Ein multimediales, zeitbasiertes Lehr- und Publikationssystem

DIPLOMARBEIT

im Studiengang Technische Informatik des Fachbereichs Informatik

Vorgelegt von Björn Feustel im WS 1999/2000
Betreuer: Prof. Ratsch
Dr. Thomas Schmidt

Berlin, 9. März 2000

Inhaltsverzeichnis

1. Einleitung

Die Bedeutung des Computers im Bereich der Lehre hat in den letzten Jahren enorm zugenommen. Begriffe wie „Computer Based Training“ oder „Multimediales Informations- und Schulungssystem“ werden mittlerweile in der Informationsindustrie beinahe inflationär verwandt. Gründe hierfür gibt es viele; bietet das computerbasierte Lernen doch die Möglichkeit, sich selbstbestimmt und problemorientiert benötigtes Wissen anzueignen. Dabei unterscheidet sich diese Art des Lernens besonders durch seine Multimedialität von den klassischen Lehrmethoden. Texte, Bilder oder Videos können auf assoziative Weise miteinander verknüpft werden und der dabei erreichte Grad der Wissenskonzentration ist beinahe unbegrenzt.

Eine weitere Triebkraft des computergestützten Lernens bildet die Verbindung zum Internet: Ein unüberschaubares, sich ständig aktualisierendes Wissenspotential bietet sich dem vernetzten Computernutzer frei an. Lernen mit Hilfe des Netzcomputers ist heute unter dem Begriff „Telelearning“ in aller Munde.

Die vorliegende Arbeit setzt sich mit den Vor- und Nachteilen der bereits vorhandenen computergestützten Lernumgebungen auseinander und macht dabei deutlich, daß keine der derzeitigen Anwendungen das gegebene Potential in bestmöglicher Weise ausnutzt. Dies motiviert die Konzeption und Umsetzung einer neuen Lernplattform, welche die wichtigsten Eigenschaften des Netzcomputers vereint und so eine sehr leistungsstarke und flexible Möglichkeit bildet, Informationen anschaulich und leicht verständlich zu präsentieren.

Die vorliegende Arbeit gliedert sich in sechs Teile.
Kapitel 2 führt die Unterschiede der wichtigsten Medientypen auf und gibt einen Einblick in die derzeit zum Einsatz kommenden Lehr- und Präsentationssysteme.
Kapitel 3 widmet sich dem Entwurf einer neuen Lernumgebung auf Basis des World Wide Web. Es werden die dabei zu berücksichtigenden Probleme dargelegt und die Konzeption der Plattform veranschaulicht.
Kapitel 4 dokumentiert die Implementierung der Lernumgebung.
Kapitel 5 stellt anhand einer Beispielpräsentation die Eigenschaften der neuentwickelten Plattform dar.
Zusammenfassung und Ausblick befinden sich schließlich am Schluß der Arbeit.

2. Lernumgebungen

In diesem Kapitel möchte ich einen Überblick über die derzeitig in der Lehre eingesetzten Medien und deren spezifische Eigenschaften geben. Dabei unterscheide ich zwischen klassischen und neuen Medien, da sich diese beiden Gruppen recht stark in ihrer Art unterscheiden.

2.1. Klassische Medien

Die Gruppe der „klassischen Medien“ könnte ebenso mit dem Begriff „nicht computergestützte Medien“ umschrieben werden. Hierzu gehören all jene Informationsträger, die schon seit geraumer Zeit zur Vermittlung von Wissen bzw. Lehrinhalten verwandt werden. Beispiele hierfür wären alle Arten von Printmedien (Skripten, Bücher, Zeitschriften), Filme, Overhead-Folien und Audiobänder. All diesen Medien liegt die Eigenschaft zugrunde, daß ihre Erstellung, Aufbewahrung, Nutzung und Wiederverwendung oft nur mit beachtlichem Aufwand betrieben werden kann.

Läßt man einmal den Film außer Betracht, können sämtliche erwähnten Medien als *unimedial* angesehen werden. Eine Multimedialität läßt sich mit ihnen nur durch ihre Kombination erreichen. So bleibt es z. B. dem Dozenten überlassen, ob und in welchem Umfang er die ihm zur Verfügung stehenden Medien nutzt, um seiner Hörerschaft die gewünschten Lehrinhalte näherzubringen.
Neben der naturgemäß nicht vorhandenen Multimedialität und der daraus resultierenden Begrenztheit der Informationskanäle, sieht sich der Lernende auch mit einem weiteren Problem konfrontiert: Er ist zumeist durch seine Lerngruppe gebunden und kann dadurch als einzelne Person nur begrenzt Einfluß auf die Geschwindigkeit und die Abfolge des Stoffes nehmen. So muß er sich der Gruppe anpassen, auch wenn er

selbst eine andere Form oder Geschwindigkeit der Informationvermittlung bevorzugen würde.

2.2. Neue Medien

Den klassischen Medien gegenüber stehen die „neuen Medien“. Hierbei handelt es sich um das Gebiet der computergestützten Informationsträger, dessen Verbreitung in den letzten Jahren stark zugenommen hat. Dabei schafft der Computer als Grundlage dieser Medien völlig neue Ansätze und Möglichkeiten, die sich natürlich auch auf das Gebiet der Lehre auswirken. Wird es mit seiner Hilfe doch erstmals möglich, Informationen beliebig zu Verknüpfen und so die Lehrinhalte assoziativ verkettet zu präsentieren. Betrachtet man in diesem Zusammenhang auch die Fähigkeit des Computers, mit den unterschiedlichsten Arten von Medien umzugehen, wird die Bedeutung klar, die eine Verkettung der Informationen darstellt. So kann beispielsweise aus einem computerbasierten Lexikon heraus auf andere Medientypen verwiesen werden, die die zugrunde liegenden Information veranschaulichen oder klarer machen. Bilder, Videos oder Klangbeispiele zu einem Eintrag direkt abzurufen, war mit den „althergebrachten“ Lexika nicht einmal ansatzweise möglich.

Neben dieser verknüpften Darstellung der Informationen bringt der Einsatz von Computern auch die Möglichkeit der programmgesteuerten Interaktion mit sich. Der Lernende kann hierdurch den Ablauf der Lehrpräsentation gezielt beeinflussen, ja er kann u. U. sogar mit dem Rechner kommunizieren. Das einfache Beispiel eines Vokabellernprogrammes verdeutlicht diesen Umstand: Der Lerner teilt dem Programm seinen Wissensstand mit und erhält eine direkte Aus- bzw. Bewertung dessen. Er kann selbst entscheiden, welche Thematiken er überspringen oder wiederholen möchte und kann dabei den Informationsfluß an sein eigenes Lerntempo anpassen.

Neben dem Computer hat auch eine weitere Technologie in den letzten Jahren die Welt der Lehre und Information maßgeblich beeinflußt: die Datennetze. Dabei stellt das Internet das bekannteste und verbreitetste Datennetz dar. In den vergangenen Jahren hat es sich zu einem riesigen Netzwerk entwickelt, an das unzählige Computer angeschlossen sind. So gibt es heute bei uns wohl kaum noch eine Universität oder Hochschule, die nicht vernetzt ist. Doch auch ein großer Teil der Haushalte hat bereits Zugang zum Netz der Netze. Wenn diese Entwicklung weiter voranschreitet wie bisher, wird in wenigen Jahren nahezu jeder Bürger das Internet als Kommunikations- und Informationsmedium nutzen. Begünstigt wird diese Entwicklung durch die relativ geringen Anforderungen, die an den potentiellen „Netzbürger“ gestellt werden. Ein normaler PC mit Modem und ein für die Einwahl erforderlicher Provider reichen aus, um Anschluß an das Internet zu erhalten.
Die Ortsunabhängigkeit beim Zugriff auf die Informationen des Netzes ist natürlich eine wichtige Eigenschaft, wenn es um die Verbreitung von Lehrmaterialien geht. Vergleicht man beispielsweise eine klassische Lehrveranstaltung mit dem computer- und netzgestützten Vermitteln von Lehrinhalten (computer based training), wird der

sich daraus ergebende Vorteil klar: Der Lerner hat Zugang zu einem weitaus größeren Fundus an Informationen, der außerdem noch beliebig Verknüpft sein kann.

All diese - gegenüber den früheren Möglichkeiten der Informationsdarbietung - wesentlich erweiterten Fähigkeiten machen den Netzcomputer zum idealen Werkzeug für den Einsatz im Bereich der Lehre. Würde man alle genannten Vorteile bei der Entwicklung einer Lern- und Präsentationsplattform berücksichtigen, könnte ein Werkzeug implementiert werden, das dem Nutzer in bisher völlig undenkbarer Art und Weise einen einfach zu bedienenden, multimedialen und von ihm steuerbaren Zugang zu den unterschiedlichsten Lehrinhalten gewährt.

Derzeit gibt es natürlich schon verschiedene Applikationen, die sich den Computer und das Netz zunutze machen, um dem Anwender Informationen und Zusammenhänge zu präsentieren. Nachfolgend möchte ich daher einen Überblick über die wichtigsten geben.

2.2.1. Lernprogramme

Lernprogramme waren eine der ersten Plattformen computergestützten Lernens. Sie wurden und werden zumeist für einen speziellen Themenschwerpunkt geschrieben und sind daher nicht universell einsetzbar. Die Erstellung solcher Software setzt neben dem für die inhaltliche Gestaltung nötigen Fachwissen auch ein recht umfangreiches Wissen über Softwareerstellung bzw. Programmierung voraus. Da es auch keine Standards oder technische Richtlinien für die Implementierung derartiger Programme gibt, wird der Lernende bei Benutzung unterschiedlicher Systeme immer wieder vor neue Umgebungen und Benutzerführungen gestellt. Des weiteren verlangt die Handhabung der Lernprogramme oft auch die unterschiedlichsten technischen Voraussetzungen für ein problemloses Arbeiten mit ihnen.
All die genannten Nachteile und Einschränkungen machen das System der spezialisierten Lernsoftware unflexibel und deren Entwicklung kostenintensiv und technisch anspruchsvoll. Der eigentliche Autor der Lehrinhalte ist aufgrund der speziellen Umsetzung der einzelnen Programme allein nicht in der Lage, seinen Lehrstoff in der Art und Weise zu präsentieren, wie er es möchte.

2.2.2. Autorensysteme

Autorensysteme bilden die Grundlage für die unterschiedlichsten Arten von Präsentationen und Lernumgebungen. Mit ihnen kann der Autor relativ frei und mit nur geringem technischen Wissen seine Lehrinhalte umsetzen und erstellen. Dazu bedient er sich eines sogenannten *Authoringtools*. Mit diesem kann er, in Abhängigkeit von den angebotenen Möglichkeiten, die Inhalte zu einer mehr oder weniger interaktiven „Lehreinheit“ zusammensetzen. Dabei werden zumeist Abläufe und Aktionen an einem Zeitstrahl ausgerichtet, der einen linearen Verlauf hat. Dem Nutzer bleibt dadurch aber nur eine beschränkte Möglichkeit der Interaktion, durch die er nur zwi-

schen fest vorgegebenen „Zweigen“ der Präsentation wählen kann.
Als Grundlage für den zeitgesteuerten Ablauf und die Interaktionsfähigkeit dient bei allen Systemen eine eigene Programmier- bzw. Skriptsprache. Nur mit ihrer Hilfe wird es überhaupt möglich, entsprechende Abhängigkeits- und Ablaufbeschreibungen zu formulieren.

Die fertige Präsentation, die mit einem Autorensystem erstellt wurde, kann nur mit einem eigens dafür vorgesehenen Programm abgespielt bzw. abgearbeitet werden. Hier ist auch schon ein Nachteil erkennbar: Es werden je nach verwendetem Autorensystem und zugrunde liegender Plattform unterschiedliche „Player“ in Form von eigenständigen Programmen oder aber Plugins für die gängigen Webbrowser benötigt. Die Hersteller solcher Systeme versuchen dieses Defizit dadurch auszugleichen, daß sie die zum Abspielen erforderliche Software „marktschreierisch“ bewerben und kostenlos für jeden interessierten Benutzer anbieten, um ihre Produkte am Markt zu etablieren.
Ein typisches Beipiel für eine per Autorensystem erstellte Lernanwendung ist die „interaktive CD-Rom“, wie sie für Einführungen und Präsentationen von Produkten der Computer- oder Unterhaltungsindustrie verwandt wird. Auf dieser CD-Rom befindet sich dann neben der eigentlichen Präsentation auch gleich der entsprechende zum Abspielen erforderliche Player. Mittels solcher multimedialer Einführungen wird z. B. dem Käufer von Softwareprodukten die Benutzung der jeweiligen Produkte vorgestellt, um so den Einstieg in die Bedienung zu erleichtern und eventuell das Nachschlagen in einem Handbuch überflüssig zu machen. Auch wenn der Name „interaktive CD-Rom“ anderes vermuten läßt, trifft auf diese Art der Informationsdarbietung der Nachteil der oben erwähnten begrenzten Interaktionsmöglichkeit zu. Der Nutzer kann nur in sehr engen Grenzen bestimmen, welchen Ablauf die Präsentation hat, da sie sich nur aus linearen Verläufen zusammensetzt.

Gerade am Beispiel der CD-Rom lässt sich ein weiterer Nachteil der heute üblichen Autorensysteme erkennen: Die mit ihnen erzeugten Lerneinheiten müssen jedem Nutzer als eigenständiges Exemplar zur Verfügung gestellt werden. Ein gemeinschaftliches Nutzen ist daher ohne zusätzlichen Verteilungsaufwand nicht möglich.

Im Laufe der Zeit haben sich einige Systeme als Standardsysteme etablieren können. Die wichtigsten Autorensysteme sind dabei die folgenden:

- Macromedia Director 8 Shockwave Studio[1]
- Macromedia Authorware 5.1[2]
- ToolBook II Instructor 7.1[3]

Für weitere Informationen zu den einzelnen Produkten sei auf die entsprechende Website verwiesen.

2.2.3. Netzmedien

Da die Netztechnologie ihrem Wesen nach im allgemeinen nicht vorgibt, welche Arten von Diensten auf ihr implementiert werden, ergibt sich ein nach oben offenes Anwendungsgebiet. Dabei muß der Entwickler und der Nutzer dieser Dienste natürlich auf gewisse Beschränkungen, wie z. B. die Bandbreite der Netzverbindung, Rücksicht nehmen. Es ist aber bei der raschen Entwicklung der Netztechnologie nur eine Frage der Zeit, bis die meisten Probleme irrelevant werden.

Kommunikationsplattformen

Die unterschiedlichsten Kommunikationsumgebungen existieren derzeit im Internet, welche mehr oder weniger auch als Lehrplattformen genutzt werden können. Einige der verbreitetsten möchte ich hier kurz beschreiben:

Chat Der Chat bietet die Möglichkeit einer direkten, unmittelbaren Kommunikation. Zwei oder mehrere Gesprächspartner können sich dabei in Echtzeit miteinander durch Textmitteilungen unterhalten. Bekanntester Vertreter ist wohl der Internet Relay Chat (IRC), welcher ein Netzwerk von miteinander verbundenen Chatservern darstellt. Der Nutzer kann mit Hilfe eines speziellen IRC-Clients mit anderen Teilnehmern kommunizieren.
Da der Chat aber eine unimediale Plattform ist, mit der nur Textmitteilungen ausgetauscht werden können, eignet er sich im Rahmen der Lehre eigentlich nur als Ergänzung zu anderen Technologien.

Discussionboards Discussionboards dienen dem Austausch von Informationen zwischen den Teilnehmern, die dabei nicht zeitgleich an der Kommunikation teilnehmen müssen, was z. B. ein Chat voraussetzt. Jeder Nutzer kann normalerweise selbst neue Einträge in das Board stellen oder die Einträge anderer beantworten. Das umfassenste Diskussionsboard ist das Usenet. Spezielle Clients, „Newsreader“ genannt, eröffnen hierbei dem Nutzer eine schier überwältigende Flut von themenbezogenen Diskussionsgruppen.
Auch Discussionboards stellen höchstens eine Ergänzung zu anderen Lehrplattformen dar. Ähnlich wie der Chat bieten sie nur die Möglichkeit des Austauschs von Textmitteilungen.

Konferenzsysteme Kommunikationsumgebungen, die in Echtzeit eine direkte Verbindung zwischen zwei oder mehreren Teilnehmern bereitstellen, nennt man Konferenzsysteme. Ein typischer Vertreter ist das Videokonferenzsystem. Die Gesprächspartner sitzen dabei zumeist an vollkommen unterschiedlichen Orten und können mit Hilfe einer Kamera und einem Mikrofon miteinander kommunizieren. Unterstützt werden solche Syteme oft auch durch sogenannte „whiteboards“, die ein gemeinsames Arbeiten an ein und demselben Dokument ermöglichen.

Konferenzsysteme werden heutzutage bereits als Lehrmedien verwandt, obwohl dabei eventuell nur eine einseitige Bildübertragung vom Dozenten zum Studenten erfolgt. Da sich gegenüber der klassischen Lehre nur der Vorteil der Ortsungebundenheit ergibt, bringen diese Systeme allerdings keine wesentlich neuen Möglichkeiten mit sich.

Proprietäre Plattformen Neben den bereits genannten Plattformen gibt es auch andere proprietäre Umgebungen, die unterschiedliche Kommunikationssysteme unter einer gemeinsamen Oberfläche zusammenfassen. Ein Beispiel wäre hier z. B. die ProVirtus-Plattform [4]. Sie vereint Discussionboard, Chat, Messagesystem und Dokumentenverwaltung mit Hilfe einer einheitlichen Benutzerführung und bietet so zwar gute Voraussetzung für ein gemeinschaftliches Arbeiten - als Lehr- oder Präsentationsplattform eignen sie sich allerdings weniger.

2.2.4. World Wide Web

Eigenschaften des WWW

Das World Wide Web als Dienst des Internet ist ein interaktives Informationssystem auf Hypertext-Basis, über welches digitale Daten weltweit veröffentlicht werden können. Das vom CERN ins Leben gerufene und vom W3C weiter entwickelte WWW ist mittlerweile zum meistgenutzten Teil des Internets herangewachsen.

Die WWW-Inhalte werden in der HyperText Markup Language (HTML)[5] beschrieben, einer von der Structured Generalized Markup Language (SGML)[6] abgeleiteten logischen Dokumentenbeschreibungssprache. Sie bietet unter anderem die Möglichkeit, Verweise auf beliebige Dokumente zu definieren. Dabei spielt es keine Rolle, wo diese Daten im Netz stehen oder welchen Typs sie sind. So kann beispielsweise aus einem Dokument heraus auf ein Video verwiesen werden, daß auf einem völlig anderem Server liegt. Eine Folge derartiger Verkettungen ist eine nichtlineare Informationsstruktur, denn ein WWW-Dokument hat keinen unmittelbaren Vorgänger oder Nachfolger. So kann der Nutzer frei entscheiden, auf welchen Wegen er sich durch die vernetzten Daten bewegt. Durch diese Verknüpfung lassen sich hervorragend Wissensbasen bilden, deren Daten nicht mehr zentral verwaltet werden müssen und vollkommen unterschiedlicher Art sein können.

Um HTML-Dateien betrachten zu können, benötigt der Endnutzer ein Programm, welches die von ihm gewünschten WWW-Seiten abruft und aufbereitet. Die weitverbreitetsten Programme dieser Art, Browser genannt, sind der Navigator der Firma Netscape und der Internet Explorer der Firma Microsoft. Da allerdings beide Unternehmen aus marketingtechnischen Gründen anfingen, unterschiedliche und nicht vereinheitlichte Elemente in ihre Browser zu integrieren, wird das Erstellen von universell verwendbaren und anspruchsvollen WWW-Inhalten zur Farce. Der Webdesigner muß sich mit den verschiedensten Eigenheiten der Programme beschäftigen,

um später festzustellen, daß in einer neueren Version der Browser doch nichts so funktioniert, wie er es sich vorgestellt hatte.

Die Datenübertragung zwischen Server und Client beruht auf dem Hypertext Transfer Protokoll (HTTP). Das in RFC 1945[7] in seiner Version 1.0 beschriebene Protokoll wurde als Grundlage für eine leichte und schnelle Datenübertragung für hypermediale Informationssysteme geschaffen. Eine Eigenschaft, die aus diesem Protokoll hervorgeht, ist seine Zustandslosigkeit. Bei der Datenübertragung zwischen Server und Client gibt es keine festen Verbindungen, so daß ein kontinuierlicher, definierter Datenstrom nicht möglich ist. Diese Zustandslosigkeit schränkt aber den Einsatz des WWW als Lehr- und Präsentationsplattform ein, denn daraus ergeben sich - auch im Hinblick auf die gewünschte Interaktionsfähigkeit - folgende Probleme:

- Dynamische Webinhalte können nur über einige Erweiterungen (DHTML[8], Server-Side-Includes[9]) und dabei auch nur in sehr begrenztem Umfang realisiert werden.
- Da die Daten im WWW mittels eines „Pull“-Verfahrens vom Server zum Client gelangen, ist eine Kommunikation vom Client zum Server nicht möglich. So gibt es beispielsweise keine Möglichkeit, eine vom Nutzer ausgelöste Interaktion während einer laufenden Präsentation zum Server zu schicken, um ihn aufzufordern, dynamische und vom Kontext abhängige Daten bereitzustellen.
- Mit HTML ist es nur sehr eingeschränkt möglich (z. B. duch Cookies[10]), eine nutzerbezogene Kommunikation herzustellen.
- Gestreamte Medien, die auf persistente Verbindungen angewiesen sind, können nur mit zusätzlicher Software abgespielt werden.

Dynamische Inhalte im WWW

Wie bereits eben erwähnt, kann das WWW an sich nur statische und nicht zeitgesteuerte Webseiten darstellen. Um dennoch dynamische Inhalte präsentieren zu können, wurden drei unterschiedliche Ansätze entwickelt, die in unterschiedlichem Maße derzeit eingesetzt werden. Als erstes wäre dabei Javascript[11] zu nennen. Diese, von der Firma Netscape entwickelte objektorientierte Skriptsprache, wird clientseitig vom Browser ausgeführt. Mit ihr kann z. B. auf einzelne Elemente der HTML-Seite zugegriffen werden. Da diese Sprache aber nicht von jedem Browser unterstützt wird, nur einen beschränkten Befehlsumfang hat und auch nicht sehr sicher ist, sollte sie nur in begrenztem Maße eingesetzt werden.

Ein weiterer und zugleich interessanter Versuch dynamische Webinhalte zu ermöglichen, ist die vom W3C als Entwurf veröffentlichte Synchronized Multimedia Integration Language[12] - kurz SMIL. Dabei handelt es sich um einer HTML-ähnlichen Beschreibungssprache, die aber als wichtigste Neuerung ihre Objekte zeitabhängig darstellen kann. Sie ist wie HTML auch multimediafähig und zustandslos, da sie

auch auf das HTTP-Protokoll aufbaut. Da sie eine Präsentation aber nur beschreibt und diese selbst nicht abspielen kann, werden dafür Player-Programme benötigt, von denen zur Zeit nur wenige existieren. Da diese Player nur bestimmte und außerdem auch unterschiedliche Medientypen unterstützen, kann SMIL z. Zt. nicht wirklich sinnvoll eingesetzt werden.

Der dritte Ansatz sind die Applets. Dabei handelt es sich um Javaprogramme, auf die durch spezielle HTML-Tags verwiesen wird und die innerhalb des Browsers ausgeführt werden. Da es sich bei den Applets um vollwertige Javaprogramme handelt, die - mit einigen Einschränkungen - auf der Clientseite ablaufen, bietet diese Lösung die größte Flexibilität, denn mit ihrer Hilfe können persistente Netzverbindungen erzeugt, umfangreiche grafische Ausgaben erstellt und viele andere komplexe Technologien innerhalb des WWW angewendet werden. Applets kommen derzeit in sehr großem Maße zum Einsatz, da sie die Grundlage für die unterschiedlichsten Webanwendungen bieten.
Ich werde das Thema Java in 3.3.2 auf Seite 30 noch genauer erläutern.

3. Entwurf einer WWW-Lernumgebung

3.1. Zielstellung

Wie in Kapitel 2 erkennbar ist, haben alle der dort dargelegten Ansätze Unzulänglichkeiten inne, die ihre Verwendung als Lernplattformen einschränken oder auf spezielle Anwendungsgebiete begrenzen. Keines der vorgestellten Systeme vereint in zufriedenstellender Weise die durch die Einführung des Computers und des Netzes erlangten Möglichkeiten.

Im Rahmen dieser Diplomarbeit soll deshalb eine Lehr- und Präsentationsplattform entwickelt werden, die das gegebene Potential auf eine neue, innovative Art ausnutzt. In den nachfolgenden Absätzen werde ich auf die einzelnen Anforderungen, die an die zu entwickelnde Plattform gestellt werden, genauer eingehen.

Um die Lernumgebung einem großen Personenkreis zugänglich zu machen, darf sie keine besonderen Plattformen oder Umgebungen voraussetzen. So würde es z. B. wenig Sinn machen, das System als nur auf einen Rechnertyp zugeschnittene Anwendung zu gestalten, da sich dann wieder die oft bei spezialisierter Lernsoftware bestehende Plattformabhängigkeit ergibt. Desweiteren muß das System so unabhängig wie möglich von weiteren Software- und auch Hardwareprodukten sein. Es kann einfach nicht verlangt werden, daß sich der Lerner, nur um die Umgebung nutzen zu können, eine große Anzahl von Programmen oder Plugins installieren muß. Je weniger Zusatzsoftware erforderlich ist, um so weniger Probleme werden sich auch bei der Installation oder dem Betrieb des Systems ergeben. Daher muß die zu entwickelnde Plattform auf bestehende und weit verbreitete Standards aufbauen.

Ein weiteres Ziel bei der Entwicklung der Plattform ist außerdem die darstellerisch aber auch inhaltlich freie Gestaltungsmöglichkeit für den Autor der Lehrinhalte. Die zu implementierende Umgebung muß flexibel und offen gestaltet werden, so daß der Autor bei der Umsetzung der Inhalte nur mit so wenig wie möglich programmtechnischen Zwängen oder Einschränkungen konfrontiert wird, um eine bestmögliche Gestaltung des Lehrstoffs zu gewährleisten. Diese Flexibilität ist auch die Voraussetzung dafür, daß mit der Plattform ein breites Spektrum an Lehrthemen präsentiert werden kann.

Eng an diese Problematik gekoppelt ist die Fähigkeit, mit einer Vielfalt von unterschiedlichen Medien umgehen zu können. Nur wenn die Umgebung in der Lage ist, Texte, Bilder, Videos und andere Arten von Medien zu verarbeiten, können die unterschiedlichen Thematiken in angemessener Weise veranschaulicht werden. Um sich aber bei der Implementierung nicht auf ein beschränktes Repertoire an Medientypen festlegen zu müssen, sollte die Plattform dahingehend leicht erweiterbar sein. So soll sich beispielsweise ohne Änderungen an der Plattform ein neuer Videocodec umsetzen lassen. Hierfür ist es erforderlich, eine definierte Schnittstelle für die Anbindung der einzelnen Medienkomponenten zu schaffen.

Ein weiterer unverzichtbarer Punkt bei der Entwicklung einer neuen Lehr- und Präsentationsplattform ist die Interaktivität. Mit ihrer Hilfe wird es möglich, den Lernenden direkt auf die Inhalte Einfluß nehmen zu lassen. Er soll, ähnlich wie es auch HTML ermöglicht, selbst den Ablauf der Präsentation steuern können. Diese Beeinflussung ist aber - im Gegensatz zu HTML - zeitabhängig. Bei der zu entwickelnden Plattform ist es daher entscheidend, *wann* der Benutzer eine Interaktion auslöst. Als Reaktion auf solch ein Ereignis werden ihm dann dynamisch neue Inhalte präsentiert.

Um dieses zeitabhängige Interaktionsverhalten zu realisieren, muß die gesamte Plattform zeitsynchron arbeiten. Die einzelnen Komponenten einer Präsentation müssen zu vorher definierten Zeiten die gewünschten Aktionen (siehe 3.2.5 auf Seite 24) ausführen. Dabei muß jedes einzelne Element sich zeitsynchron zum gesamten Präsentationsablauf verhalten. Nur so wird es z. B. möglich, auf einen „Klick“ in eine Videokomponente mit dem genau dieser Szene zugeordneten Inhalt zu reagieren.

Als letzte wichtige Eigenschaft soll die Plattform ihre Daten nach einem Komponentenmodell verwalten. Der dadurch erreichte Effekt der Wiederverwendbarkeit der einzelnen Präsentationsobjekte vereinfacht das Erstellen neuer Lehrinhalte und ermöglicht es, die unterschiedlichen Objekte themenbezogen zu verwalten. Setzt man das Komponentenmodell konsequent um, ergibt sich noch eine entscheidende Funktionalität, die bisherige Lernumgebungen so nicht bieten: Es können ganze Präsentationen als Objekte betrachtet und wiederverwendet werden und so beispielsweise in ihrer Gesamtheit in andere Präsentationen eingebettet werden.

Betrachtet man all die aufgeführten Anwendungen, fällt auf, das kein bisheriges Lehrsystem diese Eigenschaften zufriedenstellend vereint. Aus dieser Überlegung entstand die Idee zu der der Arbeit zugrunde liegenden Plattform. Nachfolgend habe ich noch einmal alle Anforderungen in Kurzform zusammengefaßt.

- Der Zugriff auf die Lehrinhalte soll ohne spezielle Soft- oder Hardware möglich sein.
- Die Plattform muß universell sein, d. h. sie darf in ihrer Struktur und Umsetzung das Spektrum der mit ihr präsentierbaren Lehrinhalte nicht auf bestimmte Themengebiete einschränken.
- Die Plattform soll Multimedialität bieten und muß daher mit unterschiedlichen Medientypen wie Text, Bild, Ton und Video umgehen können.
- Eine Erweiterbarkeit des Systems um neue Medientypen soll ohne großen Aufwand möglich sein.
- Die Verwaltung der einzelnen Präsentationsobjekte soll nach einem Komponentenmodell erfolgen.
- Die Lehrumgebung soll dem Nutzer die Möglichkeit der Interaktion bieten.
- Die Präsentationen sollen zeitsynchron ablaufen. Im Zusammenspiel mit der Interaktionsfähigkeit ergibt sich dadurch ein durch den Nutzer beeinflußbares x-y-t-Koordinatensystem.

3.2. Problemerörterung

In diesem Abschnitt möchte ich die vor dem Entwurf der Plattform zu beachtenden Probleme und deren Lösung darlegen.

3.2.1. Plattformunabhängigkeit

Die wichtigste Voraussetzung für die angestrebte Plattformunabhängigkeit ist mit der Nutzung des WWW als zugrunde liegendes Medium erfüllt. Wie bereits dargelegt,

bietet die weite Verbreitung des und der einfache Zugang zum WWW bestmögliche Bedingungen, um dem Autor der Lerninhalte einen großen Rezipientenkreis zu eröffnen. Besonders deutlich wird dieser Umstand, wenn man die Hauptzielgruppe des Lernstoffes betrachtet. Hier wird vor allem der Einsatz im universitären Bereich angestrebt, wobei davon ausgegangen werden kann, daß mittlerweile jeder Student uneingeschränkten Zugang zum Internet und damit zum WWW hat. Da aber dessen bisherige Möglichkeiten nicht ausreichen, um die bereits genannte Zielvorstellung zu erfüllen, muß außerdem auf eine der unter 2.2.4 auf Seite 11 erwähnten Erweiterungen zurückgegriffen werden, um beispielsweise dynamische Inhalte präsentieren zu können.

Wenn man davon ausgeht, daß die Plattform auch wegen der sich aus ihm ergebenden Ortsunabhängigkeit des Zugriffs auf die angebotenen Inhalte dezentral genutzt wird, darf die Frage nach der zur Verfügung stehenden Bandbreite nicht ungeklärt bleiben. Möchte ein Student z. B. von seinem heimischen Computer aus die Plattform nutzen, so wird die mögliche Übertragungsrate der Daten stark durch die heute zum Einsatz kommenden Modems begrenzt. Normalerweise kann davon ausgegangen werden, daß der Großteil der Nutzer einen nomalen Modemzugang mit einer heute üblichen Übertragungrate von 33,6 $\frac{Kb}{s}$ oder einen ISDN-Anschluß mit 64 $\frac{Kb}{s}$ sein eigen nennt.

Bei der Umsetzung der Plattform muß diesem Umstand natürlich Rechnung getragen werden, denn sollen beispielsweise auch Videos und Bilder integriert werden, kann die Übertragung derer leicht zu einer Farce für den Nutzer werden. Aus dieser Problematik ergibt sich aber eine ganz grundsätzliche Frage: Sollen die einzelnen Elemente einer Präsentation vor dem Start oder - soweit möglich - während des Ablaufs derselben zum Client übertragen werden? Wählt man erstere Variante, so kann es je nach Umfang der Daten zu einer erheblichen Wartezeit kommen. Die zweite Variante birgt aber ein anderes Problem, nämlich die u. U. auftretenden Verzögerungen oder Unterbrechungen während des Ablaufs der Präsentation, die durch Engpässe bei der Datenübertragung verursacht werden können. Letzteres würde aber nicht in den Kontext der angestrebten zeitlichen Synchronisation passen, so daß die Wahl auf die erste der beiden Möglichkeiten fällt. Zukünftig wäre es auch denkbar, dem Nutzer als erstes die Wahl zu lassen, wie die Datenübertragung erfolgen soll: Vor oder während des Ablaufs der Präsentation. Er könnte so die für seine Netzanbindung optimale Variante wählen.

3.2.2. Wiederverwendbarkeit der Komponenten

Um eine Wiederverwendbarkeit der Elemente einer Präsentation oder derer im Ganzen zu ermöglichen, stellt sich die Frage, wie die Elemente dafür überhaupt strukturiert werden sollen bzw. wie eine Präsentation aufgebaut werden muß.

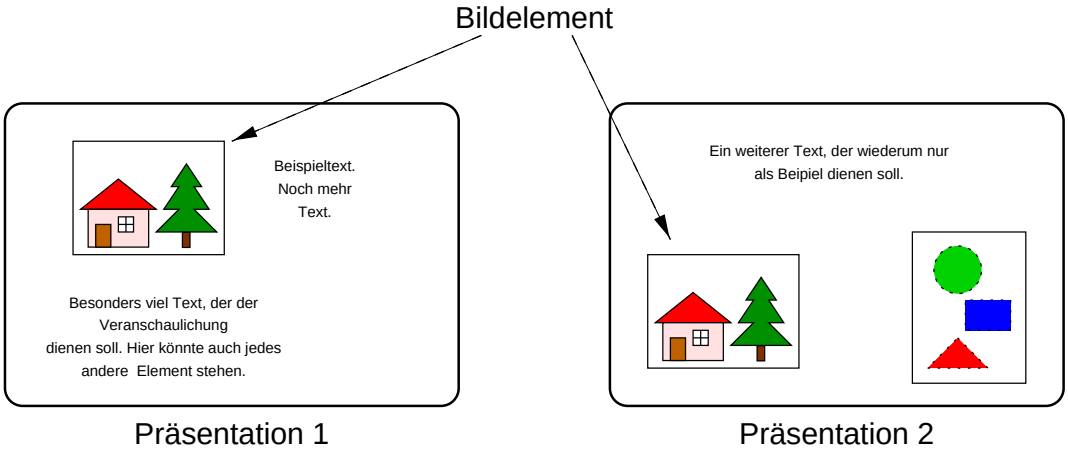


Abbildung 3.1.: Einfache Wiederverwendbarkeit von Komponenten am Beispiel eines Bildelements

Einfache Wiederverwendbarkeit der Komponenten

Der einfachste Fall ist hierbei die mehrfache Nutzung der unterschiedlichen Komponenten mit ihren ursprünglich festgelegten Parametern und Eigenschaften. Betrachtet man die unterschiedlichen Medientypen als Elemente, so könnte beispielsweise eine Grafik unverändert in mehreren Präsentationen verwendet werden (siehe Abb. 3.1). Dieser Ansatz birgt allerdings einige gravierende Unzulänglichkeiten:
Bei der Mehrfachnutzung ist man auf sämtliche bei der Erstellung des Elementes angegebene Parameter festgelegt. Wenn z. B. die Farbe eines Textelementes auf rot gesetzt wurde, erscheint der Text - egal in welchem Kontext - immer in dieser Farbe, was zumeist nicht gewünscht ist. Zieht man dann noch den Austausch der Elemente zwischen zwei vollkommen unterschiedlichen Instanzen der Plattform in Betracht, wird erkenntlich, daß bestimmte Festlegungen nicht haltbar sind. So könnte ein Textelement in einer auf dem einen Server vorhandenen Schriftart angegeben sein, die auf dem zweiten Server überhaupt nicht installiert ist. Das hätte letztendlich den Effekt, daß beinahe kein Element wiederverwendbar und somit das Ziel der Mehrfachnutzung verfehlt wäre.

Abstrahierte Wiederverwendbarkeit der Komponenten

Um nun eine wesentlich flexiblere Wiederverwendbarkeit zu erreichen, muß es möglich sein, die Elemente bei ihrer Verwendung parametrisieren zu können. Wenn man also das Element an sich abstrahiert und ihm die Fähigkeit der nachträglichen Parametrisierung gibt, erhält man einen wesentlich universelleren Ansatz. So könnte ein derart verallgemeinertes Textelement sämtliche nur denkbare Eigenschaften annehmen. In Abhängigkeit des Kontextes, in den es eingebettet ist, könnte der Autor dem Text unterschiedliche Farben, Schriftarten, Größen oder beliebige andere Parameter

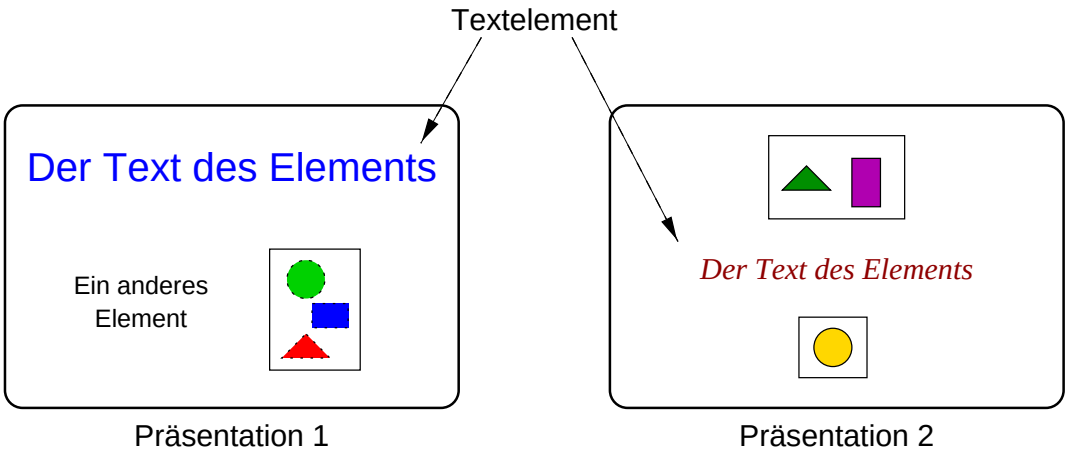


Abbildung 3.2.: abstrahierte Wiederverwendbarkeit von Elementen am Beispiel eines Textelements

zuordnen (siehe Abb. 3.2).

Universelle Wiederverwendbarkeit - Komponentenmodell

Die in 3.2.2 auf der vorherigen Seite dargelegte Form der Wiederverwendbarkeit ermöglicht nun also eine recht flexible Mehrfachnutzung der einzelnen Elemente. Wie ausgereift ist dieser Ansatz aber bei einer Präsentation, die vollständig in eine andere integriert werden soll? Diese Frage möchte ich an einem Beispiel erörtern. Nehmen wir an, es existiert bereits eine Lehreinheit, die sich mit den unterschiedlichen Griffen des Gitarrenspiels beschäftigt. Sie veranschaulicht anhand von Bildern und Text die einzelnen Griffe, wobei sich ihre Darstellung wie in Abb. 3.3 auf der nächsten Seite gestalten könnte. Wie ersichtlich ist, würde sie sich dann aus drei Elementen zusammensetzen:

- E1 - Textelement (Überschrift)
- E2 - Bildelement (Griffdarstellung)
- E3 - Textelement (Erklärung)

Möchte nun zu einem späteren Zeitpunkt ein Autor diese Präsentation um ein kleines Demonstrationsvideo erweitern, muß er aus allen drei Elementen eine nahezu identische Darstellung zusammenbasteln, die lediglich um ein Videoelement erweitert ist. Dieses Vorgehen mag in dem gegebenen Beispiel noch relativ einfach und mit wenig Aufwand verbunden sein; eine umfangreichere Ausgangspräsentation läßt sich aber auf diesem Wege nicht ohne größeren Arbeitsaufwand erweitern.
Ideal wäre in einem solchen Fall die Möglichkeit, ganze Lehrprogramme zu einzelnen

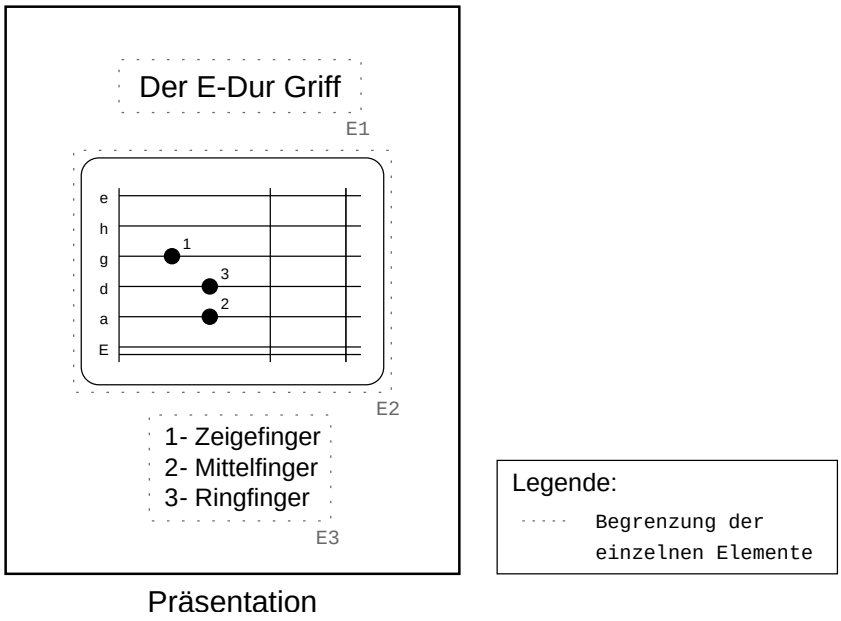


Abbildung 3.3.: Beispiel einer einfachen Präsentation

logischen Paketen[1] zusammenfassen zu können. Das daraus resultierende Komponentenmodel würde die gewünschten Möglichkeiten bieten, die für die Wiederverwendung der Elemente nötig sind.
Wenn man diese Idee aufgreift und umsetzt, wird man erkennen, daß sich daraus eine hierarchisch strukturierte Anordnung der einzelnen Objekte ergeben muß. Ein solcher wurzelartiger logischer Zusammenhang zwischen den Objekten ist in Abb. 3.4 auf der nächsten Seite ersichtlich. Mit Hilfe dieser Logik können nun ohne weiteres ganze Wurzelzweige unterhalb eines neuen Objekts gruppiert und so auf einfache Weise wiederverwendet werden. Auch das oben aufgeführte Beispiel der Gitarrengriffschule läßt sich nun mit wenig Aufwand um ein Videoelement erweitern. Dazu wird die ursprüngliche Präsentation, die sich nach der eingeführten Struktur bereits als Mob auffassen läßt, zusammen mit dem Videoelement als Zweige eines übergeordneten Objekts kombiniert. Die resultierende Strukturdarstellung und das Erscheinungsbild sind in Abb. 3.6 auf Seite 21 zu sehen.

Problem der geometrischen Skalierung

Die bisherigen Überlegungen führten zu einer sehr guten *logischen* Wiederverwendbarkeit der Elemente. Jedoch spätestens beim Versuch, eine größere Hierarchie von ganzen Präsentationen zu erstellen, stößt man auf ein weiteres Problem. Es ist nämlich nach dem bisherigen Ansatz zwingend erforderlich, die geometrischen Ausmaße eines übergeodneten Mob mindestens so groß wie die des darunter liegenden

[1] nachfolgend als (Medien-)Objekt bzw. *Mob* bezeichnet

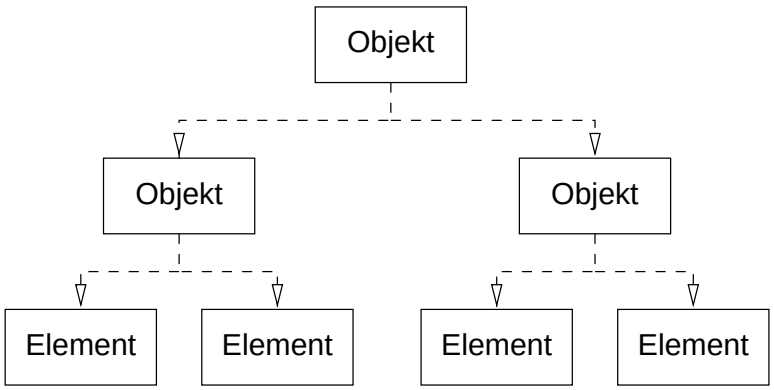


Abbildung 3.4.: Hierarchische Darstellung der logischen Abhängigkeiten zwischen Objekten

festzulegen. Das führt aber letztendlich zu einer nicht erwünschten Gesamtgröße. Um dieses Defizit zu lösen, muß es möglich sein, die eingebetteten Mobs in ihren geometrischen Ausmaßen anzupassen, unabhängig davon, ob es sich dabei um ein einzelnes Element oder eine ganze Präsentation handelt. Aus diesem Grunde muß jedes Mob und jedes Element die Fähigkeit haben, sich selbst skalierbar darzustellen. Wenn nun ein Objekt einem anderen untergeordnet wird, muß ihm ein Skalierungsfaktor zugeteilt werden, der letztlich seine Erscheinungsgröße bestimmt. Da bei einer Durchwanderung eines Wurzelzweiges von oben nach unten mehrere solcher Faktoren auftreten können, kann diese Angabe nur eine relative sein. Die effektive Größe der Darstellung muß daher aus dem Produkt der einzelnen Faktoren errechnet werden. Ein Beispiel dieser Begebenheit ist in Abbildung 3.7 auf Seite 22 zu sehen. Zum Zwecke der Vereinfachung habe ich in dem Beispiel zwei Mobs und ein Element mit identischen Ausgangsmaßen ($x = y = 120$) verwendet.
Das Element 1.1.1 ist in der Darstellung mit dem Skalierungsfaktor $s_3 = 0.333$ in das ihm übergeordnete Objekt 1.1 eingebettet. Da aber dieses selbst schon mit dem Skalierungsfaktor $s_2 = 0.5$ angezeigt wird, ergibt sich für Element 1.1.1. ein effektiver Darstellungsfaktor von:

$$s_{3_{eff}} = s_2 \cdot s_3 = 0.5 \cdot 0.666 = 0.166$$

Aus genau diesem Faktor läßt sich dann die Darstellungsgröße des Elements 1.1.1 errechnen:

$$x_{3_{eff}} = y_{3_{eff}} = s_{3_{eff}} \cdot 120 = 0.166 \cdot 120 = 20$$

Mit der Fähigkeit, Mobs unabhängig von der Anzahl weiterer enthaltener Mobs zu skalieren, können jetzt sämtliche Elemente einer Präsentation beliebig wiederverwendet werden.

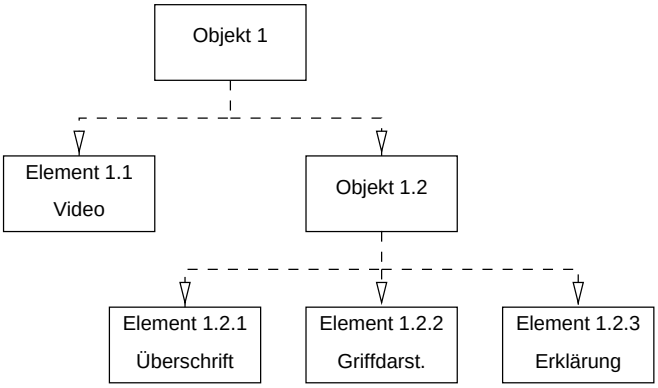


Abbildung 3.5.: Logische Struktur des Gitarrengriff-Beispiels

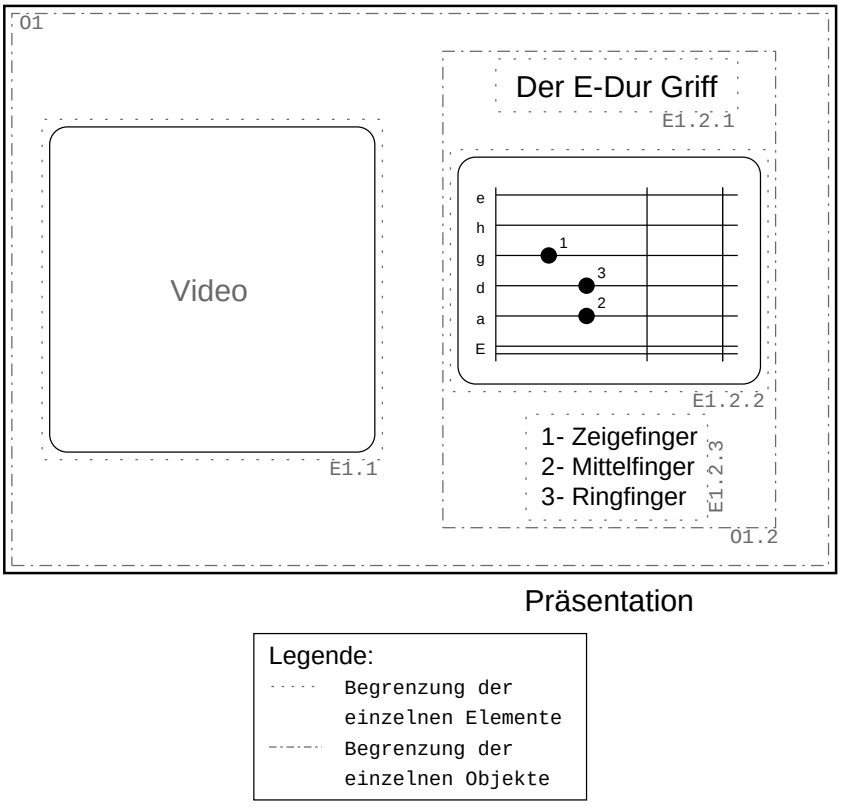


Abbildung 3.6.: Darstellung des Gitarrengriff-Beispiels

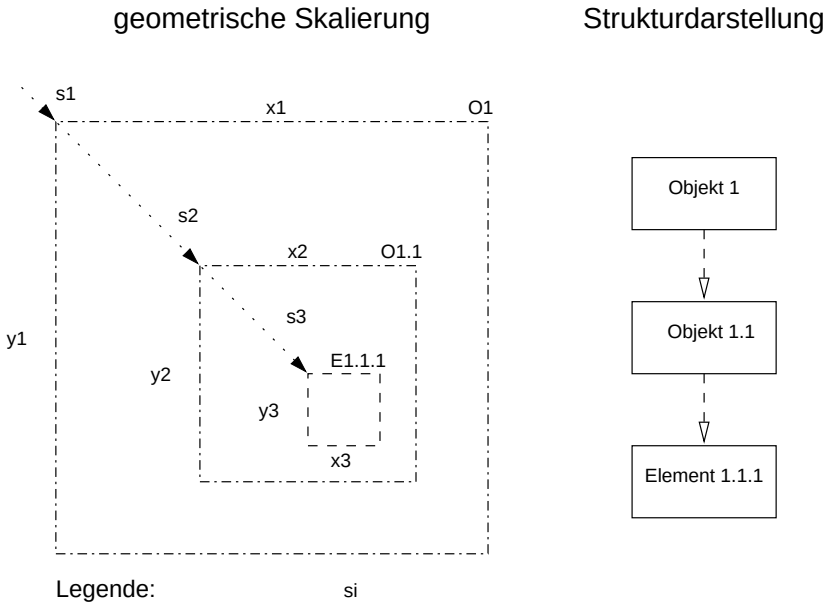


Abbildung 3.7.: Schematische Darstellung des Skalierungvorganges

3.2.3. Erweiterbarkeit der Plattform um neue Medientypen

In den bisher zur Veranschaulichung konstruierten Beispielen gab es drei unterschiedliche Typen von Elementen. Dabei handelte es sich um Text-, Bild- und Videoelemente, die neben dem bisher unerwähnten Audioelement die wesentlichen Medientypen repräsentieren, die eine Lernplattform bieten könnte. Leider gestaltet sich diese Vierteilung als nicht umsetzbar, denn es kann z. B. nicht von *dem* Element Bild gesprochen werden. So existieren eine Vielzahl von unterschiedlichen Bildformaten, die verschiedensten Videoformate bzw. -codecs und auch Audioformate sind in nicht geringer Anzahl bekannt.

Neben den unterschiedlichen Datenformaten können natürlich auch unterschiedliche Funktionalitäten nicht durch einen gemeinsamen Elementtyp realisiert werden. So wäre ein Textelement denkbar, das eine Laufschrift anzeigt, ein anderes, welches seinen Text ein- und ausblendet und ein drittes, das den Text im Blocksatz präsentiert. Es ist daher nicht ausreichend, die vier „Grundelemente“ zu implementieren.

Nun könnte man einfach die wichtigsten Medientypen bzw. -formate fest in die Plattform integrieren, was aber letztendlich zu einem festen, ohne größere Programmierarbeit nicht erweiterbaren Repertoire an darstellbaren Elementtypen führen würde. Versucht man aber die Implementierung der einzelnen Typen aus der zugrunde liegenden Plattform herauszulösen und schafft definierte Schnittstellen, über welche neue Elemente an die Plattform angekoppelt werden können, so erhält man eine äußerst flexible Lösung, die späteren Erweiterungen offen gegenübersteht. Im Idealfall könnten danach Elementtypen beliebig zwischen verschiedenen Instanzen der Plattform getauscht werden, vollkommen unabhängig von ihrem Datenformat

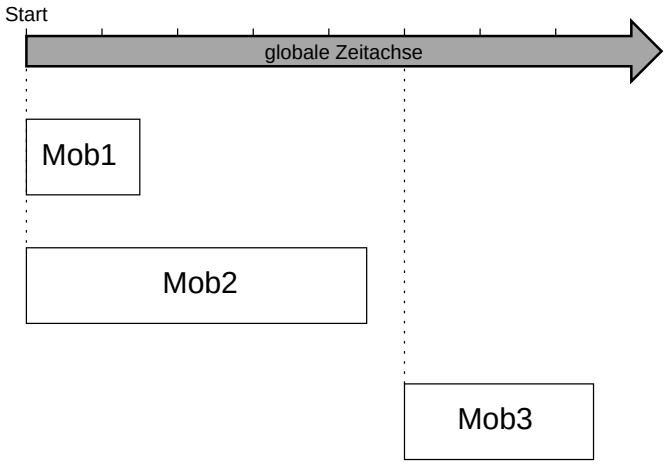


Abbildung 3.8.: Präsentationsablauf bei einer globalen Zeitachse

und der innewohnenden Funktionalität und ohne jegliche Programmierarbeit, die der Autor der Inhalte zumeist sowieso nicht erledigen könnte.

3.2.4. Zeitliche Synchronisation der Präsentationsobjekte

Die zeitliche Synchronisation der einzelnen Präsentationsobjekte untereinander ist eine äußerst wichtige Problematik, die als Voraussetzung für die Interaktionsmöglichkeit unerläßlich ist. Wie schon in 3.1 auf Seite 14 dargelegt, muß es möglich sein, den Zeitpunkt einer eintretenden Interaktion genau bestimmen zu können. Nur wenn dieser Umstand gegeben ist, kann die dem entsprechenden Zeitpunkt zugeordnete Aktion angestoßen werden.
Unabdingbar für die zeitgenaue Ermittlung der Interaktionsparameter ist natürlich der synchrone Ablauf der einzelnen Komponenten einer Präsentation. Der Autor soll z. B. die Möglichkeit haben, ein Objekt nach einer definierten Zeit sichtbar werden zu lassen. Da durch diese Fähigkeit die Mobs nicht nur durch ihre Inhalte, sondern auch den Zeitpunkt ihrer Präsentation beschrieben werden, muß die Synchronisation so genau wie möglich implementiert werden, um den gewünschten Zusammenhang zu gewährleisten.

Legt man nun der gesamten Präsentation eine einzige globale Zeitachse zugrunde, so können sämtliche Aktionen an dieser ausgerichtet werden. Eine schematische Darstellung dieser Lösung ist in Abbildung 3.8 ersichtlich.
Alle drei Mobs orientieren sich an der einzigen Zeitachse der Präsentation. Da eine derartige Achse programmtechnisch nur durch einen Timer, der in diskreten Intervallen voranschreitet, zu erzeugen ist, sind auch alle Mobs auf die Granularität dieses Timers angewiesen. Wenn aber z. B. eines der Objekte ein Videoelement ist, so wird die Wahrscheinlichkeit sehr gering sein, daß der globale Timer die für die gewünschte Frame-Rate des Videos nötige Taktweite besitzt.

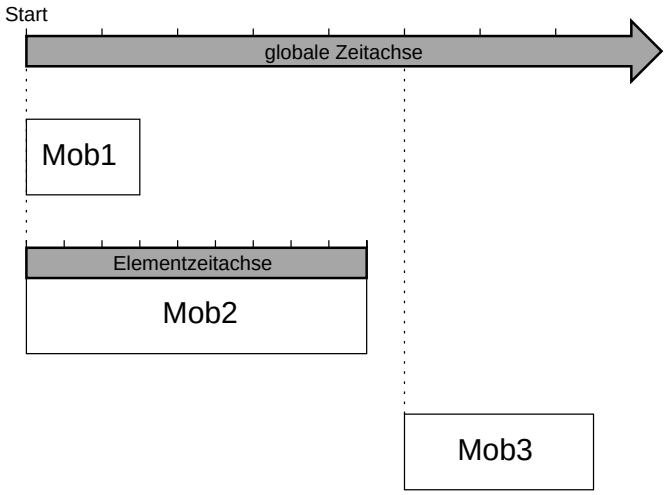


Abbildung 3.9.: Ablauf bei mehreren Zeitachse

Als Lösung bietet sich hier an, jedem Element die Möglichkeit zu geben, eine eigene Zeitachse zu benutzen. Die damit gewonnene Objekteigenständigkeit begünstigt nicht nur die Objektorientiertheit der Elementverwaltung, sondern ermöglicht überhaupt erst das Einbinden von gestreamten Medien (3.3.3 auf Seite 32). Um dabei die Gesamtsynchronisation zu gewährleisten, muß sich aber ein solches „nebenläufige“ Element durch garantierte Aktionszeiten an die globale Zeitachse binden lassen. Es ist z. B. zwingend erforderlich, daß ein Video bei einem Signal, welches ihm den Abspielstart anzeigt, wirklich mit selbigem beginnt oder zumindest seine eigene Zeitachse startet. Würde es in diesem Moment beispielsweise erst Initalisierungen durchführen und so den Beginn verzögern, würde automatisch auch sein Ende zu einem späteren Zeitpunkt erreicht sein. Andere Elemente, die vielleicht genau auf diesen Endpunkt synchronisiert wurden, erscheinen dann in ihrer Aktion dem Betrachter als verfrüht, obwohl der eigentliche Urheber der Unstimmigkeit das vorherige Videoelement ist.
Die schematische Darstellung des Systems mit mehreren Zeitachsen ist in Abbildung 3.9 zu sehen.

3.2.5. Aktionsverarbeitung

Befehle

Um nun überhaupt Nutzen aus der Synchronisationsfähigkeit zu ziehen, muß die Plattform die Fähigkeit besitzen, zu bestimmten Zeiten bestimmte Aktionen auf die Objekte anwenden zu können. Derartige Aktionen sind z. B. die Befehle zum Sichtbar machen oder Verdecken der Elemente auf dem Bildschirm. Betrachtet man in diesem Zusammenhang das unter 3.2.1 dargelegte Problem der unter Umständen vorhandenen niedrigen Datenübertragungsraten, wird klar, daß ein einfacher Befehl zum Anzeigen der Elemente nicht ausreicht, um die gewünschte Synchronität zu gewährlei-

sten. Handelt es sich beispielsweise um ein größeres Bildelement, welches darzustellen ist, kann dessen Übertragung eine gewisse und unbestimmte Zeit in Anspruch nehmen. Erhält nun dieses Element den Befehl, sich selbst darzustellen, muß es erst die Bilddaten vom Server laden und kann somit nicht zum gewünschten Zeitpunkt den Befehl ausführen. Abhilfe schafft hierbei die Einführung einer Initialisierungsanweisung. Wenn diese dem Element vor dem Befehl zum Anzeigen übergeben wird, kann es sich „in Ruhe“ darauf vorbereiten und zum angestrebten Zeitpunkt auch wirklich mit der Darstellung beginnen.

Ein weiteres Problem stellen die Objekte dar, die eine eigene Zeitachse besitzen. Da es z. B. nicht allgemeingültig festlegbar ist, was im Falle des Verdeckens eines solchen Mobs mit seiner Ablaufzeit geschehen soll (soll sie weiterlaufen oder stoppen?), müssen hierfür zwei weitere Befehle eingeführt werden, nämlich zum Starten und Stoppen des Elementtimers.

Neben diesen unabdingbaren Kommandos sind natürlich auch weitere denkbar. So bietet es sich in Hinblick auf die größtmögliche Flexibilität der Plattform an, einen „freien“ Befehl zu implementieren. Durch ihn könnten zukünftige Medienobjekte eigene Befehle verwenden, die sie dem „Platzhalter“-Befehl als Parameter mitgeben und später selbst auswerten können, ohne daß die Plattform programmtechnisch verändert werden muß.

Playlist

Um die Anweisungen zu bestimmten Zeiten an die Zielobjekte zu binden, muß eine sogenannte Playlist vorhanden sein. In ihr stehen in einer direkten Zuordnung die Aktionszeiten, das Zielobjekt und natürlich die auszuführende Aktion. Jedes Mob besitzt einen derartigen „Spielplan“, welcher den Ablauf seiner untergeordneten Objekte bestimmt. Trägt man der unter 3.2.2 auf Seite 18 eingeführten Modularisierung Rechnung, müssen sich die einzelnen Zeiten der Playlist natürlich immer um die Zeitspanne nach hinten verschieben, nach der das der Playlist zugehörige Mob von seinem übergeordneten Mob gestartet wird. Ein Beispiel ist in Abbildung 3.10 auf der nächsten Seite dargestellt. Es zeigt, daß sich der Nullpunkt der einzelnen Objekte im Vergleich zur globalen Zeitachse je nach Startzeit verschiebt.

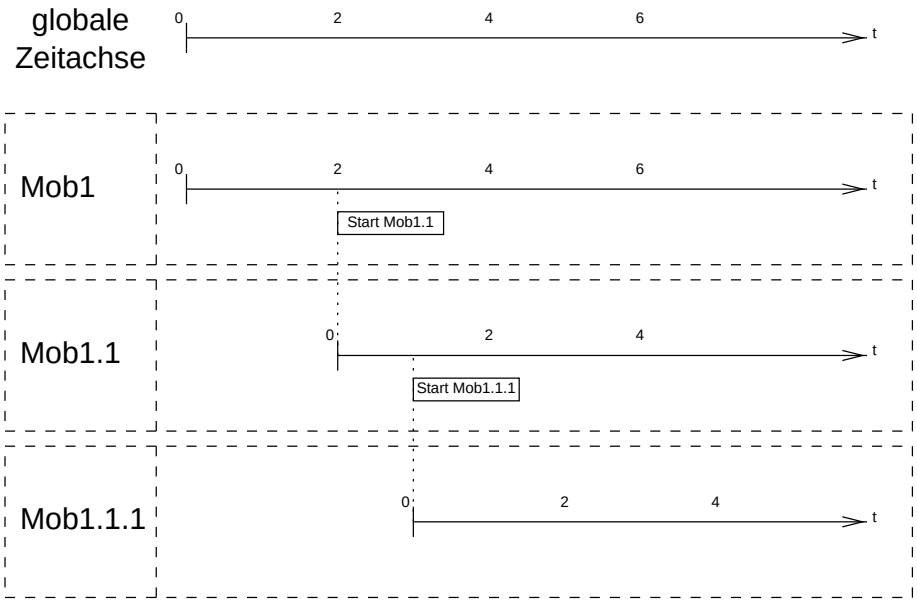


Abbildung 3.10.: Verschiebung der Aktionszeiten in einer Mob-Hierarchie

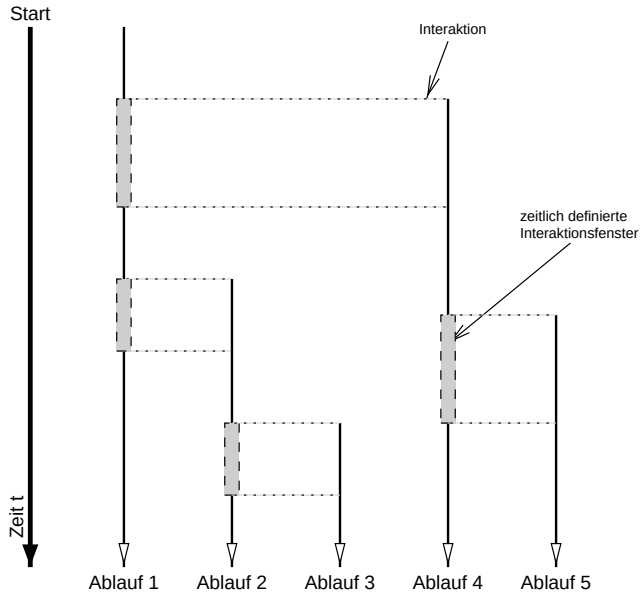


Abbildung 3.11.: Beispiel eines Präsentationsablaufs mit fest vorgegebenen Handlungssträngen

3.2.6. Interaktion

Die Interaktionsfähigkeit ist ein wesentlicher Bestandteil eines modernen und umfassenden Lehrsystems, weshalb auch die zu entwickelnde Plattform dieses Potential innehaben soll.

Mit einem Beispiel möchte ich die Bedeutung der Interaktionsfähigkeit und den Programmablauf beim Eintreten einer Interaktion deutlich machen. Geht man beispielsweise von einem Videoelement aus, soll der Nutzer durch einen „Mausklick“ in das Videoelement in Abhängigkeit von dessen Ort und Spielzeit situationsbezogene Inhalte angezeigt bekommen.

Um dieses Szenario zu ermöglichen, muß beim Erstellen der Präsentation für jedes Objekt festgelegt werden, zu welcher Zeit und an welchem Ort ein sogenannter sensitiver Bereich aktiviert werden soll. Wenn nun der Nutzer eine Interaktion in Form eines „Klicks“ auslöst, überprüft der Client, über welchem Objekt und zu welchem Zeitpunkt das geschehen ist. Dazu durchwandert er die Objekthierarchie vom Wurzelobjekt ausgehend abwärts, bis er eine Komponente findet, für die an der Position der Interaktion ein sensitiver Bereich definiert wurde. Jetzt muß der Client entscheiden, welche Konsequenzen sich daraus für den Zustand der Präsentation ergeben.

Genau das ist aber ein durchaus komplexes Problem, da die Folgeaktionen durch eine Vielzahl von Abhängigkeiten definiert sein können. Daher stellt sich die Frage, wie all diese Abhängigkeiten und die daraus möglicherweise resultierenden Inhalte beschrieben und umgesetzt werden können. Dafür bieten sich zwei Lösungen an:

- Es könnten für alle nur erdenklichen Interaktionen die daraus folgenden unter-

schiedlichen Präsentationsabläufe erstellt werden. Tritt nun eine Interaktion ein, wird das gerade aktuelle Objekt durch ein völlig anderes ersetzt und die Präsentation folgt somit einem anderen Handlungsstrang (vergleiche Abb. 3.11 auf der vorherigen Seite).
Allerdings ergeben sich aus diesem Ansatz einige Probleme. So müssen schon bei einer moderaten Anzahl von Interaktionen viele Handlungsstränge erstellt werden, die sich aber u. U. nur minimal unterscheiden. Letztendlich würde das zu einer sehr hohen Redundanz führen, die im Widerspruch mit der gewünschten Wiederverwendbarkeit der einzelnen Komponenten verursacht wird.
Ein zweites Problem stellt das hohe Datenaufkommen dar, das durch die unterschiedlichen Handlungsabläufe entsteht. Da es sich bei der Plattform um eine Client/Server-Anwendung handelt, können die zu übertragenden Datenmengen zu Verzögerungen des Präsentationsablaufs führen.

- Es wäre denkbar in die Plattform eine Sprache zu integrieren, mit der beliebig komplizierte Abhängigkeiten zwischen den Komponenten beschrieben werden können. Diesen Ansatz verfolgen auch die derzeitig erhältlichen Authoringtools. Da aber die Umsetzung einer solchen Sprache einen erheblichen Aufwand verursachen würde, eignet sich diese Vorgehensweise nicht.

Im Rahmen dieser Diplomarbeit kann ich nur die beiden eben erwähnten Denkansätze zur Lösung dieses nicht trivialen Problems darlegen. Da sich aber wie bereits gezeigt beide nicht für die zu entwickelnde Plattform eignen, muß ein anderer Weg gefunden werden, um die Interaktionsfähigkeit zu implementieren. Da der hierfür erforderliche Zeitaufwand nicht innerhalb dieser Arbeit erbracht werden kann, lasse ich diesen Punkt für eine spätere Weiterentwicklung des Systems offen.

3.2.7. Handhabung der Objektdaten

Bei einer umfangreichen Präsentation stellt sich die Frage, in welcher Form die Plattform die Daten verwaltet. Betrachtet man die Abhängigkeiten zwischen den Mobs, die Größe und Anzahl der Objekte und die Art und Weise des Zugriffs auf selbige, bietet sich die Speicherung in einer Datenbank geradezu an. In ihr können die Präsentationsstrukturen direkt abgebildet werden. Da aber das Design einer Datenbank und die Programmierung der Zugriffsroutinen den zeitlichen Rahmen der vorliegenden Arbeit sprengen würde, habe ich mich entschlossen, die Mobs als normale Dateien im Filesystem abzulegen. Die Beschreibung jedes Objekts wird dabei in einer eigenen Datei, die als Namen eine eindeutige Identifikationsnummer besitzt, abgelegt. Dabei soll das Datenformat folgenden Überlegungen gerecht werden:

- Die Objektbeschreibung sollte mit einem einfachen Texteditor erstell- und bearbeitbar sein. Dadurch kann ohne spezielle Drittsoftware eine Präsentation entworfen werden. Auch das Einlesen und Umsetzen dieser Beschreibungsdateien gestaltet sich so recht einfach.

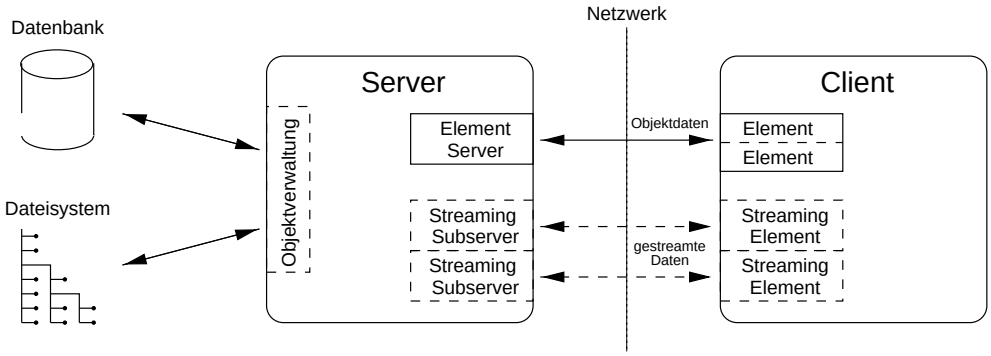


Abbildung 3.12.: Client/Server-Modell

- Da die Beschreibungen einer umfangreicheren Präsentation relativ komplex werden kann, sollte bei Bedarf auch mittels eines frei erhältlichen speziellen Editors die Bearbeitung vorgenommen werden können.
- Die Syntax der Objektbeschreibung soll einer eventuell später erfolgenden Migration in eine Datenbank entgegen kommen.

Als Lösung bietet sich hierfür idealerweise die Extensible Markup Language (XML) an. Sie ist eine Teilmenge von SGML und bietet die Möglichkeit, eine eigene Struktur „zu formen“, in der die Daten abgespeichert werden. Diese Struktur läßt sich so gestalten, daß sie später in idealer Weise in einer Datenbank abgebildet werden kann. Außerdem existieren für sie die unterschiedlichsten Editoren, Betrachter und auch Programmier-Toolkits, die den Umgang mit ihr wesentlich vereinfachen. Weitere Informationen zu XML sind unter 3.3.4 auf Seite 33 nachzulesen.

3.3. Konzeption einer Lehr- und Präsentationsplattform

3.3.1. Client/Server-Modell

Bei dem umzusetzenden System handelt es sich in seiner Architektur um eine klassische Client/Server-Anwendung. Hierbei existiert ein zentraler Server, der im Idealfall beliebig viele Clients bedient.
In diesem konkreten Beispiel einer Lehr- und Präsentationsplattform übernimmt der Server die Aufgabe, die Mobs zu verwalten und auf Anfragen des Clients entsprechend zu reagieren. Der Client seinerseits ist für den strukturellen Aufbau, die Steuerung und Darstellung der Präsentationen zuständig. Ein Modell dieser Aufgabenteilung ist in Abbildung 3.12 zu sehen. Die in der Darstellung ersichtlichen Komponenten, die für die gestreamten Elemente verantwortlich sind, werden in 3.3.3 näher erläutert.

Listing 3.1: HTML-Fragment zum Einbinden eines Java-Applets

```
<APPLET code="BeispielApplet.class" alt="Beispielapplet">
    <PARAM name=param1 value="ein Parameter">
    <PARAM name=param2 value="ein weiterer Parameter">
</APPLET>
```

3.3.2. Programmiersprache Java

Wie bereits dargelegt, bilden Java-Applets den einzigen zufriedenstellenden Ansatz, um die notwendige Funktionalität der Plattform zu implementieren. Daher habe ich mich auch für Java[13] als Programmiersprache entschieden.
Diese objektorientierte Sprache ist von der Firma Sun Microsystems entwickelt worden. Ihre Beliebtheit und Verbreitung hat seit ihrer ersten Veröffentlichung im Jahre 1996 rasant zugenommen. Mittlerweile existieren Javaumgebungen für alle gängigen Betriebssysteme. Die Sprache hebt sich besonders durch ein Merkmal von den meisten anderen Sprachen wie C oder C++ ab, welches sie gerade für den Einsatz im WWW prädestiniert: ihre Plattformunabhängigkeit. Erreicht wird diese Fähigkeit, indem der Java-Sourcecode durch den Javacompiler in einen vollständig definierten und maschinenunabhängigen Zwischencode, dem Java-Bytecode, umgesetzt wird. Dieser Bytecode kann anschließend auf jeder Plattform, die eine sogenannte Java Virtual Machine (JVM) anbietet, ausgeführt werden. Die JVM dient dabei als Interpreter, der den Zwischencode in einen maschinenabhängigen Code übersetzt. Natürlich bietet Java auch andere Vorteile, wie z. B. hohe Sicherheit, Unterstützung vieler Schnittstellen, sehr elegante Behandlung von Ausnahmen (Exceptions) oder einfach nur das angenehme Programmieren mit der Sprache, die ich aber im Rahmen dieser Diplomarbeit nicht näher erläutern möchte, da sie bereits an vielen anderen Stellen diskutiert und dargelegt wurden.

Client

Die Firma Sun hat nach Veröffentlichung der Sprache Java in ihren Browser HotJava eine eigene Syntax zum Einbinden von Javacode in HTML-Seiten integriert. Mit deren Hilfe wurde es möglich, kleine Javaprogramme - die sogenannten Applets - innerhalb der Webseiten clientseitig auszuführen. Diese Programme laufen dann innerhalb des Browsers und ermöglichen dadurch dynamische Webinhalte. Netscape implementierte eine andere Syntax, die wegen der wesentlich größeren Verbreitung ihres Browsers heute Verwendung findet. Nach ihr wird ein Applet durch den in Listing 3.1 dargestellten Code in eine HTML-Seite integriert. Da der Appletcode innerhalb des Browsers durch dessen eigene JVM ausgeführt wird, darf er in maximal der Sprachversion vorliegen, die von der JVM unterstützt wird. So kann der zur Zeit aktuelle Navigator 4.08 nur Programme nach der Java 1.1-Spezifikation ausführen.
Das derzeitige Java 1.2 bietet aber gegenüber den älteren Versionen einige Vorteile,

wie neue Klassen oder einen Just-In-Time-Compiler (JIT), so daß es sinnvoll erscheint, diese Version als Grundlage für die zu entwickelnde Plattform zu verwenden. Möglich wird das durch ein von Sun entwickeltes, für mehrere Plattformen erhältliches Java-Plugin[14]. Nach dessen Installation auf dem Clientrechner verwendet der Browser nicht mehr seine integrierte JVM, sondern er benutzt eine andere, die eine neuere Java-Version unterstützt. Das hat natürlich den Nachteil, daß jeder „Konsument“ der Lehrinhalte momentan dieses Plugin auf seinem Rechner installieren muß. Allerdings kann davon ausgegangen werden, daß die nächsten Browserversionen den Java 1.2-Standard unterstützen werden, so daß das Plugin überflüssig sein wird.

Server

Die Entscheidung, Java auch serverseitig einzusetzen, beruht auf folgenden Überlegungen:

- Da der Server auch über einen sehr langen Zeitraum stabil laufen soll, muß bei der Implementierung der Datenverwaltung sehr viel Sorgfalt aufgewendet werden, um beispielsweise nicht das Phänomen des „vollaufenden“ Speichers infolge einer unzulänglich implementierten Resourcenfreigabe zu erhalten. Diese verdeckten Fehler, die erst nach einiger Laufzeit zutage treten, entstehen oft durch fehlerhaft programmierte Zeigerarithmetik oder einfach durch versehentlich nicht freigegebenen Speicherplatz.
  Die Sprache Java bietet ideale Voraussetzungen, um dieser Problematik Herr zu werden. Sie verzichtet völlig auf Zeiger und verwaltet den durch das Programm belegten Speicher vollautomatisch. Dazu läuft bei jeder Java-Anwendung im Hintergrund ein sogenannter Garbage Collector. Bei diesem handelt es sich um einen Thread, der nicht mehr referenzierte Objekte selbstständig auflöst.

- Die Daten der Präsentationen werden, wie unter 3.2.7 auf Seite 28 dargelegt, in XML-Strukturen gespeichert. Auch hierfür bietet Java gute Voraussetzungen, gibt es doch verschiedene fertige Klassen, die das Arbeiten mit XML-Daten vereinfachen. Besonders die Firma IBM engagiert sich in den Bereichen Java und XML und bietet diesbezüglich viele, größtenteils freie, Klassen, Tools und Informationen an.
  Ebenso ergeben sich bei einer späteren Speicherung der Daten in einer Datenbank keinerlei Probleme. Java bietet auch hierfür passende Klassen an. Mit Hilfe des Java Database Connectivity Package (JDBC) kann ohne weiteres der Zugriff auf SQL-Datenbanken programmtechnisch realisiert werden.

- Die Datenübertragung zum Client läßt sich durch die gewählte Sprache sehr einfach lösen, da sie eine eingebaute Netzwerkfähigkeit besitzt. Besonders die Möglichkeit, ganze Sprachobjekte serialisieren und über eine Netzwerkverbindung senden zu können, stellt einen wichtigen Vorteil dar. Dadurch vereinfacht sich die Client/Server-Kommunikation wesentlich.

3.3.3. Gestreamte Medien / Wavelet-Element

Da die zu entwickelnde Plattform zeitgesteuert arbeiten wird, bietet sie auch das Potential für zeitbasierte Medien. Als wichtigster Vertreter wäre dabei das Video zu nennen, welches jedoch nur als Datenstrom zeitsynchronisiert verarbeitet werden kann. Um verschiedene derartige Medien unterstützen zu können, muß der Server für jeden Medientyp einen eigenen Stream-Server instanziieren. Dieser übernimmt danach die eigentliche Kommunikation mit einer Empfangskomponente auf der Clientseite, so daß sich der Plattformserver nicht um die eigentliche Datenübertragung kümmern muß.

Am Beispiel des Videos wird ein wichtiger Aspekt dieser „stromartigen“ Übertragung deutlich: sie muß in Echtzeit erfolgen. Hierbei kommt ein speziell für derartige Anwendungfälle vorgesehenes Protokoll zum Einsatz, nämlich das nach RFC1889[15] definierte Real-Time-Protokoll (RTP). Mit seiner Hilfe übertragende Daten enthalten einige Status- und Zeitinformationen, die ein echtzeitfähiges Abspielen der gestreamten Medien gewährleisten. Um eventuell auftretende Verzögerungen bei der Datenübermittlung durch ein Netzwerk zu verdecken, wird außerdem das sogenannte *Latency Hiding* eingesetzt. Hinter diesem Begriff verbirgt sich eine durch geschickte Pufferung des Streams erreichte Überbrückung von kurzen Übertragungstotzeiten.

Als Anwendungsbeispiel soll im Rahmen der vorliegenden Diplomarbeit ein Videoelement integriert werden, welches mittels eines Wavelet-Algorithmus komprimierte Daten dekodiert und am Bildschirm darstellt. Der sehr effiziente Wavelet-Algorithmus beruht auf einer Hoch-/Tiefpaßzerlegung der Bildsignale. Die Kodierung der Daten erfolgt durch ein externes Programm, welches aus bestehenden AVI-Files ein Wavelet-kodiertes File mit den Video- und ein MPEG-kodiertes (Layer 3) File mit den Audiodaten erzeugt. Die beiden so erstellten Dateien werden dann durch die Wavelet-Serverinstanz separat zum Client gestreamt, der beide Datenströme empfängt, dekodiert und abspielt. Besonders interessant ist dabei die Fähigkeit, den Videodatenstrom in Abhängigkeit von der zur Verfügung stehenden Übertragungsbandbreite skalieren zu können. Um das zu erreichen, werden bei einer niedrigeren Bandbreite Teile der waveletkodierten Videodaten ausgelassen. Das führt zwar zu einer verminderten Bildqualität, die jedoch durch den Vorteil der fortlaufenden Videodarstellung aufgewogen wird.

Die Programmierung der einzelnen Programme und Codeteile, die den Wavelet-Codec betreffen, ist nicht Teil der Diplomarbeit. Sie müssen nur in das System integriert werden. Da die entsprechenden Teile in Java geschrieben wurden, gestaltet sich die Zusammenführung mit der Plattform als relativ einfach. Für genaue Informationen zu diesem Wavelet-Algorithmus sei auf [16] verwiesen.

Listing 3.2: wohlgeformtes XML-Dokument

```
<?xml version="1.0" standalone="yes"?>
    <unterhaltung>
        <grusswort>Hello, world!</grusswort>!
        <antwort>Stop the planet, I want to get off!</antwort>!
        <wichtigkeit wert="urgent" />
    </unterhaltung>
```

3.3.4. XML

Einführung in XML

Die *Extensible Markup Language* (XML) ist eine Teilmenge von SGML, welche eine äußerst umfangreiche Dokumentenbeschreibungssprache darstellt, die zum standardisierten Dokumentenaustausch benutzt wird. Da diese Sprache aber gerade wegen ihres Umfangs nicht für z. B. den Austausch von Dokumenten im WWW geeignet ist, hat das W3C Konsortium im Februar 1998 eine Empfehlung[17] der Sprache XML veröffentlicht. Sie enthält nur einen Teil der Möglichkeiten von SGML und bildet dabei eine „Metasprache“, mit der eigene Markup-Languages definiert werden können.

Ein gültiges XML-Dokument besteht aus mehreren unterschiedlichen Elementen, die nach der XML-Syntax angeordnet sein müssen. Ein solches Dokument wird dabei als *wohlgeformt* bezeichnet. Desweiteren besteht die Möglichkeit, mit einer sogenannten Document Type Definition (DTD) grammatikalische Regeln festzulegen, nach denen die XML-Tags angeordnet sein müssen. Eine XML-Beschreibung, die solche Regeln befolgt, nennt man *gültiges* Dokument.
Ein Beispiel einer wohlgeformten XML-Struktur ist in Listing 3.2 zu sehen. Wie man erkennen kann, besteht die XML-Struktur aus (den von HTML bekannten) „Tags“, derer es zwei unterschiedliche Typen gibt:

- Start/End-Tag Paare. Ein Solches wäre im Listing z. B. das `<[/]unterhaltung>`-Tag Paar, das, wie im Beispiel zu sehen, auch weitere untergeordnete Elemente haben kann.

- Leeres-Element-Tag (z. B. `<wichtigkeit>`), das keine weiteren untergeordneten Elemente haben kann.

Im Gegensatz zu HTML ist man aber nicht auf fest definierte Tags angwiesen. So sind die in Abbildung 3.2 dargestellten Tags nach der XML-Syntax gültig, denn die Namensgebung wird nicht durch diese vorgegeben und es wird keine DTD verwandt, die grammatikalische Regeln definiert.
Da eine weitergehende Vertiefung des Themas XML nicht im Sinne dieser Arbeit ist, sei für ausführlichere Informationen auf [18, 19] verwiesen.

XML4J

Um die in XML vorliegenden Objektbeschreibungen auslesen zu können, verwende ich die XML4J-API[20] in der Version 2.0 der Firma IBM. Dabei handelt es sich um einen modularen XML-Parser, der als Java-Archiv (JAR) vorliegt.
Um mit seiner Hilfe auf eine XML-strukturierte Datei zugreifen zu können, müssen folgende Programmschritte erfolgen:

1. Es muß ein Parser instanziiert werden. Da ich mich der Einfachheit halber für wohlgeformte XML-Dateien entschieden habe, reicht es aus, einen Parser zu verwenden, der nicht die Gültigkeit der Syntax anhand einer DTD prüft:
   NonValidatingDOMParser parser = **new** NonValidatingDOMParser();

2. Dem Parser werden anschließend die zu bearbeitenden XML-Daten übergeben:
   parser.parse (**new** InputSource(**new** FileReader ("daten.xml")));

3. Mit Hilfe des vom Parser ermittelten `documents` kann dann der Zugriff auf die einzelnen Elemente der XML-Datei erfolgen:
   NodeList nodelist = parser.getDocument().getElementsByTagName("TAGNAME");

3.4. Datenmodell

3.4.1. Dynamische Datenstruktur des Clients

Die Fähigkeit des Clients, beliebige Präsentationen darstellen zu können, bedingt eine dynamische Datenstruktur, welche die Präsentationsinhalte wiederspiegelt. Wie in Abbildung 3.4 auf Seite 20 dargestellt, besteht diese Struktur aus Objekten und Elementen. Letztere haben die Aufgabe, die Elementdaten in Abhängigkeit von den auf sie angewandten Aktionen am Bildschirm wiederzugeben. Die Objekte hingegen besitzen solche Daten nicht. Sie dienen einzig und allein der logischen Gruppierung und der Steuerung ihrer untergeordneten Objekte bzw. Elemente. Um unnötigen Aufwand bei der Implementierung zu vermeiden, wäre es vorteilhaft, beide Typen in einer Datenstruktur zu vereinen. Die so leichter handhabbare Struktur wird dann aus einer einzigen Javaklasse zusammengesetzt, die im folgenden als *Mob* bezeichnet wird. Die Abbildung 3.13 auf der nächsten Seite zeigt die so leicht veränderte logische Struktur.
Mobs können je nach Typ unterschiedliche Klassen beinhalten, die nachfolgend einzeln vorgestellt werden sollen.

Playlist

Die Playlist enthält die Aktionen, die auf untergeordnete Mobs zu bestimmten Zeiten angewandt werden sollen. Die Zuordnung erfolgt dabei durch eine Tabelle, die neben dem Zeitpunkt und der Aktion auch das entsprechende Ziel-Mob kapselt. Tabelle 3.1

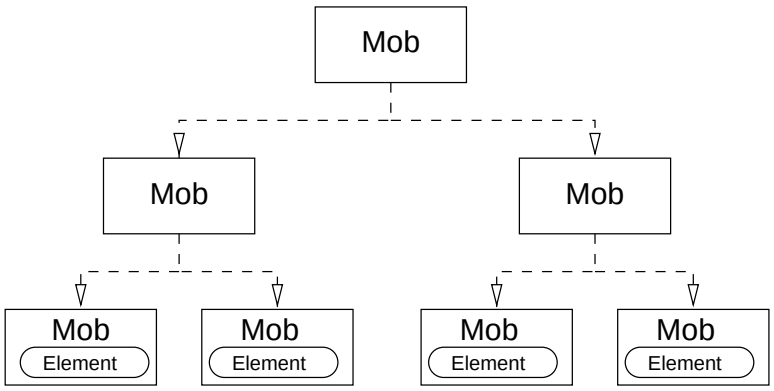


Abbildung 3.13.: Logische Struktur mit gekapselten Elementen

stellt eine derartige Playlist dar.
Die darin vorkommende Zielobjekt-ID wird aus der Moblist (siehe Tabelle 3.2) ermittelt.

Timer

Der Timer stellt eine Referenz auf den für das Mob verantwortlichen Zeitgeber dar und wird dem Medienobjekt von seinem übergeordneten Mob zugewiesen. Dadurch wird es möglich, je nach Umfang der Präsentation mehrere Timer zu verwenden, um so einen einzigen Timer nicht zu überlasten.

Datenquelle

Die Datenquelle ist dafür verantwortlich, sämtliche vom Mob benötigten Daten (z. B. Playlist, Moblist oder Elementdaten) zu liefern. Sie bildet (wie in Abb. 3.14 auf der nächsten Seite dargestellt) damit eine Pufferschicht zwischen Mob und Server und übernimmt alle für die direkte Datenübertragung wichtigen Aufgaben. Somit

Zeit t	Aktion	Zielobjekt-ID
0	init	0
0	init	1
0	start	0
0	show	0
2	start	1
2	init	2
4	show	1
7	show	2

Tabelle 3.1.: Beispiel einer Playlist

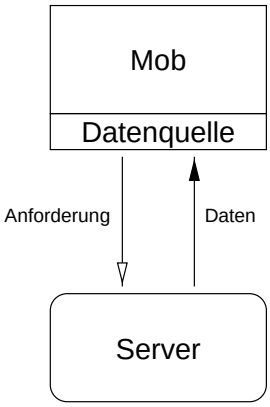


Abbildung 3.14.: Darstellung der Pufferfunktion einer Datenquelle

braucht sich das Mob nicht darum zu kümmern, woher die Daten stammen und wie diese übertragen werden.

Die angeforderten Daten erhält die Datenquelle normalerweise vom Server. Es ist aber durchaus denkbar, daß sie die Daten stattdessen direkt von der Tastatur des Benutzers bekommt. Dadurch könnte zukunftig eine weitere Form der Interaktion implementiert werden.

Da sich die Datenquelle genau zwischen dem Datenerzeuger/-zulieferer und dem Mob befindet, könnte sie idealerweise auch ein intelligentes Caching der durchgeleiteten Daten übernehmen. Dadurch würde der zu Wartezeiten führende Netzwerkverkehr auf ein Minimum reduziert.

Moblist

Um die einzelnen Mobs in einer Präsentation adressieren zu können, benötigen sie eindeutige Bezeichner, die sogenannten IDs. Diese werden u. a. als Index für die Speicherung der Objekte in Dateien oder später in einer Datenbank verwendet. Problematisch wird eine derartige Adressierung allerdings dann, wenn ein Mob zwei oder mehr gleiche „Unter-Mobs“ besitzt. Würden deren wahrscheinlich unterschiedliche Aktionen nun ihrem gemeinsamen Schlüssel zugeordnet werden, ergäben sich unerwünschte Interferenzen. Um dieses Problem zu lösen ist es erforderlich, neben der globalen ID (GID) auch eine lokale einzuführen, die nur innerhalb eines Mob gültig ist. So können zwei identische Objekte mit ein und derselben GID zwei unterschiedliche lokale IDs besitzen, über die ihnen Aktionen zugeordnet werden.

Ein Beispiel einer solchen Moblist ist in Tabelle 3.2 aufgezeigt. Sie bezieht sich direkt auf die in Tabelle 3.1 dargestellte Playlist. Hier wird ersichtlich, daß sich hinter den Zielobjekt-IDs *0* und *2* Kopien desselben Mob verbergen.

Mobs, die ein Element direkt kapseln, können keine weiteren untergeordneten Objekte besitzen und haben daher eine leere Moblist.

ID	0	1	2
GID	542	938	542

Tabelle 3.2.: Beispiel einer Moblist

Element

Das Element kann nur dann Teil eines Mob sein, wenn letzteres am Ende der Wurzelhierarchie steht (siehe Abb. 3.13 auf Seite 35).
Es übernimmt nicht nur die Darstellung der Elementdaten (wie z. B. Text- oder Bildinformationen), sondern muß auch auf eventuell eintreffende Aktionen reagieren. So sorgt es im Falle eines Videoelements beispielsweise dafür, daß das Video beim Start-Befehl auch wirklich zu spielen beginnt.

4. Implementierung

4.1. Server

Die Implementierung des Servers[1] teilt sich in die folgenden drei Schwerpunkte:

1. Die **allgemeine Datenserverfunktionalität**. Sie bildet die Grundlage des MedienServers und umfaßt die „normalen“ Merkmale eines generellen Servers, wie z. B. das Verbindungsmanagment.

2. Die für diese Plattform **spezielle Serverfunktionalität**. Sie interpretiert die vom Client stammenden Anfragen und reagiert darauf entsprechend der gewünschten Funktionalität. Desweiteren sorgt sie für das Transaktionsmanagement, welches z. B. den Zugriff auf die im Dateisystem oder einer Datenbank liegenden Daten verwaltet.

3. Der **XML-Zugriff**. Er dient dem Zugriff auf die in der Extensible Markup Language vorliegenden Objektbeschreibungen.

[1] entsprechend seiner Funktionalität nachfolgend auch als *MedienServer* bezeichnet

4. Die **Subserverfunktionalität**. Sie ermöglicht es dem MedienServer, andere Serverprozesse für gestreamte Medien zu instanziieren und zu verwalten.

4.1.1. Allgemeine Datenserverfunktionalität

Da es sich bei der Implementierung der generellen Serverfunktionalität um eine in vielerlei Literatur besprochene und aufgezeigte Programmierarbeit handelt, habe ich als Grundlage den in [21] ausführlich erklärten Code eines „generic server“ verwendet. Der dort dargelegte Server kann durch relativ wenig Aufwand um sogenannte Services erweitert werden. Dazu muß der zu integrierende Service das Interface `Service` (Listing 4.1) implementieren.

Listing 4.1: Service - Interface

```
public interface Service {
    public void serve(InputStream in, OutputStream out)
        throws IOException;
}
```

Diese Schnittstelle ermöglicht die Kommunikation zwischen Server und Service.
Um nun in den zugrunde liegenden Server einen neuen Service einzubinden, muß dieser dem Server bekannt gemacht werden. Mit Hilfe der `addService()` Methode des Servers läßt sich das bewerkstelligen. Wie in der Methodendeklaration (4.2) ersichtlich ist, werden ihr dabei der Klassenname des Services und der Port, an dem der Server auf eine Verbindung wartet, übergeben.

Listing 4.2: Deklaration der addService() Methode

```
public void addService(Service service, int port) throws IOException
```

Erhält der Server eine Verbindungsanfrage, baut er diese auf, instanziiert den entsprechenden Service und übergibt diesem zwei Datenströme, über welche die Kommunikation mit dem der entsprechenden Verbindung zugehörigen Client erfolgt.
Der verwendete Server bietet neben der angesprochenen `addService()` Methode noch weitere Möglichkeiten. So können Services wieder entfernt, die maximale Anzahl an erlaubten Verbindungen festgelegt oder auch eine Liste aller aktiven Verbindungen ausgegeben werden.

4.1.2. Spezielle Serverfunktionalität

Die für diese Plattform benötigte spezielle Serverfunktionalität habe ich als Service realisiert. Es handelt sich dabei um eine Java-Klasse, die das `Service` Interface implementiert und mit seiner Hilfe an den zugrunde liegenden Server gekoppelt werden kann.

Befehl	Zielobjekt	Parameter

Tabelle 4.1.: Syntax der vom Client geschickten Anfragen

Auswertung der Clientanfragen

Der Client formuliert seine Anfragen nach den von ihm benötigten Objekten als normale Textstrings, wobei diese die in Tabelle 4.1 dargestellte Form haben.

Dabei ist derzeit neben einem Befehl zum Beenden der Verbindung nur der Befehl `get` implementiert. Er fordert den Server auf, ein bestimmtes Objekt an den Client zu senden. Welches Objekt dabei addressiert wird, steht im Feld *Zielobjekt*. Die eventuell für den Befehl benötigten Parameter sind im Feld *Parameter* abgelegt. Zukünftig sind weitere Befehle geplant, die z. B. für das Interaktionsmanagement oder das Ablegen von Objektdaten mit Hilfe eines Authoringtools benötigt werden.

Der Programmcode, der die Anfragen des Clients auswertet, ist vereinfacht in Listing 4.3 dargestellt.

Listing 4.3: die Befehlsverarbeitung

```
1  String line;          // String, der vom Client geschickt wird
2  String cmd;           // extrahiertes Kommando
3  StringBuffer par = new StringBuffer();  // die restlichen Parameter
4
5  // Anpassung der streams
6  BufferedReader in = new BufferedReader(new InputStreamReader(_i));
7  ObjectOutputStream out = new ObjectOutputStream(_o);
8
9  out.reset();
10 while (doIt) {
11     // Kommandostring vom Client ermitteln
12     line = in.readLine();
13     if ((line == null) || line.equals(".")) {
14         // Verbindung wird beendet
15         doIt = false;
16         continue;
17     }
18
19     // Kommandostring aufteilen
20     StringTokenizer token = new StringTokenizer (line);
21     if (token.hasMoreTokens())
22         cmd = token.nextToken();
23     else
24         // leerer String -> nächsten einlesen
25         continue;
26
27     // die Kommandoparameter ermitteln
28     par.setLength(0);
29     while (token.hasMoreTokens()) {
```

```
30            par.append(token.nextToken());
31            par.append(' ');
32        }
33    }
```

Die durch das Interface definierten Ein- und Ausgabeströme werden entsprechend ihrer Verwendung durch Filterklassen angepaßt(Zeilen 6 und 7). Dabei wird der vom Client kommende Strom in einen gepufferten Zeichenstrom gewandelt. Da es sich bei den zum Client zu sendenden Daten um ganze Java-Objekte handelt, wird der Ausgabestrom in einen `ObjectOutputStream` gewandelt. Danach werden in einer while-Schleife sämtliche Clientanfragen abgearbeitet: Zuerst wird die gesamte Anfrage eingelesen und darauf geprüft, ob sie überhaupt Zeichen enthält (Z. 12 - 17). Dann wird der ermittelte String mit Hilfe eines `StringTokenizer`-Objektes in seine Einzelteile zerlegt. Dabei wird in den Zeilen 20 - 25 das eigentliche Kommando extrahiert. Der verbleibende String wird letztendlich in seine einzelnen Worte geteilt (Z. 28 - 32), die in dem Stringarray `par` abgelegt werden.

Ausführen der Clientanfragen

Um die vom Client geschickten Befehle ausführen zu können, habe ich mich der mit Java 1.1 eingeführten Reflection-Klassen bedient. Mit deren Hilfe können zur Laufzeit des Programms Informationen über Klassen oder Methoden abgefragt werden. Dadurch wird es möglich, den Server eine Methode mit dem Namen des Befehls aufrufen zu lassen. Erfolg dieser Bemühungen ist die leichte Implementierung von neuen Befehlen, ohne Änderungen am eigentlichen Server vornehmen zu müssen. Es muß einfach eine Methode geschrieben werden, deren Name identisch mit dem Befehlsnamen ist.
Der für die Reflektion erforderliche Programmabschnitt ist in Listing 4.4 dargestellt. Um die angedachte Funktionalität umzusetzen, sind folgende Programmschritte erforderlich:

Listing 4.4: die Befehlsverarbeitung

```
1  // fuer command-reflection benoetigte parameter
2  Class c, parameters[];
3  Object[] arguments;
4  Method m;
5  Object result;
6
7  // Klasse mit den Befehls-Methoden instanziieren
8  ServeMethods sMethods = new ServeMethods();
9
10 // Referenz auf die Befehls-Methoden Klasse ermitteln
11 c = sMethods.getClass();
12
13 // Parameterbeschreibung generieren
14 parameters = (par.length() == 0)
```

```
15              ? new Class[0] : new Class[] {String.class};
16 try {
17     // die zum Befehl passende Methode ermitteln
18     m = c.getMethod (cmd, parameters);
19 } catch (NoSuchMethodException e) {
20     // Befehl unbekannt
21     continue;
22 }
23
24 // Parameterliste generieren
25 arguments = (par.length() == 0)
26              ? new Object[0] : new Object[] {par.toString()};
27
28 try {
29     // Methode ausführen
30     result = m.invoke(sMethods, arguments);
31 } catch (InvocationTargetException e) {
32     // Fehler beim Ausführen der Befehlsmethode
33     System.err.println(e.getTargetException().toString());
34     e.printStackTrace (System.err);
35     continue;
36 }
```

1. Instanziieren der Klasse, die die zu den Befehlen gehörigen Methoden enthält (Z. 8). Hierbei habe ich deswegen eine Extraklasse verwendet, um ein Aufrufen von unerlaubten Methoden zu verhindern und damit die Sicherheit zu erhöhen.
2. Eine Referenz auf diese Klasse ermitteln (Z. 11).
3. Die der auszuführenden Methode entsprechende Parameterbeschreibung generieren (Z. 14).
4. Eine Referenz auf die gewünschte Methode ermitteln (Z. 15 - 21). Wenn dabei eine `NoSuchMethodException` erzeugt wird, gibt es keine dem Befehl entsprechende Methode.
5. Die Parameterliste für die Methode generieren (Z. 24).
6. Ausführen der Methode. Wenn dabei ein Fehler (`InvocationTargetException`) auftritt, wird ein Stacktrace ausgegeben, um so den Fehler eingrenzen zu können. Das von der Methode zurückgegebene Ergebnis wird in der Variablen `result` aufgenommen.

Die get()-Methode

Die `get()`-Methode (siehe Listing 4.5 auf Seite 44) ist für das Ausliefern der Objektdaten verantwortlich. Sie bildet die Schnittstelle zu den möglicherweise unterschiedlichen Arten der Datenspeicherung. Momentan werden die Objekte in XML-strukturierten Files abgelegt. Daher instanziiert die Methode in Zeile 6 die dafür

Abkürzung	Bedeutung
pl	Playlist
ml	Mobliste
mi	Mobinformation, wie z. B. die geometrischen Ausmaße des Mob.
element	Elementdaten, wie z. B. Text- oder Bildinformationen

Tabelle 4.2.: Abfragbare Objektelemente

nötige `XMLaccess` Klasse. Um die zugrunde liegende Speicherschicht zu abstrahieren, wäre es hier noch notwendig, die Anbindung mittels eines Interfaces zu gestalten.
Nachdem in den Zeilen 9 - 16 der Parameterstring in seine Worte zerlegt wurde, ruft die Methode ab Zeile 25 die unterschiedlichen Methoden der XML-Klasse auf, die wiederum das entsprechende Objektelement zurückliefern. Dieses wird in Zeile 34 an den aufrufenden Server zurückgegeben.
Als Beispiel wurde im Listing nur die Abfrage des `mi`-Objektelements aufgeführt. Die Bedeutung dessen und die weiteren möglichen Elemente sind in Tabelle 4.2 zusammengefaßt.

4.1.3. Zugriff auf die XML-strukturierten Objektbeschreibungen

Bevor ich den Zugriff auf die in XML-Notation vorliegenden Objektbeschreibungen erläutere, möchte ich zum besseren Verständnis ein Beispiel für eine derartige Beschreibungsdatei darlegen.

XML-Beispielbeschreibung

In Listing 4.6 auf Seite 45 ist die Struktur eines Mobs dargestellt, welches zwei weitere Medienobjekte steuert.
Die Bedeutung aller im Rahmen dieser Arbeit verwendeten Tags und deren möglichen Attribute sind in Abbildung 4.1 auf Seite 45 aufgezeigt.

XMLaccess-Klasse

Die `XMLaccess`-Klasse stellt die für den Zugriff auf die gespeicherten Objekte nötige Funktionalität bereit. Sie liest die Dateien mit den Objektbeschreibungen ein, interpretiert die XML-Struktur und gibt die ermittelten Daten an die aufrufende Klasse zurück.
Zur Veranschaulichung des ganzen Vorganges ist in Listing 4.7 auf Seite 46 ein vereinfachter Auszug der Klasse dargestellt. Das aufgezeigte Code-Stück ist für das Auslesen der Mobinformation zuständig, welche u. a. die Größenangaben des Mob enthält.
Als erstes wird das entsprechende Beschreibungsfile geöffnet (Z. 10) und dem Parser übergeben (Z. 11). Anschließend werden die beiden Attribute `XDIM` und `YDIM` des

Listing 4.5: die Befehlsverarbeitung

```
1 public Object get (String _what) {
2     int id;
3     Object back = null;
4 
5     // Klasse für den XML-Zugriff instanziieren.
6     XMLaccess xml = new XMLaccess();
7 
8     // Zielobjekt-ID aus Parameterstring ermitteln
9     StringTokenizer token = new StringTokenizer (_what);
10    try {
11        id = java.lang.Integer.parseInt (token.nextToken());
12    }
13    catch (NumberFormatException e){
14        // erstes Argument ist kein Integer -> Rücksprung
15        return null;
16    }
17 
18    String w;
19    // das gewünschte Objektelement ermitteln
20    try {
21        w = token.nextToken();
22    } catch (NoSuchElementException e) {
23        w = new String ("all");
24    }
25    if (w.equals ("mi")) {
26        back = xml.getMI (id);
27    }
28    // restlichen Objektelemente abarbeiten
29    else if (w.equals ...
30        .
31        .
32        .
33    }
34    return back;
35 }
```

Listing 4.6: Beispiel einer XML-strukturierten Objektbeschreibung

```
<MOB ID="102" XDIM="200" YDIM="215" BGCOLOR="#000000">
  <PLIST>
    <PLITEM TIME="-1" ACTION="init"  OID1="0" XPOS="50" YPOS="0"
                                                       SCALE="1"  />
    <PLITEM TIME="-1" ACTION="init"  OID1="1" XPOS="0"  YPOS="185"
                                                       SCALE="0.3"/>
    <PLITEM TIME="0"  ACTION="start" OID1="0"/>
    <PLITEM TIME="0"  ACTION="show"  OID1="0"/>
    <PLITEM TIME="12" ACTION="start" OID1="1"/>
    <PLITEM TIME="12" ACTION="show"  OID1="1"/>
  </PLIST>
  <MLIST>
    <MLITEM ID="0" GID="116"/>
    <MLITEM ID="1" GID="117"/>
  </MLIST>
</MOB>
```

Abbildung 4.1.: Die möglichen XML-Tags und deren Bedeutung

Listing 4.7: Die Klasse XMLaccess

```
1 NonValidatingDOMParser parser = new NonValidatingDOMParser();
2
3 public MobInfo getMI (int _id) {
4     Object data = null;
5     MobInfo mInfo = null;
6
7     String uri = new String(new Integer(_id).toString()+".mob");
8
9     try {
10        FileReader f = new FileReader ("data/"+uri);
11        parser.parse (new InputSource(f));
12        Document d = parser.getDocument();
13        NodeList nodelist = d.getElementsByTagName("MOB");
14        Node node = nodelist.item(0);
15        NamedNodeMap attribs = node.getAttributes();
16        int xdim = java.lang.Integer.parseInt
17                    (attribs.getNamedItem("XDIM").getNodeValue());
18        int ydim = java.lang.Integer.parseInt
19                    (attribs.getNamedItem("YDIM").getNodeValue());
20        mInfo = new MobInfo();
21        mInfo.xDim = xdim;
22        mInfo.yDim = ydim;
23        f.close();
24    }
25    catch (Exception e) {
26        e.printStackTrace(System.err);
27    }
28    return mInfo;
29 }
```

Mob-Elements ausgelesen und in der Mobinformationsvariable `mInfo` abgelegt. Diese wird letztendlich der aufrufenden Methode zurückgeliefert.

4.1.4. Verwaltung der Subserver

Die Notwendigkeit Subserver verwalten zu können, ist bereits in 3.3.3 erläutert worden. Um diese Fähigkeit zu erlangen, wurden folgende Schritte vollführt:

1. Im Element-Objekt, welches gestreamt werden soll, wird noch in der `XMLaccess`-Klasse ein entsprechendes Flag gesetzt und ihm ein `ServerInfo`-Objekt zugewiesen. Dieses enthält zu diesem Zeitpunkt nur die Informationen darüber, wo die Elementdaten zu finden sind.

2. Der MediaServer prüft, ob bei dem zurückgelieferten Objekt das Flag gesetzt ist und ruft in dem Fall die Methode `StartSubserver` auf.

3. Die `StartSubserver`-Methode ermittelt dann, welcher Subserver zu starten ist, ermittelt die benötigten freien Portnummern, speichert diese in das weiter oben erzeugte `ServerInfo`-Objekt und instanziiert damit letztendlich den Subserver. Letzterer übernimmt von nun an die Kommunikation mit dem auf Clientseite vorhandenen Playerelement.

4.2. Client

Als erstes möchte ich die Implementierung der wichtigsten Klassen beschreiben, die auf der Clientseite zum Einsatz kommen. Anschließend werde ich den Programmablauf in seiner Gesamtheit erläutern.

4.2.1. Timer

Der Timer ist für das zeitgerechte Anstoßen der durch die unterschiedlichen Playlisten beschriebenen Aktionen zuständig. Dabei bildet er die Grundlage für die zeitliche Synchronisation und muß daher so zuverlässig wie möglich arbeiten. Um das zu erreichen, habe ich ihn als extra Thread mit erhöhter Priorität implementiert. Der für die zeitliche Steuerung relevante Teil des Sourcecodes ist in Listing 4.8 auf der nächsten Seite ersichtlich.

In Zeile 8 wird beim ersten Durchlauf des Timers die Startzeit ausgelesen, die praktisch den Nullpunkt der gesamten Präsentation markiert. Danach wird in einer Endlosschleife die effektive, auf die Startzeit bezogene Ablaufzeit ermittelt (Z. 13 u. 15). Anschließend wird geprüft, ob ein aktuelles oder früheres (bisher nicht verarbeitetes) Ereignis ansteht (Z. 17-20). Wenn dem so ist, wird die Methode `runMobs()` aufgerufen, welche die Aktionen anstößt (siehe nachfolgenden Absatz). Da die Ausführung des Ereignisses eine nicht berechenbare Zeit in Anspruch nimmt, muß anschließend

Listing 4.8: run()-Methode der Klasse `Timer.java`

```
1  private final int granularity = 1000;    // Auflösung des Timers
2  private int relTime = -1;                // die relative Zeit
3  private int zeroTime = -1;               // absolute Startzeit des Timers
4  private int nextEvent = -1;              // Zeit des nächsten Ereignisses
5
6  public void run() {
7      // absolute Startzeit bestimmen
8      zeroTime = (int)System.currentTimeMillis();
9      long sleepTime;
10     long startTime;
11     while (true) {
12         // Zeitpunkt des Durchlaufs bestimmen
13         startTime = System.currentTimeMillis();
14         // effektive Zeit berechnen
15         relTime = ((int) startTime - zeroTime) / granularity;
16         // steht ein Ereignis an oder wurde gar übersprungen?
17         if (relTime >= nextEvent) {
18             // Ereignis ausführen
19             runMobs();
20         }
21         // Pausenzeit berechnen
22         sleepTime = granularity - (System.currentTimeMillis()
23                                                     - startTime);
24         // darf überhaupt pausiert werden?
25         if (sleepTime <= 0) {
26             continue;
27         }
28         // Thread "schlafen" legen
29         try { Thread.sleep(sleepTime); }
30         catch (InterruptedException e) { }
31     }
32 }
```

berechnet werden (Z. 22), wie lange der Timer-Thread pausieren soll, damit der durch den Variablenwert von `granularity` vorgegebene Takt eingehalten werden kann. Für die berechnete Zeitspanne wird dann der Thread schlafen gelegt (Zeilen 28 u. 29) um anschließend den nächsten Takt auszuführen.

Damit der Timer weiß, wann ein Ereignis ansteht, muß jedes Medienobjekt bei seiner Anmeldung den Zeitpunkt seines nächsten Events angeben. Der Timer verwaltet die bei ihm registrierten Mobs und die dazugehörigen Aktionszeiten in einem Vector und speichert zusätzlich den von allen Medienobjekten als nächstes auftretenden Zeitpunkt in der Variablen `nextEvent`. Wird beim Vergleich in Zeile 17 festgestellt, daß ein Ereignis ansteht, ermittelt der Timer in der Methode `runMobs()`, welches der registrierten Mobs eine Aktion auszuführen hat und ruft dessen `update()` Methode auf (siehe 4.2.5 auf Seite 55). Diese liefert nach der Ausführung die Zeit des folgenden Ereignisses zurück, so daß der Timer dem entsprechenden Mob einen neuen Aktionszeitpunkt zuordnen kann.
Ein gravierender Nachteil dieser Vorgehensweise ist die Gefahr des Blockierens, das durch eine zu zeitintensive `update()` Methode eines Mob auftreten könnte. Die einzige Möglichkeit, dieses Problem zu lösen, wäre die Verwendung weiterer Threads. Würde man dem Ausführen der Methode einen eigenen Thread zuteilen, könnte der Timer unabhängig von dessen Zustand weiterlaufen. Bei einer umfangreichen Präsentation ist es zu bestimmten Zeiten allerdings möglich, daß mehrere Objekte zeitgleich Ereignisse angemeldet haben. Die dabei u. U. auftretende Zahl an Threads kann aber in heutigen Browsern zu Instabilitäten führen, so daß ich diese Art des Vorgehens nicht angewendet habe.

4.2.2. Datenquelle

Wie in 3.4.1 auf Seite 35 dargelegt, ist die Datenquelle für Transfer und Organisation der Daten zuständig. Im Rahmen dieser Arbeit habe ich zunächst die Klasse `MobDataSourceTCP` implementiert, die ihre Daten über das Netz vom Server anfordert. Dazu muß ihr bei ihrer Instanziierung die Serveradresse und der entsprechende Port übergeben werden. Nach dem Aufbau der Netzwerkverbindung können ihr von den Medienobjekten Datenanfragen gestellt werden, die sie dann an den Server weiterleitet und die zurück erhaltenen Objekte den Mobs zur Verfühgung stellt. Bei der Umsetzung dieser Verfahrensweise habe ich ihren Aufbau so gestaltet, daß bei einer späteren Abstraktion durch ein Interface ein Queue-Mechanismus integriert werden kann. Durch diesen könnten die Medienobjekte ihre Anfragen asynchron an die Datenquelle stellen und bis zum Eintreffen der Daten weitere Programmschritte durchführen, so daß sie nicht für die Zeit der Datenermittlung blockiert werden.

Listing 4.9: `addElement()`-Methode der Klasse `Playlist.java`

```
1  public void addElement (Command _cmd) {
2      _cmd.id = this.maxCmdID++;
3      // erstes Kommando
4      if (this.cmdList.isEmpty()) {
5          this.cmdList.addElement (_cmd);
6      }
7      // weiteren Kommandos sortiert hinzufuegen
8      else {
9          for (int i = 0; i < this.cmdList.size(); i++) {
10             if (((Command)this.cmdList.elementAt(i)).time > _cmd.time) {
11                 this.cmdList.insertElementAt (_cmd, i);
12                 return;
13             }
14         }
15         this.cmdList.addElement (_cmd);
16     }
17 }
```

4.2.3. Playlist

Die Playlist speichert die einzelnen Aktionen mit den zugehörigen Zeiten und Objekten in einem Vector und beinhaltet außerdem die Funktionalität für den Zugriff auf diesen Vector. Die dafür implementierten Methoden erläutere ich nachfolgend im einzelnen.

`addElement()`

Um der Playlist eine neue Aktion zuzuweisen, muß die Methode `addElement()` (Listing 4.9) verwendet werden. Sie sorgt dafür, daß die in der Playlist gespeicherten Aktionen aufsteigend geordnet nach ihren Ausführungszeiten gespeichert werden.
In Zeile 4 wird dafür als erstes getestet, ob überhaupt schon Aktionen bekannt sind. Bei einer noch leeren Playlist wird der entsprechende Befehl an die erste Stelle des Vectors gelegt. Wenn bereits Aktionen vorgemerkt wurden, wird der neue Befehl in korrekter Zeitreihenfolge im Vector gespeichert (Zeilen 9-15).

`getNextEvent()`

Die Methode `getNextEvent()` ermittelt aus den in der Playlist gespeicherten Aktionen diejenige, die als nächstes zur Ausführung gelangen soll. Dazu werden (wie in Listing 4.10 auf der nächsten Seite erkennbar) die einzelnen Aktionen in einer Schleife durchlaufen und als erstes darauf geprüft, ob sie bereits ausgeführt wurden (Zeilen 6-8). Wenn eine noch nicht abgearbeitete Aktion vorhanden ist, wird deren Startzeit an die aufrufende Methode zurückgeliefert. Um die in 3.2.5 auf Seite 25 erläuterte

Listing 4.10: `getNextEvent()`-Methode der Klasse `Playlist.java`

```
1 public int getNextEvent (int _d) {
2     int nextEvent = Integer.MAX_VALUE;
3     Command c;
4     for (int i=0; i < this.cmdList.size(); i++) {
5         c = (Command) this.cmdList.elementAt(i);
6         if (c.done) {
7             continue;
8         }
9         nextEvent = c.time + _d;
10        break;
11    }
12    return nextEvent;
13 }
```

Verschiebung der Startzeiten zu ermöglichen, wird die der `getNextEvent()`-Methode übergebene Verzögerungszeit zur Startzeit der Aktion hinzu addiert (Z. 9).

`getEventsUpTo()`

Die Methode `getEventsUpTo()` (Listing 4.11 auf der nächsten Seite) liefert eine Liste aller Aktionen zurück, die bis zu einem bestimmten Zeitpunkt auszuführen sind. Diese Funktionalität wird z. B. im Hinblick auf eine später zu implementierende Möglichkeit des zeitlichen Sprunges in einer Präsentation benötigt. Momentan findet sie jedoch nur während der Initialisierung der einzelnen Mobs Verwendung, um alle Aktionen zu ermitteln, die vor dem eigentlichen Präsentationsstart erfolgen sollen. Dafür werden diese Aktionen mit der Startzeit `-1` versehen, so daß sie während der Initialisierungsphase mit dem Aufruf `getEventsUpTo(-1)` ermittelt werden können. Um die bis zum Zeitpunkt `_t` anstehenden Events herauszufiltern, wird eine Kopie des Aktionsvectors erzeugt (Z. 5), aus der anschließend alle Aktionen entfernt werden, die bereits abgearbeitet wurden (Z.8-13) oder deren Startzeiten nicht mehr in den Betrachtungszeitraum fallen (Z.15-19). Die verbleibenden Vectorelemente werden dann in ein Array kopiert und an die aufrufende Methode zurückgeliefert.

4.2.4. Element

`Implementable`-Interface

Wie in 3.2.3 auf Seite 22 bereits dargelegt, müssen die unterschiedlichen Medientypen bzw. deren korrespondierende Elemente über eine definierte Schnittstelle an die Plattform gebunden werden. Das dafür notwendige Interface `Elementable` gibt die notwendigen Methodendefinitionen vor (siehe Listing 4.12 auf Seite 53), deren Bedeutung ich nachfolgend kurz erläutere:

Listing 4.11: `getEventsUpTo()`-Methode der Klasse `Playlist.java`

```
1  public Object[] getEventsUpTo (int _t) {
2      int i;
3      Command c;
4      MyVector tmp = null;
5      tmp = (MyVector)this.cmdList.clone();
6      for (i = 0; i < tmp.size();) {
7          c = (Command)tmp.elementAt(i);
8          if (c.done) {
9              tmp.removeElementAt (i);
10             // da das Element entfernt wurde, rutschen nachfolgende
11             // nach vorne  => i nicht erhoehen
12             continue;
13         }
14         // liegt event erst spaeter an?
15         if (c.time > _t) {
16             // alle events der übrigen Kommandos kommen spaeter
17             tmp.setSize (i);
18             break;
19         }
20         i++;
21     }
22     return tmp.toArray();
23 }
```

Listing 4.12: `Elementable`-Interface

```
1 public interface Elementable {
2     // Methoden für den Zugriff auf den Elementtyp
3     public void setType (String _t);
4     public String getType ();
5
6     // Methoden für die Initialisierung des Elements
7     public void setData (Object _d);
8     public void setPos (Point _p);
9     public void setRelScale (float _s);
10
11    // die den einzelnen Befehlen entsprechenden Methoden
12    public void doInit();
13    public void doStart();
14    public void doStop();
15    public void doShow();
16    public void doHide();
17    public void doCustom(String _par);
18
19    // Methoden für den Zugriff auf die Elementparameter
20    public void setParams (Vector _v);
21    public void setParam (ElementParam _ep);
22    public String[] getParam (String _name);
23    public boolean hasParam (String _name);
24
25    // Methoden für den Zugriff auf das streamable-Flag
26    public void setStreammedia (boolean _f);
27    public boolean isStreammedia ();
28 }
```

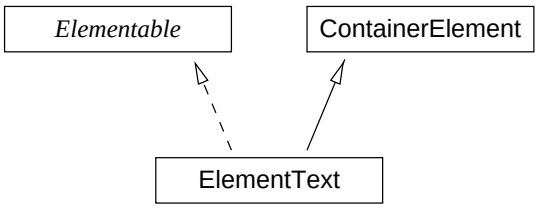


Abbildung 4.2.: Klassendiagramm der Klasse `ElementText`

Zeilen 3-4 Mit diesen Methoden kann der Elementtyp gesetzt und abgefragt werden. Er sollte die durch RFC 1341[22] definierte MIME-Type Form haben.

Zeilen 7-9 Um während der Initialisierungsphase auch die Elemente mit den nötigen Werten versorgen zu können, existieren diese drei Methoden. Sie weisen dem Element seine Daten (z. B. Bilddaten), die Position und den Skalierungsfaktor zu und werden vom übergeordneten Mob aufgerufen.

Zeilen 12-17 Um dem Element die eintretenden Aktionen übergeben zu können, werden diese Methoden benötigt. Sie werden direkt den entsprechenden Befehlen zugeordnet.

Zeilen 20-23 Mit den Methoden wird es möglich, den Elementen Parameter zu übergeben. So werden alle name/value-Paare des `<PARAMETER>`-Tags aus der serverseitigen Objektbeschreibung mit ihrer Hilfe an das Element weitergereicht.

Zeilen 26 u. 27 Diese beiden Methoden dienen dem Zugriff auf das Flag, welches anzeigt, ob das Element gestreamte Medien verarbeitet.

Implementierung eines Elements

Die Implementierung eines Elements möchte ich am Beispiel des Textelements, welches die Textdaten einfach auf dem Bildschirm darstellt, aufzeigen. Sein Klassendiagramm ist in Abbildung 4.2 ersichtlich.

Um die logische Strukturierung der Medienobjekte auch geometrisch am Bildschirm zu erreichen, werden allen Mobs und Elementen ein eigener `Container` zugeordnet. Diese vom AWT bereitgestellte Klasse ermöglicht es, die einzelnen Objekte in Abhängigkeit von ihrem strukturellen Zusammenhang auf dem Bildschirm zu positionieren. So werden Position und Größe der Medienobjekte von ihren übergeordneten Mobs während der Initialisierungsphase gesetzt, so daß jedem Mob nur der Teil des Bildschirms zur Verfügung steht, der ihm seiner hierarchischen Position gemäß zugeteilt wurde. Auch die Elementdaten - in diesem Beispiel die Textdaten - werden dem Element in dieser Phase übergeben.
Die Verwendung eines eigenen Containers bietet auch einen weiteren Vorteil: Bei Änderungen an den das Element betreffenden Bildschirmteilen sorgt das AWT durch

Listing 4.13: Beispiel einer `doInit()`-Methode eines TextElements

```
1  public void doInit () {
2      super.doInit();
3
4      // Ermitteln des Parameters für die Textgröße
5      this.size = 10; // Standardgröße
6      String[] v = getParam("size");
7      if (v != null) {
8          try {
9              // Die als Parameter übergebene Größe ermitteln
10             this.size = java.lang.Integer.parseInt (v[0]);
11         } catch (Exception e) {;}
12     }
13     // Die Größe entsprechend des Skalierungsfaktors anpassen
14     this.size = (int)(this.size * this.scale);
15     Font oF = this.getFont();
16     // neue Größe setzen
17     this.font = new Font(oF.getName(), oF.getStyle(), this.size);
18     this.setFont (this.font);
19
20     // weitere Parameter ermitteln (wie z.B. die Farbe des Textes)
21      .
22      .
23      .
24 }
```

Aufruf der `update()` oder `paint()`-Methoden des Containers für eine korrekte Darstellung der Elemente. Werden diese Methoden durch geeigneten Alternativen ersetzt, muß sich die Plattform nicht um z. B. das Neuzeichnen der Elemente kümmern. Dieser Fall tritt beispielsweise ein, nachdem das Browserfenster verdeckt und wieder sichtbar gemacht wurde.

Nachdem alle Mobs und deren Elemente von der Plattform derartig initialisiert wurden, wird die Präsentation gestartet und es werden die einzelnen Befehle der Playlist abgearbeitet. Liegt nun eine dem Element zugeordnete Aktion an, wird deren Methode aufgerufen. Am Beispiel des *INIT* Befehles (Listing 4.13) wird die Abarbeitung dieser Methoden ersichtlich.

4.2.5. Medienobjekt

Die Klasse `Mob` ist die für den Client elementare Klasse. Sie repräsentiert ein Mob mit all seinen Eigenschaften und Methoden. Das Klassendiagramm ist in Abbildung 4.3 auf der nächsten Seite dargestellt.

Ein Objekt der Klasse `Mob` beinhaltet Objekte aller Klassen, die für den Ablauf der ihm untergeordneten Mobs notwendig sind. Dazu zählt die Playliste, ein Timer, eine Datenquelle und eine entsprechende Mobliste. Letztere ist in Form eines Hashes

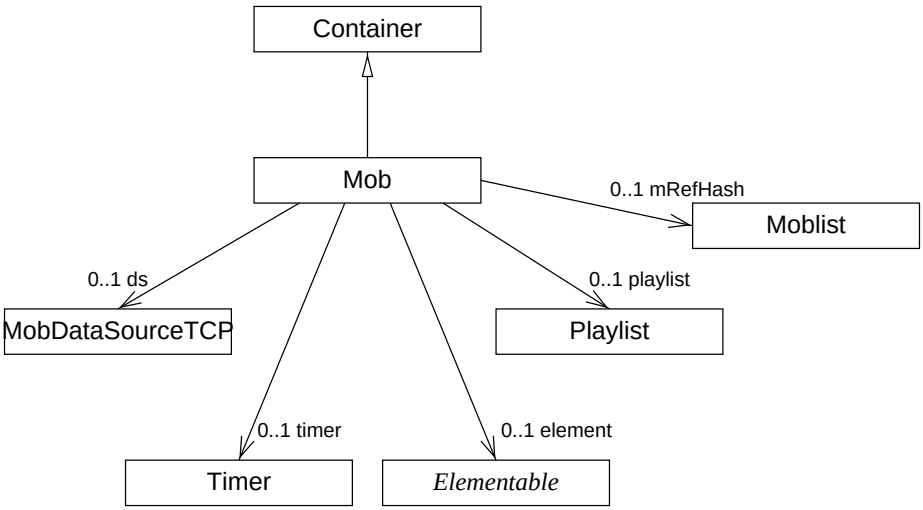


Abbildung 4.3.: Klassendiagramm der Klasse `Mob`

realisiert worden, dessen Schlüssel die lokale ID bildet.
Die Klasse `Mob` ist selbst, wie auch schon die Elementklassen, von `Container` abgeleitet, auch wenn das Mob ansich keine Daten am Bildschirm darstellt. Dieses Vorgehen ist deshalb sinnvoll, weil so die Medienobekte gemäß ihrer logischen Zusammenhänge strukturiert werden können. Dazu kann man einem Mob einfach weitere mittels der `add()`-Methode (die von `Container` geerbt wird) zuweisen. Diese liegen dann innerhalb seines eigenen Darstellungsbereichs. Somit überlasse ich dem AWT einen Großteil der Verwaltungsaufgaben, die die clientseitige Datenstruktur (vgl. 3.4.1 auf Seite 34) aufwirft.

Generation der Datenstruktur

Nachdem alle wichtigen Klassen erklärt sind, möchte ich mich nun der Initialisierungsphase des Mob und der damit verbundenen Generierung der Datenstruktur zuwenden.

Möchte man ein Objekt der Klasse `Mob` initialisieren, muß man ihm neben der GID des zu kapselnden Mobs als erstes eine Datenquelle und einen Timer zuweisen. Anschließend legt man seinen sichtbaren Bereich mit der `setBounds()`-Methode fest, die von der Klasse `Container` geerbt wird. Diese Schritte versorgen das Objekt an sich mit allen nötigen Parametern und Informationen. Allerdings enthält es sehr wahrscheinlich weitere Unterobjekte oder -elemente, die ebenfalls initialisiert werden müssen. Hierfür bietet die Klasse `Mob` die Methode `init()` (siehe Listing 4.14 auf der nächsten Seite) an. Wird sie aufgerufen, initialisiert das Medienobjekt alle ihm untergeordneten Objekte oder Elemente. Dabei ergibt sich folgender Ablauf, der rekursiv bis in die „Wurzelspitzen“ der Hierarchie (vgl. 3.13 auf Seite 35) durchgeführt wird.

Listing 4.14: `init()`-Methode der Klasse `Mob`

```
1  public void init () {
2      // Mobliste anfordern
3      this.mRefHash = this.ds.getMobHash(this.ID);
4      if (this.mRefHash.size() == 0) {
5          // keine untergeordneten Mobs vorhanden
6          return;
7      }
8      // enthält das Mob ein Element?
9      if (this.mRefHash.get(new Integer(-1)) != null) {
10         this.isElement = true;  // Elementflag setzen
11     }
12     // Playlist anfordern
13     this.pList = this.ds.getPlaylist(this.ID);
14     // wenn aktuelles Mob kein Element enthält, untergeordnete
15     // Mobs initialisieren
16     if (!this.isElement) {
17         MobReference mRef;
18         // alle untergeordneten Mobs durchlaufen
19         for (int i = 0; i < this.mRefHash.size(); i++) {
20             // Referenz auf untergeordnetes Mob ermitteln
21             mRef = (MobReference)this.mRefHash.get (new Integer(i));
22             int mID = mRef.gid;
23             // neues Mob-Objekt erzeugen und dessen Referenz
24             // abspeichern
25             Mob mob = new Mob (mID);
26             mRef.mob = mob;
27             // dem Mob eine Datenquelle zuweisen
28             mob.setDS (this.ds);
29             // dem Mob einen Timer zuweisen
30             mob.setTimer (this.timer);
31             // Größenparameter des Mob ermitteln und setzen
32             MobInfo mInfo = this.ds.getMobInfo (mID);
33             mob.setAbsDim (mInfo.xDim, mInfo.yDim);
34             // Mob der Hierarchie hinzufügen
35             this.add (mob);
36             // Mob veranlassen, untergeordnete Mobs zu initialisieren
37             mob.init();
38         }
39     }
40     // aktuelles Mob mit der nächsten Ereigniszeit beim Timer anmelden
41     this.timer.regMob(this, Integer.MAX_VALUE);
42 }
```

Z. 3-7 Moblist von der Datenquelle anfordern.

Z. 9-11 Prüfen, ob das aktuelle Mob ein Element enthält.

Z. 13 Playlist von der Datenquelle anfordern.

Z. 16-38 Wenn das aktuelle Mob kein Element enthält, folgende Schritte mit jedem untergeordneten Mob (*uMob*) durchführen.

- **Z. 21** Referenz auf uMob ermitteln.
- **Z. 24** Neues uMob-Objekt instanziieren.
- **Z. 27 u. 29** Dem uMob eine Datenquelle und einen Timer zuweisen.
- **Z. 31 u. 32** Die Größenparameter des uMob setzen.
- **Z. 34** Das uMob dem aktuellen Container hinzufügen.
- **Z. 36** Die `init()`-Methode des uMob aufrufen (hier setzt die Rekursion ein).

Z. 40 Das aktuelle Mob beim Timer mit der nächsten Aktionszeit registrieren

Die eigentliche hierarchische Datenstruktur wird durch den rekursiven Ablauf der Initialisierungsphase generiert. Wenn diese beendet ist, steht eine vollständig initialisierte Baumstruktur (vergleiche Abb. 3.13 auf Seite 35) zur Verfügung.

4.2.6. Gesamter Programmablauf

Nachdem ich die wichtigsten Klassen vorgestellt habe, wende ich mich nun dem gesamten Programmablauf zu.

Der eigentliche Client besteht aus einer Javaklasse, die die bereits erwähnten Klassen nach Bedarf instanziiert und anspricht. Diese „Hauptklasse“ bildet sozusagen den Einstiegspunkt für den Webbrowser, in welchem der Client laufen soll. Da die Vorgehensweise des Browsers beim Starten eines Applets fest vorgegeben ist, muß meine `MobIT`-Klasse dementsprechend implementiert werden. Daher enthält sie die beiden Methoden `init()` und `start()`, die der Browser auch in dieser Reihenfolge aufruft. Dabei spielt die erstere in diesem Applet keine funktionale Rolle, da alle wichtigen Programmschritte innerhalb der `start()`-Methode durchgeführt werden, die ich nachfolgend erläutern werde.

`init()`-Methode

In Listing 4.15 auf der nächsten Seite ist die `init()`-Methode vereinfacht aufgeführt. Der Browser führt nun also diese Methode nach der Abarbeitung von `init()` aus, wobei sich folgender Programmablauf ergibt:

Z. 3 Als erstes wird die ID des Mobs bestimmt, welches das Wurzelelement der gesamten Präsentation darstellt. Dabei ermittelt die Methode `getParentMob()` die als Appletparameter in der umgebenen HTML-Seite eingebettete ID.

Listing 4.15: `init()`-Methode der Klasse `MobIT.java`

```
1  public void start() {
2      // Nummer des anzuzeigenden Mobs ermitteln
3      parentMob = getParentMob();
4      // Datenquelle anlegen
5      serverName = this.getCodeBase().getHost();
6      try {
7          ds = new MobDataSourceTCP(serverName, serverPort);
8      }
9      catch (Exception e) {
10         System.out.println("no connection: " + e.getMessage());
11         this.error = 2;
12         return;
13     }
14     // Neues Mob-Objekt instanziieren und initialisieren
15     mob = new Mob(parentMob);
16     mob.setDS (ds);
17     mob.setTimer (timer);
18     mob.setBounds (0, 0, xd, yd);
19     mob.setLayout (null);
20     // Mob dem aktuellen Container hinzufügen
21     this.add(mob);
22     // Den Datenbaum der untergeordneten Mobs generieren
23     mob.init();
24     // Aktionen bis zum Zeitpunkt 0 ausführen
25     mob.runTo (timer.getTime());
26     // Mob anzeigen
27     mob.setVisible (true);
28     // Timer starten
29     timer.start();
30 }
```

Z. 5-13 Jetzt wird eine neue Datenquelle erzeugt. Dabei wird ihr der Server, von dem das Applet geladen wurde und ein von mir festgelegter Port übergeben. So weiß sie, woher sie ihre Daten beziehen kann.

Z. 15-23 Danach wird das Wurzelmob instanziiert und initialisiert, equivalent zum Initialisierungsvorgang der `Mob`-Klasse.

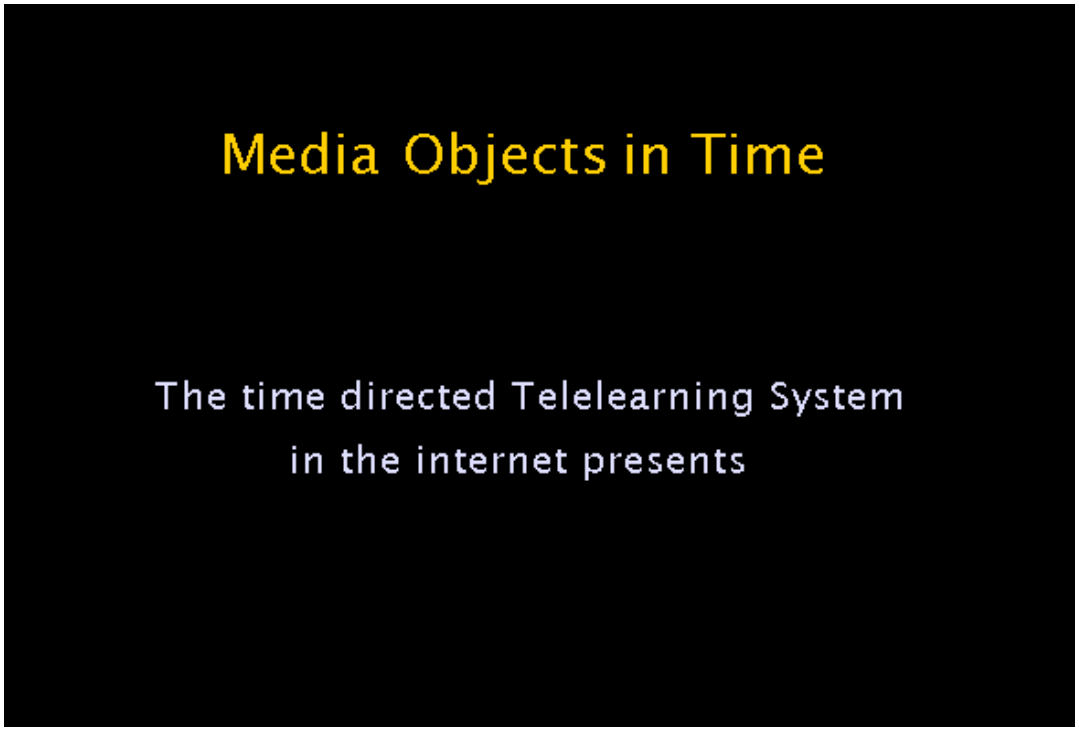
Z. 25 Anschließend werden alle Aktionen ausgeführt, die vor dem eigentlich Start der Präsentation anstehen. Das sind all jene, die die Startzeit $t_{start} = -1$ haben. Verantwortlich ist dafür die Methode `runTo()`, der als Parameter die aktuelle Zeit des Timers (beim Start der Präsentation ist diese -1) übergeben wird.

Z. 27 Als letzte Aktion bei der Initialisierung des Clients wird der Timer gestartet, der von nun an die Kontrolle über den gesamten Präsentationsablauf erhält.

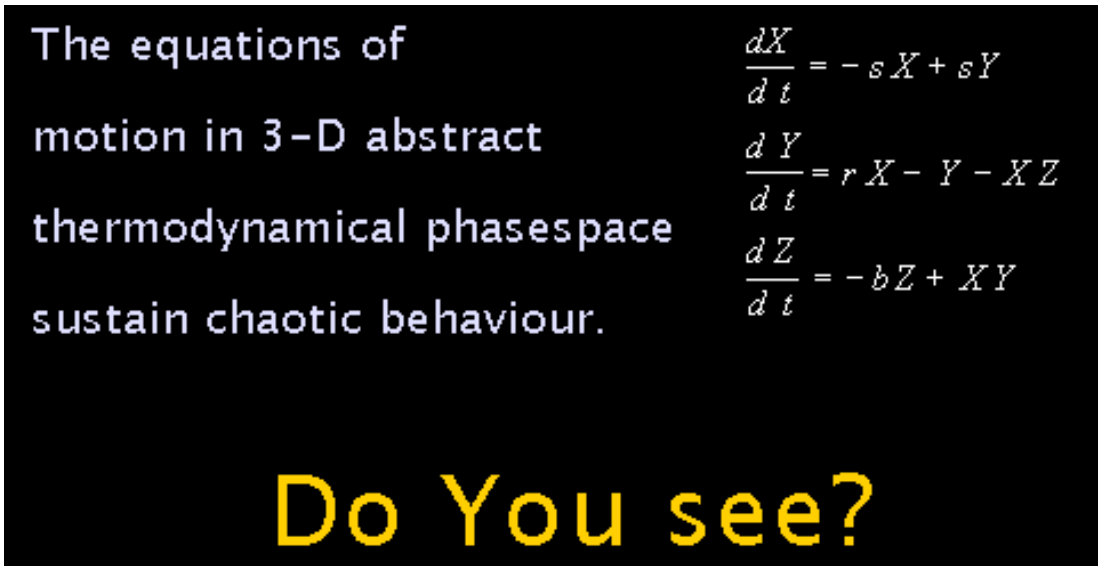
5. Beispielpräsentation

Um die im Rahmen dieser Arbeit entwickelte Plattform konzeptionell zu testen, habe ich eine Beispielpräsentation entwickelt. Diese gibt eine kurze Einführung in die Thematik des sogenannten „Lorenz-Attraktors“. Dabei wurde der gesamte Ablauf aus ca. 150 Einzelobjekten zusammengesetzt, die zeitabhängig dem Betrachter präsentiert werden.

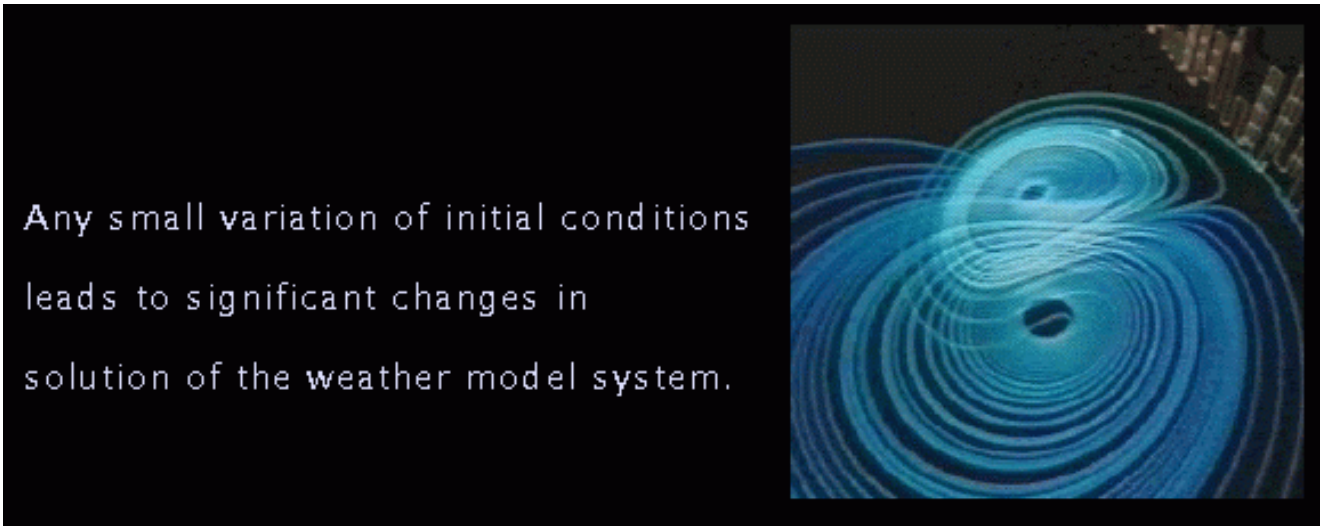
Das nachfolgende Bild zeigt ein Medienobjekt, welches nur Textelemente darstellt. Dabei sind die unteren beiden Textzeilen zu einem weiteren Mob gruppiert, so daß sich eine dreistufige Hierarchie ergibt.



Nachfolgend ist die Integration eines Bildelementes zu sehen. Die übrigen Elemente bilden eine Hierarchie von Textobjekten, die zeitgesteuert dargestellt werden.



Der unten gezeigte Ausschnitt der Präsentation zeigt die Integration eines Videos, das beispielhaft die Fähigkeit der Plattform zeigt, mit gestreamten Medien umgehen zu können.



6. Zusammenfassung

Ziel dieser Diplomarbeit war die Konzeption und Entwicklung eines multimedialen, zeitbasierten Lehr- und Informationssystems.

Um das Potential des Computers im Bereich der Lehre aufzuzeigen, habe ich die unterschiedlichen Medien und ihre Verwendungszusammenhänge analysiert. Dabei wurde deutlich, daß keine der heutzutage eingesetzten Lehrplattformen die bestehenden Möglichkeiten in umfassender Weise nutzt. Anschließend habe ich das Konzept eines innovativen Lehr- und Präsentationssystems dargelegt, welches versucht, die wichtigsten Eigenschaften der neuen Medien gemeinsam zu nutzen.

Im Rahmen dieser Arbeit wurde weiterhin der Prototyp einer Java-basierten Internetplattform implementiert, welche zeitgesteuert multimediale Inhalte darstellt, offen für Erweiterungen ist und hierdurch ein hohes Maß an Universalität bei der Präsentation unterschiedlichster Informationen bietet. Auf der Basis der hierin vorhandenen Zeitsynchronität konnten gestreamte Medien am Beispiel einer waveletbasierten Videokomponente in das System integriert werden. Die Plattform verwaltet die mit Hilfe der Extensible Markup Language (XML) beschriebenen Komponenten in einer objektorientierten Weise, die eine Wiederverwendbarkeit der einzelnen Medienobjekte in einem hohen Maße fördert.

Obwohl die der Entwicklung zugrunde liegende Java SDK-Version 2 kleinere Fehlfuntionen aufweist, konnte die konzeptionierte Lernplattform erfolgreich implementiert werden.
Zu deren Test habe ich eine Beispielpräsentation unter Einbeziehung aller derzeit möglichen Medientypen erstellt und aufgezeigt.

7. Ausblick

Die im Rahmen dieser Arbeit entwickelte Plattform bietet mehrere Anknüpfungspunkte für zukünftige Optimierungen.

So müßte z. B. über ein sogenanntes *Thread-Pooling* innerhalb des Timers nachgedacht werden, damit die Zeitsynchronität des Systems nicht durch das Blockieren von unsauber implementierten Elementtypen gefährdet wird. Dabei handelt es sich um eine feste Anzahl bereits laufender Threads, die den einzelnen Mobs zugeteilt werden könnten und somit eine berechenbare Belastung für den Webbrowser darstellen.
Eine weitere denkbare Optimierung wäre das geschwindigkeitsbezogene Optimieren der Plattform, um bei größeren Präsentationen den zeitlichen Ablauf zu gewährleisten.

Neben diesen Verbesserungen des bereits bestehenden Systems existieren natürlich auch Möglichkeiten für die zukünftige Weiterentwicklung.

Als erstes wäre dabei wohl die bereits vorbereitete Interaktionsfähigkeit zu nennen, die den praktischen Einsatz des Systems als Lehrplattform spannend macht. Ein weiterer wichtiger Punkt, den es umzusetzen gilt, ist die Anbindung des Systems an eine Datenbank. Nur so kann das bei mehreren Präsentationen zu erwartende Datenaufkommen sinnvoll bewältigt werden. Erst mit ihrer Anbindung wird auch die Wiederverwendbarkeit der Komponenten praktikabel.
Als einen weiteren wichtigen Punkt bei der Weiterentwicklung der Plattform wäre die Umsetzung eines geeigneten Authoringtools anzusehen. Der dabei zu erwartende recht erhebliche Aufwand würde die Fähigkeit zur Folge haben, daß jeder Interessierte einfach und schnell Präsentationsinhalte erstellen kann.

Das Ergebnis einer derartigen Optimierung und Weiterentwicklung des Systems wäre eine aüßerst leistungsstarke und flexible Lehr- und Präsentationsplattform, wie sie heutzutage noch nicht existiert.

Literaturverzeichnis

[1] Macromedia: Macromedia Director. http://www.macromedia.com/software/director/, 1995-2000.

[2] Macromedia: Macromedia Authorware. http://www.macromedia.com/software/authorware/, 1995-2000.

[3] Asymetrix: ToolBook II Instructor. http://www.asymetrix.com/products/toolbook2/instructor/, 2000.

[4] Feustel, Björn: ProVirtus. http://provirtus.fhtw-berlin.de/, dec 1998.

[5] Berners-Lee, T.: RFC 1866 Hypertext Markup Language - 2.0, nov 1995.

[6] International Organization for Standardization: ISO 8879: Information processing - Text and office systems - Standard Generalized Markup Language (SGML), 1986.

[7] Berners-Lee, T.: RFC 1945 Hypertext Transfer Protocol – HTTP/1.0, may 1996.

[8] Corporation, Netscape Communications: Dynamic HTML Developer Central. http://developer.netscape.com/tech/dynhtml/index.html, 1999.

[9] Münz, Stefan: SELFHTML - Server Side Includes in HTML. http://www.netzwelt.com/selfhtml/tgbe.htm, 1998.

[10] Fleishman, Glenn: Cookies: Fresh From Your Browser's Oven. http://www.webdeveloper.com/cgi-perl/cgi_fresh_cookies.html, 1999.

[11] Corporation, Netscape Communications: JavaScript Developer Central. http://developer.netscape.com/tech/javascript/index.html.

[12] Consortium, World Wide Web: Synchronized Multimedia Integration Language (SMIL) 1.0 Specification. http://www.w3.org/TR/1998/REC-smil-19980615/.

[13] Steele, James Gosling; Bill Joy; Guy: The Java Language Specification. http://java.sun.com/docs/books/jls/html/index.html, 1996.

[14] Sun Microsystems, Inc.: Java Plug-in Product. http://java.sun.com/products/plugin/index.html, 2000.

[15] Jacobson, H. Schulzrinne; S. Casner; R. Frederick; V.: RFC 1889 RTP: A Transport Protocol for Real-Time Applications, jan 1996.

[16] Palkow, Prof. H. L. Cycon; Mark: Wavelet-Based Image Compression. http://www.fhtw-berlin.de/Projekte/Wavelet/, 1998.

[17] World Wide Web Consortium: Extensible Markup Language (XML) 1.0, feb 1998.

[18] Ducharme, Bob: XML - The annotated Specifcation. Prentice Hall PTR, 1999.

[19] Verlag Heinz Heise - XML page. http://www.heise.de/ix/raven/Web/xml/, 1999.

[20] XML4J - XML Parser for Java. http://alphaworks.ibm.com/tech/xml4j, 1998.

[21] Flanagan, David: Java Examples in a Nutshell. O'Reilly Verlag, 1997.

[22] Freed, N. Borenstein; N.: RFC 1341 MIME (Multipurpose Internet Mail Extensions): Mechanisms for Specifying and Describing the Format of Internet Message Bodies., jun 1992.

[23] CERN: European Organization for Nuclear Research. http://www.cern.ch/.

A. Abkürzungsverzeichnis

AVI	Audio Video Interleave
AWT	Abstract Window Toolkit
CERN	Conseil Européen pour la Recherche Nucléaire[23]
DHTML	Dynamic HTML
DTD	Document Type Definition
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
ISDN	Integrated Services Digital Network
JDBC	Java Database Connectivity Package
JVM	Java Virtual Machine
MIME	Multipurpose Internet Mail Extensions
Mob	Medienobjekt
MPEG	Moving Picture Experts Group
RFC	Request for Comments
SDK	(Java) Software Development Kit
SGML	Standard Generalized Markup Language
W3C	World Wide Web Consortium
WWW	World Wide Web
XML	Extensible Markup Language

B. Entwicklungsumgebung

Bei der Entwicklung der Plattform wurde folgende Hardware verwendet:

- Sun UltraSparc-IIi
  1 CPU 300Mhz, 256 Mb RAM
- Silicon Graphics O2
  1 CPU R5000 180Mhz, 192 Mb RAM
- Standard PC
  1 CPU AMD Athlon 500 Mhz, 64 Mb RAM
- Wasserkocher
  Petra Cordless FixKocher

Folgende Software habe ich verwandt:

- Betriebssysteme:
  - Sun Solaris 7
  - SGI IRIX 6.5
  - Windows NT 4.0
- Java Software Development Kit:
  JavaTM 2 Platform, Standard Edition (J2SE)
- WWW-Browser:
  Netscape Communikator 4.6 mit Java-Plugin 1.2 der Firma Sun Microsystems
- Text-Editor:
  VIM - Vi IMproved 5.4
- Textsatzsystem für diese Arbeit:
  LaTeX 2_ε
- Zeichenprogramm für die grafischen Darstellungen:
  Xfig 3.2

C. Javadoc Klassenbeschreibungen

C.1. Package MobITServer

Package Contents *Page*

Classes

C.1.1. Classes

Class **MediaServer**

Diese Klasse enthält die `main()`-Methode, durch welche der Server gestartet wird.

Declaration

public class MediaServer
extends java.lang.Object

Constructors

- *MediaServer*
  `public` **MediaServer**`( )`

Methods

- *main*
  `public static void` **main**`( java.lang.String [] args )`

– **Usage**
  ▷ Die Mainmethode, die beim Start des Servers aufgerufen wird. Sie instanziiert einen neuen "generic Serveründ meldet den MediaServer bei diesem an.

Class MediaServer.MobServer

Diese Klasse enthält neben einer inneren Klasse fuer die spezielle Serverfunktionalität auch Methoden zum Debuggen und zum Starten der SubServer. Sie wird mittels des `Service`-Interface an den zugrunde liegenden "generic server"gebunden.

Declaration

public static class MediaServer.MobServer
extends java.lang.Object
implements Server.Service

Constructors

- *MediaServer.MobServer*
  `public` **MediaServer.MobServer**`( )`

Methods

- *debug*
  `public static void` **debug**`( java.lang.String` **_msg** `)`
  – **Usage**
    ▷ Methode, die f¨r die Ausgabe der Debug-Informationen zuständig ist.
  – **Parameters**
    ▷ `_msg` - die auszugebende Meldung

- *serve*
  `public void` **serve**`( java.io.InputStream` **i**`, java.io.OutputStream` **o** `)`
  – **Usage**
    ▷ Instanziiert einen neuen Medienserver, der die spezielle Serverfunktionalität bereitstellt.

- **Parameters**
  - ▷ `i` - der vom Client kommende Datenstrom
  - ▷ `o` - der zum Client gehende Datenstrom
- **Exceptions**
  - ▷ `java.io.IOException` - wird durch Fehler bei der Datenübertragung erzeugt

---

- *startSubServer*
  `public void` **startSubServer**`( java.lang.String` **_type**`, java.lang.Object` **_elem** `)`
  - **Usage**
    - ▷ Startet einen neuen Subserver, der z.B. für streaming media benötigt wird.
  - **Parameters**
    - ▷ `_type` - der Typ des zu startenden Servers
    - ▷ `_elem` - das durch den Subserver auszuliefernde Medienobjekt
  - **Exceptions**
    - ▷ `myexceptions.NoFreePortsException` - wird erzeugt, wenn kein freier Port für den Subserver gefunden werden kann
    - ▷ `java.net.UnknownHostException` - wird erzeugt, wenn der aktuelle Host, auf dem der Server läuft, nicht ermittelt werden kann

Class MediaServer.MobServer.MServer

Diese Klasse stellt alle fuer die spezielle Serverfunktionalitaet benoetigten Methoden und Klassen bereit.

Declaration

public class MediaServer.MobServer.MServer
extends java.lang.Object

Methods

- *serve*
  `public void` **serve**`( java.io.InputStream` **_i**`, java.io.OutputStream` **_o** `)`
  - **Usage**
    - ▷ Methode, welche die spezielle Serverfunktionalität implementiert

- **Parameters**
  - ▷ `_i` - der Eingabestrom
  - ▷ `_o` - der Ausgabestrom
- **Exceptions**
  - ▷ `java.io.IOException` - wird durch Fehler bei der Datenübertragung erzeugt

Class MediaServer.MobServer.MServer.ServeMethods

Diese Klasse enthält die den einzelnen Befehlen zugeordneten Methoden.

Declaration

public class MediaServer.MobServer.MServer.ServeMethods
extends java.lang.Object

Constructors

- *MediaServer.MobServer.MServer.ServeMethods*
  `public` **MediaServer.MobServer.MServer.ServeMethods(**
  `MediaServer.MobServer.MServer` **this$1**)

Methods

- *get*
  `public Object` **get(** `java.lang.String` **_what**)
  - **Usage**
    - ▷ Methode, welche die vom Client angeforderten Objekte ausliefert
  - **Parameters**
    - ▷ `_what` - die gewünschten Objektelemente (z.B. `pl` für die Playlist)
  - **Returns** - das ermittelte Objekt

- *quit*
  `public void` **quit(**)
  - **Usage**
    - ▷ Methode zum Beenden der aktuellen Verbindung

Class **XMLaccess**

Diese Klasse ist für den Zugriff auf die Objektbeschreibungen, die durch die Extensible Markup Language (XML) beschrieben vorliegen, zuständig.

Declaration

public class XMLaccess
extends java.lang.Object

Methods

- *getElement*
  `public Elementable` **getElement**`( int` **_id** `)`
  - **Usage**
    - ermittelt ein spezielles Element
  - **Parameters**
    - `_id` - die globale ID des Objektes, welches das Element kapselt
  - **Returns** - das gewünschte Element

- *getMH*
  `public Hashtable` **getMH**`( int` **_id** `)`
  - **Usage**
    - ermittelt eine spezielle Moblist
  - **Parameters**
    - `_id` - die globale ID des Objektes
  - **Returns** - die gewünschte Moblist

- *getMI*
  `public MobInfo` **getMI**`( int` **_id** `)`
  - **Usage**
    - ermittelt Informationen über ein Mob
  - **Parameters**
    - `_id` - die globale ID des Objektes
  - **Returns** - die gewünschten Informationen

- *getPL*
  `public Playlist` **getPL**`( int` **_id** `)`

- **Usage**
  - ▷ ermittelt eine spezielle Playlist
- **Parameters**
  - ▷ `_id` - die globale ID des Objektes
- **Returns** - die gewünschte Playlist

Class **NoFreePortsException**

Declaration

public class NoFreePortsException **extends** java.lang.Exception

Constructors

- *NoFreePortsException*
  `public` **NoFreePortsException**`( )`
- *NoFreePortsException*
  `public` **NoFreePortsException**`( java.lang.String s )`

Methods inherited from class `java.lang.Exception`

Methods inherited from class `java.lang.Throwable`

- *fillInStackTrace*
  `public native Throwable` **fillInStackTrace**`( )`
- *getLocalizedMessage*
  `public String` **getLocalizedMessage**`( )`
- *getMessage*
  `public String` **getMessage**`( )`
- *printStackTrace*
  `public void` **printStackTrace**`( )`
- *printStackTrace*
  `public void` **printStackTrace**`( java.io.PrintStream )`
- *printStackTrace*
  `public void` **printStackTrace**`( java.io.PrintWriter )`
- *toString*
  `public String` **toString**`( )`

C.2. Package element

Package Contents *Page*

Classes

C.2.1. Classes

Class **ElementGIF**

Declaration

public class ElementGIF
extends mobit.ContainerElement
implements mobit.Elementable

Serializable Fields

- public String type
  –

Fields

- public String type
  –

Constructors

- *ElementGIF*
  `public` **ElementGIF**`( )`

Methods

- *doHide*
  `public void` **doHide**`( )`
- *doInit*
  `public void` **doInit**`( )`
- *doShow*
  `public void` **doShow**`( )`
- *doStart*
  `public void` **doStart**`( )`
- *doStop*
  `public void` **doStop**`( )`
- *paint*
  `public void` **paint**`( java.awt.Graphics g )`

- *setData*
  `public void` **setData**`( java.lang.Object _d )`
- *setID*
  `public void` **setID**`( int _id )`
- *setParams*
  `public void` **setParams**`( java.util.Vector _v )`
- *update*
  `public void` **update**`( java.awt.Graphics g )`

Class **ElementPaccMp3**

Declaration

public class ElementPaccMp3
extends mobit.ContainerElement
implements mobit.Elementable, mobit.Streamable

Serializable Fields

Constructors

- *ElementPaccMp3*
  `public` **ElementPaccMp3**`( )`

Methods

- *doHide*
  `public void` **doHide**`( )`
- *doInit*
  `public void` **doInit**`( )`
- *doShow*
  `public void` **doShow**`( )`
- *doStart*
  `public void` **doStart**`( )`
- *doStop*
  `public void` **doStop**`( )`
- *getServerInfo*
  `public ServerInfo` **getServerInfo**`( )`

- *paint*
  `public void` **paint**`( java.awt.Graphics` **_g** `)`
- *setData*
  `public void` **setData**`( java.lang.Object` **_d** `)`
- *setID*
  `public void` **setID**`( int` **_id** `)`
- *setParams*
  `public void` **setParams**`( java.util.Vector` **_v** `)`
- *setServerInfo*
  `public void` **setServerInfo**`( mobit.ServerInfo` **_si** `)`
- *setVisible*
  `public void` **setVisible**`( boolean` **b** `)`
- *update*
  `public void` **update**`( java.awt.Graphics` **_g** `)`

C.3. Package mobit

Package Contents *Page*

C.3.1. Interfaces

Interface **Elementable**

Declaration

public abstract interface Elementable

Methods

- *doCustom*
  `public void` **doCustom**`( java.lang.String` **_par** `)`
- *doHide*
  `public void` **doHide**`( )`
- *doInit*
  `public void` **doInit**`( )`
- *doShow*
  `public void` **doShow**`( )`
- *doStart*
  `public void` **doStart**`( )`
- *doStop*
  `public void` **doStop**`( )`
- *getParam*
  `public String` **getParam**`( java.lang.String` **_name** `)`
- *getType*
  `public String` **getType**`( )`
- *hasParam*
  `public boolean` **hasParam**`( java.lang.String` **_name** `)`
- *isStreammedia*
  `public boolean` **isStreammedia**`( )`
- *setData*
  `public void` **setData**`( java.lang.Object` **_d** `)`
- *setID*
  `public void` **setID**`( int` **_id** `)`
- *setParam*
  `public void` **setParam**`( mobit.ElementParam` **_ep** `)`
- *setParams*
  `public void` **setParams**`( java.util.Vector` **_v** `)`

- *setPos*
  `public void` **setPos**`( java.awt.Point` **_p** `)`
- *setRelScale*
  `public void` **setRelScale**`( float` **_s** `)`
- *setStreammedia*
  `public void` **setStreammedia**`( boolean` **_f** `)`
- *setType*
  `public void` **setType**`( java.lang.String` **_t** `)`

Interface **Streamable**

Dieses Interface muessen alle Elemente implementieren, die einen eigenen Subserver benoetigen.

Declaration

public abstract interface Streamable

Methods

- *getServerInfo*
  `public ServerInfo` **getServerInfo**`( )`
- *setServerInfo*
  `public void` **setServerInfo**`( mobit.ServerInfo` **_si** `)`

C.3.2. Classes

Class **Command**

Diese Klasse entspricht einer einzelnen Aktion

Declaration

public class Command **extends** java.lang.Object **implements** java.io.Serializable

Serializable Fields

- public int cmd

  –

- public int time

–

- public int id

  –

- public int object1

  –

- public int object2

  –

- public Point pos

  –

- public int layer

  –

- public float scale

  –

- public boolean done

  –

Fields

- public static final int cmdInit

  –

- public static final int cmdStart

  –

- public static final int cmdStop

  –

- public static final int cmdShow

  –

- public static final int cmdHide

  –

- public static final int cmdCustom

  –

- public static final int cmdSetPos

  –

- public static final int cmdSetScale
  –
- public int cmd
  –
- public int time
  –
- public int id
  –
- public int object1
  –
- public int object2
  –
- public Point pos
  –
- public int layer
  –
- public float scale
  –
- public boolean done
  –

Constructors

- *Command*
  `public` **Command**`( )`
- *Command*
  `public` **Command**`( int` **_cmd** `)`

Methods

- *toString*
  `public String` **toString**`( )`
  - **Usage**
    - ▷ Gibt das Kommando in gut lesbarer Form aus.

Class **ContainerElement**

Diese Klasse bildet die Basis fuer die meisten Elementtypen.

Declaration

public class ContainerElement
extends java.awt.Container
implements java.io.Serializable

Serializable Fields

- public int id
  –
- public boolean isVisible
  –
- public boolean isActive
  –
- public Point pos
  –
- public float scale
  –

Fields

- public int id
  –
- public boolean isVisible
  –
- public boolean isActive
  –
- public Point pos
  –
- public float scale
  –

Constructors

- *ContainerElement*
  `public` **ContainerElement**`( )`

Methods

- *doCustom*
  `public void` **doCustom**`( java.lang.String` **_par** `)`
  - **Usage**
    - ▷ die dem Befehl `CUSTOM` entsprechende Methode. Sie sollte von den Elementtypen entsprechend ihrer Funktionalitaet ueberschrieben werden
  - **Parameters**
    - ▷ `_par` - der Parameterstring

- *doHide*
  `public void` **doHide**`( )`
  - **Usage**
    - ▷ die dem Befehl `HIDE` entsprechende Methode. Sie sollte von den Elementtypen entsprechend ihrer Funktionalitaet ueberschrieben werden

---

- *doInit*
  `public void` **doInit**`( )`
  - **Usage**
    - ▷ die dem Befehl `INIT` entsprechende Methode. Sie sollte von den Elementtypen entsprechend ihrer Funktionalitaet ueberschrieben werden

---

- *doShow*
  `public void` **doShow**`( )`
  - **Usage**
    - ▷ die dem Befehl `SHOW` entsprechende Methode. Sie sollte von den Elementtypen entsprechend ihrer Funktionalitaet ueberschrieben werden

---

- *doStart*
  `public void` **doStart**`( )`
  - **Usage**

▷ die dem Befehl `START` entsprechende Methode. Sie sollte von den Elementtypen entsprechend ihrer Funktionalitaet ueberschrieben werden

---

- *doStop*
  `public void` **doStop**`( )`
  - **Usage**
    ▷ die dem Befehl `STOP` entsprechende Methode. Sie sollte von den Elementtypen entsprechend ihrer Funktionalitaet ueberschrieben werden

---

- *getParam*
  `public String` **getParam**`( java.lang.String` **_name** `)`
  - **Usage**
    ▷ liefert einen bestimmten Parameter.
  - **Parameters**
    ▷ `_name` - der Name des Parameters, zu dem die Werte ermittelt werden sollen
  - **Returns** - die dem Parameter zugeordneten Werte oder `null`, wenn der Parameter nicht existiert

---

- *getType*
  `public String` **getType**`( )`
  - **Usage**
    ▷ liefert den Typ des Elements
  - **Returns** - der Typ (MIME-kodiert)

- *hasParam*
  `public boolean` **hasParam**`( java.lang.String` **_name** `)`
  - **Usage**
    ▷ prueft, ob ein bestimmter Parameter vorhanden ist.
  - **Parameters**
    ▷ `_name` - der Name des Parameters
  - **Returns** - `true` wenn der Parameter vorhanden ist, sonst `false`

- *isStreammedia*
  `public boolean` **isStreammedia**`( )`
  - **Usage**
    ▷ ermittelt das Flag, welches das Element als gestreamtes Medium ausweist.
  - **Returns** - `true` wenn es sich um ein gestreamtes Medium handelt, sonst `false`

- *setData*
  `public void` **setData**`( )`
  - **Usage**
    - ▷ setzt die Daten des Elements. Dabei sollte diese Methode von den einzelnen Elementen entsprechend ueberschrieben werden.

- *setID*
  `public void` **setID**`( int` **_id** `)`
  - **Usage**
    - ▷ setzt die ID des Elements
  - **Parameters**
    - ▷ `_id` - die ID

- *setParam*
  `public void` **setParam**`( mobit.ElementParam` **_ep** `)`
  - **Usage**
    - ▷ setzt einen einzelnen Parameter.
  - **Parameters**
    - ▷ `_ep` - der Parameter

- *setParams*
  `public void` **setParams**`( java.util.Vector` **_v** `)`
  - **Usage**
    - ▷ setzt die Parameterliste des Elements.
  - **Parameters**
    - ▷ `_v` - der Parametervector

- *setPos*
  `public void` **setPos**`( java.awt.Point` **_p** `)`
  - **Usage**
    - ▷ setzt die geometrische Position
  - **Parameters**
    - ▷ `_p` - die Position

- *setRelScale*
  `public void` **setRelScale**`( float` **_s** `)`
  - **Usage**
    - ▷ setzt den geometrische Skalierungsfaktor
  - **Parameters**
    - ▷ `_s` - der Skalierungsfaktor

- *setStreammedia*
  `public void` **setStreammedia**`( boolean _f )`
  - **Usage**
    - ▷ setzt das Flag, welches das Element als gestreamtes Medium ausweist.
  - **Parameters**
    - ▷ `_f` - `true` wenn es sich um ein gestreamtes Medium handelt, sonst `false`

- *setType*
  `public void` **setType**`( java.lang.String _t )`
  - **Usage**
    - ▷ setzt den Typ des Elements
  - **Parameters**
    - ▷ `_t` - der Typ (MIME-kodiert)

Class Globals

Diese Klasse enthaelt globale Variablen und Methoden fuer das gesamte Projekt.

Declaration

public class Globals
extends java.lang.Object

Fields

- public static final boolean DEBUG
  –

Constructors

- *Globals*
  `public` **Globals**`( )`

Methods

- *Debug*
  `public static final void` **Debug**`( int _level, java.lang.String _msg )`

- *Debug*
  `public static final void` **Debug**`( java.lang.String` **_msg** `)`
- *getSpaces*
  `public static final String` **getSpaces**`( int` **_i** `)`

Class **Mob**

Diese Klasse repraesentiert ein Medienobjekt mit all seinen Methoden und Eigenschaften.

Declaration

public class Mob **extends** java.awt.Container

Serializable Fields

- public int inst
  –

Fields

- public int inst
  –

Constructors

- *Mob*
  `public` **Mob**`( int` **_id** `)`
  - **Usage**
    - ▷ setzt die ID.
  - **Parameters**
    - ▷ `_id` - die ID

Methods

- *doHide*
  `public void` **doHide**`( mobit.Command` **_c** `)`
  - **Usage**
    - ▷ die dem `HIDE` Befehl zugeordnete Methode.
  - **Parameters**

▷ _c - das hide-Kommando mit den entsprechenden Parametern

- *doInit*
  `public void` **doInit**`( mobit.Command _c )`
  - **Usage**
    ▷ die dem `INIT` Befehl zugeordnete Methode.
  - **Parameters**
    ▷ _c - das init-Kommando mit den entsprechenden Parametern

---

- *doShow*
  `public void` **doShow**`( mobit.Command _c )`
  - **Usage**
    ▷ die dem `SHOW` Befehl zugeordnete Methode.
  - **Parameters**
    ▷ _c - das show-Kommando mit den entsprechenden Parametern

---

- *doStart*
  `public void` **doStart**`( mobit.Command _c )`
  - **Usage**
    ▷ die dem `START` Befehl zugeordnete Methode.
  - **Parameters**
    ▷ _c - das start-Kommando mit den entsprechenden Parametern

---

- *doStop*
  `public void` **doStop**`( mobit.Command _c )`
  - **Usage**
    ▷ die dem `STOP` Befehl zugeordnete Methode.
  - **Parameters**
    ▷ _c - das stop-Kommando mit den entsprechenden Parametern

---

- *getID*
  `public int` **getID**`( )`
  - **Usage**
    ▷ liefert die ID.
  - **Returns** - _id die ID

- *init*
  `public void` **init**`( )`
  - **Usage**
    ▷ initialisiert das Objekt, einschliesslich der ihm untergeordneten Mobs. Hierbei wird die Datenstruktur erstellt.

- *isInitiated*
  `public boolean` **isInitiated**`( )`
  - **Usage**
    - ▷ zeigt den Initialisierungszustand an.
  - **Returns** - `true` wenn das Objekt initialisiert wurde, sonst `false`

- *paint*
  `public void` **paint**`( java.awt.Graphics g )`
  - **Usage**
    - ▷ stellt das Objekt mit seinen untergeordneten Komponenten dar.
  - **Parameters**
    - ▷ `g` - der entsprechende Grafikkontext

- *printStatus*
  `public void` **printStatus**`( )`

- *processCmd*
  `public void` **processCmd**`( mobit.Command _c )`
  - **Usage**
    - ▷ verarbeitet eine anstehende Aktion.
  - **Parameters**
    - ▷ `_c` - das auszufuehrende Kommando

- *runTo*
  `public void` **runTo**`( int _t )`
  - **Usage**
    - ▷ fuehrt alle Aktionen bis zu einem bestimmten Zeitpunkt aus.
  - **Parameters**
    - ▷ `_t` - der Zeitpunkt, bis zu dem die Aktionen ausgefuehrt werden sollen

---

- *setAbsDim*
  `public void` **setAbsDim**`( int _xd, int _yd )`
  - **Usage**
    - ▷ setzt die absoluten geometrischen Ausmasse.
  - **Parameters**
    - ▷ `_xd` - die Breite
    - ▷ `_yd` - die Hoehe

- *setAbsScale*
  `public void` **setAbsScale**`( float _s )`

  - **Usage**
    ▷ setzt den absoluten geometrischen Skalierungsfaktor.
  - **Parameters**
    ▷ `_s` - der Faktor

- *setActive*
  `public void` **setActive**`( boolean _f )`
  - **Usage**
    ▷ de-/aktiviert das Objekt.
  - **Parameters**
    ▷ `_f` - das den Zustand beschreibende Flag

- *setBaseTime*
  `public void` **setBaseTime**`( int _t )`
  - **Usage**
    ▷ setzt die Basiszeit.
  - **Parameters**
    ▷ `_t` - die Basiszeit

- *setDS*
  `public void` **setDS**`( mobit.MobDataSourceTCP _ds )`
  - **Usage**
    ▷ setzt die Datenquelle.
  - **Parameters**
    ▷ `_ds` - die Datenquelle

- *setPanel*
  `public void` **setPanel**`( java.awt.Panel _p )`
  - **Usage**
    ▷ setzt das Panel, zu dem das Objekt gehoert.
  - **Parameters**
    ▷ `_p` - das Panel

- *setRelDim*
  `public void` **setRelDim**`( int _xd, int _yd )`
  - **Usage**
    ▷ setzt die relativen geometrischen Ausmasse.
  - **Parameters**
    ▷ `_xd` - die Breite
    ▷ `_yd` - die Hoehe

- *setRelPos*
  `public void` **setRelPos**`( java.awt.Point _p )`
  - **Usage**
    - ▷ setzt die effektive Position.
  - **Parameters**
    - ▷ `_p` - die Position

- *setRelScale*
  `public void` **setRelScale**`( float _s )`
  - **Usage**
    - ▷ setzt den effektiven geometrischen Skalierungsfaktor.
  - **Parameters**
    - ▷ `_s` - der Faktor

- *setStartTime*
  `public void` **setStartTime**`( int _t )`
  - **Usage**
    - ▷ setzt die Startzeit, bei der das Objekt erstmalig aktiv wurde.
  - **Parameters**
    - ▷ `_t` - die Startzeit

- *setTimer*
  `public void` **setTimer**`( mobit.Timer _t )`
  - **Usage**
    - ▷ setzt den zustaendigen Timer.
  - **Parameters**
    - ▷ `_t` - der Timer

- *setVisible*
  `public void` **setVisible**`( boolean b )`
  - **Usage**
    - ▷ zeigt das Objekt an oder verdeckt es.
  - **Parameters**
    - ▷ `b` - das den Zustand anzeigende Flag

- *update*
  `public int` **update**`( )`
  - **Usage**
    - ▷ diese Methode wird vom Timer beim Eintreten einer bestimmten Aktion angesprochen.

- *update*
  `public void` **update**`( java.awt.Graphics  g )`
  - **Usage**
    - ▷ aktualisiert die Darstellung.
  - **Parameters**
    - ▷ `g` - der entsprechende Grafikkontext

Class **MobDataSourceTCP**

Diese Klasse handhabt die Transaktion der Daten vom Server zum Client.

Declaration

public class MobDataSourceTCP **extends** mobit.MobDataSource **implements** MobDataSourceInterface, java.lang.Runnable

Fields

- public boolean connected
  –
- public boolean doIt
  –

Constructors

- *MobDataSourceTCP*
  `public` **MobDataSourceTCP**`( java.lang.String  _server, int _port )`
  - **Usage**
    - ▷ der Konstruktor.
  - **Parameters**
    - ▷ `_server` - der Server, von dem die Daten geholt werden sollen
    - ▷ `_port` - der fuer die Verbindung vorgesehene Port

Methods

- *getElement*
  `public Elementable` **getElement**`( int  _id )`
  - **Usage**

▷ liefert das zu einem Objekt gehoerende Element.

– **Parameters**

▷ `_id` - die ID des Mobs, dessen Element ermittelt werden soll

– **Returns** - das angeforderte Element

- *getMobHash*
  `public Hashtable` **getMobHash**`( int _id )`

  – **Usage**

  ▷ liefert die zu einem Objekt gehoerende Moblist in Form eines Hashes.

  – **Parameters**

  ▷ `_id` - die ID des Mobs, dessen Moblist ermittelt werden soll

  – **Returns** - die angeforderte Moblist

- *getMobInfo*
  `public MobInfo` **getMobInfo**`( int _id )`

  – **Usage**

  ▷ liefert Informationen zu einem Objekt.

  – **Parameters**

  ▷ `_id` - die ID des Mobs, von dem Informationen abgefragt werden sollen

  – **Returns** - die ermittelten Informationen

- *getPlaylist*
  `public Playlist` **getPlaylist**`( int _id )`

  – **Usage**

  ▷ liefert die zu einem Objekt gehoerende Playlist.

  – **Parameters**

  ▷ `_id` - die ID des Mobs, dessen Playlist ermittelt werden soll

  – **Returns** - die angeforderte Playlist

- *run*
  `public void` **run**`( )`

  – **Usage**

  ▷ wird vom Thread ausgefuehrt und uebernimmt das Management der Datenanfragen

---

- *sendRequest*
  `public void` **sendRequest**`( mobit.MobDataSource.ProcessInst _p )`

  – **Usage**

  ▷ nimmt Datenanforderungen an und speichert diese.

  – **Parameters**

▷ _p - die Beschreibung der gewuenschten Daten

- *stop*
  `public void` **stop**`( )`
  - **Usage**
    ▷ stopt den Thread.

Class **MobInfo**

Diese Klasse beschreibt verschiedene Informationen eines Mob.

Declaration

public class MobInfo **extends** java.lang.Object **implements** java.io.Serializable

Serializable Fields

- public int xDim
  –
- public int yDim
  –
- public Color color
  –

Fields

- public int xDim
  –
- public int yDim
  –
- public Color color
  –

Constructors

- *MobInfo*
  `public` **MobInfo**`( )`

Class **MobReference**

Diese Klasse repraesentiert eine referenz auf ein Mob. Dabei wird die Referenz, die GID und die Basiszeit erfasst.

Declaration

public class MobReference
extends java.lang.Object
implements java.io.Serializable

Serializable Fields

- public int baseTime
  –
- public int gid
  –
- public Mob mob
  –

Fields

- public int baseTime
  –
- public int gid
  –
- public Mob mob
  –

Constructors

- *MobReference*
  `public` **MobReference**`( )`

Class **Playlist**

Diese Klasse beschreibt eine Playlist mit ihren Eigenschaften und Methoden.

Declaration

public class Playlist
extends java.lang.Object
implements java.io.Serializable

Serializable Fields

Constructors

- *Playlist*
  `public` **Playlist**`( )`
  - **Usage**
    - ▷ Konstruktor.

- *Playlist*
  `public` **Playlist**`( int` **_cap** `)`
  - **Usage**
    - ▷ Konstruktor.
  - **Parameters**
    - ▷ `_cap` - die Anfangskapazitaet des zugrunde Liegenden Vectors

Methods

- *addElement*
  `public void` **addElement**`( mobit.Command` **_cmd** `)`
  - **Usage**
    - ▷ fuegt eine Aktion zur Playlist hinzu.
  - **Parameters**
    - ▷ `_cmd` - der das einzutragende Kommando

- *capacity*
  `public int` **capacity**`( )`

- *debugShowCmds*
  `public void` **debugShowCmds**`( )`

- *getEventsUpTo*
  `public Object` **getEventsUpTo**`( int` **_t** `)`
  - **Usage**

▷ ermittelt alle Ereignisse, die bis zu einem bestimmten Zeitpunkt eintreten.

– **Parameters**

▷ `_t` - der Zeitpunkt

– **Returns** - alle in Frage kommenden Ereignisse.

- *getNextEvent*
`public int` **getNextEvent**`( int _d )`

  – **Usage**

  ▷ ermittelt naechstes Event unter Beruecksichtigung der eventuell vorhandenen Verzoegerungszeit.

  – **Parameters**

  ▷ `_d` - die zu beruecksichtigende Verzoegerungszeit

  – **Returns** - das als naechstes anstehende Ereignis

- *removeElement*
`public boolean` **removeElement**`( java.lang.Object _o )`

  – **Usage**

  ▷ entfernt eine Aktion aus der Playlist.

  – **Parameters**

  ▷ `_o` - das zu entfernende Objekt bzw. Kommando.

- *setBaseTime*
`public void` **setBaseTime**`( int _t )`

  – **Usage**

  ▷ setzt die Basiszeit.

Class **ServerInfo**

Diese Klasse enthaelt alle noetigen Angaben, die fuer den Start eines Subservers noetig sind.

Declaration

public class ServerInfo **extends** java.lang.Object **implements** java.io.Serializable

Serializable Fields

- public int sockets
  –
- public InetAddress serverAddress
  –
- public String source
  –

Fields

- public int sockets
  –
- public InetAddress serverAddress
  –
- public String source
  –

Constructors

- *ServerInfo*
  `public` **ServerInfo**`( )`

Methods

- *printAll*
  `public void` **printAll**`( )`

Class **Timer**

Diese Klasse stellt die notwendige Timerfunktionalitaet zur Verfuegung.

Declaration

public class Timer
extends java.lang.Thread

Constructors

- *Timer*
  `public` **Timer**`( )`
  - **Usage**
    - ▷ der Konstruktor.

Methods

- *getTime*
  `public int` **getTime**`( )`
  - **Usage**
    - ▷ liefert die effektive Praesentationszeit.

- *regMob*
  `public void` **regMob**`( mobit.Mob` **_mob**`, int` **_nextEvent** `)`
  - **Usage**
    - ▷ mittels dieser Methode muessen sich Mobs bei dem Timer registrieren.
  - **Parameters**
    - ▷ `_mob` - eine Referenz auf das entsprechende Medienobjekt
    - ▷ `_nextEvent` - der Zeitpunkt des naechsten, zum sich anmeldenden Mob gehoerende Ereigniss

---

- *run*
  `public void` **run**`( )`
  - **Usage**
    - ▷ bildet den Kern des Timers und prueft, um eventuell anstehende Ereignisse auszufuehren, bei jedem Timerduchlauf auf anstehende Aktionen.

- *runMobs*
  `public void` **runMobs**`( )`
  - **Usage**
    - ▷ fuert die `update()` Methode der untergeordneten Mobs aus.

- *updateMobEvent*
  `public void` **updateMobEvent**`( mobit.Mob` **_mob**`, int` **_nextEvent** `)`
  - **Usage**
    - ▷ Methode, mit der Mobs beim Timer neue Aktionszeiten anmelden koenne.

– **Parameters**
  ▷ `_mob` - eine Referenz auf das entsprechende Medienobjekt
  ▷ `_nextEvent` - der Zeitpunkt des naechsten, zum sich anmeldenden Mob gehoerende Ereigniss

Class **Timer.MoblistElement**

Declaration

public class Timer.MoblistElement
extends java.lang.Object

Fields

- public Mob mob
  –
- public int nextEvent
  –

Constructors

- *Timer.MoblistElement*
  `public` **Timer.MoblistElement**`( mobit.Timer` **this$0** `)`

C.4. Package MobITClient

Package Contents *Page*

Classes

C.4.1. Classes

Class **MobIT**

Diese Klasse wird vom Browser instanziiert und bildet daher die Hauptklasse des MobIT-Clients

Declaration

public class MobIT
extends java.applet.Applet
implements java.lang.Runnable

Constructors

- *MobIT*
  `public` **MobIT**`( )`

Methods

- *destroy*
  `public void` **destroy**`( )`
  - **Usage**
    - ▷ wird vom Browser z.B. beim Verlassen der HTML-Seite aufgerufen.

- *getBgColor*
  `public void` **getBgColor**`( )`
  - **Usage**
    - ▷ ermittelt die Hintergrundfarbe des darzustellenden Mobs

---

- *getParentMob*
  `public void` **getParentMob**`( )`
  - **Usage**
    - ▷ ermittelt die ID des darzustellenden Mobs aus dem `applet`-Tag der HTML-Seite

---

- *init*
  `public void` **init**`( )`
  - **Usage**
    - ▷ wird vom Browser als erstes aufgerufen und ermittelt und setzt einige Darstellungsparameter

---

- *paint*
  `public void` **paint**`( java.awt.Graphics  g )`
  - **Usage**
    - ▷ dient der Darstellung des Applets. Diese Methode wird vom Browser aufgerufen, wenn der Appletinhalt neu gezeichnet werden soll.

- *run*
  `public void` **run**`( )`

- *showStatus*
  `public void` **showStatus**`( java.awt.Graphics  g )`

- *start*
  `public void` **start**`( )`
  - **Usage**
    - ▷ wird vom Browser nach dem Aufruf von `init()` angesprochen. Die Methode instanziiert das darzustellende Mob und initialisiert es danach.

- *stop*
  `public void` **stop**`( )`
  - **Usage**
    - ▷ wird vom Browser z.B. beim Verdecken des Browserfensters aufgerufen. Die Methode sorgt dafuer, dass alle Threads ordnungsgemaess gestoppt werden.

- *update*
  `public void` **update**`( java.awt.Graphics  g )`

D. Anlagen

Die beiliegende CD-Rom hat den folgenden Inhalt:

Verzeichnis server In diesem Verzeichnis befinden sich die Quelldateien des Plattformservers.

Verzeichnis client Hier sind die Quelldateien des Clients zu finden.

Verzeichnis xml4j Die für den Zugriff auf die XML-Beschreibungsdateien verwendeten `xml4j`-Klassen befinden sich in diesem Verzeichnis.