Realized` ya se conocen los recursos necesarios, en este estado ya se pueden visualizar los controles, acceder a los datos del objeto y modificar sus características.

Para pasar al siguiente estado (`Prefetching`) se llama al método asíncrono (vuelve inmediatamente) `prefetch()`, en este estado se van adquiriendo los distintos recursos necesarios, cuando se termina se envía el evento `PrefetchCompleteEvent` pasando al estado `Prefetched` donde ya se está dispuesto a realizar la reproducción.

Tras llamar al método `start()` se empieza la reproducción entrando en el estado `Started`.

Con el método `Stop` se deja de reproducir y se pasa del estado `Started` a `Prefetched`. Tanto si ocurre así como si se termina la reproducción por otro motivo al pasar de `Started` a `Stop` se lanza un evento (`StopEvent`).

Si estando en los estados `Prefetched` o `Prefetching` se cambian los parámetros del `Player` o se invoca al método `deallocate()` se pasa al estado `Realized`. liberando los recursos que se hayan obtenido (esto es muy útil cuando no se pretende utilizar el objeto en un tiempo elevado). En caso de utilizar `deallocate()` en `Realizing` se pasaría a `Unrealized`.

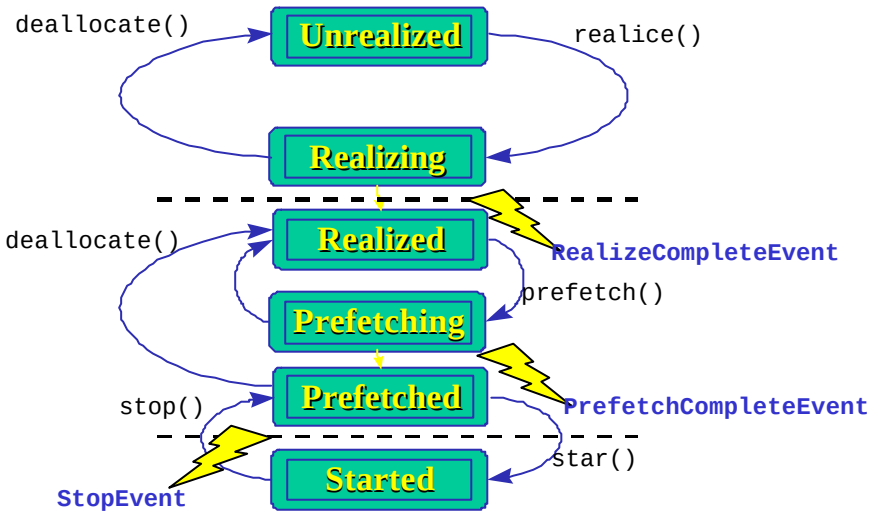

Figura 3.8: Estados de `Player`.

Es importante tener en cuenta que el utilizar funciones en los estados en los que no están permitidas hace que el método lance una excepción.

Por último hay que comentar que no hace falta pasar por todos los estados, así, en una primera aproximación para la reproducción de audio basta con crear un `Player` y llamar al método `start()`, este método hará que el `Player` pase por todos los estados hasta reproducir el sonido. En otros casos (por ejemplo vídeo) se puede crear el `Player` con el método de `Manager` `createRealizedPlayer()` que devuelve el `Player` en el estado `Realized`, se puede obtener aquí los controles y componentes visuales necesarios e invocar, como antes, al método `start()`.

En el caso del `Processor` los estados `Unrealized` y `Realizing` se divide en `Unrealized`, `Configuring`, `Configured` y `Realizing`. El primero de los estados es idéntico al del `Player` pero con la función `configure()` se pasa al estado `Configuring` donde se van conectando los componentes, como esta función es asíncrona, y retorna tras su llamada, para avisar del fin de esta operación el `Processor` lanza un evento (`ConfigureCompleteEvent`) antes de pasar al estado `Configured`, que es el siguiente.

En este estado se pueden obtener con `getTrackControls()` referencias a los distintos controles de cada *track* pudiendo configurar cada uno de ellos (por ejemplo con `setFormat()`). Si se llama en `Unrealized` o en `Configured` al método `realized()` del `Processor` se pasa al estado `Realizing` que termina esta parte del proceso al igual que ocurría en el `Player`.

La secuencia de estados en el `Processor` queda como se muestra en la Figura 3.9.

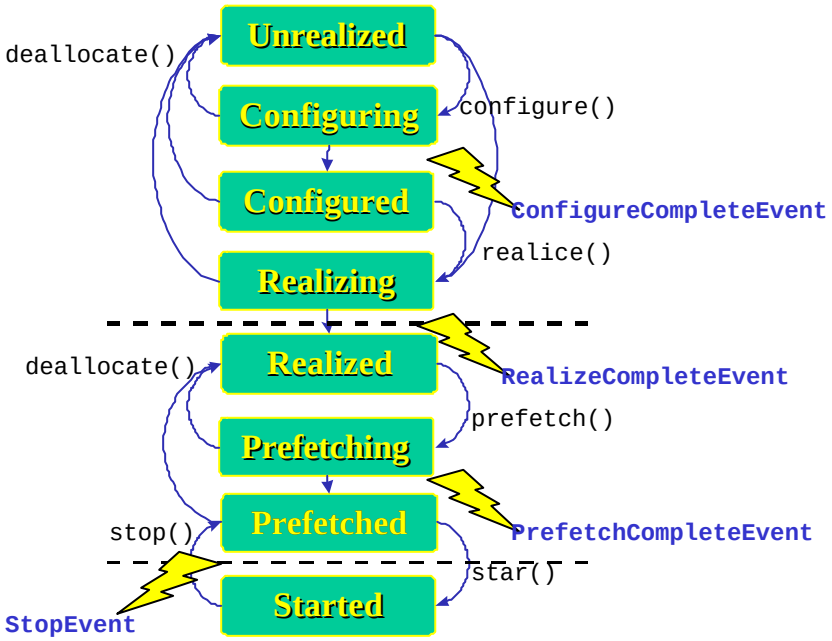

Figura 3.9: Estados del `Processor`.

Otra forma de configurar el `Processor` es utilizar un objeto `ProcessorModel` que define completamente la estructura del `Processor` (en cuanto al flujo de datos se refiere). Este objeto se pasa a la función del creación del *manager* que

construirá el `Processor` con estas características no siendo necesaria la configuración manual.

4 Utilización y control de los objetos.

Ya se ha comentado como se utiliza cada objeto, sin embargo hay opciones adicionales que pueden ser de gran interés, por otro lado en este capítulo se intentará ver desde un punto de vista más práctico algunas de las funciones comentadas.

4.1 Reproducción de multimedia.

En este caso lo primero es crear el `Player`, aunque se puede hacer con un `DataSource` lo más simple es utilizar el `URL` o `MediaLocator` directamente. Si no se utiliza `createRealizedPlayer()` podemos incluir un `ControllerListener` en el `Player` y llamar a `realize()`. Cuando el método `controllerUpdate()` de `ControllerListener` reciba un evento `RealizeCompleteEvent` se pueden obtener los componentes `VisualComponet` y `ControlPanelComponent` (con las funciones `getVisualComponet` y `getControlPanelComponent()`), asegurarse que los componentes existen y añadirlos en la ventana principal de la aplicación.

Si fuese necesario se puede comprobar si existen otros controles intentando obtenerlos con los métodos `getGainControl()`, `getCachingControl()`, etc. o bien consiguiendo una lista de controles con `getControls()`. A cada uno de esos controles podemos "pedirle" su componente visual con `getControlComponent()` y presentarlo si existiese.

Para fijar otras características de la reproducción se puede usar el método `setRate()` al que se le pasa un `float` con la velocidad relativa a la que queremos la presentación (si el número es negativo irá hacia atrás en el tiempo, si es 0 permanecerá en pausa, si es positivo y menor que 1 irá a "cámara lenta", si es 1 la reproducción será normal y si es mayor irá más deprisa), hay que comentar que los `Player` no tienen por qué soportar cualquier número en este método; con `setMediaTime()` se puede modificar la posición inicial de la reproducción; etc.

4.2 Procesado de multimedia.

En el caso de la configuración del `Processor` en el estado `Configured` se pueden utilizar funciones como `setOutputContentDescriptor()` que define el formato de salida del `Processor` o, si se quiere modificar "a mano" el `Processor` pedir al `PlugInManager` una lista con los `PlugIn` disponibles mediante `getPlugInList()` y en cada `TrackControl` utilizar `setCodecChain()` para especificar el *plug-in* del *track* o `setFormat()` para especificar el formato de salida del *track*.

4.3 Captura.

Aquí se necesita la ayuda del *manager* `CaptureDeviceManager` para localizar el dispositivo. Para ello, pasando un objeto `Format` a la función de este *manager* `getDeviceList()`, se obtiene un vector con la información de los dispositivos existentes (encapsulada en objetos `DeviceInfo`) que pueden capturar señal y mostrarla en el formato deseado, se selecciona de esta lista el objeto deseado o bien, si conocemos el dispositivo, podemos obtener su información con `getDevice()` del mismo *manager*. De una forma u otra podemos llamar al método `getLocator()` del objeto seleccionado obteniendo un `MediaLocator` útil para la creación de un `DataSource`, un `Player` o un `Processor`. A partir de aquí será el `MediaHandler` el que maneje el flujo de datos capturados.

4.4 Almacenamiento.

Es necesario crear un `URL` o `MediaLocator` del destino, por supuesto también es necesario el `DataSource` que servirá de fuente. Con estos dos objetos podemos llamar a `createDataSink()` de `Manager` obteniendo nuestro objeto destino. Ahora basta con abrirlo mediante `open()` y , con `start()`, empezar la operación de almacenamiento. En el caso de que el `DataSource` sea la salida de un `Processor` es necesario, para no perder información, llamar antes a la función `start()` del `DataSink` que a la función

`start()` del `Processor` (el cual debe estar en el estado `Started` para poder sacar datos hacia el `DataSource` de salida y, por lo tanto, hacia el `DataSink`).

Cuando el flujo de datos termina el `DataSink` envía un evento `EndOfStreamEvent` que será necesario atender para destruir el `DataSink` utilizando su método `close()`. En el caso de una captura de datos multimedia, al llamar a los métodos `stop()` y `close()` del `Processor` también se lanza el evento. En el supuesto de saber cuando queremos terminar el almacenamiento no es necesario la implementación del `Listener`.

5 Otras posibilidades.

A parte de lo comentado JMF tiene otras posibilidades que se escapan del ámbito del documento, en este capítulo se resume algunas de estas características de la API JMF.

5.1 Herencia.

El hecho de poder heredar de las clases definidas en JMF va a permitir que podamos ampliar las posibilidades del sistema o personalizar los objetos.

En el caso de heredar de `PlugIn` podremos realizar un procesado propio de la señal o implementar nuevos protocolos y formatos para el manejo de nuevos tipos de ficheros, *streams*, `DataSource`, etc. además nos permitirán añadir objetos `Control` aumentando su funcionalidad. Para heredar de `PlugIn`, dependiendo del tipo que sea (`Demultiplexer`, `Codec` o `Effect`, `Multiplexer` o `Renderer`) hay que implementar una serie de funciones (ver 3.4.4) y después registrarla con ayuda de la clase `PlugInManager`, para ello hay que usar el método `addPlugIn()`, en caso de querer eliminar del registro algún elemento basta con llamar a `remove()` de la misma clase, para que los cambios sean perennes en el sistema hay que llamar a `commit()` que graba el resultado.

Para implementar nuevos `DataSource` hay que heredar de alguno de los hijos de esta clase (Figura 3.1), además hay que implementar el correspondiente `SourceStream`. Esto permitirá implementar nuevos protocolos de adquisición de datos y manejar nuevos tipos multimedia. El nombre de las clases tiene que ser estándar para poder registrarse en el sistema, con un prefijo de paquete, seguido de `media.protocol`, el nombre del protocolo que implementa y el nombre de la clase que será `DataSource` como se indicó en 3.1.3, en el caso de los `SourceStream` están situados en un escalón más arriba en la jerarquía de paquetes (`<prefijo de protocolo>.media.protocol`) y no tienen un nombre concreto.

Estas son las clases más útiles ya que permiten añadir funcionalidad a JMF, implementar `Player`, `Component` o `Control` permitirán variar y personalizar JMF.

5.2 Objeto `MediaPlayerBean`.

`MediaPlayerBeam` es un *Java Bean* que encapsula JMF. Este es fácil de construir e implementa funciones como `setMediaLocator()`, que especifica la fuente del *stream*, y `start()` y `stop()` para la reproducción además de las correspondientes por ser *Java Bean*. Esto permite incluir este tipo de objetos en otros módulos, incluso utilizarlos por programadores que no conozcan JMF.

5.3 RTP.

Hay que comentar la posibilidad de transmisión y recepción de *streams* por la red utilizando el protocolo RTP (*Real-time Transfer Protocol*). Al ser transmisión en tiempo real este protocolo se apoya en UDP y permite realizar "*unicast*" cuando se realiza transmisión punto a punto o "*multicast*" cuando se realiza una transmisión a una red de distribución. Entre los servicios ofrecidos por RTP se encuentra la identificación del tipo de datos, la ordenación y sincronización de los datos (incluso cuando se reciben de varias fuentes), proporcionan control y monitorización de la transmisión, etc. JMF permite transmisión, reproducción y almacenamiento de este tipo de datos. Para ello existen objetos `SesionManager` que controlan toda la sesión, existe un modelo de eventos adicional, distintos flujos de datos, nuevas clases derivadas de `Format`, reproductores especiales, `DataSink` para la transmisión, `RTPSocket` para la comunicación, etc.

6 Resumen y conclusiones.

A modo se resumen se puede decir que JMF es una API que permite, de forma muy simple, añadir multimedia a los programas Java. A su vez permite el tratamiento, la captura, el almacenamiento y la transmisión de este tipo de datos. El corazón de esta interfaz es el `Player` que permite la presentación de multimedia y, su hijo, el `Processor` que permite el tratamiento de la señal. Otros objetos importantes son los `PlugIn` que componen los `Player` y `Processor` encargandose de dividir, procesar, unir y presentar la señal. Desde el punto de vista de la señal quedan por comentar `DataSink` y `DataSource` que se encargan de enviar y obtener el flujo de datos de ficheros, URLs, etc.

Desde el punto de vista del sistema son muy importantes las clases *manager* que permiten la reutilización de los componentes de los componentes desarrollados una vez por cualquier usuario, además tienen importancia el modelo de eventos con los `MediaEvent` y los correspondientes `Listener` para controlar el estado del sistema y el estado de cada objeto.

Como hemos visto la interfaz está muy definida no dejándole al usuario o al desarrollador libertad para la toma de decisiones, esto hace que pueda haber un gran entendimiento entre unos y otros. Este hecho unido a la posibilidad de desarrollar parte del sistema, registrarlo y poder utilizarlo por cualquier usuario sin la necesidad de conocer las clases implementadas por el distribuidor permiten facilitar mucho el trabajo.

Otro punto importante de esta interfaz es que permite utilizarla desde varios puntos de vista, así un usuario básico puede no saber absolutamente nada del funcionamiento de multimedia y muy poco de JMF, sin embargo ser capaz de incluir elementos multimedia en sus programas utilizando los objetos básicos de este sistema. Un usuario experto tiene un punto de vista intermedio y puede llevar el control de los objetos JMF que cree así como de los estados del sistema, permitiendo realizar una programación óptima con el control de los recursos del sistema, presentación de los objetos más importantes en cada caso, etc. a su vez puede ser capaz de heredar de algunos objetos para personalizar la aplicación a sus gusto. Por último el punto de vista más avanzado corresponde al desarrollador que tiene que

implementar tanto `PlugIn`, `Player`, `DataSouce`, etc. este tiene que tener en cuenta cómo funciona el sistema, para implementar objetos de captura, y a veces `Codec`, tendrá que tener en cuenta el hardware del sistema y el funcionamiento de otros paquetes Java.

Hay que comentar como otro aspecto importante de JMF que es in interfaz y no una implementación de todos los codificadores, capturadores, etc. ahora le corresponde a los fabricantes de hardware hacer su parte para poder capturar señal y que algunos desarrolladores realicen codificadores para los diversos formatos que vayan apareciendo, si finalmente se consolida como un estándar no habrá ningún problema, pero si no ocurre así los desarrolladores de aplicaciones estarán muy limitados para aprovechar la potencia de este intefaz.

Otra desventaja es que el API JMF no es parte de la MV Java estándar y para poder utilizarlo es necesario instalar en el sistema del usuario este añadido, esto representa un grave problema en algunos casos ya que no puedes pedirle a alguien que para la correcta visualización de un pequeño *Applet* se instale un programa de tamaño mucho mayor que la propia página por la que se transita "de paso", por otro lado el hecho de pedir la instalación de un programa hace que se pierda la confianza en la seguridad del *Applet* en cuestión.

Por último comentar otra de las desventajas de este sistema, debido a su complejidad no puede estar escrito completamente en Java por lo que no vamos a poder desarrollar aplicaciones para cualquier plataforma si antes no se ha desarrollado este API para ella.

Por todo esto esta interfaz se perfila como uno de los hitos del desarrollo de la programación multimedia, sin embargo todavía parece demasiado pronto para hacer una valoración justa de su impacto en el mundo comercial.

Bibliografía.

[1] Java Media Framework Home Page http://java.sun.com/products/java-media/jmf/.

[2] IBM Media Software Home Page. http://www.software.ibm.com/net.media/.

[3] Java Sound Home Page. Diciembre 2000. http://java.sun.com/products/java-media/sound/index.html.

[4] Página sobre `Player` http://java.sun.com/marketing/collateral/jmf.html.

[5] Java 2. Curse de Programación. Fco. Javier Ceballos. Edi. Ra-Ma.

[6] Gifmanía. http://gifmanía.com/ y http://www.gifmania.com/.

Aviso legal.