

## 8. Digitale Filmverarbeitung

- 8.1 Klassische Filmtechnik
- 8.2 Analoge TV- und Videotechnik
  - TV-Technik
  - Videoaufnahme- und Speichertechnik
- 8.3 Digitale Videotechnik
- 8.4 Digitale Videoproduktion
- 8.5 Software zur Videoverarbeitung (Fortsetzung) 
  - (Bsp. Java Media Framework)

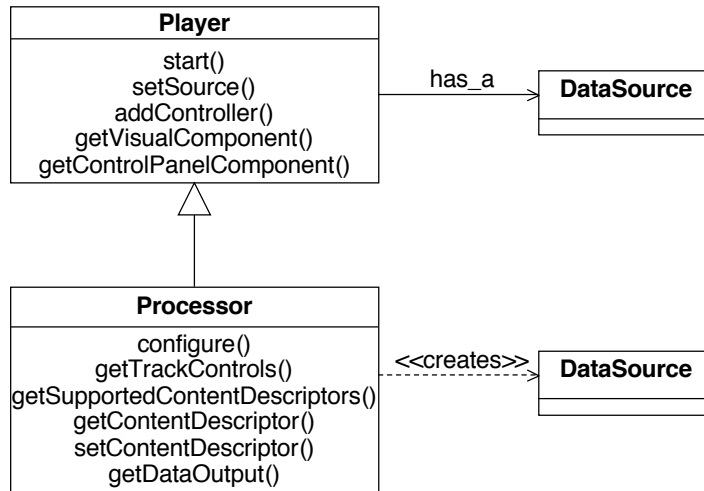
### Literatur:

H. M. Eidenberger, R. Divotkey: Medienverarbeitung in Java.  
dpunkt.Verlag 2004

## Processor

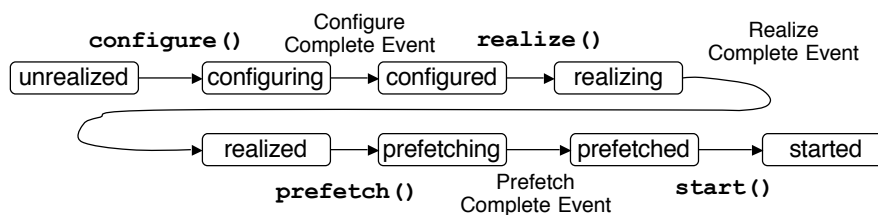
- **Processor** ist die Abstraktion einer medienverarbeitenden Einheit
- Funktionsmöglichkeiten:
  - Nimmt Datenquellen als Eingabe
  - Führt beliebige benutzerdefinierte Transformationen aus
  - Liefert bearbeitete Daten ab
    - » Auf Ausgabegerät (Rendering, analog Player)
    - » Als **DataSource**
- Vom **Processor** gelieferte **DataSource** kann weiterverarbeitet werden
  - In einem weiteren **Processor**
  - In einer **DataSink** (z.B. Speicherung in einer Datei)
- Wichtigster Unterschied zu **Player**:
  - Separate Bearbeitung verschiedener Tracks der Quelle

## Processor vs. Player



UML

## Zustandsmodell von Processor



- **Configuring:**
  - Die Eingabe wird auf die enthaltenen Medien (Spuren, *tracks*) analysiert
- **Configured:**
  - Bearbeitung für die einzelnen Spuren kann separat definiert werden
- **Realizing:**
  - Wie beim Player: Verarbeitungskette wird abhängig von den konkreten Mediendaten bereitgestellt
- *(alles andere wie beim Player)*

## DataSource

- Gemeinsame Abstraktion für Medienzugriffsmethoden
- „Pull“ = passives Medium, „push“ = aktives Medium
- Verpackungsart: einfache Datenstrukturen oder durch **Buffer**-Objekte
- Vier Typen:
  - **PullDataSource**
  - **PullBufferDataSource**
  - **PushDataSource**
  - **PushBufferDataSource**
- Schnittstellen, die zusätzlich von einer **DataSource** implementiert werden können, z.B.:
  - **Positionable** (positionierbar)
  - **RateConfigurable** (positionierbar)
  - **CaptureDevice** (Aufnahmegerät)

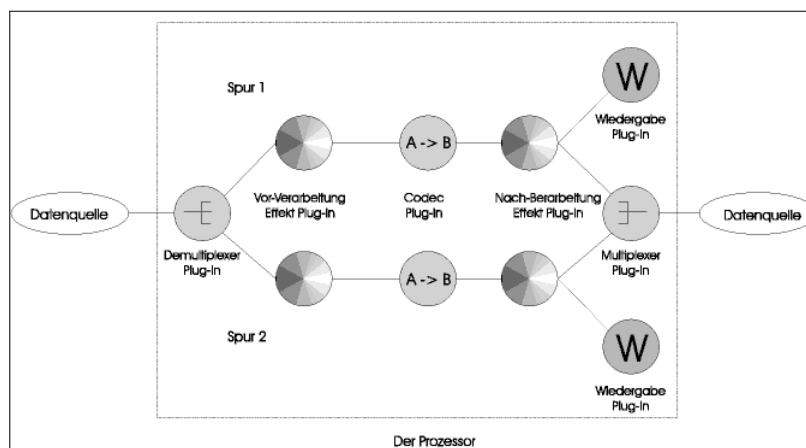
## DataSink

- Basisschnittstelle für Objekte, die Daten aus einer Verarbeitungskette lesen und ausgeben
  - Schreiben in Datei
  - Versenden über Netz an definierte Stelle
  - Verbreiten (*broadcast*) über Netz

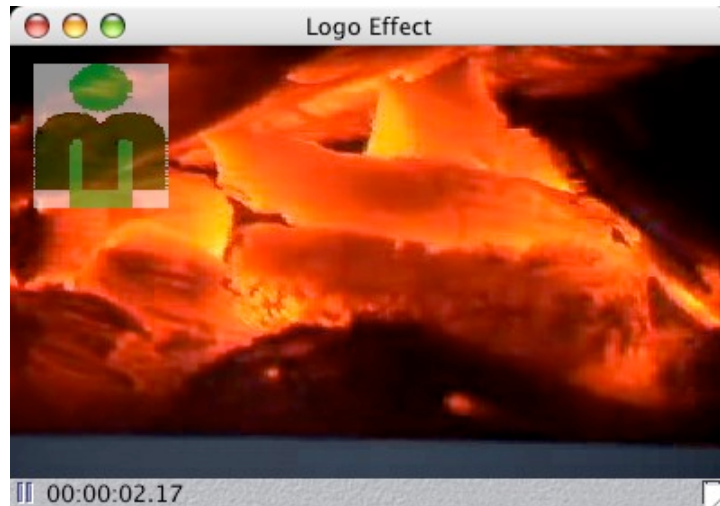
## Buffer

- Reiner Datenbehälter zum Transport von Medieninformation
- Zusatzinformation
  - Flags (z.B. KeyFrame, Verfallsstrategie, ...)
  - Format
  - Header
  - Länge, Dauer
  - Lfd. Nummer
  - Zeitstempel
- Hilfreiche Operationen bei **Buffer**-Objekten:
  - Kopieren (z.B. in Ausgabepuffer)
  - Konversion in Bild (**Image**) und aus Bild
    - » Klassen **ImageToBuffer**, **BufferToImage**
    - » Schnittstelle zu Java 2D-Verarbeitung

## Schema einer Verarbeitungskette



## Beispiel 2: Logo-Effekt



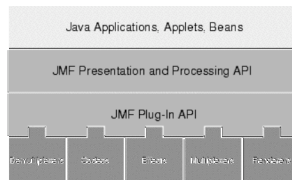
## Beispiel 2: Logo-Effekt (1)

```
class videoPlayerFrame extends JFrame
    implements ControllerListener {
    Processor p = null;
    Player player = null;

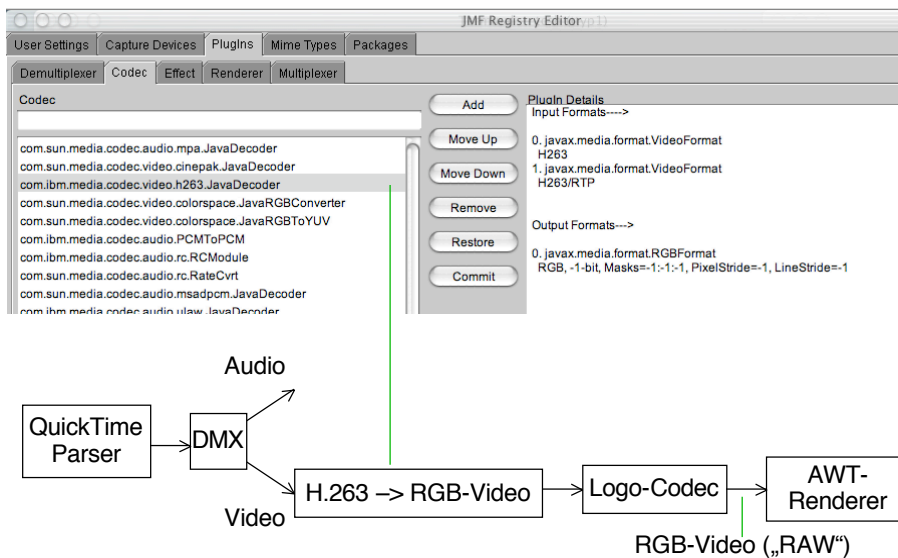
    public videoPlayerFrame(String file) {
        setTitle("Logo Effect");
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent event) {
                p.stop();
                p.deallocate();
                System.exit(0);
            }
        });
        try {
            p = Manager.createProcessor(
                new MediaLocator("file:"+file));
            p.addControllerListener(this);
            p.configure();
        } catch (Exception e) { ... }
    }
    ...
}
```

## Plugin

- Basisschnittstelle für alle Elemente der Verarbeitungskette, die Daten in einem bestimmten Format ein- und/oder ausgeben
- Unterschnittstellen:
  - Codec
    - » Unterschnittstelle **Effect**
  - Multiplexer, Demultiplexer
  - Renderer, VideoRenderer
- Plugin-Schnittstelle unterstützt allgemeine Operationen `open()`, `close()`, `reset()`, die bei Konfiguration aufgerufen werden
  - Weitere Details in den Unterschnittstellen definiert



## Verarbeitungskette für Logo-Effekt



## Track, TrackControl

- **Track:**
  - Ergebnis eines Demultiplexers
  - Strom von gleichartigen Mediendaten (**Buffer**)
- **TrackControl:**
  - Schnittstelle zur Steuerung der Verarbeitung eines einzelnen Tracks
  - `Processor.getTrackControls()`
    - » Liefert Array von `TrackControl`-Objekten
  - `void setCodecChain() (Codec[] codecs)`
    - » Spezifikation einer Folge von `Codec`-Plugins, die angewendet werden sollen.

## Beispiel 2: Logo-Effekt (2)

```
public synchronized void controllerUpdate
(ControllerEvent event) {
    if (event instanceof ConfigureCompleteEvent) {
        TrackControl[] tracks = p.getTrackControls();
        boolean found = false;
        for(int i=0; i < tracks.length; i++) {
            if (!found && tracks[i].getFormat()
                instanceof VideoFormat) {
                Codec[] videoConversion = new Codec[] {
                    new com.ibm.media.codec.video.h263.JavaDecoder(),
                    new logoCodec()
                };
                logoCodecControlIF lcc = (logoCodecControlIF)
                    (videoConversion[1].getControl("logoCodecControl"));
                lcc.setAlphaChannel(0.5);
                try {
                    tracks[i].setCodecChain(videoConversion);
                    p.setContentDescriptor(
                        new ContentDescriptor(ContentDescriptor.RAW));
                    found = true;
                } catch (Exception e) {...}
            } else tracks[i].setEnabled(false);
        };...
    }
}
```

spezifisch  
für neuen  
„Logo-Codec“

## Beispiel 2: Logo-Effekt (3)

```
        if (found) p.realize();
        else { Fehlerbehandlung }
    }
    else if (event instanceof RealizeCompleteEvent) {
        try {
            player = Manager.createRealizedPlayer(p.getDataOutput());
            getContentPane().setLayout(new BorderLayout());
            getContentPane().add(player.getControlPanelComponent(),
                BorderLayout.SOUTH);
            getContentPane().add(player.getVisualComponent(),
                BorderLayout.NORTH);
            setSize(400,280);
            setVisible(true);
            player.start();
            p.start();
        } catch(Exception e) { Fehlerbehandlung }
    }
    else if (event instanceof EndOfMediaEvent) {
        player.stop();
        p.stop();    ... Z.B. Neustart
    }
    }...
```

**Hauptprogramm: Erzeugen eines videoPlayerFrame-Objekts**

## Wie realisiert man „steckbare“ Komponenten?

- Plugin-Mechanismus
  - Beispiel für generellen Mechanismus in Frameworks
  - Austauschbare und vom Benutzer ergänzbare Komponenten
- Idee:
  - Einheitliche Schnittstelle für alle Komponenten (hier **Plugin**)
    - » Insbesondere einheitliche Schnittstelle für fachlichen Kern
  - Schnittstelle enthält Methoden zur *Selbstbeschreibung*:
    - » Zulässige Datentypen bei Eingabeparametern (hier Eingabeströmen)
    - » Zu erwartende Datentypen bei Ergebnissen (hier Ausgabeströme)
    - » Dokumentation (z.B. Name, Kurzbeschreibung, ...)



## Implementierung eines eigenen Codec/Effect

- Implementierung der Schnittstelle `Codec` bzw. ihrer trivialen Unterschnittstelle `Effect`
  - Selbstbeschreibung bezüglich Strom-Datenformaten
  - Realisierung der Konfigurations-Operationen `open()`, `close()`, `reset()`
  - Fachlicher Kern: `process()`
- Implementierung mindestens eines `Controls` zur Steuerung

```
public interface Codec { // Auszug!  
    Format[] getSupportedInputFormats()  
    Format[] getSupportedOutputFormats()  
    int process(Buffer input, Buffer output)  
    ...  
    Object getControl(String controlType)  
    ...  
}
```

## Beispiel 2: Logo-Effekt (4)

```
public class logoCodec implements Effect {  
  
    private Format[] inputFormats;  
    private Format[] outputFormats; ...  
  
    private logoCodecControl control = new logoCodecControl();  
    private int [] logoArray = null;  
  
    private String logoFile = "logo.jpg";  
    private int logoWidth = 60;  
    private int logoHeight = 80;  
    private int offsetX = 10;  
    private int offsetY = 10;  
    private double alphaChannel = 1.0;  
  
    public void open() {  
        ...  
        JPEGImageDecode decoder= JPEGCodec.createJPEGDecoder  
            (new FileInputStream(logoFile));  
        ... Einlesen von logoArray  
    }  
    ...  
}
```

## Beispiel 2: Logo-Effekt (5)

```
public void close() {}
public void reset() {
    close();
    open();
}
public synchronized int process
(Buffer in, Buffer out) {
    siehe später
}
public synchronized Object getControl
(String controlType) {
    return(control);
}
private class logoCodecControl
implements logoCodecControlIF {
    siehe nächste Folien
}
}
```

## Control

- **Control:**
    - Basisschnittstelle für alle Funktionen, die die Verarbeitung steuern
- ```
public interface Control {
    java.awt.Component getControlComponent()
}
}
```
- `getControlComponent` liefert eine Swing-Komponente, die als GUI der Steuerung dient, kann `null` sein, wenn kein GUI vorhanden
  - Jedes Plugin (also auch jeder Codec) muss anbieten:

```
Object getControl (String classname)
```

    - Das Resultat-Objekt muss `Control` implementieren und `classname`-Objekte steuern können
  - Das Control-Interface umfasst fachliche Funktionen für das spezifische Plugin neben der optionalen Konstruktion eines GUI-Objekts
  - Beispiel:

```
logoCodecControlIF lcc = (logoCodecControlIF)
(videoConversion[1].getControl("logoCodecControl"));
```

## „Trick“: Erhöhung der Flexibilität in Frameworks

- Bei Sprachen, die *reflektive* Sprachmittel anbieten (z.B. Ermittlung des Klassennamens als Zeichenreihe)
- Statt einer speziellen (tatsächlich geforderten) Operation:  
`Control LogoCodec.getControl ()`  
wird eine allgemeinere Operation realisiert bzw. gefordert:  
`Object getControl (String classname)`
- Aufruf z.B.:  
`lc.getControl ("logoCodecControl") ;`
- Idealerweise wird zur Laufzeit die Typrichtigkeit überprüft

## Beispiel 2: Logo-Effekt (4)

```
public interface logoCodecControlIF
    extends javax.media.Control {

    public java.awt.Component getControlComponent();
    public String getLogoFileName();
    public void setLogoFileName(String fileName);
    public int getLogoWidth();
    public void setLogoWidth(int w);
    public int getLogoHeight();
    public void setLogoHeight(int h);
    public int getXOffset();
    public void setXOffset(int ox);
    public int getYOffset();
    public void setYOffset(int oy);
    public double getAlphaChannel();
    public void setAlphaChannel(double ac);
}
```

aus Interface  
javax.media.  
Control

spezifisch  
für neuen  
„Logo-Codec“

## Beispiel 2: Logo-Effekt (5)

```
private class logoCodecControl implements logoCodecControlIF {  
    public java.awt.Component getControlComponent() {  
        return(null);  
    }  
  
    public int getLogoHeight() {  
        return(logoHeight);  
    }  
  
    public void setLogoHeight(int h) {  
        logoHeight=h;  
        reset();  
    }  
  
    ... Usw. (analog)  
}
```

Beispiel: `lcc.setAlphaChannel(0.5);`

## Beispiel 2: Logo-Effekt (6)

```
public synchronized int process  
    (Buffer in, Buffer out) {  
  
    out.copy(in);  
    int [] data = (int[]) out.getData();  
  
    RGBFormat inFormat = (RGBFormat)in.getFormat();  
    int redMask = inFormat.getRedMask();  
    int greenMask = inFormat.getGreenMask();  
    int blueMask = inFormat.getBlueMask();  
  
    Dimension inSize = inFormat.getSize();  
    int w = (int)inSize.getWidth();  
    int h = (int)inSize.getHeight();  
  
    int x,y, offBuffer,offLogo, pixelData;  
    ...  
}
```

## Beispiel 2: Logo-Effekt (7)

```
...
for(y=0; y<logoHeight; y++) {
for(x=0; x<logoWidth; x++) {
    offBuffer = (y+offsetY)*w + offsetX + x;
    offLogo = (y * logoWidth + x);

    pixelData = ((int)
        ((data[offBuffer] & redMask) * (1-alphaChannel)
        + (logoArray[offLogo] & 0x000000ff)
        * (alphaChannel))) & redMask;
    pixelData += ((int)
        ((data[offBuffer] & greenMask) * (1-alphaChannel)
        + (logoArray[offLogo] & 0x0000ff00)
        * (alphaChannel))) & greenMask;
    pixelData += ((int)
        ((data[offBuffer] & blueMask) * (1-alphaChannel)
        + (logoArray[offLogo] & 0x00ff0000)
        * (alphaChannel))) & blueMask;

    data[offBuffer] = pixelData;
    }
}
return BUFFER_PROCESSED_OK;
}
```

## Streaming: Definition

- Kontinuierliche Übertragung von Daten von einem Netz-Endgerät zum anderen
  - Dienstgüte-Kriterien:
    - » Durchsatz (*throughput*)
    - » Verzögerung (*delay*)
    - » Schwankung (*jitter*)
    - » Fehlerrate (*error rate*)
  - Grundproblem: Einhaltung von Zeitbedingung unter begrenzten Ressourcen
- Verschiedene Dienst-Topologien:
  - *Unicast*: Ein Empfänger
  - *Multicast*: Mehrere Empfänger (z.B. Abonnenten)
  - *Broadcast*: Alle Endgeräte sind Empfänger (nur in begrenzten Teilnetzen sinnvoll und denkbar)

## Streaming: Grundaufgaben

- Signalisierung:
  - Benachrichtigung der Partner vor Start der Übertragung (Verbindungsaufbau)
  - Steuerung von Verbindungen
    - » Z.B. Start, Pause, Stop, Vorwärts/rückwärts
  - Aufbau von Sitzungen (*sessions*)
    - » Realisiert für zeitlich und im Teilnehmerkreis begrenzten Bereich bestimmte Dienstgüte
  - Standards: SIP, H.323, RTSP
- Datenkompression:
  - Z.B. adaptiv je nach Breite des Übertragungskanals zu einer Station
- Verpackung zum Transport:
  - Zeitstempel, Numerierung etc.
  - Standard: RTP (*Realtime Streaming Protocol*)

## Streaming in JMF

- Datenquellen und -senken mit Netzübertragung
  - **RTPManager**
  - Standardmäßige Unterstützung für RTP und RTSP
  - Unterstützung für Datenformate MPEG-1 und H.263
    - » Hohe Plattformunabhängigkeit
    - » Realisierung von Diensten wie *Video on Demand* oder *Video/AudioConference*
- Nutzung des Java-Ereignisdelegations-Modells für
  - Ereignisse beim Sitzungsaufbau
  - Änderung der aktuellen Bedingungen
- Standard-Programm zum Senden und Empfangen von Medienströmen:
  - **JMStudio**

## Java Mobile Media API

- Abgespeckte und vereinfachte Version des JMF
- 2001 für die Java2 Micro Edition entwickelt
- Seit 2002 in einzelnen mobilen Geräten unterstützt
  - Z.B. Nokia-Mobiltelefone
- Unterstützt z.B.:
  - **Manager**, **DataSource**, **Control**, **Player**,
  - Aber z.B. kein **MediaLocator**
  - Streaming-Funktionen
- Zusätzlich:
  - MIDI-Tonerzeugung
- Möglicherweise die wichtigste Entwicklung zur Verankerung von JMF als Basistechnologie
  - Z.B. UMTS-Mobilgeräte