Media Asset Management in the Broadcasting Industry using Documentum
Derrick Lau

EMC Proven Professional Knowledge Sharing 2010

Derrick Lau
EMC Proven Professional
lau_derrick@yahoo.com
Table of Contents

INTRODUCTION ............................................................................................................... 3

THE OBJECT MODEL ........................................................................................................ 4

BROADCASTING OBJECT MODEL PATTERN ................................................................. 5

  Executive Summary .................................................................................................. 5
  Technical Details ....................................................................................................... 5

OBJECT LIFECYCLE ....................................................................................................... 14

  TRAFFICKING INTEGRATION .................................................................................... 14

  Traffic System Integration Pattern ......................................................................... 15
  Asrun Translation Pattern ...................................................................................... 18

PLAYOUT SYSTEMS INTEGRATION ............................................................................ 22

  PLAYLIST TRANSLATION PATTERN ........................................................................... 23
  CUESHEET TRANSLATION PATTERN ....................................................................... 26

MEDIA WORKFLOWS ................................................................................................. 28

  MEDIA PRODUCTION PATTERN ................................................................................ 29

UNDERSTANDING THE USER BASE ......................................................................... 31

  CUSTOMIZE THE LOGIN PAGE TO ALLOW CASE-INSENSITIVE LOGIN .................. 31
  KEEP THE PROPERTIES PAGE FROM POPPING UP ............................................... 32
  DFS (DOCUMENTUM FOUNDATION SERVICES) CONSUMPTION PATTERN .... 32
  FORCE NEW CONTENT IMPORTS TO BE ONLY OF CUSTOM TYPES ..................... 36
    Default Object Type Pattern .................................................................................. 36
  PROCESS FOR DOCUMENTUM DEPLOYMENTS ....................................................... 38
  SUGGESTED ORDER OF IMPLEMENTATION OF SOLUTIONS .......................... 39
    Phase 1 (Initial rollout) .......................................................................................... 39
    Phase 2 ................................................................................................................... 39

SUMMARY ..................................................................................................................... 40

BIBLIOGRAPHY ............................................................................................................ 41

Disclaimer: The views, processes, or methodologies published in this compilation are those of the authors. They do not necessarily reflect EMC Corporation’s views, processes, or methodologies.
Introduction

Media Asset Management consists of management processes and decisions surrounding the ingestion, annotation, cataloguing, storage, retrieval, and distribution of audio or video assets. These may include small web-sized mp3 files or Quicktime video, as well as large mpeg files and high quality grass valley video files.

Television broadcasters traditionally store vast amounts of media data in the form of commercial spots, secondary event media, television episodes, and feature films. These media assets must be catalogued, reviewed, and eventually played on air. They also need to be accessible by different systems. There are many different users within the broadcast who must ingest and input metadata on the media, and prepare cuesheets for each media asset to ascertain when to play each segment.

Traditionally, Documentum® has been used as an enterprise document management system. Recently, with its rich media product suite, Documentum has expanded into the media and publishing industry. It also can be integrated into broadcasting environments. This opens a whole new market for Documentum contractors and, in particular, those with EMC Proven™ Professional certification, to implement Documentum and integrate it with other broadcasting systems to provide competitive advantage to broadcasters.

This article provides a high-level methodology showing how Documentum can fulfill media asset management requirements that are specific to broadcasters. The goal is to describe solutions as suggested design patterns. These solutions are drawn from my personal experience.

Although there are many broadcasters who use Documentum as an enterprise document management system, this article will focus solely on its media asset management potential. It will not assume that you have Documentum technical knowledge, as readers may include senior executives as business users. Therefore, I will identify technical sections.
After reading this article, you should have a high-level understanding of how to present Documentum to a broadcaster as a solution to their media asset management needs, and be able to list the high-level steps required to integrate Documentum into their technical environment. It is important to ensure that the broadcaster gets the full value of a Documentum implementation. Finally, this article provides a launching point for EMC Proven Professionals in the hardware space as they will learn how to structure a solutions package with EMC hardware alongside EMC software to effectively meet broadcaster requirements.

The Object Model
Documentum is an object-based Enterprise Content Management (ECM) system. It manages all the content within, whether structured or unstructured, as a collection of objects. Each of these objects can be classified into types that contain attributes pertaining to the objects and to the content itself. This provides a hierarchical view of the content within Documentum and allows users to organize and classify their content for retrieval. In addition to classifying objects, Documentum can file content within cabinets and folders of its repositories.

The first step to incorporate Documentum into a broadcasting environment is to develop a practical object model to categorize and catalogue the media. The suggested solution is the broadcasting object model pattern that forms the foundations for the other solutions in this article. I suggest standard metadata and inheritance patterns while ensuring important performance factors are not adversely impacted. This solution will provide a means for relating media assets within Documentum to other broadcasting systems. We will discuss the taxonomy and the easiest way to organize the cabinet and folder structure within the repository to support broadcast media asset management.
Broadcasting Object Model Pattern

Executive Summary
This is a suggested solution for how operations analysts can initially set up Documentum.

Technical Details

Problem
Most television network broadcasters are content distributors. They purchase licenses from different production studios to play their shows received as either Sony Beta tape, DVD, or any other media. These tapes are stored, but eventually need to be converted into digital files for longevity. The best way to store these tapes is in an Enterprise Content Management system such as Documentum. However, in addition to storing tapes, they must be stored with their associated metadata for easy search and retrieval.

Solution
The object model must provide the following:

1) A method to link different episode cuesheets with the correct episode title. There may be several cuesheets for a single episode. This is true in cases where the episode contains sensitive content, but will be aired during primetime as well as later at night.
2) A way to organize episodes of a series under the correct series and season.
3) A way to link episode media to the trafficking system.
4) A way to store movies or other special events that are not aired live.
5) A lean hierarchy that won’t impact Documentum Query Language (DQL) performance.
6) A taxonomy that will allow users to easily browse the folder organization of a Documentum cabinet to find what they require.
**Description**
The primary business users for this pattern are operations staff in the television broadcasting industry, though other industry verticals may use this pattern as well. As a precursor, it is assumed that the current IT environment contains a Documentum repository and a box running Documentum Foundation Classes (DFC).

This pattern provides a means to logically store broadcast media within Documentum, track the required metadata for each episode of a show or a special event, and provide a best practice for organizing these objects within the Documentum file system.

**Functional How-To**
There are 3 types of objects required:

1) Title objects represent the television series name, the season and the episode title, as well as the title of a movie or special event.
2) Content objects represent the media that will be played out.
3) A document object holds information on uploaded documents that relates them to the broadcasting business.

The best approach is to inherit all the objects that would represent media directly from the dm_document type, even though they may all have the same cuesheet attributes. This makes the media object hierarchy simple and easy to maintain. Although this should result in better performance of SELECT DQL statements, I have not been able to verify this in my own testing. Title objects should inherit from dm_folder as they may contain multiple content versions, such as versions for late-night television or primetime.

The media object requires a foreign key to the trafficking system, as well as an attribute to hold the tape number of the media, and attributes to hold cuesheet related data. The title object should contain more general information, such as the episode name, a foreign key back to the trafficking system, timecodes where questionable content appears, audio, and quality control information.
Based on the above criteria, the object model should be:

**Media_Asset:**
Create the type media_asset to represent tapes and media assets within the system.

The media_asset object type must relate itself to the other systems to ensure that Documentum can interact with broadcast systems. These relations will be tracked via attributes in the object that serve as foreign keys to the same data in another system:

**Foreign Keys:**
<Type> trafficking_system_key
The object must have a foreign key so it can relate to the trafficking system in use. The type of foreign key data attribute must be the same primitive data type as the corresponding value in the trafficking system. In addition to trafficking systems, there may be other systems that the object may require tracking foreign keys to.

**Attributes:**
There is metadata associated with every tape or video. Track this information in the object’s attributes.

String tape_number
This is the number assigned to the Sony Beta videotape of the content sent from the production company.

String streaming_url
This attribute will contain the streaming server Uniform Resource Locator (URL) that media player applications may use to stream the object’s content.

**Cuesheet Related Data**
Cuesheets contain information for the trafficking system. They detail how to divide a piece of television content into separate segments so commercials may be inserted.
Repeating Attribute, String timecode_segments
In Documentum, a repeating data type means that this attribute will store a list of values of type string. This list contains the start times for each segment of the content, and will be used later by the trafficking system.

Timecode offset_time
This will store the credit squeeze offset that tracks when the credits will play.

String cc_attribute
This attribute will store a string representation of the type of captioning information included with this content, such as whether it is described video, or if there is no captioning information included.

Timecode squeeze
This provides the timecode of the location where a promo has been squeezed to play during the credits roll.

String operator
This attribute tracks the operator who entered the cuesheet information for auditing purposes.

String recording_vtr
This attribute will remember which VTR the operator used to ingest this tape into the automation system.

String playback_vtr
This attribute notes the VTR used to playback a tape.

String edit_room
This attribute holds the location of where the content edit took place.

String aspect_ratio
This attribute represents the aspect ratio and resolution of the content of this object.
**Title**
The object type title will represent series, seasons, episodes, and special events titles.

**Foreign Keys:**
String contract_number
This attribute stores the corresponding contract number associated with the license for broadcasting this title. It is provided by the production company. This information may be of use for rights management systems. With this information, custom applications can associate content with their broadcast license.

<Type> trafficking_system_key
This attribute will store a foreign key that corresponds to the same title record in the trafficking system if such a record exists. It allows custom applications access to the relevant data in the trafficking system.

**Attributes:**
String episode_number
This attribute holds the number of the episode if the title is for a specific episode in a series.

String episode_title
This attribute will hold the name of the episode if this title is an episode.

**Questionable Content Data:**
Repeating Attribute, timecodes segments
This attribute contains the timecodes of all areas of the content that contain material that may not be suitable for televised broadcasting.

**QC (Quality Control) Data:**
Integer number_of_tapes
This attribute will tell the users the total number of tapes in possession that hold the content represented by this title.
String tape_number
The broadcaster's assigned unique identifier, usually written on the case of the media.

String aspect_ratio
This attribute will contain the original aspect ratio of the episode content.

Boolean bars_tone
This attribute determines whether the content sent from the distributor includes a few seconds of color bars and test tone.

Boolean distributor_logo
This determines if the tape has a few frames depicting the distributor's logo.

Boolean opening_credits
This attribute will determine whether the content has an opening credits sequence.

Boolean recap
This determines if the media comes with a recap.

Boolean captions
This attribute indicates whether the content came with an embedded closed captioning signal or not.

Boolean VITC
Vertical Interval Time Code (VITC) represents the timecode that is already burned into the tape. Therefore, this attribute determines if the media includes VITC or not.

Boolean subtitles
This indicates whether the content has subtitles. It is used to determine if translation is needed based on where the broadcast will play.

String hd_or_sd
This will hold either the values HD (High Definition) or SD (Standard Definition).
Boolean `textless_elements`
This attribute is used by quality control (QC) to mark whether there are graphical logos appearing on the media that were not part of the original content provided by the producer.

String `duration`
String duration is the length of the content represented by this title.

Boolean `slate`
This is another QC attribute. It determines whether this media includes a clapperboard at the beginning to designate and mark particular scenes and takes recorded during production. The clapperboard will typically include the date, the production title, the director's name, the photography director's name, and the scene information. It is normally held in front of the camera, with a top portion like a lever that is clapped down to inform the cameramen to begin shooting.

Repeating Attribute, `timecodes quality_issues`
This attribute contains a list of timecodes indicating when the content contains quality issues, such as corrupted video, corrupted audio, etc.

Boolean `end_credits`
These credits determine if the content has end credits.

String `credit_length`
This stores the duration of the credit sequence.

Boolean `credits_squeezable`
This attribute determines if the credits are squeezable.

String `max_squeezable_length`
This attribute marks how long a promo can be squeezed into the credits. The typical values are n/a, 15s, and 30s.
Boolean easily_separated
This attribute refers to the credits and flags and whether they can be run as their own segment or not (easily separated from their program/content). Thus, users can examine this attribute to determine if the credits included with the media can be replaced with credits generated by the broadcaster specifically for this content.

Boolean acceptable_broadcast_quality
Is the content quality high enough for broadcasting?

String tape_format
This determines the type of media in which the original content was provided. The usual formats are beta, VHS, HD Cam, Dbeta, Dub from PAL, SP, SX, and HD Cam SR. These values can be set using value assistance.

String frame_rate
This attribute notes the frame rate of the media. The typical values are 23.967fps, 24p, 29.97 (NTSC SD), 59.94i, 50i (PAL HD), and 25 (PAL SD).

String operator
This attribute remembers the operator who inputted information.

String QC_Solution
This describes what was done to resolve quality control issues.

Audio Channels:
Repeating Attribute of String v_audio_channels
This attribute holds channel number, the mix (described, dialogue, discrete (for SR), Dolby E, full mix, M/E, mix minus narration, MOS (silence), narration & dialogue, surround), a description (Dolby 2.0, SMPTE (Dolby E 5.1 Standard), Stereo, Mono, MOS (Silence), Lt, Rt, L, R, C, LFE, Ls, Rs), and user comments.
Media Document is the final object type required /

**Media Document**

This type is a descendant of dm_document, but contains one custom attribute.

String (Repeating) channels

This attribute holds a list of channels related to the document. It would normally be used by Marketing or the Creative Agency to help organize promos by permitting searching of all documents related to a specific set of channels. The values for this attribute should be set using value assistance, and forbid user entry, so that only valid channel values can be added.

Implement these custom types within a single Documentum composer project for ease of maintenance.

**Taxonomy:**

The organization of the Documentum file system must support a natural organization of the media so that users browsing for an episode or tape can easily find it. I suggest putting all the media within one cabinet, and organizing by the title of the media first, be it a special event or television show. Special event titles will have their related media objects directly underneath. Television titles will start with the series title, which will be broken into season titles, which will then be broken into subsequent episode titles. The respective media objects will be within the episode title folders. There can be multiple media objects within a title folder as some media will be edited for primetime playout while others may contain adult content and intended for late night playout.

**When to use it**

It is best to base all media asset management objects on this design pattern.
Object Lifecycle

Documentum supports lifecycles that are “a set of states that define the stages in an object’s life.”1 Similar to how people age from toddler, teenager, adult, and mature, objects progress linearly throughout their lifecycles. Administrators may use lifecycles to define storage policies or to determine at what stage in an object’s lifecycle it should be moved to archival storage to free up expensive but high-speed disk space.

Archiving media assets is necessary in broadcasting since Documentum will be storing rich media assets that are traditionally large in size. A regular archiving policy must be employed to prevent expensive fast disk storage from depletion. Based on current experience, it’s best to move assets over 30 days old to digital tape archives. This can be done using lifecycle policies in Documentum.

Trafficking Integration

Integration with broadcast trafficking systems is the third major area addressed in this article. These systems are used to schedule sponsored content (commercials), build out the daily TV schedule, and eventually reconcile what the broadcaster played against what they were supposed to play. Proper integration between them and Documentum is essential since these systems require information about media assets and often contain data about the television series, movies, and sports events broadcast daily.

Part of this solution is addressed in the broadcasting object model pattern, while the traffic system integration pattern focuses on keeping both Documentum and the trafficking system in sync. Finally, the asrun translation pattern will discuss the need to make sure the asrun logs produced by the playout server are compatible with the trafficking system so the broadcaster can get paid.

---

1 EMC Documentum Content Server Version 6.5 Fundamentals (EMC Corporation, 2008), p301.
Traffic System Integration Pattern:

Executive Summary
This solution requires custom application development resources to implement, and will ensure that information within Documentum is compatible with the trafficking system. It will also ensure the efficient flow of information from the broadcasting system to Documentum.

Technical Details

Problem
In a typical broadcast organization workflow, users enter all data regarding content to be aired into a trafficking system so that content can be scheduled and sponsored media can be billed. The trafficking system can hold information such as tape number, series name, special event name, and episode name, but usually cannot contain links or pointers to low-res copies of the media. The trafficking system is usually the first system in the broadcast workflow, so any information entered into it should be accessible by other systems down the pipeline. How should this be implemented?

Solution
At a high level, any integration between a trafficking system and Documentum must accomplish 3 things:

1) Keep the title objects and the Documentum taxonomy synchronized with the trafficking system’s special events, series, seasons, and episode titles.

2) Keep media objects properly associated with their corresponding titles.

3) Keep enough information in Documentum to link back to the related record in the trafficking system.
Description
Although this is a technical design pattern, its purpose is to ensure that work is not duplicated between trafficking and broadcast operations. It provides a solution for keeping the trafficking system’s asset management component synchronized with Documentum.

Prerequisite
This pattern requires a Documentum repository using the Broadcasting Object Model pattern.

Functional How-To
There are two approaches to ensure that Documentum is in sync with the traffic system. One approach is to load the trafficking system records into a dataset, traverse through the set, and ensure the links of the corresponding Documentum objects reflect the hierarchy. If not, then re-link the objects until they do. In addition, populate the Documentum objects with the trafficking system’s foreign key information.

Another approach is to assign triggers to Structured Query Language (SQL) statements that change the information of the trafficking system, and use the triggers to make the corresponding changes to Documentum. This would avoid batch processing, reduce server load, and be easier to maintain. However, this method involves tampering with vendor owned databases. This is not a recommended practice, particularly if regular support and maintenance from the vendor is required.

The data processing work remains the same regardless of the approach. Once the data is loaded from the trafficking system’s database, the records must be traversed and the information used to create and populate an object containing the following attributes:

Title ID
This refers to the unique identifier of the title object being created. The title object may be a series title, season title or episode title.
Parent Title ID
The unique identifier of the title object this object should be a child of.

Parent Name
The name of the parent object.

Title Name
The name of this title object.

Series Name
The name of the television series it is associated with, if any.

Episode Number
The number of the episode for the season.

Episode Code
A value in the trafficking system that represents the episode object.

Title Type
The type of content, whether it is a movie, sports event, television season, etc.

Production Year
The year the content was created.

Store these objects in an array or collection, and traverse to create the corresponding Documentum objects with the proper metadata values. Use the values of Parent Title ID to determine the r_object_id of the appropriate title object within Documentum.

However, the reverse must also be true where data entered into Documentum should be reflected in the trafficking system. There are two possible approaches to this:

1. customize the front-end GUI to also populate the trafficking system (which I have done),
2. or create a Type Based Object (TBO) for the media custom type and override the save method to populate the trafficking system as well.
When to use it
This should be part of any broadcast workflow.

The asrun translation pattern describes a way to ensure the asrun logs generated by the playout servers can be understood by the trafficking system. Although the general translation algorithm does not require Documentum, it is a good practice to keep copies of the logs within the repository for archival. If the logs will be stored in Documentum, it is feasible to place the asrun translation code into a plugin for Content Transformation Services (CTS).

**Asrun Translation Pattern:**

**Executive Summary**
This solution will require custom application development resources for implementation, but will allow information sharing between incompatible automation/playout systems and trafficking systems.

**Technical Details**

**Problem**
The broadcast trafficking and automation systems are not always fully compatible.

**Solution**
Provide an intermediary application that sits between the automation system and trafficking system that will read the asrun logs output by the playout server and translate them into a format understood by the trafficking system.

**Description**
Operations staff in the television broadcasting industry are the primary business users for this pattern. Other industry verticals may find use for this pattern as well.
The asrun translator must provide a translated asrun that the trafficking system will understand, and perform basic reconciliation checks to ensure the log can be properly imported.

**Prerequisites**
The playlist translation pattern is already in use.

**Functional How-To**
Regardless of the architecture decision, the pattern to accept and translate the asrun log to a format understood by the trafficking system remains constant. The pattern is very similar to the playlist translation pattern as it must read each event, organize the events by primary and secondary, ensure the secondary events are associated with the correct primary events, perform a reconciliation check, and update the output event. The pattern can be implemented as a scheduled task set to read a directory periodically, or as a service that detects when files have been dropped into the directory.

To accomplish the translation functionality, the pattern is essentially the same as the playlist translation pattern, except that there is a reconciliation member function associated with the Event class. This method must be called before each event is written as it will be used to set certain flag values for the trafficking system to understand.

Much like the playlist translation pattern, the translated asrun logs will be written to a temporary directory before being copied over to the directory monitored by the trafficking system for importing the asrun logs.
Therefore, the following high-level design is recommended:

**Class Diagrams:**

Class Event

```java
{ 
    Protected String ID; 
    Protected DateTime startTime; 
    Protected DateTime, duration 
    //other information as required.
    Public List<Event> SecondaryEvents;

    Public void setReconKey();
}
```

Abstract class Asrun

```java
{ 
    Protected List<Event> m_events;
    Public abstract Asrun read(String filename);
    Public abstract Asrun write(String filename);
}
```

Class D-series Asrun: inherits from Asrun

```java
{ 
    Public Asrun read(String filename) 
    { //method reads a D-series asrun and uses the information to populate m_events; } 

    Public Asrun write(String filename)
    {
        Foreach (Event e in m_events)
        {
            String recordLine;

            Foreach (Event se in e.SecondaryEvents)
            {
```
Se.setReconKey();
    Write se information to appropriate spot in recordLine;
}

e.setReconKey();
    Write e information to appropriate spot in recordLine;
    Write recordLine to the file filename;
}

Class H-series Asrun: inherits from Asrun
{
    Public Asrun read(String filename)
    { //method reads a H-series asrun and uses the information to populate m_events; }

    Public Asrun write(String filename)
    {
        Foreach (Event e in m_events)
        {
            String recordLine;

            Foreach (Event se in e.SecondaryEvents)
            {
                Se.setReconKey();
                Write se information to appropriate spot in recordLine;
            }

            e.setReconKey();
            Write e information to appropriate spot in recordLine;
            Write recordLine to the file filename;
        }
    }
}
Public void TranslateDseriesToHseries(String filenameD, String filenameH)
{
    Asrun currAsrun = new D-series Asrun();
    currAsrun.read(filenameD);
    ((H-series Asrun)currAsrun).write(filenameH); //write output to temporary folder
    Copy H-series asrun to the final output folder.
}

When to use it
Use this pattern only when the trafficking system does not naturally support the
automation system.

Playout Systems Integration
The fourth major solution area is integration with broadcast automation systems, or more
specifically the servers that broadcast the content viewers watch on their television sets.
The solutions covered in this section include the playlist translation pattern for ensuring
the trafficking systems’ playlists are readable by automation, and the need to archive
these playlists for future reference. Playlists can be well-formed XML (Extensible Markup
Language) or flat text files; we will discuss the optimal storage solution for these files.
Finally, before playlists can be generated, cuesheets must be created for each piece of
media. A solution for managing cuesheets will be covered partially in the broadcasting
object model pattern, while the cuesheet translation pattern will cover how to keep the
information current in the trafficking system for playlist creation.

Keeping archives of playlists for thirty days is essential. Therefore, store the original
playlists in the Documentum repository. One way to implement this pattern is to place its
functionality into a CTS (Content Transformation Services) plugin and treat the different
formats of playlists for different playout servers as renditions.
Playlist Translation Pattern:

Executive Summary
This solution will require a custom application development resource to implement a means of translating a trafficking system’s playlist into a format the playout server in the automation system can understand.

Technical Details
Problem
Most trafficking systems generate a single broadcast date’s playlist based on current user data entry, but there is no standard playlist format. If using the S4M trafficking system, the playlist it generates will not play on a Harris D-series playout server. This problem should be infrequent because the trafficking system and playout system are usually purchased by the same management team. They will ensure the systems are compatible in their vanilla state, but if a broadcaster is bought out by another company, the parent company can introduce a new trafficking system that may not produce a playlist form that the current playout servers understand.

Solution
An application can be placed between the trafficking system and the playout system to convert the trafficking system’s playlist into a format the playout system will understand.

Description
The primary business users for this pattern would be operations staff in the television broadcasting industry. This pattern will provide a means to convert a playlist from one format to another.

Functional How-To
First, the playlist output of the trafficking system will be read into memory, with records organized into different events with start times, durations, unique ID of media to play out, video source, media title, and vchip information. The translator will then write these events into the required output format, and make the required calculations in case the output record requires information such as offset from primary event time.
Second, each event and its particulars should be written to a database table for reconciliation. This table should be indexed by the event identifier so reconciliation queries performed against it will be quick. The database should also contain a separate logging table to allow users to see the errors encountered for each event per playlist or per translation session.

Third, archive the original playlist and assign a unique ID for future reference. Harris H-series playout servers use XML formatted playlists, so in that particular case, Documentum would be an ideal archive area as it naturally supports XML. In fact, since most playlists are files that are read by the automation system, Documentum is a natural fit for capturing these playlists for archival purposes. Performance issues were encountered when inserting the full playlist into Microsoft SQL Server’s XML data type.

Finally, the translated playlist will be output to a temporary directory. The translator may either notify the automation system that the playlist is ready for input or copy the translated playlist to a monitored directory for the automation system to process. The temporary directory is necessary because if the playlist is translated directly to a monitored directory, there is a risk that the automation system will ingest the playlist before the translation is complete.

Therefore, the following high-level design is recommended.

**Class Diagrams:**

Class Event

```java
{  
    Protected String ID;  
    Protected DateTime startTime;  
    Protected DateTime duration;  
    Protected EventType m_eventType;  
    Protected String m_houseNumber;  
    Protected DateTime broadcastDate;  
    //other information as required.  
    Public List<Event> secondaryEvents;
}
```
Abstract class Playlist
{
    Protected List<Event> m_events;
    Public abstract Playlist read(String filename);
    Public abstract Playlist write(String filename);    
}

Class H-series Playlist: inherits from Playlist
{
    Public Playlist read(String filename)
    {  //method reads a H-series Playlist and uses the information to populate m_events; }

    Public Playlist write(String filename)
    {  //method writes m_events information as H-series playlist format; }
}

Class D-series Playlist : inherits from Playlist
{
    Public Playlist read(String filename)
    {  //method reads a D-series Playlist and uses the information to populate m_events; }

    Public Playlist write(String filename)
    {  //method writes m_events information as D-series playlist format; }
}

Public void TranslateHseriesToDseries(String filenameH, String filenameD)
{
    Playlist currPlaylist = new H-series Playlist();
    currPlaylist.read(filenameH);
    ((D-series Playlist)currPlaylist).write(filenameD);  //write output to temporary folder
    Copy D-series playlist to the final output folder.
**Performance Considerations**

Regardless of how many channels a broadcaster may own, translation performance must be optimal. The key performance factor is how many web services the translation pattern must call in order to complete its tasks. The fewer calls to out of process code, such as web services, the better performance will be. Database interactions for the translation should not consume much time, even if the database calls do not share a single persistent connection, so long as the columns being searched are indexed.

**When to use it**

This pattern is only required when a trafficking system is implemented that does not support the playlist format of the current automation playout system.

**Cuesheet Translation Pattern:**

**Executive Summary**

This solution requires a custom application development resource to implement it so that information within Documentum and the trafficking system will remain up to date and synchronized.

**Technical Details**

**Problem**

To incorporate an Enterprise Content Management (ECM) system like Documentum into the media production workflow, media and their associated metadata should be stored in the ECM system. Thus the Broadcasting Object Model pattern uses Documentum as the repository for the electronic copy of the media, plus its closed captioning, quality, and segment information. Segment information is particularly important to the automation system as it defines how the main content is to be divided and what commercial spots to fit in between. It would be ideal if Documentum was able to naturally generate playlists, as once the cuesheet information is entered into Documentum the proper playlist could be output and presented to the automation system. However, there are no Documentum products or add-ons that do so, and to implement such a customization would be resource intensive. Fortunately, most broadcast trafficking systems have built-in functionality to generate playlists, so the challenge becomes populating the trafficking system with the updated cuesheet information.
Solution:
The solution consists of the following:
1) The trafficking system must have a means of accepting cuesheet information, such as a web service to call, a markup language or file format that it understands, or even a stored procedure that can be called.

2) Documentum must be customized such that the segment information associated with an object can be output in the format required by the trafficking system.

3) There is a custom interface that allows users to enter cuesheet segment information into Documentum.

Description
Operations staff in the television broadcasting industry would be the primary business users for this pattern. Other industry verticals may find use for this pattern as well.

As a precursor, it is assumed that the current IT environment contains a Documentum repository and a box running either DFS (Documentum Foundation Services) or DFC (Documentum Foundation Classes). This pattern can be implemented in a scheduled task architecture, where once a segment information is updated, a cuesheet file is written to a directory that is monitored by a scheduled task running a program that reads the cuesheet and imports the data into the trafficking system. Another approach is to call the trafficking system’s application programming interface (API) and feed its parameters with the cuesheet information.

Prerequisites
1) The trafficking system must have a means to allow outside processes to update its program segment information.

2) The Broadcasting Object Model pattern must be already implemented within Documentum.
Functional How-To
Regardless of the architecture decision, the pattern to accept and translate the cuesheet to a format understood by the trafficking system remains constant. The user interface, in addition to providing input controls to enter information like aspect ratio, standard or high definition, and closed captioning, must provide a dynamically expanding grid to hold the start and end timecodes of each segment. Once this information has been entered and confirmed, the interface must populate the corresponding Documentum object with the cuesheet information, and either call the trafficking system API to update that system with the segment information, or call a web service to do so. The custom interface can either be implemented in the desktop client or web development kit (WDK). Since implementing it in the desktop client requires installing and supporting the client on every user’s desktop, I recommend implementing the customization using WDK unless a custom interface already exists for tracking broadcast titles and episodes. In that case, you can use DFS or DFC to add this functionality into the existing interface.

Most likely a custom interface will not already exist (hence the need to purchase Documentum), so this pattern will usually be implemented using WDK. However, a web service will be required if the broadcast trafficking system’s API doesn’t support Java, or can only read cuesheets as files from a Windows-based filesystem. The WDK customization should then call the web service and feed it the cuesheet information. The web service will then either call the appropriate API in the trafficking system or, while running under a provided credential, create a cuesheet file on a file server directory for the trafficking system to read and import.

When to use it
Use this pattern as a precursor to implementing a media production system using Documentum as the core content engine.

Media Workflows
Media production is the fifth solution area: the use of Documentum as an engine to help build television news shows. My solution covers how to track annotations in Documentum and transfer them to the broadcast quality content on the video server, as well as some technical details such as media streaming to provide a cut preview. This solution will be designated as the media production pattern.
**Media Production Pattern:**

**Executive Summary**
This solution requires a cross-functional project team consisting of project management, custom application development resources, and operations staff to construct a system to enable users to edit media content without physically moving between offices and archived tape libraries. The solution also provides a means for long term archival storage of decaying legacy videotape media.

**Technical Details**

**Problem**
A great deal of editing requires human effort to find the correct tape from the library, bring it up to the office, watch it, and fill out paper cuesheets with the start and end timecodes of each segment. This effort is a waste. Digitized content can be brought directly to an end-user’s desktop with the help of Documentum and some customization.

**Solution**
The solution is an application that coordinates ingesting video content, loading it into the Documentum repository, and playback to decide on how to segment the content for playout. This application should also be able to ingest broadcast quality video into the automation system for playout.

**Description**
Primary business users for this pattern are operations staff in the television broadcasting industry. It provides a high-level design for integrating an ingest system and video editing system with Documentum to provide editing at the user’s fingertips.

**Functional How-To**
Consider two groups of users:

1) The operations staff who ingest the content into electronic format for the automation system
2) The editors/trafficking staff that segment the shows and build the playlist schedule
The ingest station should be provided with a PC with an audio and video capture card to transcode the original content into low-res video for storage in Documentum in parallel with the transcoding to broadcast quality video for the automation system. A custom application should allow these two ingests to run in parallel, as well as capture metadata like speech to text and closed caption text. The metadata will be stored with the corresponding video file in Documentum. For simplicity, the low-res video should be recorded with the video recorder timecode displayed, to make it easier for editing/trafficking staff to build up the cuesheets. Therefore, the ingest station could use an integration that interacted with the application programming interfaces (APIs) of the video capture application (such as Virage), and communicate the information to Documentum via Documentum Foundation Services (DFS).

Editing/trafficking only requires a regular workstation since all content will be ingested to Documentum before building the cuesheets. To build the cuesheet, they may simply browse to the proper content, and play it back via a Documentum client like Digital Asset Manager (DAM). Since the timecodes are displayed on the low-res video, they can use the information to fill out the cuesheets within Documentum. To make things easier, DAM could be customized to allow cuesheet entry on the same web page that displays the content. If necessary, users should be provided a means to generate a cuesheet formatted for input into an editing system such as Avid.

**Performance Considerations**

Depending on how many channels a broadcaster may own, translation performance must be optimal. The key performance factor is how many web services the translation pattern must call to complete its tasks. The fewer calls to out of process code, such as web services, the better the performance. Database interactions for translation should not consume much time, even if the database calls do not share a single persistent connection, so long as the columns being searched are indexed.
When to use it:
This pattern is only required when there is a need to reduce the time spent having users to the tape library, looking up the required tape from the many shelves, and fetching the correct tape from the archive for editing. The physical tape library should be large enough so that it takes users little time to find the right tape from the right shelf.

Understanding the user base
Every broadcaster will encounter some user interface issues. However, Documentum customizations may be quickly provided to compensate for them. This will essentially be a catalogue of quick wins to help gain user acceptance of media asset management.

Customize the login page to allow case-insensitive login.
Some broadcast operations users had difficulty with the Digital Asset Manager (DAM) login page during user acceptance. They were used to case-insensitive usernames for logins, a standard employed by Windows but not by Documentum. Now, there are two ways a user may access Documentum, and one does not involve visiting the login page if credentials were saved during a previous login. Therefore, the solution requires a method of authentication that would be called each time an user passed in a Digital Asset Manager URL (Uniform Resource Locator) into their browser.

Executive Summary
This solution will allow users to log into Documentum similarly to how they log on to their Windows workstations, reducing user confusion and the time required for user training. It requires custom application development resources.

Technical Details
It is simplest to create a custom authentication service to capture both methods of authentication. This is done by extending the AuthenticationService class, and adding it as a <service_class> node within the <authentication> node of the app.xml. Within the custom authentication service class, use Documentum Foundation Classes (DFC) to query the repository and match the entered username to a username within the system, regardless of case. Then substitute the system’s username in place of the one the user entered. Thus, however a user accesses Digital Asset Manager (DAM), whether directly through the login page or indirectly by clicking on a link to a repository object, their credentials will be authenticated without regard to letter case.
Keep the Properties page from Popping Up

Executive Summary
This solution will modify DAM 6.5 so it will behave similarly to DAM 5.3, providing an interface that will not confuse users who have become accustomed to DAM 5.3, and make the properties page full-screen within DAM and thus easier to read.

Technical Details
By default, the new DAM 6.5 displays the Properties of an object in a popup Javascript Window. The user base requested we change the Properties page since they were already accustomed to the Properties page not appearing. This was a simple configuration of copying the dm_sysobject_actions.xml within $CATALINA\webapps/dam/webcomponent/config/actions to $CATALINA/webapps/dam/custom/config/actions. Then editing the configuration file so that in the <action> node with id of properties, the <invocation> node is removed.

DFS (Documentum Foundation Services) Consumption Pattern:

Executive Summary
This solution requires a custom application development resource, but places reusable DFS code in a single area, allowing for code sharing and higher developer productivity.

Technical Details

Problem
A Documentum system must have its products deployed across multiple servers to support the needs of the television broadcasting business and to provide a combination of core media asset management functionality along with a level of acceptable performance. This results in separate hardware boxes to run the content server, index server, Digital Asset Manager (DAM), Documentum Administrator (DA), and Content Transformation Services. In addition, there is usually a separate hardware box for running customized Documentum applications that were not implemented as Web Development Kit (WDK) applications. Pre-D6, these boxes usually held the
Documentum Foundation Classes (DFC) and possibly the Primary Interop Assemblies (PIA) for integration with Microsoft.NET applications. This IT infrastructure can be challenging to upgrade due to the number of hardware servers involved plus the limited amount of downtime the broadcast business can tolerate. Is there a way to reduce the number of servers used by the Documentum system while still providing the media asset management capabilities a broadcaster requires?

Solution

There are several ways to reduce the upgrade count by one server:

1) Use WDK to implement all customized Documentum applications. This depends on whether WDK can meet the business requirements of a particular project. In the broadcast industry, requirements to automate the integration of trafficking systems with media asset management systems might not be satisfied through WDK.

2) Provide a service to encapsulate the Documentum Foundation Services (DFS) productivity layer and place it on the content server box. The risk is that any problem with the customized application may cripple the entire box. Although the service can be implemented using either Java or Microsoft.NET and should be safely executing within its own virtual machine, there is no guarantee, and it’s better safe than sorry.

3) Provide a service to encapsulate DFS (without the productivity layer) and place it on a web services box. A problem with this service will not impact the content server and entire Documentum system; it requires another server instance to execute upon.

4) Implement custom applications that consume DFS without the productivity layer. DFS can be called as a web service so this solution would fit into the layered services architectural pattern, and would not require the customized application to run alongside the content server.

From a broadcast perspective, the fourth solution offers the least risk so I will focus on that one.

DFS Consumption Pattern
Description
Application development staff in the television broadcasting industry are the primary business users for this pattern. Other industry verticals may find use for this pattern as well. As a precursor, it is assumed that the current IT environment contains a Documentum repository and that DFS is installed on this box.

Using a design based on basic object oriented principles, this pattern provides a simple method to easily manage the consumption of DFS services.

Functional How-To
At a high level, the DFS Consumption Pattern consists of an abstract class that will implement the DFS service calls, and an abstract method that must be implemented with the purpose of processing the results. The rationale is that all DFS methods return either a DataPackage containing DataObjects or a QueryResult containing a collection of DataPackage objects, each with a different collection of Property objects depending on what the query returns. Therefore, the code to process the results of the DFS call differs per situation, but the general setup and call code is the same. The abstract class also contains an object to represent the message header on all DFS messages, so that the same header and Documentum credentials can be used on all DFS calls.

The abstract class should also contain methods to perform basic DFS operations, such as return the results of a DQL query or return a specific object as a DataObject. These methods will call the abstract method for processing the results.

Finally, the class that implements the abstract class should contain a public identifier, such as an enumeration, to let users tell it what query or DFS operation it is executing. This identifier can be used in the implementation of the abstract method to process the results of the DFS call appropriately. These results will be exposed to other objects via public methods that return the information in a format independent of DFS and that other objects in the application can use.
Public class ServiceContextHeader : MessageHeader
{
    Private List of RepositoryIdentity objects;
    Private ContentTransferProfile member;

    Public void Add(RepositoryIdentity);
    Public void Add(ContentTransferProfile);

    Protected void OnWriteHeaderContents(XmlDictionaryWriter, MessageVersion)
    {
        Attaches appropriate Documentum information to web service message header for
        use in communication with DFS.
    }
}

Public abstract class DfsConsumer
{
    Protected QueryService.QueryResult m_result;

    Protected abstract void processResults();

    Public void executeDfsMethod(parameters)
    {
        Using (OperationContext ocx = OperationContext.Current())
        {
            DFS calls…
            processResults();
        }
    }

    Public T getResults()
    {
        //load data from m_result into primitive data types
When to use it
Use this pattern when there is a requirement to minimize the number of servers in an IT environment.

Examples:
I implemented an example on http://www.codeplex.com/documentum.

The requirements to make the example work are:
Content Server 6.5 SP2.
DFS 6.5 SP2.
Microsoft.NET 3.5.

**Force new content imports to be only of custom types.**
Previously, the Broadcasting Object Model pattern was introduced for broadcasters using Documentum. They implemented it to set things up for long term corporate dependence on Documentum as an Enterprise Content Management system. However, use of the pattern must be enforced. For ease of system maintenance, it is important to ensure that users always create new content within the repository as one of the custom types. However, this must also be allowed to pick another type in case the custom type does not provide the attributes or desired functionality. This solution can be summarized as the Default Object Type pattern.

**Default Object Type Pattern**

**Executive Summary**
This solution requires custom application development resources but reduces the possibility of bad or unusable data being entered into Documentum.
Technical Details

Problem
Users of Documentum, especially new users, are not fully aware of all the features and functionality of this Enterprise Content Management system. Even with user training, they may not be aware of the importance of setting the correct object type for the content they are inserting into Documentum.

Solution
Customize the front end interface the users will be using to import content into Documentum. For broadcasters, the most likely interface to be used is Digital Asset Manager (DAM).

Description
Since any user in a broadcasting organization may be using Documentum, there will be a wide range of different technical competencies with the userbase. Some users, even with intensive hands-on training, may not be able to fully grasp all the fundamentals of Documentum usage immediately. Therefore, it is important that when these users start importing their content into the system, they are given relevant defaults.

Functional How-to

Customize import so that a custom type is always the default type to import an object:

Although this high-level description assumes the user is using Tomcat as their Java application server, very similar directory structures should exist for WebSphere, WebLogic, and other Java 2 Enterprise Edition (J2EE) servers.

Go to:
$CATALINA\webapps\dam\webcomponent\config\library\contenttransfer\importcontent

Copy httpimport_component.xml and import_component.xml to
$CATALINA\webapps\dam\custom\config\webcomponent\config\library\contenttransfer\importcontent
Import_component.xml
Update the <document-docbase-base-type> and <document-docbase-type> nodes so the text values are the base custom type of the object model in the repository.
Update the <folder-docbase-type> to the base custom folder type of the object model in the repository.

Httpimport_component.xml
Update the <document-docbase-type> to the desired custom type.

Copy ucf.server.config.xml from $CATALINA/webapps/wdk/WEB-INF/classes to $CATALINA/webapps/dam/WEB-INF/classes.

**Process for Documentum Deployments**
Documentum is a highly complex system, and to rush implementations, upgrades, or customizations would be foolish. All Documentum projects should involve a project manager and business stakeholders, in addition to technical resources. Maintain development and staging environments of Documentum that match the production implementation to improve the quality and efficiency of the release process. It is also extremely important that all changes to Documentum are properly documented, from installing upgrades to custom code, so that changes can be very quickly rolled back and examined if need be. Documentum customizations can become very complex and IT operations will require detailed documentation on what the customizations are in order to troubleshoot and, if necessary, re-deploy. Taking this into consideration, it is prudent to include EMC-certified Documentum resources as part of the IT team of the broadcaster investing in Documentum.
**Suggested Order of Implementation of Solutions**

The sequence of solution implementation becomes relevant when the broadcaster cannot afford to invest millions of dollars on an Enterprise Content Management system. In this scenario, probably the average as opposed to the exception, it is only feasible to roll the system out in phases. Which solutions are provided in which phase has a major impact on user acceptance and ultimately, the success of the rollout. Here are some recommended phases and what solutions to provide in the phase.

**Phase 1 (Initial rollout)**

This phase involves rollout of the basic system components, mainly the content server, Digital Asset Manager (DAM), Documentum Administrator (DA), and Documentum Composer. The Broadcasting Object Model pattern and the Default Object Type pattern are recommended solutions for this phase.

**Phase 2**

The second phase focuses on some important productivity gains for the broadcaster. In this phase, I recommend that the broadcaster deploy the Traffic System Integration pattern and Cuesheet Translation pattern. Any type of small change that would appease the user base should be considered here. Some prime examples include the case-insensitive login functionality as well as suppressing the properties page popup.

At this point, future changes should be rolled out in phases conducive to the requirements of the particular broadcaster. Different requirements normally mean different priorities, so it is up to the individual broadcaster's senior management team to determine the next logical steps. Phase 1 and 2 are mentioned simply because the functions suggested for rollout should be generic and very similar to the requirements of most broadcasters. Obviously every broadcaster is different, so I recommend that you gather detailed requirements before any major decisions are made, as the suggested implementation schedule may not necessarily suit every broadcaster’s needs.
Summary
This article addressed some of the basic and immediate problems confronted when introducing Documentum to the broadcasting environment, as well as tried solutions. Not all broadcasters will have the problems discussed here, but this article provides a good starting point for senior management when considering how to integrate Documentum into the broadcasting workflow. It focused primarily on the technical aspects of a Documentum implementation, but there are almost always political and social aspects as well. Introducing a system like Documentum will bring about massive change, and user acceptance of the change should not be taken lightly. It is extremely important that the implementation be thoroughly planned, and a high degree of importance be given to customer satisfaction. Part of the rationale behind understanding the user base section was to offer suggestions on how to quickly adapt Digital Asset Manager (DAM) to satisfy some users, and therefore speed customer acceptance of Documentum.

This article contains solutions for problems that are most likely common for broadcasters. Still, it is wise to remember that every business, even those within the same industry vertical, is very unique. Don’t adapt the suggestions word for word, but rather take the time to gather user requirements and understand business needs first. Then, adapt the solutions in this article to meet those requirements. The most important voice in any business problem is the customer, so ensure that you listen to that voice before implementing anything.

System performance always weighs heavily on users’ perspectives on the success of an IT implementation. Therefore, in every quality assurance check during a Documentum rollout, performance in basic user operations should be tested to ensure any changes to Documentum, be it the object model or custom code, do not negatively impact the user experience. Always time basic DAM operations such as searches or browsing the contents of a folder, so there are benchmarks. Compare them against the time to execute similar queries in a relational database server.

Finally, approaching Documentum projects with the right processes and procedures will ensure that the basic necessary quality controls are in place, so that customers will view the system as an improvement to their work life, as opposed to a burden. Processes
ensure quality control, which is very important when rolling out a complex system like Documentum, even if efficiency must be somewhat sacrificed.

Customized Documentum systems are involved and tedious but, if managed properly, they can provide a powerful media asset management solution that can empower the broadcaster with a competitive advantage over others who have not invested in media asset management.

**Bibliography**