

Entwicklung einer interaktiven Oberfläche zur plattformunabhängigen Präsentation von annotierten Videos

Schriftliche Hausarbeit
vorgelegt im Rahmen der ersten Staatsprüfung für das
Lehramt an Gymnasien und Gesamtschulen in Informatik

von

André Hoffmann

Paderborn, den 8. September 2008

Gutachter: Prof. Dr. Johann S. Magenheimer

Universität Paderborn

Fakultät EIM, Didaktik der Informatik

Inhaltsverzeichnis

1	Einleitung.....	3
1.1	Projektidee	3
1.2	Anwendungsfälle für den <i>ViLM</i> -Player.....	5
2	Hauptteil.....	7
2.1	Theoretische Aspekte zum Lernen mit dem Medium Video.....	7
2.1.1	Lerntheorien.....	7
2.1.2	Eigenschaften des Mediums Video	9
2.2	Technologische Basis für die Entwicklung des <i>ViLM</i> -Players	11
2.2.1	Adobe Flex.....	11
2.2.2	Actionscript und MXML.....	12
2.2.3	XML.....	13
2.2.4	CSS	14
2.3	CSS-Layout- und Designmöglichkeiten mit MXML.....	16
2.4	Video-Streaming.....	18
2.5	Programmierkonzepte.....	19
2.5.1	Model View Control.....	20
2.5.2	Ereignisbasierte Programmierung.....	21
2.5.3	Objektorientierung.....	23
2.6	Analyse	28
2.6.1	Anforderungen an den Player	28
2.6.2	Gestaltungskriterien der DIN EN ISO 9241	29
2.6.3	XML im <i>ViLM</i> -Player	30
2.7	Modellierung.....	31
2.8	Entwurf.....	33
2.8.1	Unterschiedliche Ansätze	33
2.9	Implementierung.....	35
2.9.1	Auslesen von XML (E4X)	37
2.9.2	Grundsätzliches zur Umsetzung	40
2.9.3	Video-Initialisierungsprozess.....	41
2.9.4	Ursachen der Nicht-Synchronität	45
2.9.5	Prozeß des Synchronisierens.....	46
2.9.6	Implementierung der Dragging-Funktionalität	47
2.9.7	Einbindung des Aperture-Frameworks	49
3	Schlussbemerkung.....	50
	Literaturverzeichnis.....	52

1 Einleitung

1.1 Projektidee

In Rahmen dieser Examensarbeit wurde eine Videoplayer-Software zur synchronen Präsentation von annotierten Videos entwickelt. Dies geschah aufbauend auf dem Projekt *ViLM* „Visualization of Learning and Teaching Strategies with Multimedia in Teacher Education“ (Magenheim 1999) der Fachgruppe Didaktik der Informatik an der Universität Paderborn, dass bei der Lehrerausbildung zum Einsatz kommt.

Angehende Informatiklehrer müssen ein Fachpraktikum innerhalb der Veranstaltung „Methoden des Informatikunterrichts“ absolvieren. Dabei entwickeln die Studenten eine Unterrichtsreihe oder – stunde, die sie dann an einer Schule im Kreis Paderborn halten. In einer oder mehreren ausgewählten Stunden wird das unterrichtliche Geschehen dabei mit Videokameras gefilmt. Hierbei kommen mehrere Kameras zum Einsatz, um verschiedene Perspektiven eines Unterrichts festzuhalten. Meist wird eine Kamera so positioniert, dass die Lehrperson und ihr überwiegender Arbeits- und Aufenthaltsbereich in der Klasse (Lehrerpult, Lehrercomputer, Tafel etc.) aufgenommen werden kann. Eine zweite Kamera filmt das Verhalten der Schüler. Die so entstandenen Videos sollen den Studenten im weiteren Verlauf des Seminars zur Evaluation ihres Unterrichts dienen. In dieser Situation ist es erforderlich, die aufgenommenen Videos parallel und synchronisiert betrachten zu können, um die unmittelbare Aufeinanderfolge von Lehrer- und Schülerverhalten möglichst realgetreu und zeitgleich wiederzugeben.

Deshalb wurde in der Studienarbeit „Markierungs- und Schnittwerkzeug zur videounterstützten Analyse von Kommunikationsszenarien“ aus dem Jahre 2006 von Oliver Buschjost ([Buschjost, 2006](#)) das Tool *ViLM* des gleichnamigen Projektes neu entwickelt. Es enthält eine Software zur Synchronisation von zwei Videos, die es ermöglicht, Synchronisationszeitpunkte für jedes Video festzulegen und damit das Playback so zu justieren, dass beide Videos zu jedem Zeitpunkt die gleiche Situation zeigen. Als Produkt dieser Software entsteht dabei ein Projektordner mit den Videodateien, sowie Daten über deren Synchronisationszeiten. Er ist die Basis für eine weitere Software, die innerhalb dieser Studienarbeit entwickelt wurde, nämlich ein Annotationstool. Darin sind die Videos nicht mehr unabhängig verfügbar, sondern werden als synchrone Einheit bearbeitet. Das Annotationstool ermöglicht das Betrachten der synchronen Videos und das Setzen von zeitlichen Markierungen, um die Videos in Phasen zu gliedern, wobei zu jeder Phase mindestens ein Name, sowie ihr Start- und Endzeitpunkt gespeichert wird, sich zudem aber noch eine Beschreibung und ein Kommentar hinzufügen lassen. Beide Tools werden im Seminar „Methoden des Informatikunterrichts“ eingesetzt.

Die Studenten sollen ihren selbst gehaltenen Unterricht evaluieren und hinsichtlich der vorher definierten Ziele reflektieren. Mittels der synchronen Wiedergabe mehrerer Perspektiven des Klassengeschehens lassen sich parallele Prozesse des Unterrichts untersuchen, die erst auf diese Weise sichtbar und bewusst gemacht werden können. Innerhalb des Annotationstools gliedern die Studenten ihre Videos dazu in Phasen, entsprechend der tatsächlichen Unterrichtsabschnitte. Diese Phasen können nun hinsichtlich der in der Unterrichtsplanung definierten Ziele mit einer Beschreibung versehen werden und die Realisierung dieser Ziele dann zu jeder Phase zusätzlich kommentiert werden. Dadurch entsteht für den Studenten ein differenziertes Bild seines eigenen Unterrichts und er kann eine begründete, kritische Selbsteinschätzung vornehmen, da er in einer so gestalteten Reflexionsphase die eingeschränkte Sicht als Lehrperson einer Klasse verlassen kann und stattdessen eine Metaperspektive einnimmt.

Als Produkt des Annotationstools entsteht somit ein Paket von zwei synchronisiert spielenden Videos zu einem Unterricht aus verschiedenen Perspektiven, das zusammen mit den eingebetteten Informationen multiple Aspekte der Unterrichtsdurchführung und Unterrichtsreflektion wiedergeben soll. Zusätzlich können in diesem Paket noch Materialien der Unterrichtsplanung und -durchführung gespeichert werden, so dass sich ein einmal gehaltener Unterricht verhältnismäßig detailgenau konservieren lässt.

Bisher waren das Synchronisations- und Annotationstool auf zwei Videos beschränkt, deshalb wird diese Software derzeit in einer weiteren Arbeit auf den Umgang mit beliebig vielen Videos erweitert. Zusammengefasst bedeutet dies, dass nun zwei Tools existieren, mit denen sich mehrere Videos synchronisieren lassen und sich über den zeitlichen Ablauf Phasen festlegen lassen, in denen sich relativ beliebige Informationen als Beschreibung, Kommentar oder angehängte Materialdateien einbinden lassen.

Um diese Medien dann allerdings betrachten zu können, musste ein Nutzer bisher wenigstens das Annotationstool lokal installieren und auch den Projektordner in dem sich die Videodateien, die Projektdatei und auch die Materialdateien befinden, lokal auf seiner Festplatte haben. Zudem ist diese Software nur auf Windows lauffähig. Daher existierte der Wunsch nach einer plattformunabhängigen Videoplayer-Software, mit der ein Nutzer eine im Folgenden „VILM-Projektdatei“ genannte Datei öffnet, die alle erforderlichen Daten über ein Paket von synchronisierten und annotierten Videos enthält, und auf diese Weise die synchronisierten Videos zusammen mit den eingebetteten Informationen betrachten kann. Die Videos sollen dabei als Streams geladen werden, so dass der lokale Rechner des Nutzers nicht durch große Datenmengen der Videos überfrachtet wird. Der Videoplayer soll dabei eine plattformunabhängige Desktopanwendung sein, die unabhängig vom Synchronisations- und Annotationstool ist. Für eine

spätere Reflexion existiert dann eine kompakte Software für Zuhause, die eine komfortable Rekapitulationsmöglichkeit der Unterrichtsstunde für den Lehramtsstudenten bedeutet.

1.2 Anwendungsfälle für den ViLM-Player

Im Seminar „Methoden des Informatikunterrichts in Theorie und Praxis“ ist die unterrichtspraktische Durchführung einer oder mehrerer Unterrichtsstunden in einem realen Informatikunterricht ein zentraler Bestandteil. Um einen erfolgreichen Lerneffekt von Unterrichtspraxis bei den Studenten zu garantieren, muss die Durchführung dieses Unterrichts im Nachhinein ausführlich analysiert und reflektiert werden. In der Unterrichtsvorbereitung einer zu haltenden Stunde müssen neben didaktischen und fachlichen Fragestellungen auch vielfältige soziale und psychologische Aspekte berücksichtigt werden. So steht der Lehrer in einer sozialen Beziehung zu den Schülern und diese wird durch das gegenseitige Verhalten permanent modelliert. Deswegen muss sich der Lehrer seines eigenen Verhaltens bewusst werden und hinsichtlich der dadurch in der Klasse hervorgerufenen Reaktionen der Schüler adaptieren. Solche Verhaltensaspekte sind z.B. die Organisation des Gruppenverhaltens durch den Lehrer, die Art wie Anweisungen an die Schüler gegeben werden, wie der Lehrer die Schüler motiviert, Unterrichtsgespräche lenkt, Gruppen in Arbeitsphasen betreut, wie er bei Ausnahmesituationen in der Klasse interveniert oder ob die eingesetzten Medien der Erreichung der Lernziele dienlich sind u.a. (vgl. Arbeitsblatt „Gesichtspunkte zur Beobachtung von Unterricht“, 2006, FG Didaktik der Informatik, Universität Paderborn). Die Anpassung dieses Verhaltens erfolgt aber gerade bei ungeübten Lehrern aus einer sehr beschränkten und subjektiven ICH-Perspektive. Um dies wenigstens im Nachhinein ausgleichen zu können und den Studenten eine bessere Selbsteinschätzung der Unterrichtssituationen zu ermöglichen, ist eine ausführliche Reflektion des Unterrichts zwingend geboten. Der Unterricht muss dann aber möglichst vollständig wiedergegeben werden können, deshalb bietet sich die Videoanalyse mit verschiedenen Perspektiven innerhalb einer Klasse an.

Die Idee, ein Videoanalysetool bereit zu stellen, das synchrone Wiedergabe mehrerer Videos und das Einbetten diverser Informationen ermöglicht, lässt sich auch auf andere Anwendungsfälle übertragen. In Bewerbungsgesprächen zwischen Stellenbewerbern und Firmen liegt eine Situation vor, in der ein Bewerber sehr genau und akkurat auf Fragen durch einen Personalbeauftragten eines Unternehmens reagieren muss und dabei hinsichtlich diverser Aspekte wie seiner Mimik und Gestik beobachtet wird. Sicherlich wäre es für die Schulung von Bewerbungsgesprächen sinnvoll, die Perspektive des Personalbeauftragten und die des Bewerbers gleichzeitig auf Video festzuhalten und nachfolgend zu analysieren. Ein weiterer Anwendungsfall, bei dem nur eine Perspektive notwendig

ist, aber eingebettete Informationen sinnvoll sind, ist gegeben, wenn eine Person einen Vortrag oder ein Referat halten muss. Mit dem skizzierten Videoanalysetool ließen sich beispielhaft Mustervideos zu Vorträgen zeigen, die dann an exponierten Stellen wichtige Informationen anzeigen, warum der Vortragende an einer Stelle eine bestimmte Handlung vollzieht etc. Auch bei Verkaufsschulungen könnte ein solches Videoanalysetool zum Einsatz kommen. Dort könnte man die Perspektive des Verkäufers und die eines potenziellen Käufers auf Video festhalten und dann analysieren, welche Gesprächsinhalte oder Verhaltensweisen des Verkäufers positive Reaktionen beim Käufer hervorrufen. Hier könnten auch mehr als zwei Videos zum Einsatz kommen, denn mit einer Kamera würde man den Verkäufer filmen, um seine Mimik und Gestik festzuhalten, eine zweite Kamera fokussiert dagegen den Verkaufsgegenstand, an dem vielleicht etwas demonstriert wird und die dritte Kamera filmt schließlich den potenziellen Käufer. Es gibt also offensichtlich mehrere Anwendungsfälle für ein Videoanalysetool mit den beschriebenen Features.

2 Hauptteil

2.1 Theoretische Aspekte zum Lernen mit dem Medium Video

2.1.1 Lerntheorien

Unter dem Aspekt, dass der *ViLM*-Player in der Lehrerbildung für Studenten eingesetzt werden soll, ist es naheliegend, Anforderungen an diese Software auch unter lerntheoretischen Aspekten zu untersuchen. Wie bereits beschrieben wurde, soll die Software u. a. dahingehend zum Einsatz kommen, dass eine möglichst vollständige Aufzeichnung von unterrichtspraktischen Stunden von Lehramtsstudenten vorgenommen wird und diese mit Hilfe des *ViLM*-Tools und des *ViLM*-Players im Nachhinein analysiert und evaluiert wird. Welche Zwecke dies erfüllen soll und was das für Konsequenzen für die zu entwickelnde Software hat, soll im Folgenden beschrieben werden. Dafür möchte ich zunächst einige Theorien des Lernens vorstellen, aus denen man Grundlagen für das Lernen mit hypermedialen Systemen ableiten kann.

2.1.1.1 Kognitivismus

Die kognitive Psychologie geht auf die Theorien von Jean Piaget ¹ und Jerome S. Bruner ² zurück, nach denen Lernen auf kognitiven Strukturen beruht und durch kognitive Prozesse ständig modelliert wird. Bei Piaget resultiert diese Vorstellung aus einer genetisch-epistemologischen Theorie, dass Organismen sich ständig an ihre Umwelt anpassen. Diese Beobachtung biologischer Evolution wird auch auf kognitiver Ebene fortgesetzt, weil sich die Entwicklung eines Individuums durch Austauschprozesse mit der Umwelt permanent reguliert, die Piaget als „Akkommodation“ (Anpassung kognitiver Strukturen an neue pragmatische Gegebenheiten) und „Assimilation“ (Aufnahme externer Objekte und Zustände in die innere Struktur eines Individuums unter Modifikation vorhandener kognitiver Strukturen) bezeichnet. Das Ziel dieser Prozesse ist nach Piaget die Herstellung eines Gleichgewichts in der Auseinandersetzung eines Individuums mit seiner Umwelt. Aus diesen Fundamenten der kognitiven Psychologie hat Bruner das pädagogisch-methodische Konzept des entdeckenden Lernens entwickelt. Es impliziert Lernvorgänge, die sich nach der Heuristik menschlichen Denkens richten (Versuch-und-Irrtum), aber auch konzeptgeleitete Denkprozesse, sowie konstruktivistisches Problemlösen. Ziel dabei ist es, vorhandene kognitive

¹ Jean Piaget (* 9. August 1896 in Neuchâtel; † 16. September 1980 in Genf) war ein Schweizer Entwicklungspsychologe, der mit seinen Arbeiten Grundlagen der konstruktivistischen Lerntheorie gelegt hat.

² Jérôme Seymour Bruner (* 1. Oktober 1915 in New York) ist ein Psychologe, der konstruktivistische Lerntheorien vertritt. In der Pädagogik geht zum Beispiel das Konzept des Spiralcurriculums auf ihn zurück.

Konzepte zu aktivieren und Neue zu entwickeln. Daraus leitet man beim entdeckenden Lernen Aufgabentypen ab, die dem Suchen, Probieren und Explorieren weiten Raum geben, sowie die Simulation kognitiver Prozesse ermöglichen. Es kommt darauf an, kognitive Strukturen durch Effekte von Akkomodation und Assimilation zu erweitern, indem dem Lernenden die Freiheit gegeben wird, Wege und Strategien im Umgang mit Problemen selbst wählen zu können.

2.1.1.2 Konstruktivismus

Die Theorie der Kognition bildet zugleich die psychologisch-philosophische Grundlage für den Konstruktivismus, der die Genese des Wissens von den Dingen beschreibt. Zentral darin ist die Aussage, dass Wissen kein Abbild einer objektiv existierenden, externen Realität ist, sondern im Akt des Erkennens konstruiert wird. Es existiert nicht objektiv und unabhängig vom erkennenden Subjekt, ist nicht fest gespeichert, sondern wird stattdessen dynamisch generiert und kann darum auch nicht an andere Individuen ohne einen Prozess der Rekonstruktion übermittelt werden. Daraus ergibt sich, dass Wissen keine feste Repräsentation der Dinge ist. Wissen stellt sukzessiv neuinterpretierte Darstellungsformen von Sinneseindrücken dar. Daraus resultiert, dass es immer mehrere Möglichkeiten der Konstruktion von Wissen gibt, so dass für jeden Lernenden die aktive und individuelle Auseinandersetzung mit Aufgaben notwendig wird. Daher dürfen Lerninhalte nicht als objektive, isolierte Wahrheiten in Form von Lernzielen mit bestimmten kognitiven Konzepten vermittelt werden, sondern müssen in ihre jeweiligen Kontexte eingebunden sein. Unter diesen Prämissen konstituiert sich der Lernprozess dann aus den Bestimmungen des Erkenntnisprozesses. D.h. Lernen entwickelt sich durch Handeln und Handeln geschieht immer in sozialen Situationen, so dass jeder Lernprozess situativ, also in einen Kontext eingebunden sein sollte. Nicht Instruktion, sondern Selbstreflexion und Selbstregulation zählen zu den entscheidenden Merkmalen konstruktivistischer Erkenntnistheorie (Schulmeister 1996, 67-80). Daraus ist z.B. das Konzept des „Cognitive Apprenticeship“ entstanden. Es impliziert ein Lernen in einem Meister-Lehrlingsverhältnis, das in einen sozialen Kontext eingebunden ist. Dabei findet das Lernen sukzessive über die Phasen des *modeling* (Beobachtung des Meisters durch den Lehrling mit dem Ziel einer Modellbildung), des *coaching* (Selbstständiges Üben des Lehrlings unter Beratung des Meisters) und des *fadings* (Hilfestellungen des Meisters werden zunehmend zurück genommen) statt (Collins, Brown und Newman 1989).

Bezogen auf Lernvorgänge für die Unterrichtspraxis von Lehramtsstudenten kann man auf Basis dieser Theorien festhalten, dass es zweckmäßig ist, Wissen über Unterrichtsmethoden und unterrichtliche Abläufe nicht als starres Wissen zu vermitteln, sondern kognitive Strukturen und Vorerfahrungen die die Studenten in diesem Bereich vielleicht schon mitbringen, zu adaptieren und

günstig zu modellieren. Nach der konstruktivistischen Grundposition konstituiert der Lernende sein Wissen als einen dauernden Akt des Erkennens in einem bestimmten Kontext. Dieses Erkennen findet nicht allein in bestimmten Handlungssituationen vor Ort, also in der Schule oder Universität statt, sondern auch in Evaluations- und Reflexionsphasen danach. Im Fall von unterrichtspraktischen Erfahrungen ist das Lernobjekt aber sehr umfang- und facettenreich. Daher benötigt man für die Evaluation von Unterrichtsstunden eine möglichst detailgenaue und realistische Reproduktionsmöglichkeit. Darin liegt der Wunsch einer Videoanalyse begründet. Der *ViLM*-Player kann dazu dienen, die im Annotationstool gemachte Reflexion zu einem späteren Zeitpunkt erneut anzuschauen und der Erfahrungen, die man dabei gesammelt hat, wieder bewusst zu werden. Aus dieser ersten Analyse ergibt sich für den *ViLM*-Player, dass er möglichst viele Informationen des annotierten unterrichtlichen Geschehens gleichzeitig darstellen sollte. Das betrifft nicht nur die Synchronität aller laufenden Videos, sondern auch die parallele Einblendung von Phaseninformationen zu bestimmten Abschnitten des Unterrichts.

2.1.2 Eigenschaften des Mediums Video

Das Medium Video steht in diesem Fall für digital übermittelte bzw. gespeicherte Filmaufnahmen. Die Kombination von Codierungsarten³ und angesprochenen Sinnesmodalitäten⁴ ist die gleiche wie beim Medium Film. Video ist in diesem Fall ein digitales Medium, da es hier in Form von Daten gespeichert und übermittelt wird und moderne Technologien, wie das Videostreaming zum Einsatz kommen. Für das Seminar „Methoden des Informatikunterrichts in Theorie und Praxis“ stellt der *ViLM*-Player einen Teil einer E-Learning-Umgebung dar. Im Folgenden werden kurz die Funktionen des Mediums Video dafür skizziert.

Video stellt eine multimediale Repräsentationsform für Informationen dar und weist gegenüber rein bild- oder textbasierten Formen der Informationsvermittlung eine Reihe von Vorteilen auf. Ein Video ist als Darstellungsmedium nur Sprache oder nur Text weit überlegen, was die Informationsdichte angeht. Für die Umsetzung des *ViLM*-Players impliziert dies die grundlegende Eigenschaft, dass er vor allen Dingen ein Videoplayer ist und das Hauptaugenmerk auf der Präsentation der Videos liegt. Deshalb sind alle Bildelemente dieser Software ausblendbar, bis auf die Videopanel. Für das Medium Video ist darüber hinaus der Zeitbezug ein entscheidender Vorteil, denn jeder dargestellte

³ Der Begriff Sinnesmodalität bezeichnet die angesprochenen Sinneskanäle eines Mediums. Unterschieden werden auditiv und visuell-statisch (Bild), sowie visuell-dynamisch (Film) (Tulodziecki und Herzig 2004, 36).

⁴ Der Begriff Codierungsart bezeichnet das Symbolsystem, auf dem die in einem Medium dargestellten Inhalte basieren. Unterschieden werden ikonische und symbolische Formen. Eine ikonische Codierung ist abbildhaft und kann realgetreu oder schematisch sein. Eine symbolische Codierung kann in verbaler oder nicht-verbaler Form vorliegen (ebda.)

Vorgang lässt sich hierbei in Echtzeit abbilden. Dadurch wird es möglich, Abläufe und Handlungen zu verstehen, die zeitgebunden und dynamisch sind. Daher ist die deutliche Kennzeichnung der Phasen während dem Playback eine ebenso wichtige zu realisierende Funktionalität des *VILM*-Players. Das Video eines Vorgangs stellt die beste Möglichkeit dar, Realität abzubilden und damit vielfältige Facetten, wie Stimmungen und Gefühle, Verhaltensweisen und Ausdrücke, sowie die dadurch hervorgerufenen Reaktionen zu konservieren. Die Darstellung gleicher Information durch verschiedene Symbolsysteme führt zu einer kognitiven Summation, d.h. zur Generierung multipler, modalitätsspezifischer mentaler Repräsentationen (Niegemann 2004, 147-151). Ein Student, der sich mit Hilfe des *VILM*-Players die von ihm gehaltene Unterrichtsstunde zur Reflexion nochmals vor Augen führt, kann beispielsweise beobachten, welche Lernziele in einer Phase nicht erreicht wurden und dies mit seiner in der zugehörigen Phasenbeschreibung dargestellten Planung vergleichen. In dieser Situation liegt einerseits ein Text vor, der die Unterrichtsplanung beschreibt und andererseits eine filmische Aufnahme, die über Ton und Bild den tatsächlichen Ablauf dazu stellt. In einem möglichen Kommentar zur Phase können dazu noch didaktische Aspekte erläutert werden, die in diesem Zusammenhang von Bedeutung sind. Weitere Materialien, die zudem noch andere Sinnesmodalitäten ansprechen können, vertiefen so ein differenziertes Bild (zum Beispiel Links zu Webseiten oder Textdokument mit lehr-/lerntheoretischen Hintergründen oder didaktischen Aspekten). Auf diese Weise sind nahezu alle Facetten eines Unterrichtsaspektes für den Studenten nahezu gleichzeitig verfügbar und sehr gut nachvollziehbar. Man kann also sagen, dass die Ergänzung von textbasierten Lernangeboten durch authentische Audio- und Videobeispiele zu einem verbesserten Wissenserwerb und Wissenstransfer führt (ebda.). Da das Videoplayback jederzeit pausiert werden kann und man immer wieder zu bestimmten Phasen springen kann, kann ein Nutzer des *VILM*-Players die damit verbundene Informationsaufnahme an persönliche Lernziele, seine Lernstrategien und sein Lerntempo anpassen. Im Fall von unterrichtspraktischen Stunden ist der Betrachter der Videos möglicherweise auch die Lehrperson vor der Klasse. Auf diese Weise erfüllt der Player eine Motivierungsfunktion für den Studenten: wenn er merkt und beobachtet, welche planerischen Aspekte nicht umgesetzt werden konnten, steigt die Qualität der geistigen Auseinandersetzung mit dem Gezeigten und er wird versucht sein, die bewusst-gemachte Negativverfahren in Zukunft vermeiden zu wollen.

2.2 Technologische Basis für die Entwicklung des *ViLM*-Players

Als technologische Basis des *ViLM*-Players wurde frühzeitig die Adobe AIR Technologie gewählt. Sie stellt eine von Adobe kostenlos vertriebene Software für die Plattformen Mac und Windows dar, mit der Webanwendungen auf dem Desktop lauffähig gemacht werden. AIR Anwendungen lassen sich mit dem Adobe Flex Framework entwickeln, welches eine reichhaltige open-source Klassenbibliothek zur Entwicklung von Rich Internet Applications⁵ (RIA) bereitstellt und wesentliche Technologien der Adobe Flash Plattform vereint. Innerhalb von Flex werden über die XML-basierte Programmiersprache MXML mächtige Klassen für die Realisierung typischer Inhaltselemente von Flash-basierten Webanwendungen bereit gestellt, die in eigene Anwendungen eingebunden werden können. Das umfasst zum Beispiel Klassen, die Funktionalitäten im Bereich von Audio und Video ermöglichen (Brückmann und Butterer 2006).

2.2.1 Adobe Flex

Flex ist ein inzwischen von Adobe veröffentlichtes und kostenlos vertriebenes Framework auf der Basis von Flash zur Erstellung von RIAs. Zur Anwendungsentwicklung stellt Flex die Programmiersprachen MXML und ActionScript bereit. MXML bildet eine XML-basierte Programmiersprache zur Erstellung von Webanwendungen, die von der Firma Macromedia entwickelt wurde, inzwischen aber zu Adobe gehört und deswegen innerhalb des Flex Frameworks verfügbar ist. Flex implementiert die vollständige Klassenbibliothek der Flash Entwicklungsplattform, so dass Flex Anwendungen auf dem Flashplayer lauffähig sind. Flash ist eine ursprünglich von Macromedia stammende Technologie für animierte, multimediale Inhalte. Diese Inhalte werden in einem Vektorgrafikbasiertem Format, dem SWF-Format⁶ abgelegt und sind zur Anzeige in einem Webbrowser gedacht, der dafür noch das Flashplayer Plug-In benötigt. Seit der Version 4 besitzt die Flash Entwicklungsplattform einen Interpreter für die Programmiersprache ActionScript, die bis zur aktuellen Version 9 von Flash stark erweitert und verändert wurde (Webseite, Get oriented to Flex - Flex 3 Getting Started - Adobe Learning Resources 2008). Inzwischen integriert Flash ActionScript 3.0, das eine Spezialisierung der allgemeinen Skriptsprache ECMAScript⁷ darstellt und inzwischen neben seinem ursprünglich prozeduralen Charakter auch objektorientiert ist. Innerhalb des Flex-

⁵ Webanwendung, die gleichzeitig reichhaltige Möglichkeiten von Desktopanwendungen anbietet, wie z.B. Drag & Drop. Eine RIA kann auch zwischen Browser und Desktop angesiedelt sein, nutzt aber die Technologien des Internets und muss nicht installiert werden.

⁶ Vom Adobe Flash Player und Adobe Air benutztes Dateiformat, mit dem Vektorgrafiken, Text, Sound und Video über das Internet übertragen werden (Webseite, Adobe / Developer Connection / SWF Technology Center 2008).

⁷ Gemeinsamer Sprachstandard vieler Skriptsprachen, für den alle Typen, Werte, Objekte, Parameter, Funktionen und die Programmsyntax und -semantik unterstützt werden müssen (Webseite, Standard ECMA-262 2008).

Frameworks ist ActionScript unmittelbar verfügbar und wird beim Kompilieren einer Flex-Anwendung automatisch eingebunden. Auf diese Weise ermöglicht ActionScript die Integration und den Austausch der Bibliotheken von Flash und MXML und stellt somit eine Dynamisierung der MXML Elemente bereit, indem Objekte von MXML- oder Flash-Klassen durch ActionScript-Funktionen (im Folgenden als Methoden bezeichnet) gesteuert werden. Grundlage der Entwicklung mit Flex ist das Flex-SDK (Software Development Kit), das alle erforderlichen Bibliotheken und den Compiler enthält. Daneben gibt es den Flex Builder (aktuell in der Version 3), eine auf Eclipse⁸ aufsetzende, integrierte Entwicklungsumgebung, die dem Flex-Entwickler eine komfortable grafische Benutzeroberfläche und die Vorteile des Syntax-Check und der automatischen Code-Vervollständigung während der Entwicklung bietet (Webseite, Eclipse Tutorial 2008).

2.2.2 Actionscript und MXML

MXML ist eine XML-basierte Programmiersprache, die vorrangig das Erzeugen von Benutzeroberflächen mit typischen Inhaltselementen, die man von Flash her kennt, ermöglicht. Außerdem werden mittels MXML auch deklarative, nichtoptische Aspekte einer Anwendung, wie zum Beispiel Datenbindungen zwischen Oberflächenkomponenten und Datenobjekten realisiert. MXML-Befehle werden über XML-basierte Tags beschrieben, die über den Namensraum⁹ «mx» beim Kompilieren als solche erkannt werden. Dabei werden sie in Objekte von Actionscript-Klassen übersetzt, so dass die deklarativen Elemente noch angepasst werden können. Durch Vererbung der Originalklassen besteht zudem die Möglichkeit, neue MXML-Befehle auf der Basis bereits bestehender zu erstellen. Auch vollständig neue MXML-Befehle sind über andere Namensräume realisierbar. Die von der Flex-Anwendung erstellten Actionscript-Klassen werden anschließend in SWF-Bytecode kompiliert und in einer SWF-Datei gespeichert. Dieser Bytecode wird vom Flashplayer in einem AVM (Actionscript Virtual Machine) genannten Runtime-Environment ausgeführt (Webseite, Adobe - Programmieren mit MXML und ActionScript 2008). Insgesamt bietet MXML keine zusätzlichen Möglichkeiten zu Actionscript, vereinfacht aber den Schreibaufwand für das Erstellen einfacher multimedialer Inhalte in Actionscript erheblich. Wesentliche dafür geeignete MXML Elemente finden sich in Abschnitt 2.9.

Actionscript nimmt in Bezug auf MXML die Rolle des „Code Behind“ ein (Webseite, Adobe - Flex Quick Start Basics: Building components by using code behind 2008). Mit seiner Hilfe wird das

⁸ Eclipse ist eine Open Source Entwicklungsumgebung für verschiedene Konstruktionsziele (z.B. unterschiedliche Programmiersprachen) sowohl visueller als auch nicht-visueller Art, die plattformneutral und erweiterbar auf der Basis von Plug-Ins ist, sowie offen gegenüber diversen Formaten, standardisierten Verfahren und Softwarekomponenten (Assisi 2004).

⁹ siehe Abschnitt 2.2.3

Eventhandling der Oberflächenobjekte realisiert und daneben die Geschäftsprozesse im Hintergrund durch entsprechende Kontrollstrukturen realisiert. Actionscript-Sourcecode kann entweder direkt in eine MXML-Datei integriert oder in eigene Pakete gekapselt werden, die dann von entsprechenden MXML-Dateien eingebunden werden. Häufig wird dies in Form von kleineren Actionscript-Dateien «.as» realisiert, die bestimmte Codeabschnitte enthalten und über

```
<mx:Script source="dateiname.as"/>
```

innerhalb einer MXML-Datei eingebunden werden.

Eine weitere wichtige Anwendung von Actionscript ist durch die Unterstützung für *ECMAScript for XML*¹⁰ (E4X) gegeben. Dadurch ist es möglich, XML-basierte Datenquellen in eine MXML-Anwendung einzubinden, indem XML-Daten innerhalb von Actionscript ausgelesen und bearbeitet werden (siehe dazu Abschnitt 2.6.2).

2.2.3 XML

Die *Extensible Markup Language* (XML) ist eine Auszeichnungssprache, die ebenso wie viele andere Auszeichnungssprachen ihre Wurzeln bei der Standard Generalized Markup Language¹¹ (SGML) hat und grundsätzlich dazu dient, Informationen mit Typ und Struktur auszustatten. Während SGML ursprünglich dem standardisierten Austausch von elektronischen Dokumenten diente und daraus später HTML hervorging, um Dokumente standardisiert zu layouten, ist XML entwickelt worden, um dokumenteninterne Daten zu beschreiben und auszutauschen. Sie wurde in der aktuellen Version 1.0 vom World Wide Web Konsortium (W3C) standardisiert und veröffentlicht als ein Regelwerk, das die Syntax und die Semantik von XML und verwandten Technologien spezifiziert (Benz und Durant 2003). Kern der XML-Spezifikation ist das *XML Information Set* (<http://www.w3.org/TR/xml-infoset>), das die grundlegendsten Abstraktionen dieser Sprache beschreibt, die sogenannten Strukturellen Informationselemente, die eine oder mehrere benannte Eigenschaften haben. Semantische Details der darin beschriebenen Ausdrücke werden in der *Extensible Markup Language* (<http://www.w3.org/TR/REC-xml>) festgelegt. Mittels des *XML Information Set* wird jedes XML-Dokument mittels eines sogenannten *Document-Information-Items* modelliert.

¹⁰ Standard, der über einfache Syntax die XML-Verarbeitung in Skriptsprachen ermöglicht, die auf ECMAScript basieren (Webseite, Standard ECMA-357 2008).

¹¹ SGML stellt eine Metasprache dar, über die sich Auszeichnungssprachen definieren lassen. Dies betrifft das Vorhandensein bestimmter Datenelemente und eine festgelegte Syntax (Webseite, Cover Pages: Standard Generalized Markup Language (SGML) 2008).

Dabei fungiert das *Document-Information-Item* als Wurzel eines Baum-Graphen, der aus *Element-Information-Items*, *Processing-Instruction-Items*, *Kommentar-* und *Zeichen-Information-Items* besteht. Dabei sind die *Processing-Instruction-Items*, sowie die *Kommentar-* und *Zeichen-Information-Items* Blätter dieses Baumes und die *Element-Information-Items* stellen interne Knoten dar. In diesem Elementmodell gibt es weiterhin zwei Eigenschaften die die Beziehung zwischen Elementen festlegen. Zum einen die *parent*-Eigenschaft, die für jedes Element auf das enthaltende Information-Item zeigt und die *children*-Eigenschaft, die die geordnete Menge der direkten Nachfahren eines Elements im Graphen angibt. Das *Document-Information-Item* hat als Wurzel keine *parent*-Eigenschaft, während *Processing-Instruction-*, *Kommentar-* und *Zeichen-Information-Items* Blätter im Baum sind und dementsprechend keine *children*-Eigenschaft haben. Damit ist die Struktur von XML offen gelegt.

Das *Element-Information-Item* ist der komplexeste Datentyp in XML, der in der Praxis auch am häufigsten genutzt wird. Unterschieden werden können zwei Typen. Zum einen Elemente atomaren Typs, die also einen Wert enthalten, der in Zeichendaten kodiert wird. Zum anderen Elemente, die typischerweise keine *Zeichen-Information-Items* als *children* enthalten, sondern ein oder mehrere *child*-Elemente, die die Struktur festlegen. Man bezeichnet solche *Element-Information-Item* als „element-only“. Zur Modellierung traditioneller Datenstrukturen sind diese beiden Formen bereits hinreichend. XML erlaubt aber auch die Verschachtelung von *child*-Elementen und Zeichendaten. Jedes *Element-Information-Item* verfügt über eine *attributes*-Eigenschaft, die aus einer ungeordneten Liste eindeutig benannter *Attribut-Information-Items* besteht. Attribute dienen dazu, das Element, auf das sie sich beziehen, genauer zu charakterisieren. Der Zugriff auf ein Attribut eines Elements geschieht über einen eindeutigen Namen, den jedes Attribut besitzt. Attribute können durch *Element-Information-Items* ersetzt werden, da sie nur eine Untermenge von Elementen darstellen. Dies macht in einigen Situationen auch Sinn, da Attribute keine Elemente, Kommentare oder Processing-Instructions enthalten (Don Box 2001). Allerdings lassen sich im E4X-Standard Attribute sehr komfortabel auslesen, was sonst nur über Aufrufe an *child*-Elemente möglich wäre.

2.2.4 CSS

Cascading Style Sheets (CSS) bilden eine Sprache zur Definition von Stylesheets. Ein Stylesheet ist ein Block von Anweisungen, die die Darstellung eines Dokuments spezifizieren. Es formatiert Elemente in beliebig strukturierten Dokumentenformaten (zum Beispiel in Anwendungen von XML), indem es Formatierungsregeln für sie enthält, die entsprechend der CSS Grammatik geschrieben sein müssen. Eine CSS Regel besteht aus zwei Teilen, dem Selektor und der Deklaration. Eine Deklaration besteht

selbst wiederum aus zwei Teilen, nämlich Eigenschaft und Wert. Nachfolgend ist ein typischer Anweisungsblock dargestellt, der dies verdeutlicht:

```
iframe {
  align: top;
  background: #cccccc;
  frameborder: 1;
  width: 300;
  height: 200;
}
```

Wird das Stylesheet dieses Anweisungsblocks innerhalb einer HTML-Datei eingebunden, so wird damit über den Selektor «*iframe*» jedes *iframe*-Element so formatiert, dass es an der oberen Grundlinie seines *parent*-Elementes verankert ist, die Hintergrundfarbe hellgrau und eine Größe von 300 x 200 Pixel hat, sowie einen 1 Pixel starken Rahmen besitzt.

Die Selektoren eines Stylesheets können sich auf die folgenden Teile eines Dokuments beziehen:

- Elemente im Dokumentbaum und bestimmte Beziehungen zwischen ihnen
- Attribute von Elementen im Dokumentbaum und ihre Werte
- Teile des Elementinhalts (über Pseudo-Elemente¹²)
- Elemente im Dokumentbaum in einem bestimmten Status (über Pseudo-Klassen)

u.a.

Generell lässt sich sagen, dass Stylesheets strukturierte Dokumente (z.B. XML basierte Anwendungen) ergänzen und stilistische Informationen für den mit Markup ausgezeichneten Text bereitstellen. Ein Stylesheet ist änderbar ohne Einfluss auf das Markup. Stylesheets sind unabhängig von Hersteller und Plattform und erhöhen die Wartbarkeit von Software, da Dokumente nur auf sie verweisen. Stylesheets können verändert werden, ohne Einfluss auf das Markup zu haben. Die CSS-Eigenschaften eines Elements werden daher unabhängig verwaltet und es gibt im Allgemeinen nur eine einzige Regel, um einen bestimmten Effekt zu erzielen.

¹² Pseudoklassen und -elemente legen Style-Eigenschaften für HTML-Elemente fest, die sich in einem bestimmten Zustand befinden. So verfügt zum Beispiel ein Link <a> unter anderem über die Zustände "link" und "visited", die angeben, ob es sich um einen bereits besuchten Link handelt oder nicht. Über die Selektoren "a:link" und "a:visited" lassen sich damit unterschiedliche Styles für beide Zustände realisieren.

2.3 CSS-Layout- und Designmöglichkeiten mit MXML

MXML ist eine deklarative Sprache zur Gestaltung von Benutzeroberflächen innerhalb von Flex. MXML ermöglicht reichhaltige Oberflächen mit typischen Flash-Komponenten, die sehr schnell entwickelt werden können. Das Layouten und Designen der verschiedenen Elemente kann entweder mit Hilfe von CSS oder teils direkt in den MXML Tags erfolgen. Nutzt man CSS, ist auf diese Weise eine klare Trennung der Struktur der Tags und des Layouts möglich. Ein MXML-Objekt besitzt genau dann eine Formatierungsmöglichkeit für Styles, wenn seine zugehörige Klasse das «*IStyleClient*»-Interface implementiert oder von der Klasse «*UIComponent*» erbt, die dieses Interface implementiert. Dann sind die Style-Eigenschaften des Objekts genau über CSS ansprechbar. Dabei kann über den Elementnamen entweder die ganze Klasse dieser Objekte gewählt werden, oder man setzt für ein MXML-Element lokal das Attribut

```
styleName="myStyle"
```

und formatiert das Element vermöge des Stylesheet-Blocks

```
.myStyle { ... }
```

Da MXML-Elemente durch den Flex-Interpreter in Objekte von ActionScript-Klassen umgewandelt werden, lassen sich damit Style-Attribute auch innerhalb von ActionScript durch die Objektvariable «*styleName*» zuweisen. Am Beispiel des MXML-Elementes «*Panel*» werden dazu nun einmal typische Style-Attribute vorgestellt, mit denen sich das Aussehen dieser Komponente stark variieren lässt.

Tabelle 1: Style-Attribute des MXML-Elementes «Panel»

Attributname	Beschreibung
<code>backgroundAlpha</code>	Durchsichtigkeit der in « <code>backgroundColor</code> » gesetzten Farbe
<code>backgroundColor</code>	Hintergrundfarbe des Panels

<code>backgroundDisabledColor</code>	Hintergrundfarbe des Panels, wenn es über die Attributzuweisung « <i>enabled = false</i> » für Benutzereingaben gesperrt wurde.
<code>color</code>	Textfarbe in jedem Bereich des Panels
<code>cornerRadius</code>	Radius der Fensterecken des Panels
<code>borderAlpha</code>	Durchsichtigkeit eines Rahmens, der um das Panel gezogen wird
<code>borderColor</code>	Farbe dieses Rahmens
<code>borderStyle</code>	Linienstil diese Rahmens
<code>borderThickness</code>	Strichdicke dieses Rahmens
<code>disabledOverlayAlpha</code>	Durchsichtigkeit eines Overlays, das erzeugt wird, wenn die Komponente für Benutzereingaben gesperrt wird
<code>dropShadowColor</code>	Farbe für einen zusätzlich einblendbaren Schatten des Fensters des Panels
<code>dropShadowEnabled</code>	Boolean, um diesen Schatten ein- oder auszublenden
<code>fontFamiliy</code>	Textfamilie innerhalb des Panels
<code>fontSize</code>	Textgröße
<code>fontStyle</code>	Kursiveigenschaft für Text
<code>fontThickness</code>	Textdicke
<code>fontWeight</code>	Textformatierung
<code>footerColors</code>	Array, in dem zwei Farbwerte abgelegt werden, die den Fußbereich des Panels mit einem Farbverlauf füllen
<code>headerColors</code>	Array, in dem zwei Farbwerte abgelegt werden, die den Kopfbereich des Panels mit einem Farbverlauf füllen
<code>headerHeight</code>	Höhe des Kopfbereiches innerhalb des Panels
<code>highlightAlphas</code>	Array, durch das der Farbverlauf für den Hintergrund des Panels in zwei Bereiche aufgeteilt werden kann, die für sich nochmals einen Farbverlauf aufweisen. Dadurch entsteht ein dreidimensionales Aussehen
<code>horizontalAlign</code>	Boolean, ob <i>child</i> -Elemente innerhalb des Panels horizontal ausgerichtet werden

<code>horizontalGap</code>	Festlegung des Abstandes in Pixeln, die <i>child</i> -Elemente in horizontaler Richtung voneinander haben
<code>paddingLeft,</code> <code>paddingTop,</code> <code>paddingRight,</code> <code>paddingBottom</code>	<i>Child</i> -Elemente innerhalb des Panel-Elementes werden in einer sogenannten Content-Area platziert. Über diese Variablen lassen sich noch Abstände für die Content-Area zum Rand des Panels festlegen
<code>roundedBottomCorners</code>	Boolean, ob auch die Ecken des Fußbereiches des Panels abgerundete Ecken haben soll
<code>shadowDirection</code>	Richtung, in die der Drop-Shadow fällt
<code>shadowDistance</code>	Abstand des Drop-Shadow zum Fensterrahmen
<code>verticalAlign</code>	Boolean, ob <i>child</i> -Elemente innerhalb des Panels vertikal ausgerichtet werden
<code>verticalGap</code>	Festlegung des Abstandes in Pixel, die <i>child</i> -Elemente in vertikaler Richtung voneinander haben
<code>titleStyleName</code>	Style-Name für ein explizites Stylesheet, das den Titelbereich des Panels gesondert formatiert

2.4 Video-Streaming

Für den Transport von Videodaten über das Internet sind vor allen Dingen zwei Verfahren maßgeblich. Dies sind zum einen der progressive Download und zum anderen das Videostreaming. Beim progressiven Download wird eine Videodatei wie jede andere Datei im Netz behandelt, was bedeutet: Fordert ein Nutzer ein Video an, so findet eine Übertragung der entsprechenden Videodatei auf HTTP¹³-Basis zum Client statt, ohne dass ihre Eigenschaft, ein Video zu sein, spezieller beachtet wird. Demgegenüber steht das Videostreaming, bei dem die Videodaten durch einen Streamingserver über bestimmte Protokolle zum Client übertragen werden, wobei eine dynamische Anpassung an seine Verbindungsgeschwindigkeit vorgenommen wird, z.B. über die Reduzierung der Bildqualität (Bösken 2007). Die progressiven Downloads von Videos sind insbesondere durch Plattformen wie Youtube in Mode gekommen. Im Fall von Youtube ist der Client ein Flashplayer Plug-In im Browser, der die Video- und Audiodateien über HTTP herunterlädt (Petzold 2008). Videodateien liegen dabei im Flash-Video-Format vor, einem Containerformat für MPEG-4-

¹³ Hypertext Transfer Protocol – Protokoll um Daten in einem Netzwerk auf der Anwendungsschicht zu übertragen. Als unterliegendes Transportprotokoll wird meist TCP genutzt.

Videodaten von Adobe, das den standardisierten Videocodec H.264¹⁴ und zum Beispiel MP3 für Audiodaten nutzt. Es ist streamfähig, zum Beispiel über den von Adobe vertriebenen Flash Media Server. Gestreamte Flash-Videos bieten sich als Quelle für den *ViLM*-Player an, da sie durch die in Flex bereits implementierten Videofunktionalitäten voll unterstützt werden und zudem folgende Verwendung durch Adobe nahe gelegt wird:

Tabelle 2: Übermittlungsoptionen für Flash-Videos¹⁵

	Progressiver Download	Streaming
Clip ist unter 5 Sekunden lang	•	
Clip ist zwischen 5 und 30 Sekunden lang	•	•
Clip ist länger als 30 Sekunden		•
Wenig Zuschauer	•	
Einige bis viele Zuschauer		•
Unmittelbarer Start		•
Variable Datenraten in Abhängigkeit von der Bandbreite beim Client		•

Grundsätzlich ist anzumerken, dass sich das Flash-Video-Format sehr schnell verbreitet hat, nicht zuletzt weil es sehr flexibel ist und es durch den Flash Player, der eine hohe Verbreitung im Internet hat, vollständig integriert wird (Ozer 2007).

2.5 Programmierkonzepte

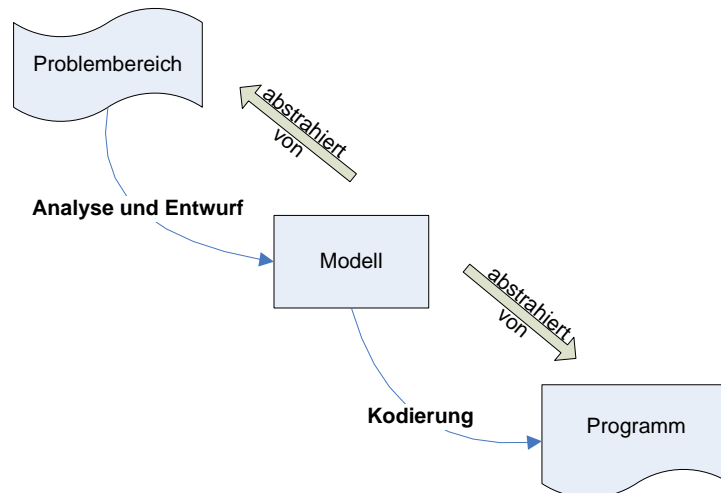
Am Anfang jeder Softwareentwicklung steht ein Problembereich, zu dem eine Lösung durch eine Software gefunden werden soll. Dabei ist ein zeitgemäßer Begriff von Softwareentwicklung nicht gleichzusetzen mit einer unmittelbaren Implementierung aus diesem Problembereich heraus. Stattdessen verfolgt man die Grundidee der modellbasierten Softwareentwicklung, bei der man vom Problembereich zunächst durch die Schritte der Analyse und des Entwurfs zu einem Modell des

¹⁴ Video-Codec von Apple, der offiziell im MPEG-4-Standard unterstützt wird. Liefert in etwa die Qualität von MPEG-2, bei etwa halbierten Datenrate (<http://www.apple.com/quicktime/technologies/h264/>)

¹⁵ Quelle: http://www.adobe.com/devnet/flash/articles/video_guide.html

Problembereichs kommt, dass eine Abstraktion des Selben ist. Von diesem Modell aus entsteht die Software dann durch Kodierung, als eine andere Konkretisierung des Modells.

Abbildung 1: Modellbasierte Softwareentwicklung



Durch modellbasierte Softwareentwicklung kann Rücksicht genommen werden auf bestimmte Konzepte in der Programmierung, die interne oder externe Softwarequalitäten positiv beeinflussen können. Dazu gehört zum Beispiel die Entscheidung für einen bestimmten Architekturstil zu Beginn einer Softwareentwicklung, aus dem heraus sich bestimmte, bewährte Pattern und Muster für die Implementierung ableiten lassen (Shaw und Garlan 1994). Eine solche Softwarearchitektur ist etwa die Schichtenarchitektur¹⁶. Dabei müssen die Verantwortlichkeiten, Aufgaben und Besonderheiten für jede Schicht beschrieben werden.

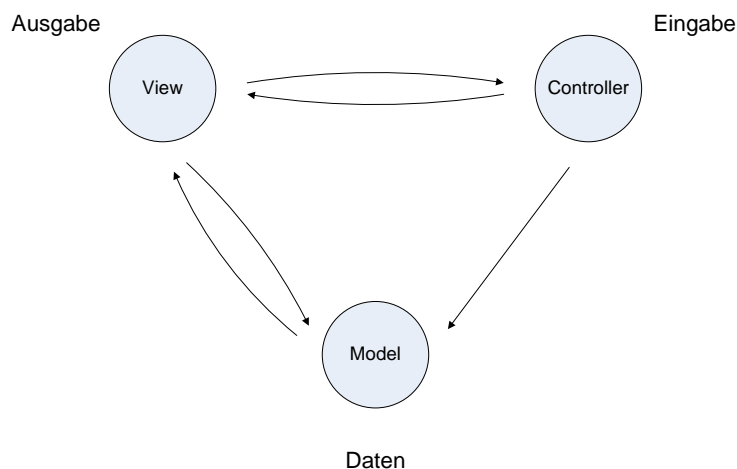
2.5.1 Model View Control

Das Konzept des Model-View-Control (MVC) beschreibt einen grundsätzlichen Architekturstil für eine Software. Dabei handelt es sich um das Prinzip, die Repräsentation von Daten (View), die Manipulation von Daten (Controller) und die Datenhaltung (Model) voneinander zu trennen. Der Vorteil bei dieser Vorgehensweise ist, dass es zu einem späteren Zeitpunkt sehr leicht möglich ist, die vorhandenen Daten neu zu visualisieren, in dem man einfach nur diejenigen Softwarekomponenten austauscht, die der *View-Schicht* zugeordnet sind. Die datenverarbeitenden Teile der Software sind

¹⁶ Festlegen von trennscharfen Schichten mit bestimmten grundsätzlichen Eigenschaften, denen alle Komponenten einer Software zugeordnet werden. Jede Komponente ist dabei genau einer Schicht zugeordnet und für Zugriffe und Abhängigkeiten gilt, dass Elemente der unteren Schichten auf Elemente der oberen Schichten zugreifen, aber nicht umgekehrt (Oestereich 2004, 158).

dabei entkoppelt und von dieser Änderung nicht betroffen. Ändert man den Controller oder das Model ab, verhält sich das natürlich vergleichbar. Im *Model* ist der aktuelle Zustand und das Verhalten des gesamten Systems repräsentiert. Es ist also gleichsam das Datenmodell einer Software. Der *Controller* stellt die Verbindung des Modells zur Außenwelt her, indem er eingehende Anforderungen (z.B. Eingaben von der Tastatur) verarbeitet, und das Model dazu veranlasst, die Daten entsprechen zu verändern. Aufgabe der View ist es, die Daten des Modells auf irgendeine Art darzustellen, wobei dies die einzige Aufgabe der View ist. Das Model und die View sind häufig eng verknüpft und man bezeichnet diese Verknüpfung oft als Benutzerschnittstelle oder UI (User Interface) bzw. GUI (Graphical User Interface). Sie bezeichnet daher den Teil einer Software, der vorhandene Daten auf einem Ausgabegerät visualisiert und dabei gleichzeitig Datenänderungen über Eingabegeräte entgegen nimmt. Insbesondere die Trennung der Controller-Schicht von den beiden anderen Schichten ist durch das Flex-Framework geradezu vorgegeben. In den als *Code-Behind* bezeichneten ActionScript-Methoden findet die eigentliche Geschäftslogik statt und die View wird häufig über MXML-Elemente im Zusammenspiel mit CSS realisiert.

Abbildung 2: MVC-Konzept



2.5.2 Ereignisbasierte Programmierung

Das Konzept der Ereignisbasierten Programmierung bezeichnet eine Programmiermethode, deren grundlegendes Konzept in der Document Object Model (DOM) Level 3 Events-Spezifikation¹⁷ beschrieben ist und die vor allen Dingen im Bereich der Benutzerschnittstellen von Bedeutung ist.

¹⁷ siehe: <http://www.w3.org/TR/DOM-Level-3-Events/events.html>

Kern dieses Konzeptes ist das Aufrufen von Operationen einer Softwarekomponente durch externe Ereignisse. Drei grundlegende Elemente sind dabei zu nennen: das Ereignis, die Ereignisquelle und der Ereignisempfänger. Ereignisse sind sogenannte Event-Objekte, die von den Ereignisquellen an die Ereignisempfänger verschickt werden, und zwar genau dann, wenn die dem Ereignis entsprechende Aktion eintritt. Dazu gibt es einen sogenannten Eventhandler, der vom unterliegenden Framework, einem Treiber oder dem Betriebssystem bereitgestellt wird, bei dem die eigenen Operationen registriert werden. Ereignisquellen sind häufig Buttons, Textfelder oder andere Elemente der Benutzeroberfläche, können aber auch programminterne Objekte sein, die dann über eine Zustandsänderung informieren. Bei ihnen melden sich die Ereignisempfänger für ein bestimmtes Ereignis an und werden immer dann benachrichtigt, wenn das Ereignis bei der Ereignisquelle auftritt.

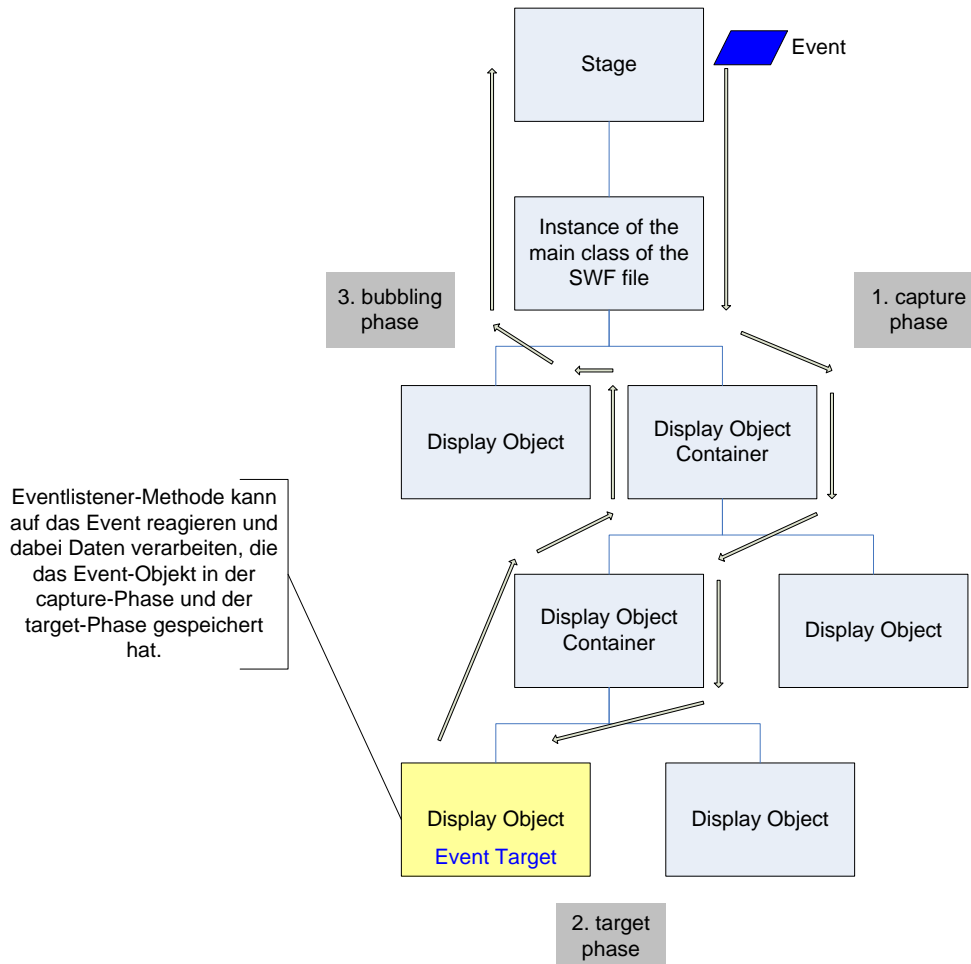
Programmtechnisch wird dieses Ereignismodell zum Beispiel in JAVA wie folgt realisiert: Ereignisse sind feste Objekte von sogenannten Event-Klassen. Einem Ereignis wird nun als Beobachter ein sogenannter Eventlistener zugeordnet, der eine Klasse darstellt, die bestimmte Methoden eines zu dem betreffenden Ereignis gehörenden Interface implementieren muss. Darin kann dann sinnvoll auf das Ereignis reagiert werden.

Die Implementierung des Eventmodells von Flash, das auf der DOM Level 3 Events - Spezifikation beruht, weicht davon etwas ab. Dabei wird jedes Ereignis durch ein Objekt repräsentiert, das eine Instanz der «*Event*»-Klasse oder eine ihrer Unterklassen ist. Ein Event-Objekt speichert dabei nicht nur Informationen über sich selbst, sondern auch Objektvariablen desjenigen Objektes, bei dem das Event aufgetreten ist. Im Flashplayer wird dieses Objekt als Event-Target bezeichnet. Event Targets sind Objekte von solchen Klassen, die von der «*EventDispatcher*»-Klasse erben, welche selbst das «*IEventDispatcher*»-Interface implementiert und damit die Basisklasse für alle Klassen darstellt, die Ereignisse auslösen. Die Klasse «*DisplayObject*», die für Klassen von UI-Elementen Superklasse ist, erbt selbst von der «*EventDispatcher*»-Klasse. Ein vom Framework in den sogenannten Eventflow geschicktes Event, wandert in der sogenannten „capture phase“ von der Wurzel der *DisplayList*¹⁸ bis zum Event-Target, wo es in der sogenannten „target phase“ über das Aufrufen einer «*addEventListener*»-Methode für den entsprechenden Event-Typ abgefangen werden kann. Diese Methode bekommt als Parameter zunächst das Event-Objekt auf das man reagieren möchte, übergeben. Als weiteren Parameter muss man darin eine „EventHandler“ genannte Methode angeben, die aufgerufen wird, wenn das entsprechende Ereignis auftritt. Sie bekommt dann als Parameter allein das Event-Objekt übergeben, welches aber selbst wiederum Eigenschaften und Zustände der Komponente speichert, in der es auftrat. In der „bubbling-phase“ wandert das Event

¹⁸ Der Begriff *DisplayList* beschreibt die hierarchische Anordnung von Elementen der Klasse «*DisplayObject*» in Flex. Dahinter steht ein auf dem Document Object Model aufbauende Baumstruktur.

dann über alle *parent*-Elemente zurück zur Wurzel (Rich Shupe 2008). Das folgende Modell verdeutlicht den Ablauf:

Abbildung 3: Event-Modell in Flex



2.5.3 Objektorientierung

Das Konzept der objektorientierten Programmierung fußt auf dem Paradigma der Objektorientierung. Dies ist eine grundlegende epistemologische Sichtweise von Wirklichkeit, bei der Gegenstände als Objekte aufgefasst werden und Objekte mit ähnlichen Eigenschaften in sogenannten Klassen zusammengefasst werden. Einzelne Entitäten solcher Klassen sind dann genau die Objekte einer Klasse. Die Variabilität von real existierenden Dingen, wird für eine gemeinsame Klasse durch Parameter (Attribute) spezifiziert, die eine Klasse für ihre Objekte vorschreibt. In einer Klasse können weiterhin Operationen festgelegt werden, die das Verhalten ihrer Objekte festlegen. Dabei gilt das Kapselungsprinzip, nach dem Klassen Attribute und Operationen von Objekten in einer Einheit zusammenfassen und die Attribute nur indirekt über die Operationen der Klasse zugänglich

sind (Oestereich 2004). Operationen einer Klasse werden häufig als Methoden bezeichnet. In diesem Konzept gibt es weiterhin die sogenannte Polymorphie, bei der Klassen ihre Eigenschaften an andere Klassen vererben können. Das bedeutet, dass einige Klassen Spezialisierungen anderer Klassen sind, mithin ihre Objekte also die Eigenschaften und Methoden der Superklasse implementieren, aber noch erweitern. Dabei gilt das Substitutionsprinzip, nach dem Objekte von Unterklassen jederzeit anstelle von Objekten der Oberklasse eingesetzt werden können. Die Grundidee der objektorientierten Programmierung ist demnach, Daten und Funktionen, die auf diese Daten angewandt werden können, möglichst eng in einem sogenannten *Objekt* zusammenzufassen und nach außen hin zu *kapseln*, so dass diese Daten nicht durch Methoden fremder Objekte manipuliert werden können. Dabei ist jedes Objekt per Definition unabhängig von seinen konkreten Attributwerten von allen anderen Objekten eindeutig zu unterscheiden (Oestereich 2004, 44). Für die Festlegung von Klassen, die die Klassifizierung von Objekten vorgeben, gilt das Kohärenzprinzip, nach dem jede Klasse für genau einen sachlich-logischen Aspekt des Gesamtsystems verantwortlich sein soll. Die in diesem Verantwortungsbereich liegenden Eigenschaften sollen innerhalb einer eindeutigen Klasse zusammengefasst sein. Zudem soll eine Klasse keine Eigenschaften enthalten, die nicht zu ihrem Verantwortungsbereich gehören. Das folgende einfache Beispiel in JAVA zeigt dieses Programmierkonzept:

```
class Person {  
  
    private String name;  
    private char geschlecht;  
    private int alter;  
    private Date geburtsdatum;  
    private String geburtsort;  
    private Adress wohnsitz;  
  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
    public char getGeschlecht() {  
        return geschlecht;  
    }  
    public void setGeschlecht(char geschlecht) {  
        this.geschlecht = geschlecht;  
    }  
    public int getAlter() {  
        return alter;  
    }  
    public void setAlter(int alter) {  
        this.alter = alter;  
    }  
    public Date getGeburtsdatum() {
```

```

        return geburtsdatum;
    }
    public void setGeburtsdatum(Date geburtsdatum) {
        this.geburtsdatum = geburtsdatum;
    }
    public String getGeburtsort() {
        return geburtsort;
    }
    public void setGeburtsort(String geburtsort) {
        this.geburtsort = geburtsort;
    }
    public Adress getWohnsitz() {
        return wohnsitz;
    }
    public void setWohnsitz(Adress wohnsitz) {
        this.wohnsitz = wohnsitz;
    }
}

public Person(String n) {
    this.name = n;
}
}

class Student extends Person {

    private String hochschule;
    private String studiengang;
    private int semester;
    private String[] fachrichtungen;

    public String getHochschule() {
        return hochschule;
    }

    public void setHochschule(String hochschule) {
        this.hochschule = hochschule;
    }

    public String getStudiengang() {
        return studiengang;
    }

    public void setStudiengang(String studiengang) {
        this.studiengang = studiengang;
    }

    public int getSemester() {
        return semester;
    }

    public void setSemester(int semester) {
        this.semester = semester;
    }

    public String[] getFachrichtungen() {
        return fachrichtungen;
    }

    public void setFachrichtungen(String[] fachrichtungen) {
        this.fachrichtungen = fachrichtungen;
    }
}

```

```

    public void createStudentenDaten(String h, String sg,
                                     int s, String[] f) {
        this.setHochschule(h);
        this.setStudiengang(sg);
        this.setSemester(s);
        this.setFachrichtungen(f);
    }

    public Student(String n) {
        super(n);
    }
}

class Beispiel {

    public static void main(String args[]) {

        Person rentner = new Person("Max Musermann");
        Student student = new Student("Moritz Mustermann");

        rentner.setGeschlecht('m');
        rentner.setAlter(67);
        rentner.setGeburtsdatum(new Date(1,1,1940));
        rentner.setGeburtsort("Musterstadt");
        rentner.setWohnsitz(new Adress("Musterweg 1",
                                       12345, "Musterstadt"));

        student.setGeschlecht('m');
        student.setAlter(25);
        student.setGeburtsdatum(new Date(1,1,1982));
        student.setGeburtsort("Musterstadt");
        student.setWohnsitz(new Adress("Musterweg 2",
                                       12345, "Musterstadt"));

        String[] faecher = {"Deutsch", "Englisch"};

        student.createStudentenDaten(
            "Universität Musterstadt",
            "Bachelor",
            4,
            faecher);
    }
}

```

Hierbei werden die Attribute von Objekten der Klassen «*Person*» und «*Student*» gekapselt und sind von außen nur über die Setter- und Getter-Methoden zugänglich. In der Klasse «*Beispiel*» wird nun ein Objekt namens «*Rentner*» der Klasse «*Person*» erzeugt und mit den Setter-Methoden die Variablen mit den entsprechenden Daten der Person „Rentner“ gespeichert. Das gleiche geschieht für das Objekt namens «*student*» der Klasse «*Student*», die eine Unterklasse von «*Person*» ist. Zusätzlich wird hier eine nur für Objekte der Klasse «*Student*» zugängliche Methode

«*createStudentenDaten*» aufgerufen, in der zusätzlich typische Daten über Studenten gespeichert werden.

Im Fall des *ViLM*-Players hat sich relativ schnell herauskristallisiert, dass eine vollständig objektorientierte Implementierung nicht notwendig ist. Wie bereits angedeutet, kommt die objektorientierte Modellierung und Entwicklung von Software dann zum Einsatz, wenn bestimmte Objekte eines realen Problembereichs identifizierbar sind und diese in einer Software repräsentiert werden müssen. Dabei müssen Prozesse der Interaktion und des Austausch von Objekten modelliert und implementiert werden. Eine solche Analogie zu realen Objekten und Strukturen lag aber beim *ViLM*-Player nicht vor. Stattdessen stellt das Flex-Framework ja bereits eine hochwertige Klassenbibliothek bereit, deren Objekte nahezu das gesamte Spektrum an Elementen und Prozessen abdeckt, die man für eine typische Webanwendung inklusive Audio- und Videofunktionalitäten benötigt, zur Verfügung. Das Verhalten von MXML-Objekten ist dabei ereignisbasiert über Actionscript steuerbar, so dass auch bezüglich der Controller-Schicht für den *ViLM*-Player kein Mangel an Funktionalität durch das Framework vorhanden ist. Man kann aber auch argumentieren, dass es so gut wie keine Speicherung persistenter Daten gibt, die eine Datenbank erforderlich machen würden. Wäre dies der Fall, könnte man eine objektorientierte Modellierung und Implementierung aufgrund der Äquivalenz des Objekt- und Datenmodells rechtfertigen (Brössler und Siedersleben 2000, 197). Es lassen sich aber zum Beispiel über Instanzen der Toplevel-Klasse «*Object*» des Flex-Frameworks sehr komfortabel Datenstrukturen kreieren, die man sonst über eigene Klassen realisieren würde. Um zum Beispiel die aus der *ViLM*-Projektdatei ausgelesenen Daten über die URLs der Videodateien, der Namen der Videos und ihrer Synchronisationszeitpunkte für die Anwendung zu speichern, werden über

```
videoPreviews.addItem({videoSource: videoSources[i].file,  
                        videoName: videoSources[i].name,  
                        videoNumber: i+1,  
                        videoSyncTime: videoSources[i].sync});
```

in dem Datenarray «*videoPreviews*» «*Object*»-Instanzen der folgenden Struktur gespeichert:

```
{videoSource: videoSources[i].file,  
  videoName: videoSources[i].name,  
  videoNumber: i+1,  
  videoSyncTime: videoSources[i].sync}
```

Sie sind dann über die Punktnotation der Objektvariablen «*videoSource*», «*videoName*», «*videoNumber*» und «*videoSyncTime*» zugreifbar.

2.6 Analyse

2.6.1 Anforderungen an den Player

Der *ViLM*-Player soll das Abspielen mehrerer Videos ermöglichen und dabei Synchronität sicherstellen. Da der Player in engem Zusammenhang mit dem Synchronisations- und Annotationstool steht, muss es gleichzeitig möglich sein, alle zur Wiedergabe notwendigen Informationen aus der *ViLM*-Projektdatei auszulesen. Dies betrifft Informationen über Phasen und die in ihnen enthaltene Beschreibung und einen eventuellen Kommentar. Es soll möglich sein, die definierten Phasen innerhalb der Videos anzuspringen.

Während des Playbacks soll die Sichtbarkeit der Videos einstellbar sein, so dass zum Beispiel bei einem Projekt mit vier Videos nur zwei oder eins angezeigt werden. Zur Anzeige der Videos soll es maximal vier Videopanel geben, die gleichzeitig dargestellt werden. Für den Fall, dass mehr Videoquellen existieren, soll es möglich sein, für die Videopanel neue Quellen auszuwählen.

Die *ViLM*-Projektdatei soll dem Projektcharakter nach als entfernte Datei über HTTP offenbar sein, für Test- oder Anwendungsfälle in denen lokale Projektdateien vorliegen, soll man sie aber auch lokal öffnen können.

Eine weitere zentrale Anforderung ist die Bereitstellung einer klassischen Videosteuerung, ähnlich einer DVD-Player-Software. Das bedeutet neben den üblichen Steuerungselementen Play, Pause, Stop auch eine Möglichkeit, während dem Playback direkt die Phasen anzuspringen.

Die Oberfläche des Players soll dabei ein zeitgemäßes Design erhalten, das Gestaltungskriterien der DIN-EN 9241¹⁹ berücksichtigt. Vorhandene Metadaten innerhalb der Phasen, also ihren Namen, ihre Zeiten, sowie die darin enthaltene Beschreibung und der Kommentar sollen während des Playbacks eingeblendet werden und dem Nutzer so die Annotationen jederzeit zugänglich machen.

Des Weiteren ist gefordert, die in einer *ViLM*-Projektdatei gespeicherten URLs zu Materialien öffnen zu können, so dass diese durch die dem jeweiligen Format zugeordnete Systemsoftware geöffnet werden.

2.6.2 Gestaltungskriterien der DIN EN ISO 9241

Um ein barrierefreies Arbeiten mit dem *ViLM*-Player zu ermöglichen, wurde bei der Gestaltung der Benutzerschnittstelle Rücksicht auf fünf Grundsätze der DIN-Norm 9241 genommen.

1. Um **Aufgabenangemessenheit** für die Software zu garantieren, liegt das Augenmerk vorrangig beim synchronen Abspielen der Videos. Dies stellt den zentralen Anwendungsfall des Players dar, und sollte daher effektiv und effizient möglich sein.
2. Um **Selbstbeschreibungsfähigkeit** herzustellen, wird eine Hilfe angeboten, in der die Software und ihre Funktionen erläutert werden. Zudem werden einige UI-Elemente mit Tooltips ausgestattet, die Rückmeldung über ihre jeweilige Funktionalität geben.
3. Damit die **Steuerbarkeit** des Players zu jeder Zeit gewährleistet ist, werden Mechanismen realisiert, die robust gegenüber beliebigen Interaktionen mit dem Benutzer machen sollen. Das bedeutet zum Beispiel, dass übereiltes Hin- und Herspringen im Video vermieden wird, um Problemen mit den Streams und der internen Arbeitsweise der *MXML*-Klasse «*VideoDisplay*» aus dem Weg zu gehen.
4. Der Grundsatz der **Individualisierbarkeit** wird dahingehend berücksichtigt, dass innerhalb des Players UI-Elemente ausgeblendet werden können, die nichts mit dem eigentlichen Playback der Videos und dessen Steuerung zu tun haben. Damit kann ein Nutzer seinen Fokus auf das Betrachten der Videos legen, indem der zur Verfügung stehende Bildschirmplatz besser ausgenutzt wird.
5. Um **Erwartungskonformität** zu garantieren, sollen alle UI-Elemente, die Benutzereingaben entgegen nehmen und die Steuerung des Playbacks realisieren, die mit ihnen implizierte Funktionalität, in der Art und Weise bereitstellen, wie man das bereits von ähnlicher Software zum Abspielen von Medien gewöhnt ist (zum Beispiel sollen der Prev- und Next-Button dem Anspringen der jeweils vorherigen, bzw. nächsten Phase dienen, ähnlich wie man das von einem DVD-Kapitel-Menü her kennt).
6. Die Grundsätze der **Fehlertoleranz** und **Lernförderlichkeit** sind bei der Umsetzung des *ViLM*-Players nicht so sehr von Bedeutung.

¹⁹ Die DIN EN ISO 9241 – Norm spezifiziert Grundsätze für die Gestaltung von Benutzerschnittstellen von Software. Sie besteht aus 17 Teilen, wovon vor allem der Teil 10, „Grundsätze der Dialoggestaltung“ wichtig ist (Webseite, OpenUse - DIN EN ISO 9241 2008).

2.6.3 XML im ViLM-Player

XML bildet die zentrale Schnittstelle des Synchronisations- und Annotationstools und des ViLM-Players. Am Ende des Synchronisierungs- und Annotationsprozesses steht eine XML-Datei, die das Annotationstool mit allen notwendigen Daten erstellt. Nachfolgend ist die Struktur einer solchen Datei dargestellt:

```
<?xml version="1.0" encoding="utf-8"?>
<ROOT>
<Meta project="Beispiel"/>

<Einheit>
  <Materialien>
    <Material fileName="material_1.txt"/>
  </Materialien>
  <TextNode nodeName="Beschreibung">ViLM-Projektdatei</TextNode>
  <TextNode nodeName="Kommentar"></TextNode>
</Einheit>

<VideoSources>
  <Source videoname="Trailer 1"
    videoID="0"
    syncStartTime="0.0"
    endTime="100.0">video.flv</Source>
</VideoSources>

<Phasen>
  <Phase name="Phase 1" startTime="0" endTime="100">
    <TextNodes>
      <TextNode nodeName="Beschreibung">
        Phasenbeschreibung</TextNode>
      <TextNode nodeName="Kommentar">Phasenkommentar</TextNode>
    </TextNodes>
    <Phase name="Subphase 1" startTime="10" endTime="90"/>
  </Phase>
</Phasen>
</ROOT>
```

In Abschnitt 2.9.1 wird beschrieben, dass sich Attribute von XML-Elementen über E4X sehr komfortabel auslesen lassen. Daher nutzen die ViLM-Projektdateien Attribute, zum Beispiel bei dem Element "Phase". Es besitzt die Attribute "name", "startTime" und "endTime", die Information über Start- und Endzeiten einer Phase, sowie ihren Namen enthalten. Sowohl Elemente, als auch Attribute haben Namen, für die *Uniform Resource Identifiers* (URI) genutzt werden. Die Menge der URIs innerhalb eines XML-Dokumentes lassen sich als Bezeichner innerhalb einer abstrakten Menge von Namen, dem Namensraum (*namespace*) auffassen. Der Namensraum ist für XML-Dokumente entscheidend, die zum Austausch von Daten über verschiedene Technologien hinweg dienen, denn er bestimmt den Kontext, innerhalb dessen ein Element oder Attribut Bedeutung hat. So werden *Element-Information-Items* mit dem Präfix "myNs" durch die Angabe des Attributs

```
xmlns:myNs="myNameSpaceURI"
```

im Namensraum mit der URI "myNameSpaceURI" bedeutsam. In einem anderen Namensraum existieren Sie möglicherweise gar nicht. Innerhalb des *ViLM*-Players gibt es zwei Namensräume. Zum einen den durch Flex standardmäßig vorgegebenen Namensraum

```
xmlns:mx="http://www.adobe.com/2006/mxml"
```

mit dem über das Präfix "mx" auf die Elemente der Programmiersprache MXML zugegriffen werden kann und zum anderen den Namensraum

```
xmlns:aperture="com.fluorinefx.aperture.*"
```

über das man Zugriff auf das Aperture-Framework erhält (siehe Abschnitt 2.9.2).

2.7 Modellierung

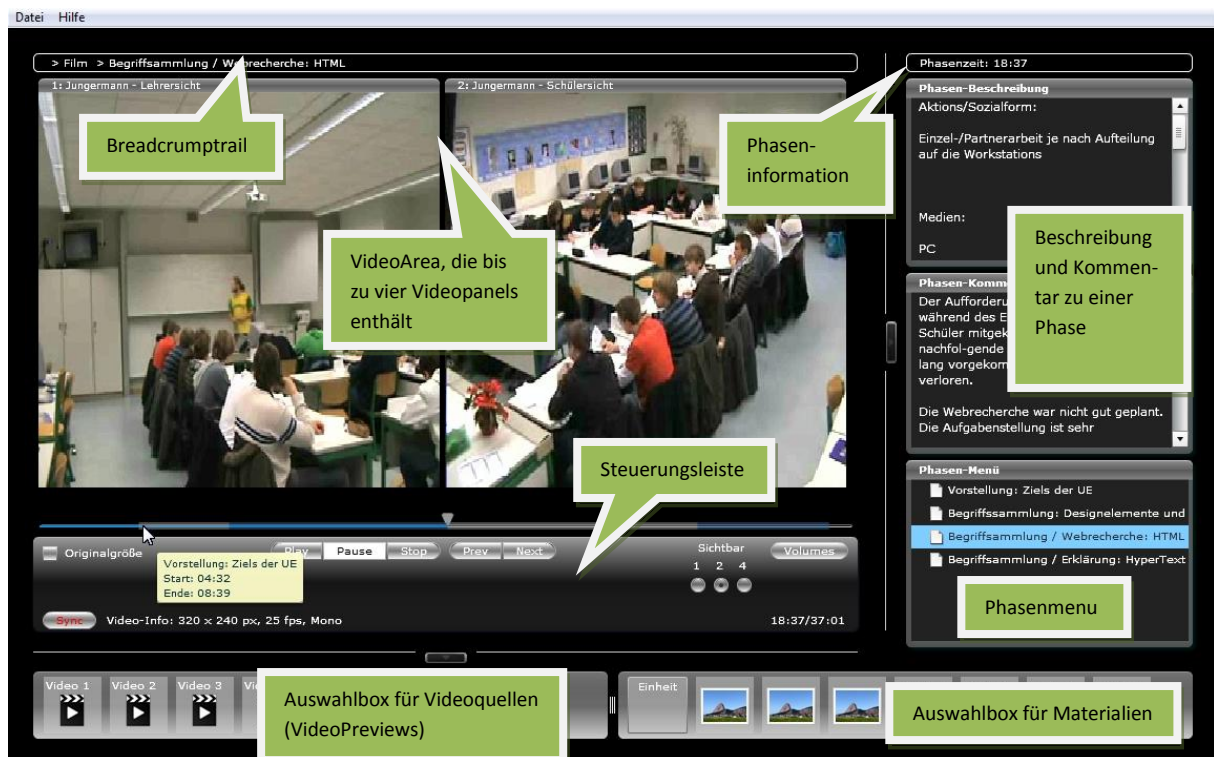
Die bereits in der Analyse geschilderten Anforderungen leiten zur Modellierung für den *ViLM*-Player über. Dabei steht vor allen Dingen die Modellierung einer sinnvollen Oberfläche im Vordergrund, die sich an den erwähnten Kriterien zur Dialoggestaltung aus Abschnitt 2.6.2 orientiert. Folgende Elemente wurden für die Benutzeroberfläche des *ViLM*-Players identifiziert:

- 1 – 4 parallel laufende Videopanel mit Anzeige des Videonamens
- Steuerungsleiste mit folgenden Buttons: Play, Pause, Stop, Positions-Slider
- Daneben einen Prev- und Next-Button zum Anspringen der jeweils vorherigen, bzw. nächsten Phase
- Phasenmenu zur direkten Auswahl einzelner Phasen

- Auswahlbuttons, ob 1, 2 oder 4 Videopaneln sichtbar sind
- Regler zur Einstellung der Lautstärke jedes Videos
- Checkbox, ob die Größe der Videopaneln an den zur Verfügung stehenden Platz angepasst wird, oder das originale Seitenverhältnis erhalten bleibt
- Breadcrumbtrail²⁰ zur Darstellung der Ableitung der aktuellen Phase
- Textfeld zur Einblendung des Namens und der Startzeit der aktuellen Phase
- Textfelder zur Einblendung von Beschreibung und Kommentar der aktuellen Phase
- Auswahlbox, aus der sich neue Videoquellen per Drag & Drop auf die Videopaneln ziehen lassen
- Auswahlbox mit Links zu den verschiedenen Materialien, die dann durch Klicken zur Anzeige gebracht werden

Die folgenden Screenshots des ViLM-Players zeigen und benennen diese Elemente auf der Anwendungsoberfläche:

Abbildung 4: Screenshot der Anwendungsoberfläche



²⁰ Navigationstechnik in Benutzerschnittstellen, die dem Nutzer eine Rückverfolgungsmöglichkeit bis zu seiner aktuellen Position innerhalb der Dokumentenstruktur zeigt und anbietet. Im Fall des ViLM-Players ist der Breadcrumbtrail nur zur Information da.

2.8 Entwurf

2.8.1 Unterschiedliche Ansätze

Zu Beginn der Entwicklung des *ViLM*-Players stand eine relative unscharfe Beschreibung des Problembereichs. Zunächst waren nur wesentliche funktionale Eigenschaften im Zusammenhang mit der Erweiterung des *ViLM*-Synchronisations- und Annotationstools klar definiert. Demnach sollte eine plattformunabhängige Videoplayer-Software entwickelt werden, die über eine XML-Datei eine Schnittstelle zum Annotationstool besitzt und die darin annotierten Videos präsentieren soll. Dabei war aber noch nicht festgelegt, ob die Videos als progressiver Download über HTTP oder als Streams für den Player zur Verfügung gestellt werden.

Für die Entwicklung wurde von Anfang an Adobe AIR als technologische Basis gewählt. Zunächst bestand die Aufgabe darin, die XML-Struktur der *ViLM*-Projektdateien aus dem Annotationstool so anzupassen, dass der Player daraus alle relevanten Daten, die zur Wiedergabe notwendig sind, auslesen kann. Bisher waren in dieser Datei Informationen über Phasen mit einer Beschreibung und einem Kommentar zu jeder Phase, sowie Materialien abgelegt. Die Erweiterung des Synchronisations- und Annotationstools von zwei auf beliebig viele Videos führt nicht unmittelbar zu Änderungen an dieser Datei. Jedoch mussten zum Beispiel für den *ViLM*-Player die URLs der Videoquellen zusätzlich darin gespeichert werden.

Zu Beginn der Implementierung erfolgten zunächst einmal einige Tests, wie rechenlastig es ist, mehrere Instanzen der Klasse `VideoDisplay` (im Folgenden `Videodisplay` genannt) nebenläufig zu nutzen. Zunächst lagen dafür nur lokale Flash-Video-Dateien vor und die Auslastung des Rechners (Windows Plattform, Intel Centrino Duo CPU (1,8 Ghz), 1,5 GB RAM) bei vier nebenläufigen `Videodisplay`s war relativ gering. Also entschied ich mich dafür, diese Komponente zu nutzen und Videodaten zunächst einmal über einen progressiven Download in einer Flex-Anwendung wiederzugeben. Zu diesem Zeitpunkt lagen noch keine großen Flash-Video-Dateien mit Unterrichtsstunden vor, so dass zunächst nur mit kleinen Dateien über HTTP von einem Apache Webserver getestet wurde. Dieses Vorgehen stellte die Flex-Anwendung vor keine Schwierigkeiten, so dass es dann um die aus der Erweiterung für zusätzliche Videoquellen ging. Besonders intuitiv und komfortabel schien es dabei zu sein, alle in der *ViLM*-Projektdatei angegebenen Videoquellen als kleine Previews abzuspielen, die dann per Drag & Drop auf die Videopaneln gezogen werden können und dort in voller Größe abgespielt werden. Dabei entstanden allerdings die ersten Performanceprobleme. Zur Darstellung der Previews wurden wieder `Videodisplay`s genutzt, da sich empirisch herausgestellt hatte, dass die Prozessor- und Speicherauslastung bei der Nutzung weiterer `Videodisplay`s nicht signifikant steigt. Bei vier Videoquellen mussten dann anwendungsintern

allerdings bereits acht Videodisplays genutzt werden und gleichzeitig fanden in diesem Szenario dann acht progressive Downloads statt. Das hat die Auslastung des Rechners doch merklich erhöht. Deutlich schlimmer war aber das Problem, dass acht unterschiedliche progressive Downloads gestartet wurden, was zu einer erheblichen Verlangsamung der Gesamtgeschwindigkeit führte. Ein Ausweg schien zu sein, nur für die Menge aller Videoquellen tatsächlich neue Videodisplays als Videopreviews zu erzeugen und die eigentlichen Videopaneln nur als Objektkopien dieser Videodisplays zu erzeugen. Bei einer Drag & Drop Operation eines Videopreview-Elementes auf ein Videopanel wird eine Kopie des Videodisplays des Preview-Elements erstellt und an das Videopanel angefügt, in der Hoffnung, die Netzwerkauslastung damit zu senken und die Gesamtperformance wieder zu steigern. Realisiert wurde eine solche Objektkopie über die Klasse «*ByteArray*» von Actionscript 3.0, wie die folgende, jedoch nicht mehr verwendete Methode zeigt:

```
private function cloneVideoDisplay(source :  
    VideoDisplay) : VideoDisplay {  
  
    var byteArray : ByteArray = new ByteArray();  
  
    byteArray.writeObject(Object(source));  
    byteArray.position = 0;  
    return VideoDisplay(byteArray.readObject());  
  
}
```

Gleichzeitig kristallisierte sich dann aber heraus, dass progressives Downloaden von Videodaten für die eigentlichen Anwendungsfälle des *ViLM*-Players überhaupt nicht sinnvoll sind. Bei dem Anwendungsfall von gefilmten Unterrichtsstunden werden moderne, digitale Videokameras eingesetzt, die die Videos meist auf MPEG-4-Basis abspeichern, in üblichen Containerformaten wie MP4 oder H.264. Auch wenn die in MPEG-4 beschriebenen Videokompressionsalgorithmen bereits eine hohe Datenkompression ermöglichen, so sind die für den *ViLM*-Player bereitgestellten Videodateien dann trotzdem meist noch etliche hundert Megabytes groß. Für einen progressiven Download ist das eindeutig zu viel, da die Daten ja auf den Rechner des Videoclient heruntergeladen werden. Ein weiteres gravierendes Problem ist die Tatsache, dass das punktgenaue Anspringen bestimmter Positionen in einem Flash-Video dann nicht möglich ist, weil zunächst erst einmal die Daten bis zu der gewünschten Stelle heruntergeladen worden sein müssen und außerdem nur zu

definierten Keyframes in den Videos springen kann, die aber nicht mit der gewünschten Position exakt übereinstimmen müssen. Ein Streamingserver berechnet dagegen für jede beliebige Position aus den Keyframes und den dazwischen liegenden Bildinformationen ein Vollbild, ab dem das Playback punktgenau gestartet werden kann. Damit war klar, dass gestreamte Flash-Videos das Rohmaterial für den *ViLM*-Player bilden. Da als technologische Basis bereits das Flex-Framework festgelegt worden war, entschieden wir uns, den von Adobe stammenden Flash Media Server 3 als Streamingserver zu nutzen, da er auf Basis des Protokoll RTMP echtes Streaming von Flash Video Dateien ermöglicht und in einer kostenlosen Variante für den privaten Gebrauch mit bis zu 10 gleichzeitigen Verbindungen verfügbar ist (Bösken 2007, 37).

Nun zeigte sich aber, dass wenn zunächst n Streams auf den Videodisplays für die Previews geladen werden und gleichzeitig noch bis zu vier Videopaneln als Objektkopie mitlaufen, die Recherauslastung durch das Dekodieren der Streams auf bis zu 100% steigt. Außerdem werden dabei für die Videopaneln neue Streams vom Streamingserver angefordert. Demnach hat man durch das Erzeugen der Videopaneln als Objektkopien der Videodisplays in den Previews nichts gewonnen. Also wurde schließlich die Idee der synchron mitlaufenden Videopreviews verworfen und stattdessen folgendes Modell gewählt:

Es gibt bis zu vier Videopaneln, die Flash-Video-Dateien von einem Flash Media Server streamen. Die übrigen Videoquellen werden durch ein grafisches, nicht-animiertes Element dargestellt und können mit einer Drag & Drop - Operation auf die Videopaneln gezogen werden, worauf diese das entsprechende Video abspielen. Diese gefundene Lösung läuft stabil und berechnet den Client-Rechner nur minimal, allerdings ist eine schnelle Internetverbindung dafür von Nöten.

2.9 Implementierung

Die Klassenbibliothek von Flex stellt in der Sprache MXML bereits mächtige Klassen zur Gestaltung von typischen Webanwendungen zur Verfügung. Im Folgenden sollen daraus Klassen vorgestellt werden, die für die Implementierung des *ViLM*-Players bedeutsame Objekte bereitstellen.

`mx.core.Application`: Container der gesamten Flex-Anwendung, in den Content eingefügt werden kann, ohne einen anderen Container zu benutzen.

`mx.controls.VideoDisplay`: UI-Komponente, die ein lokales oder gestreamtes Flash Video abspielt.

`mx.containers.Panel`: UI-Komponente mit einer Titelzeile, einem Rand und einem Inhaltsbereich für ihre *child*-Elemente. Stellt eine Container-Klasse für weitere UI-Komponenten dar.

`mx.containers.TitleWindow`: UI-Komponente ähnlich einem *Panel*, vorwiegend benutzt für Popups.

`mx.containers.Canvas`: UI-Komponente, die einen rechteckigen Bereich darstellt, in welchem *child*-Elemente und *Control*-Elemente platziert werden.

`mx.containers.ControlBar`: UI-Komponente, die es ermöglicht, Steuerungselemente im Fußbereich eines *TitleWindow*- oder *Panel*-Containers zu platzieren.

`mx.containers.HBox`/`mx.containers.VBox`: UI-Komponente, die ihre *child*-Elemente in einer horizontalen oder vertikalen Liste platziert. Stellt eine Container-Klasse für weitere UI-Komponenten dar.

`mx.containers.HDividedBox`/`mx.containers.VDividedBox`: UI-Komponente ähnlich zu *HBox* und *VBox*, jedoch wird zwischen jedem *child*-Element innerhalb der Box ein verschiebbarer Trennstrich angezeigt.

`mx.controls.HSlider`/`mx.controls.VSlider`: UI-Komponente, mit der sich ein Wert zwischen zwei Grenzen mittels eines verschiebbaren Reglers auswählen lässt.

`mx.controls.Text`: UI-Komponente, die mehrzeiligen, nicht-editierbaren Text anzeigt.

`mx.controls.TextArea`: UI-Komponente, die einen Rand und gegebenenfalls Scrollbalken besitzt und mehrzeiligen, editierbaren Text anzeigt.

`mx.controls.TextInput`: UI-Komponente, über die ein String eingelesen werden kann.

`mx.controls.Alert`: Popup-Dialog-Box, die eine Nachricht, einen Titel, Buttons und ein Icon enthalten kann.

`mx.controls.RadioButton`: UI-Komponente, mit der genau ein Element aus einer Menge von Elementen ausgewählt werden kann. Stellt einen Radiobutton dar.

`mx.controls.Button`: UI-Komponente, die einen Button darstellt (speziellere Buttonkombinationen sind zum Beispiel *ButtonBar* oder *ToggleButtonBar*).

`mx.controls.ToolTip`: Tooltip-Popup-Box, die Informationen über Felder und Objekte anzeigt, wenn die Maus darauf zeigt.

`mx.controls.Repeater`: MXML-Komponente, die mehrfache Instanzen ihrer Subkomponente in Abhängigkeit eines Dataproviders²¹ erzeugt. Solche Subkomponenten können standardmäßige UI-Elemente sein.

Aus diesen wenigen Elementen lässt sich die Oberfläche des ViLM-Players bereits nahezu zusammensetzen:



2.9.1 Auslesen von XML (E4X)

Actionscript enthält eine Erweiterung zum Auslesen von XML-Dateien, die eine Alternative zum Parsen über DOM darstellt. Es ist das sogenannte ECMAScript für XML (E4X), das XML-Daten wie primitive Datentypen behandelt, was einen einfachen Zugriff auf Elemente ermöglicht. E4X wurde durch ECMA International standardisiert und ihre aktuelle Version Ende 2005 veröffentlicht. Das folgende Beispiel verdeutlicht, wie der Zugriff auf XML-Daten durch E4X aussieht:

²¹ Ein Dataprovider stellt ein Attribut zum Beispiel der Klasse «Repeater» dar, das als Wert einen Datenobjekt (zum Beispiel ein Array) übergeben bekommt, anhand dessen der Repeater seine Elemente erzeugt.

```

<Phasen>
  <Phase name="start" startTime="0" endTime="50">
    <TextNodes>
      <TextNode>Hier steht ein Text</TextNode>
    </TextNodes>
  </Phase>
  <Phase name="mitte" startTime="15" endTime="30"/>
  <Phase name="ende" startTime="30" endTime="100"/>
</Phasen>

```

Eine solche XML-Struktur kann in Actionscript dem Konstruktor der Klassen XMLList oder XML als Argument übergeben werden, wodurch ein Objekt mit einer Flex-internen XML-Repräsentation dieser Daten entsteht. Um diese XML-Daten basierend auf E4X auszulesen, gibt es folgende Möglichkeiten:

Tabelle 3: XML-Verarbeitung in ActionScript 3.0

	AS-Ausdruck	Rückgabewert
1.	<code>Phasen.Phase.(@name == "start").TextNodes.TextNode</code>	Liefert die Zeichendaten zurück, die innerhalb des "TextNode"-Knotens der ersten Phase stehen
2.	<code>Phasen.Phase.(@name == "mitte").@startTime</code>	Liefert den Wert des Attributes « <i>startTime</i> » für den zweiten "Phase"-Knoten zurück (dessen Attribut « <i>name</i> » gleich "mitte" ist).
3.	<code>Phasen.Phase.@startTime</code>	Liefert eine Liste mit den Werten des Attributes « <i>startTime</i> » für alle "Phase"-Knoten zurück
4.	<code>for each (var t in Phasen..@endTime) { ... }</code>	Ermöglicht für jedes Vorkommen eines « <i>endTime</i> »-Attributes in einem beliebigen <i>child</i> -Element von "Phasen" einen Durchlauf einer Schleife, in der die Schleifenvariable jeweils der Wert dieses Attributes ist

Das Auslesen der *ViLM*-Projektdatei wird nach dem Öffnen einer Datei durch die Methoden «*readXML*» oder «*readXMLFromRemote*» gestartet. Dadurch wird die Methode «*setXMLDataObjects*» aufgerufen, die das Auslesen der verschiedenen XML-Knoten in unterschiedlichen Methoden auslöst.

In der Methode «*createCuepointsAndPhaseTimesFromXML*» wird für jeden Knoten mit dem Namen "*Phase*" ein *CuePoint*-Objekt (siehe dazu Abschnitt 2.9.2) in dem Array «*xmlCuePoints*» angelegt, das die Attribute «*name*» und «*time*» der entsprechenden Phase speichert. Im Array «*rulersData*», das die Basis für die Phasen-Overlays auf dem Playback-Slider darstellt, wird zu jeder Phase über die Attribute «*name*», «*startTime*» und «*endTime*» ein Objekt gespeichert, das diese drei Informationen enthält. Hat eine Phase mindestens eine Unterphase, so wird noch überprüft, ob diese beim gleichen Zeitpunkt wie die *parent*-Phase anfängt. Ist das der Fall, wird für die *parent*-Phase kein *Cuepoint* erzeugt, sondern nur für die Unterphase. Auf diese entstehen nicht zwei *Cuepoints* mit derselben *startTime*, für die dann vom *Cuepoint*-Manager gleichzeitig zwei *Cuepoint*-Events ausgelöst werden, durch die zeitkritische Methoden eventuell doppelt aufgerufen würden.

In der Methode «*readVideoSourcesFromXML*» werden die *child*-Elemente des XML-Knotens "*VideoSources*" ausgelesen. In dem Fall, dass eine lokale Projektdatei geöffnet worden ist, wird als Videoquelle die String-Konkatenation des Dateipfadens der *ViLM*-Projektdatei, des Strings „\Videos\“ und des Inhalts dieses *child*-Elementes erzeugt. Somit wird als Videoquelle der Dateiname einer Flash-Video-Datei, die in einem Unterordner „Videos“ des Ordners der *ViLM*-Projektdatei liegt, gespeichert. Das Datenarray «*videoSources*» bekommt daraufhin für jede Videoquelle ein Objekt als Eintrag, das über das Attribut «*file*» den Dateinamen, über «*sync*» die zu einer Quelle gehörende Synchronisationszeit und über «*name*» den Videonamen speichert. Im Fall einer entfernten *ViLM*-Projektdatei wird als Eintrag für «*file*» direkt der Elementinhalt des Knotens "*VideoSource*" gespeichert, weil darin dann bereits die vollständige URL zu einer Datei auf einem Streamingserver steht.

In ähnlicher Form werden auch die sonstigen Informationen aus der *ViLM*-Projektdatei ausgelesen. Bei der Erzeugung der Einträge für das «*xmlPhaseMenu*» gibt es einige Besonderheiten. Das «*xmlPhaseMenu*» stellt einen Baum dar, in dem die verschiedenen Phasen mit ihren Unterphasen als Elemente wie in einem klassischen Dateibaum abgebildet werden. Dies liegt vorrangig darin begründet, dass die Einteilung in Phasen ja letztlich bereits eine solche Struktur impliziert und sie gleichzeitig für den Nutzer gewohnt und intuitiv sein sollte. Um das zu realisieren wird ein Element der Klasse «*Tree*» erzeugt, dem als *DataProvider* eine XML-Struktur übergeben wird, aus der der Baum seine Knoten über ein Attribut «*label*» erzeugt. Dabei wird aus jedem Knoten der XML, der ein Attribut besitzt, dessen Name dem Inhalt der Variablen "*label*" entspricht, ein Element des Baumes. Innere Knoten mit diesem Label werden dann zu inneren Knoten im Baum des «*xmlPhaseMenu*». Für

die *ViLM*-Projektdatei ist in diesem Fall das Attribut «*name*» relevant, welches den meisten Knoten darin einen eindeutigen Bezeichner zuweist. So haben alle "*Phasen*"-Knoten ein Attribut «*name*», deren Inhalt genau der Phasenname ist, den ein Nutzer im «*xmlPhaseMenu*» anklicken möchte. Allerdings liegen in der ursprünglichen Projektdatei noch weitere Knoten mit einem Attribut «*name*» vor, die mit dem beschriebene Vorgehen auch Elemente des Phasenbaumes werden würden (zum Beispiel die "*TextNode*"-Knoten mit der Attributzuweisung «*name*="Beschreibung"», bzw. «*name*="Kommentar"»). Diese sollen natürlich nicht angezeigt werden. Deshalb wird in der Methode «*setXMLPhaseMenu*» in der AS-Datei "*xmlManager.as*" eine XML-Kopie «*xmlPhaseData*» der ursprünglichen *ViLM*-Projektdatei erzeugt, in der diese irrelevanten Knoten und zudem Knoten mit dem Namen "*Material*" entfernt werden. Zuletzt kann dem Dataprovider des Baumes diese formatierte XML übergeben werden, was das «*xmlPhaseMenu*» schließlich erzeugt:

```
xmlPhaseData.source=formattedXMLData.Phasen.elements();  
xmlPhaseMenu.labelField="@name";  
xmlPhaseMenu.dataProvider=xmlPhaseData;
```

2.9.2 Grundsätzliches zur Umsetzung

Das Zusammenspiel von Phasen und dem Playback der Videos wird über sogenannte Cuepoints geregelt. Cuepoints sind Datenobjekte, die Zeitmarken innerhalb des Playbacks repräsentieren und die Grundlage für Ereignisobjekte der Klasse «*CuePointEvent*» (nachfolgend als „Cuepoint-Event“ bezeichnet) bilden, die von Videodisplays automatisch und zeitgesteuert während des Playbacks erzeugt werden können. Sie müssen zwingend zwei Attribute enthalten: einen Namen und einen Zeitpunkt. Sie werden über eine Instanz der Klasse «*CuePointManager*» an ein Videodisplay angehängt. Erreicht das Playback eines Videos den in einem bestimmten Cuepoint über das Attribut «*time*» festgelegten Zeitpunkt, wird ein Cuepoint-Event erzeugt und in den Eventflow versendet. In diesem Cuepoint-Event sind die Attributwerte, also Name und Zeit des betreffenden Cuepoints gespeichert. Ein Listener für die Cuepoint-Events wird an das Videodisplay angehängt. Um Redundanz durch das Auftreten mehrerer Cuepoint-Events, die die gleichen Daten enthalten, aber von unterschiedlichen Videodisplays stammen, zu vermeiden, ist es demnach notwendig, die Cuepoints nur an ein Videodisplay anzuhängen. Im *ViLM*-Player sind immer maximal vier VideoDisplays sichtbar. Durch die mittels Drag & Drop beschriebene Austauschmöglichkeit der Videoquellen ist es nicht sinnvoll, die Cuepoints einem dieser Videodisplays zuzuweisen. Daher wird

stattdessen ein im Folgenden „Mastervideo“ genanntes Videodisplay erzeugt, das zu jeder Zeit im Hintergrund mitläuft und Referenz für die eigentlichen, sichtbaren Videodisplays ist. Dieses Mastervideo bekommt die aus den Phasen erzeugten Cuepoints über die Klasse «*CuePointManager*» zugewiesen und löst die Cuepoint-Events aus.

2.9.3 Video-Initialisierungsprozess

Der Initialisierungsprozess der Videos wird gestartet, wenn durch das Öffnen einer *ViLM*-Projektdatei neue XML-Daten gelesen werden und die damit verbundenen Datenobjekte gefüllt werden. In der Methode «*setXMLDataObjects*» der AS-Datei "xmlManager.as" wird zunächst die Methode «*initializeVideos*» aufgerufen. Darin wird die Container-Komponente «*videoArea*» (nachfolgend als „Videoarea“ bezeichnet), die die Videopaneln und die Controller-Komponente «*videoControl*» („Videocontroller“) zur Steuerung des Playbacks enthält, als auch die beiden Leisten im Fußbereich des *ViLM*-Players für Benutzereingaben gesperrt. Dadurch wird ein grüliches Overlay über diese UI-Elemente gelegt und sie reagieren nicht mehr auf Maus- oder Tastaturereignisse. Möglich wird dies, weil alle Komponenten, die von der Klasse «*UIComponent*» erben ein Boolean-Attribut «*enabled*» besitzen, für das die betreffende Komponente freigegeben oder gesperrt wird. Innerhalb des *ViLM*-Players wird das genutzt, um Fehlermeldungen vorzubeugen, die durch übereilte Methodenaufrufe (zum Beispiel durch Benutzereingaben) an die Videodisplays entstehen würden.

Zu Beginn des Initialisierungsprozesses der Videodisplays steht die Anzeige von Informationen zur gesamten Unterrichtseinheit, durch Einblendung des sogenannten «*unitInformationPopup*». Dieses Popup kann aber auch später durch Klicken auf den Einheit-Button wieder eingeblendet werden. Um eventuelle Artefakte von Videoinformationen von vormals geöffneten Projekten zu vermeiden, werden daraufhin die Methoden «*initBreadCrumpTrail*» und «*setVideoInfoString*» aufgerufen. Sie löschen veraltete Einträge im Breadcrumptrail und im «*videoInfoString*». Da während des XML-Leseprozesses das Datenarray «*assets*» mit den URLs zu den Materialien bereits gefüllt worden ist, wird nun die Methode «*initialize*» auf dem Repeater-Objekt «*assetRepeater*» aufgerufen. Dadurch wird für jedes Material ein seinem Dateityp entsprechendes Icon in der rechten Fußleiste erzeugt, welches das Material symbolisiert und anklickbar ist. In der Variablen «*videoDisplayNumber*» wird die Anzahl der maximal sichtbaren Videodisplays festgelegt. Ist die Anzahl der Videoquellen größer als 4, so ist «*videoDisplayNumber*» gleich 4. Für jedes zu erzeugende Videodisplay werden nun die Arrays «*videoReady*» und «*videoMeta*» mit dem Boolean-Wert "false" gefüllt. Der Zweck der dahinter steckt, wird im Folgenden verdeutlicht.

Um Fehler beim Playback der Videos vorzubeugen, soll die Freischaltung von Steuerungsmöglichkeiten im Videocontroller für den Nutzer erst dann erfolgen, wenn alle Videodisplays nach Setzen ihres Attributs «source» ein «VideoEvent.READY» („Videoready-Event“)

und ein «MetaDataEvent.METADATA_RECEIVED» gesendet haben. Da aber ein Eventhandler in ActionScript als Parameter immer nur einzig das auslösende Event als Parameter hat, weiß ein bestimmter Eventhandler für ein Videoready-Event eines Videodisplays nichts von einem ebensolchen Event auf einem anderen Videodisplay. Daher musste eine andere Lösung gefunden werden, die Menge aller Videoready-Events abzuwarten, bevor der Initialisierungsprozess der Videos fortgesetzt werden kann.

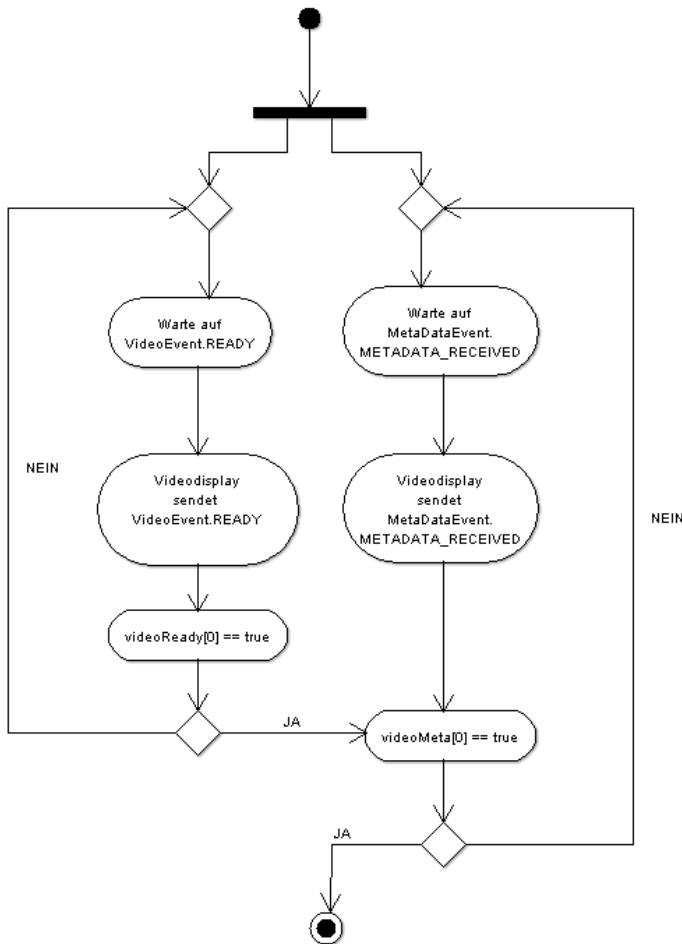


Abbildung 5: Video-Initialisierungsprozess

Alle Videodisplays rufen bei Auftritt des Videoready-Events den gleichen Eventhandler «onVideoReady» auf. Darin wird ein "false" - Element aus dem Array «videoReady» entfernt. In ActionScript lassen sich Arrays wie Listen behandeln, das bedeutet es gibt eine Methode «pop», die das erste Element entfernt und zurückliefert und eine Methode «push», die ein neues Element hinten anhängt. Vermöge der «push» - Methode wird nun ein "true" - Element am Ende von «videoReady» eingefügt. Der Eventlistener für das Videoready-Event wird für das betreffende Videodisplay noch entfernt. Schließlich wird überprüft, ob das erste Element in der Liste ein "true" ist. Das ist nämlich genau dann der Fall, wenn bei alle Videodisplays dieses Event aufgetreten ist und damit die gleichen Schritte durchlaufen worden sind.

Das Gleiche geschieht innerhalb des Video-Initialisierungsprozesses mit den sogenannten «MetaDataEvent.METADATA_RECEIVED» - Events („Metadata-Event“), die ein Videodisplay ebenfalls nach Setzen des «source»-Attributs versendet.

Dies hat seinen Grund vorrangig darin, dass eine entscheidende Größe zur Initialisierung der diversen Elemente des Videocontrollers die Dauer der einzelnen Videos ist. Die Klasse «*VideoDisplay*» stellt dafür im einfachsten Fall das Attribut «*totalTime*» zur Verfügung, das bei einem lokalen Flash-Video ab Version 1.1 die Dauer des enkodierten Videos speichert. Allerdings lässt sich die Länge eines Flash-Videos, das als Stream vorliegt, nicht so leicht bestimmen. Hier muss man stattdessen darauf warten, dass nach Aufbau des Streams Metadaten über das Video vom Streamingserver verschickt werden. Denn diese enthalten immer die Variable "*duration*", welche die Dauer eines Videos enthält. Die Metadaten werden aber auch bei lokalen Videodateien aus den Flash-Video-Dateien selber ausgelesen. Erst wenn man die Dauer jedes Videos kennt, lässt sich z.B. das Mastervideo als dasjenige mit der längsten Spieldauer (abgezogen seiner Synchronisationszeit) bestimmen.

Zunächst wird jedoch die Methode «*initializeVideoDisplays*» aufgerufen, in der das Datenarray «*videoPreviews*» mit «*Object*»-Elementen gefüllt wird, die die Attribute: «*videoSource*», «*videoName*», «*videoNumber*» und «*videoSyncTime*» enthalten, welche die Daten aus den "VideoSource" - Knoten der *ViLM*-Projektdatei enthalten. Das Array «*videoPreviews*» bildet den Dataprovider für die MXML-Komponente «*videoPreviewRepeater*».

Auf dieser Komponente wird nun die standardmäßig vorhandene Methode «*initialize*» aufgerufen, die unter dem geänderten Dataprovider das Rendern und Initialisieren der UI-Komponenten vornimmt, die den Videopreview-Bereich repräsentieren. Wenn die Initialisierung dieser Komponente beendet ist, was durch ein "*FlexEvent.INITIALIZE*" registriert wird, wird die Methode «*onVideoPreviewRepeaterInitialize*» aufgerufen, die zunächst, um Redundanz zu vermeiden und Artefakte zu entfernen, alle noch vorhandenen *child*-Elemente der Videoarea entfernt. Nun werden die Videopaneln, die die Videodisplays enthalten, erzeugt. Sie sind Objekte der Klasse «*Panel*», die als *child*-Element in ihrem Inhaltsbereich jeweils ein Objekt der Klasse «*VideoDisplay*» haben. Die Videodisplays werden mit Daten aus dem Dataprovider des «*videoPreviewRepeater*» versorgt. Für das Attribut «*source*» wird der Wert von "*videoSource*", für «*title*» der Wert von "*videoName*" zugewiesen. Jedes Videodisplay bekommt darüber hinaus folgende initialen Attributzuweisungen:

- `autoPlay = "false"`
- `autoRewind = "false"`
- `bufferTime = "3.0"`
- `maintainAspectRatio = "false"`
- `styleName = "videoPanel"`

Um auch später Zugriff auf diese Videodisplays zu haben, wird für jedes Videodisplay ein Objekt erzeugt, das im Attribut «*video*» eine Referenz auf das jeweilige Videodisplay enthält und darüber

hinaus die aus der *ViLM*-Projektdatei ausgelesene Synchronisationszeit im Attribut «*offset*» speichert. Wenn, wie oben bereits beschrieben, bei allen Videodisplays das *Videoready*-Event aufgetreten ist, ruft der Eventhandler «*onVideoReady*» die Methode «*setVideoDisplays*» auf. In ihr werden die «*offset*» - Attribute aller Videodisplays um die geringste Synchronisationszeit eines Videodisplays verkürzt, was nachher die Synchronisierung der Videos bei Positionssprüngen vereinfacht. Denn dann müssen bei Steuerungsaufrufen nur noch die relativen Offsets bezogen auf dieses Startoffset berücksichtigt werden.

Weiterhin werden hier die Eventlistener für die Drag & Drop - Operationen mit den Preview-Elementen gesetzt.

Wie bereits Eingangs dieses Abschnitts angedeutet, ist zudem das Entsenden aller "*MetaDataEvent_METADATA_RECEIVE*" - Ereignisse der Videodisplays von substantieller Bedeutung für das synchrone Playback der Videos. Wenn nun das Array «*videoMeta*» an erster Position ein "*true*" stehen hat, dann ist bei allen Videodisplays dieses Event aufgetreten. Empirisch hat sich dabei herausgestellt, dass dies in nahezu hundert Prozent der Fälle immer erst nach dem Auftreten der *Videoready*-Events passiert. Deshalb wird in der Methode «*receiveMetaData*» der AS-Datei "*metaManager.as*" ein Objekt der Klasse «*EventDispatcher*» benutzt, um ein "*Event.COMPLETE*" in den Eventflow zu entsenden. Das Entsenden dieses Events durch das Objekt «*videoMetaDispatcher*» wird durch einen Eventlistener in der Methode «*setVideoDisplays*» registriert und als Eventhandler die Methode «*onAllMetaEvents*» aufgerufen.

Sie stellt den nächsten Schritt innerhalb des Initialisierungsprozesses der Videos dar. Dann liegt nämlich für alle Videodisplays die *Metadata*-Variable "*duration*" vor, mit der nun der Index desjenigen Videos innerhalb des Arrays «*videoDisplay*» bestimmt werden kann, das am längsten ist. Die Variable "*totalVideoTime*" speichert diese Zeit in Sekunden und in "*totalDisplayTime*" ist ein im Format "*hh:mm:ss*" formatiertes «*Date*» - Objekt mit dieser Zeit hinterlegt.

Damit wird nun «*createMasterDisplay*» aufgerufen, wo das entsprechende Video nochmal als nicht-sichtbares Videodisplay im Hintergrund erzeugt wird. Dieses im Folgenden „*Mastervideo*“ genannte Videodisplay ist die Grundlage aller Synchronitätmethoden und ihm ist eine Instanz der Klasse «*CuePointManager*» zugewiesen. Es löst also bei Erreichen der entsprechenden Zeiten im Playback die *Cuepoint*-Events aus. Hat das *Mastervideo* dann ebenfalls sein *Videoready*-Event empfangen, wird als letztes Glied in der Kette die Methode «*setVideoControls*» aufgerufen, in der unter anderem verschiedene Eventlistener an das *Mastervideo* angehängt werden, deren Eventhandler die UI-Elemente updaten, die Informationen während des Playbacks anzeigen.

Die in *videoNumberButtons* repräsentierten Buttons dafür, ob 1,2 oder 4 Videopaneln sichtbar sind, werden in Abhängigkeit des Wertes in *videoDisplayNumber* angezeigt. Die Methode *«layoutVideoDisplays»* layoutet die Videopaneln in Abhängigkeit davon, wie viele dieser Panels überhaupt existieren und wie viel Platz die Videoarea bereitstellt. Damit sich keine Überschneidungen der Tonspuren ergeben, ist zu Beginn immer das erste Videodisplay auf eine Lautstärke von "0.5" (50 %) eingestellt, während alle anderen Videos stumm geregelt sind.

Falls in der *ViLM*-Projektdatei überhaupt Phasen existieren, wird außerdem die Methode *«initPhaseInformation»* aufgerufen, die mögliche Inhalte der UI-Komponenten *«xmlPhases»*, *«xmlPhaseDescription»* und *«xmlPhaseComment»* löscht und gegebenenfalls Phaseninformationen einer Phase anzeigt, die zum Zeitpunkt 0.0 s beginnt.

Weiterhin werden in *«setPlayheadStartTimes»* nun die eigentlichen Synchronisationszeiten aus der *ViLM*-Projektdatei als Playback-Startzeiten für die Videos gesetzt. Dann wird noch das Maximum des Objektes *«sliderPlaybackDragging»* der Klasse *HSlider*, das den *Playback_Slider* darstellt gesetzt, so dass ein Verschieben des Reglers nun sinnvolle Werte zurückliefert. Zuletzt werden alle mit der Videosteuerung befassten UI-Komponenten mit einem Eventlistener belegt, die als Eventhandler die mit den Buttons assoziierten Methoden aufrufen.

2.9.4 Ursachen der Nicht-Synchronität

Jedes Videodisplay verfügt über ein Attribut *«state»*, das seinen aktuellen Zustand speichert. Wenn die Startpositionen für alle Videos eines *ViLM*-Projektes richtig gesetzt sind und man das Playback startet, laufen die Videos insbesondere im Fall von gestreamten Videos häufig nicht synchron, sondern im Zehntelsekundenbereich versetzt. Dies resultiert daher, dass der Aufruf der *«play»* - Methode der Klasse *«VideoDisplay»* das Playback nicht unmittelbar startet, sondern für eine kurze Weile immer erst in den Zuständen *"seeking"* oder *"buffering"* mündet, in denen das Video noch pausiert. Der Zeitraum bis das Video dann tatsächlich im Zustand *"playing"* ist, erstreckt sich im Millisekunden- bis Sekundenbereich. Je nach Netzwerkauslastung mal mehr, mal weniger. Als Lösungsansatz für dieses Problem wurde folgender Ansatz versucht:

Ein Eventlistener für ein *"VideoEvent.STATE_CHANGE"* - Event wird jedem Videodisplay angehängt. Dieses Ereignis wird unmittelbar ausgesendet, wenn ein Videodisplay eine Zustandsänderung erfährt. Als Eventhandler diente eine Methode, in der der neue Zustand abgefragt wurde. Wenn sich alle Videodisplays im Zustand *"playing"* befinden sollten, ein Videodisplay seinen Zustand aber auf *"seeking"* oder *"buffering"* änderte, wurden alle anderen Videodisplays pausiert. Änderte nun das im Zustand *"seeking"* oder *"buffering"* befindliche Videodisplay seinen Zustand zurück auf *"playing"*,

wurde auf den übrigen Videodisplays wieder die «*play*» - Methode aufgerufen. Das führte aber nicht zu dem gewünschten Erfolg, weil dann andere Videodisplays auch wieder zunächst in die Zustände "*seeking*" und "*buffering*" wechselten, wodurch sich letztlich eine Art Deadlock-Prozess ergab. Möglicherweise ist ein Videostream netzwerkbedingt nicht soweit aufrecht zu erhalten, dass das Playback nach Unterbrechungen immer unmittelbar fortsetzbar ist.

Allerdings hat die Nicht-Synchronität intern auch damit zu tun, dass für die Videodisplays das Attribut «*bufferTime*» initial auf einen Wert von "3.0" Sekunden festgelegt wird. Diese 3 Sekunden an Videodaten werden dann von jedem Videodisplay nach Aufrufen der «*play*» - Methode zunächst geladen und damit vorgehalten. Zweck dieser Bufferzeit ist es, ein flüssiges Playback zu ermöglichen, indem immer genügend Videodaten bereits im Hauptspeicher des Client-Rechners vorhanden sind. Deswegen wurde neben dem im Folgenden beschriebenen Synchronisationsprozess bei Klicken auf den "*Sync*" - Button, noch im Anwendungsmenü unter „Extras“ > „Bufferzeit“ ein Dialog in den *ViLM*-Player integriert, der es dem Anwender ermöglicht, unter der Prämisse einer schnellen Netzwerkverbindung die Bufferzeit der Videodisplays manuell im Intervall [0, 3.0] Sekunden einzustellen und damit gegebenenfalls zu reduzieren. Eine Verkleinerung der Bufferzeit wird für alle Videodisplays in der Videoarea durchgeführt, nicht jedoch für das im Hintergrund mitlaufende Mastervideo. Da es nicht sichtbar ist und nur als zeitliche Referenz dient, also alle übrigen Videodisplays nur dann spielen, wenn das Mastervideo spielt, hat hier eine kleinere Bufferzeit keine Auswirkungen auf die Synchronität der sichtbaren Videodisplays. Einige Tests dieses Vorgehens wurden mit Bandbreiten im T1 bis T3-Bereich durchgeführt. Eine Bufferzeit von 0.0 Sekunden führt dann zu einer erheblichen Verbesserung der Synchronität, allerdings steigt damit natürlich die Wahrscheinlichkeit von Aussetzern in Korrelation mit der Qualität der Netzwerkverbindung.

2.9.5 Prozeß des Synchronisierens

Der Synchronisationsprozess wirkt der im vorherigen Abschnitt beschriebenen Problematik entgegen. Er läuft nach folgendem Schema ab:

Zu Beginn wird in der Variablen "*flagMasterPlaying*" gespeichert, ob sich das Mastervideo (und damit alle Videodisplays) gerade im Zustand "*playing*" befand. Danach wird auf allen Videos inklusive des Mastervideos die Methode «*pause*» aufgerufen und mittels eines Eventlisteners für ein "*VideoEvent.STATE_CHANGE*" im Eventhandler «*onPause*» überprüft, dass tatsächlich alle Videos in den Zustand "*paused*" gewechselt haben. Erst dann geht es im Programmfluss weiter. Nun wird die aktuelle Spielposition des Mastervideos bestimmt und die Playback-Positionen aller anderen Videodisplays auf diesen Wert synchronisiert. Dadurch wechseln sie in den Zustand "*seeking*" und

sobald die Videodaten ab der entsprechenden Position vom Streamingserver eintreffen, bzw. im lokalen Fall die entsprechende Stelle in den Videodaten gefunden wurde, wechselt jedes Videodisplay wieder in den Zustand *"paused"*. Dies wird wiederum für alle Videodisplays in *«onPlayheadChange»* überprüft. Im Fall, dass *"flagMasterPlaying"* gleich *"true"* war, wird nun die *«play»* - Methode auf allen Videodisplays inklusive des Mastervideos aufgerufen und getestet, dass alle Videodisplays in den Zustand *"playing"* wechseln. Erst wenn das eingetreten ist, wird die Videoarea mitsamt des Videocontrollers für den Benutzerzugriff freigegeben.

2.9.6 Implementierung der Dragging-Funktionalität

Das Drag & Drop - Feature der Videosource-Icons auf die Videodisplays liefert einen wesentlichen Beitrag zur Herstellung von Benutzerfreundlichkeit, weil dies eine sehr intuitive Möglichkeit bedeutet. Um es umzusetzen, wird zu Beginn des Video-Initialisierungsprozesses zunächst das Datenarray *«videoPreviews»* mit den Daten aller "VideoSource" - Knoten der *ViLM*-Projektdatei gefüllt. Daraufhin wird der *«videoPreviewsRepeater»* initialisiert, was die erwähnten Icons und dazu die bereits erwähnten Tooltip-Popups erzeugt. Um nun einen Drag-Drop mit diesen Icons zu ermöglichen, registriert ein Eventlistener darauf ein "MouseEvent.MOUSE_DOWN", was bedeutet, dass der Nutzer auf ein Icon klickt und die Maustaste gedrückt hält. Als Eventhandler dient die Methode *«beginDragDrop»*, die die Variable *"name"* des Icons als sogenannte *«DragSource»* speichert. Dieser Variable wurde beim Erzeugen jedes Preview-Icons die URL der Videoquelle des damit implizierten Videos übergeben. Zudem wird für die nachfolgend stattfindende Bewegung des Drag-Objektes eine Bitmap-Kopie des Icons erstellt und während des Drag-Drop angezeigt. Dies erledigt eine Instanz der Klasse *«DragManager»*, welche zudem dafür zuständig ist, Ziele für einen Drag-Drop zu definieren, die jegliche sichtbare UI-Komponenten der Anwendung sein können. Bewegt man das Drag-Objekt außerhalb eines solchen Ziels, wird auf dem Bitmap ein Stop-Symbol eingeblendet, um anzudeuten, dass eine Drop-Operation hier nicht möglich ist.

Statt diese Icons in ein Bitmap zu kopieren, könnte man dem *«DragManager»* auch ein festes Bitmap übergeben, allerdings wird in der implementierten Version auf jedem Preview-Icon noch ein Text "Video" zusammen mit einer laufenden Nummer der jeweiligen Videoquelle angezeigt. Dies wird auf dem kopierten *«Bitmap»* - Objekt dann mit angezeigt. Zudem stammt diese Version aus einer Umsetzung des *ViLM*-Players, bei der die Icons tatsächliche Video-Previews waren (siehe Abschnitt 2.8.1), das heißt:

Für alle Videoquellen wurden Videodisplays erzeugt, die synchron mit dem Hauptplayback liefen und die die Video-Previews darstellten. Bei dieser Realisierung war der Einsatz von Bitmap-Kopien sehr sinnvoll, weil in dem Bitmap ja die letzte Szene dargestellt wurde, die beim Auslösen der Drag & Drop

- Operation gerade sichtbar war. Dies unterstützte den Nutzer darin, zu wissen, welche Videoquelle er für den Drag-Drop ausgewählt hatte.

Um eine Bitmap-Kopie eines beliebigen *child*-Elementes der Klasse «*DisplayObject*» zu ermöglichen, gibt es innerhalb von Flex die Klasse «*ImageSnapshot*» mit der Methode «*captureBitmapData*», die als Parameter genau das gewünschte Element übergeben bekommt und daraus die Bitmap-Daten in Form eines Objekts der Klasse «*BitmapData*» speichert. Dem «*DragManager*» übergibt man dies dann als gecastetes «*Bitmap*» - Objekt. Wenn man die Drag-Objekte über eine sichtbare UI-Komponente bewegt, sendet sie ein "DragEvent.DRAG_ENTER" - Ereignis. Für dieses Event wird auf jedem Videodisplay ein Eventlistener gesetzt, der als Eventhandler die Methode «*dragEnterHandler*» aufruft. Darin wird der «*DragManager*» angewiesen, dasjenige Element, das das Ereignis gesendet hat, als Ziel eines Drag-Drops zu akzeptieren, wodurch das eingeblendete Stop-Symbol auf dem Drag-Objekt verschwindet. Lässt der Nutzer hier die Maus wieder los, so sendet das betreffende Element ein "DragEvent.DRAG_DROP" - Ereignis, für das ein Listener schließlich den Eventhandler «*finishDragDrop*» aufruft. Darin werden alle Videodisplays inklusive des Mastervideos pausiert und die letzte Playback-Position des Mastervideos gespeichert. Da sich aus dem Drag-Drop als Ziel nur das Videodisplay, nicht aber sein übergeordnetes Objekt (das Objekt innerhalb des Datenarrays «*videoDisplay*», das neben dem Videodisplay-Objekt selbst auch das zum Video gehörende Synchronisationsoffset und seinen Namen enthält) gewinnen lässt, wird nun in der Menge der maximal vier Objekte von «*videoDisplay*» dasjenige gesucht, was das Ziel-Videodisplay enthält. Dann wird die Methode «*refreshVideoDisplay*» aufgerufen, die den eigentlichen Refresh eines Videodisplays nach einem "DragEvent.DRAG_DROP" vornimmt und das Datenobjekt aus «*videoDisplay*» übergeben bekommt. Für den Refresh eines Videodisplays werden genau die darin enthaltenen Informationen aktualisiert. Die neuen Daten werden aus dem Array "videoPreviews" gewonnen, wo genau dasjenige Objekte gesucht wird, was die im Objekt «*DragSource*» gespeicherte Variable als Attribut «*videoSource*» hat.

Zu beachten ist, dass auf dem aktualisierten Videodisplay die Methode «*play*» aufgerufen wird. In einigen Tests hat sich herausgestellt, dass Flex-intern die Klasse *VideoDisplay* eine Änderung des «*source*» - Attributs nur dann vollzieht, wenn das Playback auf dieser Komponente gestartet wird. Dies hat insbesondere mit den Methoden innerhalb der Flex-Klassen «*Video*» und «*VideoPlayer*» zu tun, auf denen die Klasse «*VideoDisplay*» aufsetzt. Ruft man für ein Videodisplay nur «*pause*» oder «*stop*» auf oder lässt das Videodisplay einfach weiterlaufen, während man das «*source*»-Attribut ändert, wird das Video nicht gewechselt. Erst der Aufruf von «*play*» ändert dies. Daher geschieht tatsächlich Folgendes:

Nach dem Aufruf von «*play*» auf dem aktualisierten Videodisplay wird relativ unmittelbar die Methode «*pause*» aufgerufen, nämlich dann, wenn das Objekt kurzzeitig im Zustand "playing" war.

Wenn das Videodisplay dann wieder im Zustand *"paused"* ist, wird es wieder sichtbar gemacht. Schließlich werden für alle Videodisplays die Playback-Positionen auf den gemeinsamen Wert der Variablen *"playheadTimeAfterDrag"* synchronisiert und gegebenenfalls wieder gestartet.

2.9.7 Einbindung des Aperture-Frameworks

Eine Anforderung, die an die Entwicklung des *ViLM*-Players gestellt wurde, war die Möglichkeit externe Materialien einzubinden. Das bedeutete, dass die in der XML-Datei gespeicherten URLs zu Materialdateien im Player zum Öffnen angeboten werden. Zunächst schien eine Umsetzung dieser Anforderung nicht möglich, weil das Flex-Framework keine Möglichkeit bietet, Anwendungen auf Betriebssystemebene durch einen entsprechenden *"execute"* - Befehl zu starten. Innerhalb von Flex werden zwar Klassen angeboten, die zum Beispiel PDF-Dateien zur Anzeige bringen oder typische Grafikformate des Web (*"JPEG"*, *"PNG"*, *"GIF"* ...) darstellen. Allerdings hebt das den Mangel nicht auf, weil viele Materialien zu Annotationen bei dem Beispiel der Unterrichtsstunden in typischen Office-Formaten vorliegen (zum Beispiel *"DOC"*). Man kann aber davon ausgehen, dass in den entsprechenden Einsatz-Kontexten des *ViLM*-Players die für solche Dateiformate zuständige Software lokal auf den Clients installiert ist. Also lag es nahe, für die Materialdateien die Möglichkeit anzubieten, sie durch Klicken auf das entsprechende Symbol im *ViLM*-Player durch die mit ihrem Dateityp assoziierte Anwendung über das Betriebssystem öffnen zu lassen. Dafür gab es vor der Version 3.0 von ActionScript, den *"exec"* - Befehl (Webseite, ::Chapter 1- Actionscript Command Lines:: 2008), der als Parameter der Top-Level-Methode *«fscommand»* aufgerufen werden konnte. Für ActionScript 3.0 existiert diese Möglichkeit aufgrund gestiegener Sicherheitsanforderungen nicht mehr (Webseite, flash.system package 2008). Daher musste eine andere Möglichkeit gefunden werden. Es bot sich das Aperture-Framework an, das genau diesen Mangel von Flex beseitigt. „Launch native applications and documents with the provided *apssystem* library“ wird hier als Feature dieses freien Frameworks genannt (<http://aperture.fluorinefx.com/>).

Um das Aperture-Framework einzubinden, müssen einige Schritte durchgeführt werden. Zunächst müssen die beiden Dateien *"apssystem.dll"* und *"fluorinepp.dll"* des Frameworks in den Source-Ordner des *ViLM*-Players kopiert werden. In der MXML-Anwendungsdatei *"vilmpay.mxml"* legt man dann über den Eintrag:

```
xmlns:aperture="com.fluorinefx.aperture.*"
```

einen Namensraum für Objekte des Aperture-Frameworks an. Zudem wird ein Objekt der Aperture-Klasse «*LocalObject*» innerhalb der Anwendung angelegt, das in ActionScript-Methoden über die «*id*» "lc" ansprechbar ist. Darin befindet sich ein Objekt «*method*» mit dem Namen "Execute", welches entsprechende Aufrufe an das Framework weiterleitet und eine Datei über das Betriebssystem öffnet. Zudem wird noch eine ActionScript-Datei "manager.as" im Ordner "aperture" angelegt, die die Methode «*execute*» mit den Parametern "operation", "execInfo" und "parameters" beherbergt. Setzt man für den Parameter "operation" den String "open" und übergibt dem Parameter "execInfo" einen String mit der vollständigen URL einer Datei, so wird sie damit über das Betriebssystem geöffnet. In der Methode «*downloadAsset*» der AS-Datei "assetManager.as" lassen sich nun über den Befehl

```
execute("open", downloadURL, "")
```

Materialdateien mit der URL "downloadURL" über die mit ihren Dateitypen gegebenenfalls assoziierten Anwendungen über das Betriebssystem öffnen.

3 Schlussbemerkung

Auch wenn sich anfänglich erhebliche Schwierigkeiten bei der Entwicklung des *ViLM*-Players ergeben haben, so ist nun doch eine Realisierung gelungen, die zwar absolute Synchronität des Videoplaybacks aufgrund der geschilderten technischen Probleme nicht realisiert, aber dennoch den geforderten Anwendungsszenarien genügt. Es existieren allerdings Unzulänglichkeiten der Software, die noch behoben werden müssen.

Zum einen wäre es wünschenswert, mehrere Instanzen des *ViLM*-Players öffnen zu können, um so einen schnellen Zugriff auf verschiedene Projekte oder Situationen zu haben. AIR stellt bis jetzt aber die Möglichkeit, die gleiche AIR-Anwendung mehrmals zu öffnen, nicht zur Verfügung. Daher muss für jede Situation, die betrachtet werden soll, eine andere Situation durch Öffnen einer neuen Datei gelöscht werden. Es können nicht mehrere Situationen in mehreren Instanzen des *ViLM*-Players gleichzeitig auf einem Rechner betrachtet werden. Weiterhin wäre es wünschenswert, die Materialien, die in der rechten Leiste eingeblendet werden, im Falle, dass ein entferntes *ViLM*-Projekt vorliegt (die Videos also gestreamt werden und die Dateien sich auf einem Webserver

befinden) herunter zu laden, zum Beispiel über einen Kontextmenü durch einen Mausklick mit der rechten Maustaste.

Weiterhin führt das Öffnen einer *VILM*-Projektdatei nicht immer einen spielbereiten Zustand des Players. Vermutlich gibt es intern noch Schwierigkeiten mit der Verarbeitung aller Events der Videodisplays, deren zeitliches Auftreten oft nicht exakt bestimmt werden kann. In einigen Fällen passiert es zum Beispiel, dass nach dem Auswählen einer entfernten *VILM*-Projektdatei, die Videos zwar geladen werden, der Initialisierungsprozess dann aber stoppt und die Videoarea für Benutzereingaben nicht freigegeben wird. Für diesen Fall wurde deswegen ein Reset-Feature implementiert, das den Zustand des Players zurücksetzt und damit eine neue Datei geöffnet werden kann.

Für eine Beschreibung zur Bedienung des Players und seiner Installation sei auf die in der Anwendung erreichbare Hilfeseite unter „Hilfe“ > „Inhalt“ verwiesen, die sich auch im Anhang dieser Arbeit findet.

Literaturverzeichnis

Assisi, Ramin. *Eclipse Java Entwicklung mit der Open-Source-Plattform*. München: Carl Hanser Verlag, 2004.

Benz, Brian, and John R. Durant. *XML Programming Bible*. Indianapolis, Indiana: Wiley, 2003.

Bösken, Michael. *Streaming Video & Web TV*. Hamburg: Druck Diplomica® Verlag GmbH, 2007.

Brössler, Peter, and Johannes Siedersleben. *Softwaretechnik*. München: Carl Hanser Verlag, 2000.

Brückmann, Albert, and Patrick Butterer. "Tutorial zu Adobe Air & Flex." *Digitale Medien*. BA Mosbach, 2006.

Collins, A., J. S. Brown, and S. E. Newman. *Cognitive Apprenticeship: Teaching the Crafts Of Reading, Writing and Mathematics*. 1989.

Don Box, Aaron Skonnard, John Lam. *Essential XML, XML für die Softwareentwicklung*. München: Addison-Wesley, Pearson Education Deutschland GmbH, 2001.

Fachgruppe Didaktik der Informatik. *Arbeitsblatt: Gesichtspunkte zur Beobachtung von Unterricht*. Universität Paderborn, Seminar "Methoden des Informatikunterrichts in Theorie und Praxis", 2006.

Magenheim, Johannes. "Vilm: Visualization of learning and teaching." In *Proceedings of World Conference on Educational Multimedia, Hypermedia and Telecommunications*, by Piet Kommers and Griff Richards, 1593-1594. Chesapeake, VA, 1999.

Niegemann, H. M. *Kompendium E-Learning*. Heidelberg: Springer, 2004.

Oestereich, Bernd. *Objektorientierte Softwareentwicklung*. München: Oldenbourg, 2004.

Ozer, Jan. "selecting the right codec." *streaming media magazine*, Februar 2007, S. 140-144.

Petzold, Jan. "Film ab für Flash (Know how: Flash Media Server 3)." *Linux Magazin*, Juli 2008, S. 100-105.

Rich Shupe, Zevan Rosser. *Learning ActionScript 3.0, A Beginners Guide*. Sebastopol, CA: O'Reilly, 2008.

Schulmeister, Rolf. *Grundlagen hypermedialer Lernsysteme*. Bonn: Addison-Wesley, 1996.

Shaw, Mary, and David Garlan. *Software Architecture*. Prentice Hall, 1994.

Tulodziecki, Gerhard, and Bardo Herzig. *Handbuch Medienpädagogik, Band 2*. Stuttgart: Klett-Cotta, 2004.

Webseite: "Adobe - Flex Quick Start Basics: Building components by using code behind." *Adobe*. 2008. http://www.adobe.com/devnet/flex/quickstart/building_components_using_code_behind/ (Zugriff am 4. September, 2008).

Webseite. "Adobe - Programmieren mit MXML und ActionScript." *Adobe*. 2008. http://www.adobe.com/de/devnet/flex/quickstart/coding_with_mxml_and_actionscript/ (Zugriff am 18. August, 2008).

Webseite: *Flash Video Guide*. Adobe. 2008.

http://www.adobe.com/devnet/flash/articles/video_guide.html (Zugriff am 20. August, 2008).

Webseite. "Chapter 1- Actionscript Command Lines:" *Beyond Flash*. 2008.

<http://www.beyondflash.com/content/ch01.htm> (Zugriff am 6. September, 2008).

Webseite: "Cover Pages: Standard Generalized Markup Language (SGML)." *Cover Pages*. 2008.

<http://xml.coverpages.org//sgml.html> (Zugriff am 30. August, 2008).

Webseite: *EventDispatcher*. Adobe. 2008.

<http://livedocs.adobe.com/flex/3/langref/flash/events/EventDispatcher.html> (Zugriff am 3. September, 2008).

Webseite: "flash.system package". Adobe. 2008.

<http://livedocs.adobe.com/flash/9.0/ActionScriptLangRefV3/flash/system/package.html> (Zugriff am 6. September, 2008).

Webseite: "Get oriented to Flex - Flex 3 Getting Started - Adobe Learning Resources." Adobe. 2008.

<http://learn.adobe.com/wiki/display/Flex/Get+oriented+to+Flex> (Zugriff am 30. August, 2008).

Webseite: *OpenUse - DIN EN ISO 9241*. 2008. <http://www.openuse.de/infobox/din9241.html> (Zugriff am 3. September, 2008).

Webseite: "Part 3: Useful Programming Functions." *Eclipse Tutorial*. 2008.

<http://eclipsetutorial.forge.os4os.org/programming.htm> (Zugriff am 1. September, 2008).

Webseite: "Standard ECMA-262." *Ecma International*. 2008. <http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf>

(Zugriff am 6. September, 2008).

Webseite: "Standard ECMA-357." *Ecma International*. 2008. <http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-357.pdf>

(Zugriff am 7. September, 2008).

Anhang

Ich versichere, dass ich die schriftliche Hausarbeit selbstständig verfasst und keine anderen Quellen und Hilfsmittel als die angegebenen benutzt habe. Alle Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen sind, habe ich in jedem einzelnen Fall unter genauer Angabe der Quelle deutlich als Entlehnung kenntlich gemacht. Das Gleiche gilt auch für die beigegebenen Zeichnungen, Kartenskizzen und Darstellungen.

Datum

Unterschrift

ViLM Player Hilfe

Über ViLM Player

- [Was ist ViLM Player?](#)
- [Mindestanforderungen](#)
- [Kontakt](#)

Kurzbeschreibung

- [Installation](#)
- [Anwendungsoberfläche](#)
- [Projektdatei öffnen](#)
- [Betrachten eines Videos](#)

Bedienung

- [Menü](#)
- [Playback](#)
- [Phaseninformationen](#)
- [Videoquellen](#)
- [Materialien](#)

Tastaturkürzel

[Über ViLM Player](#) [Zurück](#)

1. Was ist ViLM Player?

ViLM Player ist eine Videoplayer-Software zur plattformunabhängigen Präsentation von annotierten Videos (bis jetzt auf Windows und Mac). Sie basiert auf der Adobe® AIR™ Technologie und benötigt das AIR™-Framework zur Installation. Innerhalb des ViLM-Players können keine Videodateien geöffnet werden, sondern sogenannte ViLM-Projektdateien (.ann), die durch ein an der Universität Paderborn entwickeltes Synchronisations- und Annotationstool erzeugt werden. Sie enthalten alle notwendigen Informationen über die URLs der in einem Projekt enthaltenen Videodateien, die Phaseninformationen, sowie weitere URLs zu assoziierten Materialien. Die Videos können dabei entweder innerhalb eines ViLM-Projektes lokal abgelegt sein oder sie stammen von einem Streaming-Server.

2. Mindestanforderungen

Für den ViLM Player gelten mindestens die Systemanforderungen von Adobe® AIR™:

Windows

- Intel-Prozessor der Pentium-Klasse mit mindestens 2 GHz

- Windows 2000 mit Service Pack 4, Windows XP mit Service Pack 2 oder Windows Vista Home Premium, Business, Ultimate oder Enterprise
- 512 MB RAM; 32 MB VRAM

Mac OS X

- PowerPC G4 mit 1,8 GHz bzw. Intel Core Duo mit 1,33 GHz oder schneller
- Mac OS X Version 10.4.9 bis 10.5.1 (Intel oder PowerPC; Intel-Prozessor für H.264-Video erforderlich)
- 512 MB RAM; 32 MB VRAM

3. Kontakt

ViLM Player ist eine im Rahmen einer Staatsarbeit an der Universität Paderborn entwickelte Software, die u.a. der Unterrichtsevaluation für Lehramtsstudenten dienen soll. Sie wurde innerhalb der Fachgruppe Didaktik der Informatik konzipiert und umgesetzt. Ansprechpartner:

Prof. Dr. J. S. Magenheim (Sprecher)
AG Didaktik der Informatik
Universität Paderborn
jsm@uni-paderborn.de

[Kurzbeschreibung](#) [Zurück](#)

4. Installation

Windows

Bevor Sie den ViLM Player installieren können, müssen Sie zunächst Adobe® AIR™ auf ihrem Rechner installieren. AIR™ (Adobe Integrated Runtime) ist eine Software der Firma Adobe®, die es ermöglicht, Webanwendungen im Stil von Web 2.0 als lokale Desktopanwendungen zu nutzen. Man kann es unter <http://get.adobe.com/de/air/> kostenlos downloaden. Programme auf der Basis von AIR™ werden in wenigen Schritten installiert. Das Installationsprogramm des ViLM Players liegt in der Datei install.air bereits vollständig vor. Nachdem Sie AIR™ installiert haben, öffnen Sie diese Datei (durch ein Paket-Icon mit dem Namen ".air" wird angezeigt, dass es sich um ein installierbares Programm für AIR™ handelt). Dadurch startet ein automatisches Installationsprogramm, was den ViLM Player standardmäßig im Ordner "C:\Program Files (x86)" installiert. Sie können aber auch jeden anderen Ordner wählen. Es erscheint der folgende Dialog, wählen Sie hier "Installieren":



Wurde der selbe ViLM Player bereits einmal installiert, bietet das Installationsprogramm stattdessen die Optionen "Deinstallieren" und "Jetzt ausführen" an. Die Möglichkeit den ViLM Player zu deinstallieren besteht auch alternativ über die Windows-Menüs "Systemsteuerung" > "Programme und Funktionen" > "Deinstallieren/ändern". Liegt eine neue Version der Software vor, so klicken Sie wiederum auf "install.air" - das Installationsprogramm nimmt die Aktualisierung des ViLM Players vor.

5. ViLM Player Oberfläche



Im Kopfbereich des ViLM Players befindet sich das Menü der Anwendung. Der Bereich in der Mitte des Anwendungsfensters ist für die VideoPanels reserviert. Hier werden in Abhängigkeit der verfügbaren Videoquellen bis zur vier Videos angezeigt.

Direkt darunter befinden sich die Steuerungselemente für das Playback der Videos (siehe dazu [9.](#)). Im Fußbereich befinden sich zwei Leisten. In der linken Leiste werden bei geöffneter ViLM Projektdatei die verschiedenen Videoquellen als Icons angezeigt, die dann über einen Drag-Drop auf die VideoPanels gezogen werden können (siehe dazu [11.](#)). In der rechten Leiste werden Links zu den in einem Projekt enthaltenen Materialien als Icons angezeigt. Sie werden in den assoziierten Anwendungen auf Betriebssystemebene über einen Doppelklick geöffnet (siehe dazu [12.](#)). Rechts im Anwendungsfenster finden sich zwei Panels, die die in einer Phase enthaltenen Informationen, also ihre Beschreibung und ihren Kommentar anzeigen.

6. ViLM Projektdatei öffnen

ViLM Player erlaubt nicht das direkte Öffnen von Videodateien in von AIR™ unterstützten Videoformaten (".flv", ".mp4"). Stattdessen öffnet man im ViLM Player eine sogenannte ViLM Projektdatei (".ann"), die einen XML-basierten Container darstellt, der alle relevanten Informationen eines ViLM Projektes enthält. In einer ".ann"-Datei sind die URLs der Videodateien und Materialien gespeichert, sowie die eigentlichen Annotationen der Videos, also Informationen über Phasen, Start- und Endzeiten, sowie darin abgelegte Beschreibungen und Kommentare. ViLM Player unterstützt zwei Arten, Projektdateien zu öffnen. Zum einen über einen Dateibrowser-Dialog, um eine lokale ".ann"-Datei zu öffnen, oder als URL-Request einer entfernten Datei über das HTTP-Protokoll. Für ersteres wählen Sie "Datei" > "Lokale Projektdatei öffnen", für letzteres "Datei" > "Remote Projektdatei öffnen". Wenn Sie eine entfernte ViLM Projektdatei öffnen wollen, müssen Sie eine vollständige URL vom Typ "http://www.server.xyz/pfad/./datei.ann" angeben. Bestätigen Sie ihre Eingabe durch Enter. Das Öffnen und Initialisieren der Videos dauert immer einen kurzen Moment, solange dies nicht erreicht ist, ist der gesamte Videobereich verdunkelt und für Benutzereingaben gesperrt.

7. Abspielen der Videos

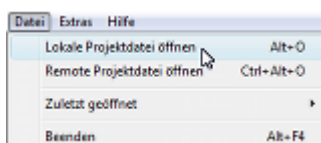


Nachdem die Videos eines ViLM Projektes vollständig geladen sind, wird der Steuerungsbereich der Videos entsperrt und die Anfangsbilder der Videos sind bereits sichtbar. Klicken Sie dann auf den "Play"-Button, startet das Playback der Videos. Im Falle, dass die Videos sehr groß sind oder gestreamt werden, starten die Videos (abhängig von der Geschwindigkeit des Rechners und der Bandbreite der Netzwerkverbindung) möglicherweise nicht gleichzeitig. In diesem Fall hilft der "Sync"-Button weiter, er setzt die Position aller Videos auf die Position des längsten Videos und startet das Playback erneut, so dass eventuell wiederholtes Klickes auf diesen Button die Synchronität der Videos erhöht.

Bedienung [Zurück](#)

8. Menü

Das Menü des ViLM Players beinhaltet die drei Einträge "Datei", "Extras" und "Hilfe". Darüber erreicht man die folgenden Funktionen:



Wählen Sie "Lokale Projektdatei öffnen" oder drücken Sie innerhalb des ViLM Players die Tastenkombination "ALT-O", um eine lokale ViLM Projektdatei auf ihrem Rechner zu öffnen. Sie befindet sich normalerweise in einem ViLM Projektordner, der mindestens den Unterordner "Videos" besitzt. Wählen Sie "Remote Projektdatei öffnen" oder drücken Sie innerhalb des ViLM Players die Tastenkombination "STRG-ALT-O", um eine entfernte Projektdatei auf einem HTTP-Server zu öffnen. Achten Sie auf die Angabe einer vollständigen URL und der Dateierdung ".ann". Im Menüpunkt "Zuletzt geöffnet" finden Sie die letzten fünf erfolgreich geöffneten Projektdateien. Wählen Sie "Beenden" oder die Tastenkombination "ALT+F4", um den ViLM Player zu beenden.



Wählen Sie den Menüpunkt "Reset", um einen vollständigen Reset des ViLM Players durchzuführen. Dies bietet sich z.B. an, wenn sich der Player durch Netzwerkprobleme in einen Zustand befindet, in dem er nicht mehr auf Benutzereingaben reagiert. Im Menüeintrag "Phasen" finden sich nach dem Öffnen einer ViLM-Projektdatei die verschiedenen Phasen als Untereinträge. Wählt man hier eine Phase aus, so wird unabhängig von der aktuellen Position der Videos ein Fenster eingeblendet, in dem die Beschreibung und der Kommentar zu dieser Phase eingeblendet wird. Wählen Sie "Inhalt" aus dem Menüeintrag "Hilfe" aus, um diese Hilfeseite angezeigt zu bekommen.

9. Playback

Die wesentlichen Steuerelemente "Play", "Pause" und "Stop" des ViLM Players sind klassischerweise benannt und selbsterklärend. Darüber hinaus befindet sich in der Steuerungsleiste noch eine Checkbox "Originalgröße". Setzen Sie hier ein Häkchen, um das originale Seitenverhältnis der Videodateien herzustellen. Das Buttonpaar "Prev", "Next" ist nur dann entsperrt, wenn in einer geöffneten Projektdatei tatsächlich Phasen enthalten sind. In diesem Fall dienen die Buttons dann dem Anspringen der nächsten bzw. vorherigen Phase. Die "Sichtbar"-Checkboxen "1", "2", "4" sind dazu da, VideoPanels aus- und wieder einzublenden. Enthält eine Projektdatei nur eine Videoquelle sind die Checkboxen "2" und "4", im Falle von zwei Videoquellen die Checkbox "4" nicht sichtbar. Der "Sync"-Button dient, wie in [7](#) beschrieben, dem Synchronisieren der Videos. Daneben finden Sie im Menü "Extras" noch den Eintrag "Buffer". Klicken Sie bei einem geöffneten ViLM Projekt darauf, um einen Dialog zu öffnen, in den Sie einen Wert zwischen 0 und 3.0 eingeben können. Er legt die Bufferzeit für alle Videos fest, also wieviele Sekunden an Videodaten im Speicher vorgehalten werden, bevor das Playback beginnt. Standardmäßig sind alle VideoPanels auf einen Wert von 3.0s eingestellt. Haben Sie jedoch eine sehr schnelle Internetverbindung, können sie diesen Wert deutlich reduzieren (Bei Breitbandverbindungen im T1-T3-Bereich können Sie diesen Wert auch auf 0 setzen) und erreichen damit eine deutlich gesteigerte Synchronität aller Videos. Im unteren Bereich der Videosteuerung werden bei gestreamten Videos noch Metainformationen über sie angezeigt. Da dies für alle Videos einen Overhead an Informationen bedeuten würde, wird hier als Referenz nur das längste Video genommen. Ein

zentrales Steuerelement für das Videoplayback ist weiterhin ein Positionsslider. Er läuft während des Playbacks mit und zeigt so die aktuelle Position der Videos an. Verschieben Sie ihn, ändern Sie diese Position. Zudem werden gegebenenfalls eingeblendete Phaseninformationen aktualisiert. Über den Button "Volumes" öffnet sich ein Popup, in dem sich die Lautstärken für die Videos zwischen 0% und 100% stufenlos einstellen lassen. Zu Beginn ist immer nur das erste Video auf eine Lautstärke von 50% eingestellt, während alle übrigen Videos stumm sind. Haben Sie den anderen Videos eine höhere Lautstärke zugewiesen und wählen nur noch 1 oder 2 VideoPanels als sichtbar, werden die Lautstärken der nicht sichtbaren VideoPanels stumm geschaltet. Blenden Sie dann diese VideoPanels wieder ein, werden ihre ursprünglichen Lautstärken wieder hergestellt.

10. Phaseninformationen

Sind in einer ViLM Projektdatei Phasen gespeichert, so werden diese nach dem Laden der Videos im Phasenmenü angezeigt. Zudem sind dann die Buttons "Prev" und "Next" entsperrt, mit denen sich die Position der Videos auf die Anfänge der Phasen setzen lässt. Es gibt mehrere Möglichkeiten, die die Einblendung von Phaseninformationen triggern. Wählen Sie im Phasenmenü eine Phase aus oder klicken Sie auf den "Prev"- oder "Next"-Button und gelangen so zu einer Phase, so werden ihre Beschreibung und ein eventueller Kommentar in den dafür vorgesehenen Panels auf der rechten Seite eingeblendet. Um einen Überblick über die Länge einer Phase und ihrer Position innerhalb der gesamten Spielzeit zu bekommen, ist der Positionsslider für jede Phase abwechselnd mit einer Farbe unterlegt, die den zeitlichen Verlauf der Phase symbolisiert. Gehen Sie mit der Maus auf diese Markierung, wird zudem ein Tooltip eingeblendet, der den Namen der Phase, sowie ihre Start- und Endzeit einblendet. Unabhängig vom Playback der Videos, lassen sich alle Informationen zu jeder vorhandenen Phase auch anzeigen, indem man im Hauptmenü unter "Extras" > "Phasen" eine Phase auswählt. Dann wird ein Popup mit den relevanten Informationen einer Phase eingeblendet.

11. Videoquellen

Für jede Videoquelle, die in einer ViLM Projektdatei gefunden wird, wird in der linken Fußleiste des Players ein VideoIcon erzeugt. Wenn Sie mit der Maus auf ein solches Icon gehen, wird der Name des zugehörigen Videos als Tooltip eingeblendet. Klicken Sie nun auf ein solches Icon und ziehen Sie es bis zu einem VideoPanel (Drag-Drop-Operation), so wird in dem betreffenden VideoPanel nun das Video mit dieser Quelle abgespielt, und zwar ab der Position, in der die Videos zuletzt waren.

12. Materialien

Innerhalb eines ViLM-Projektordners gibt es neben dem Unterordner "Videos" gegebenenfalls noch den Unterordner "Materialien". Darin befinden sich zusätzliche Materialien, die mit dem Projekt assoziiert sind. Ist das der Fall, wird beim Öffnen der ViLM Projektdatei für jedes Material ein Icon im rechten Fußbereich des Players erzeugt, das den entsprechenden Dateityp symbolisiert. Doppelklicken Sie auf ein solches Icon, wird das entsprechende Material aus dem Projektordner geöffnet. Dies geschieht auf zwei Weisen:

- In dem Fall, dass es sich um ein lokales Projekt handelt, wo also Videodateien und Materialdateien im ViLM-Projektordner vorliegen, werden diese Dateien entsprechenden ihres Dateityps durch die mit ihnen verknüpfte Betriebssystemanwendung geöffnet.
- In dem Fall, dass es sich um entferntes Projekt handelt, wo also dieselbe Ordnerstruktur auf einem HTTP-Server vorliegt, werden die entsprechenden Materialien innerhalb eines Browsers geöffnet.



Tastaturkürzel [Zurück](#)

- ALT-O Öffnen einer lokalen ViLM Projektdatei
- STRG-ALT-O Öffnen einer entfernten ViLM Projektdatei über HTTP
- ALT-F4 Schließen des ViLM Players