D I P L O M A R B E I T

Entwurf und Implementierung eines Softwarearchivs für die digitale Langzeitarchivierung

Mario Philipps

Matr.-Nr. 1110381

23. Juli 2010

Gutachter:

Prof. Dr. Gerhard Schneider
Lehrstuhl für Kommunikationssysteme

Prof. Dr. Peter Thiemann
Arbeitsbereich Programmiersprachen

an der

Technischen Fakultät
der Albert-Ludwigs-Universität Freiburg

Erklärung

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen/Hilfsmittel verwendet habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.

Ort, Datum Unterschrift

Inhaltsverzeichnis

1 Einleitung

Seit der Erfindung des ersten turingmächtigen Digitalrechners durch Konrad Zuse im Jahr 1941 erlebt die Informationstechnologie einen beispiellosen Siegeszug. Es gibt heutzutage kaum noch einen mit Datenverarbeitung befassten Arbeitsbereich, der ohne IT-Unterstützung auskommt. Damit einhergehend liegen immer mehr Dokumente nur noch in digitaler Form vor. Dies wird noch verstärkt durch das rasante Anwachsen des verfügbaren Speicherplatzes, was die Digitalisierung auch großer Datenmengen ermöglicht. So werden seit neuestem sogar Kinofilme rein digital produziert, verteilt und vorgeführt.

Leider ist es um die Haltbarkeit derart gespeicherter Informationen nicht gut bestellt. So ist einerseits die Lebensdauer der Datenträger begrenzt. Während Papyri aus dem alten Ägypten auch heute, nach mehreren tausend Jahren, noch lesbar sind, wird die Lebensdauer beispielsweise optischer Speichermedien selbst von Herstellern nur auf 100 bis 200 Jahre geschätzt [Iraci, 2005]. Aber auch die verwendete Technologie ist ständigen Änderungen unterworfen, so dass oft schon nach kurzer Zeit keine Lesegeräte für bestimmte Medien mehr vorhanden sein können[1]. Neben der Hardware ist außerdem das Format der Daten einem rapiden Alterungsprozess unterzogen: Während eine Schallplatte das analoge Tonsignal noch mehr oder weniger direkt auf der Hardware abbildet, müssen die Informationen auf einer digitalen CD aufwendig kodiert werden. Ist das Wissen um die Art dieser Kodierung nicht mehr verfügbar, so ist die Information verloren.

Während die fortschreitende Digitalisierung für die Effzienz von Arbeitsabläufen höchst vorteilhaft ist, bringt sie also hinsichtlich der langfristigen Verfügbarkeit große Probleme mit sich. Neben der Wirtschaft ist es vor allem die Gesellschaft, die hiervon stark betroffen ist: Man denke nur an den vor einigen Jahren berühmt gewordenen Fall der Rosenholz-Dateien[2]. Um zu verhindern, dass unsere Epoche für zukünftige Generationen nicht mehr erschließbar ist („Digitales Vergessen“), müssen also Strategien entworfen werden, um die Langzeitverfügbarkeit von digitalen Informationen zu sichern.

Im Rahmen dieser Arbeit ist vor allem das Problem der veraltenden Datenformate relevant, zu seiner Lösung existieren zwei Herangehensweisen: Die Migrationsstrategie setzt darauf, digitale Objekte kontinuierlich in ein jeweils aktuelles Format zu überführen. Die Emulationsstrategie hingegen versucht, die Umgebung, in der das digitale Objekt ursprünglich genutzt wurde, wiederherzustellen.

[1] So existieren Laufwerke für die noch in den 80er Jahre weit verbreiteten 5,25"Disketten heutzutage nur noch in Form von Restbeständen, ein nahezu vollständiges Verschwinden ist absehbar.

[2] Hierbei handelt es sich um 381 CD-ROMs mit Angaben über Mitarbeiter des Auslandsnachrichtendienstes der ehemaligen DDR.

1.1 Ziel und Aufbau der Arbeit

Zentraler Teil der Emulationsstrategie ist eine Komponente, welche die zur Darstellung der digitalen Objekte nötigen Hilfsobjekte, wie Betriebssysteme oder Anwendungsprogramme, archiviert. Diese Komponente ist das Softwarearchiv. In der vorliegenden Arbeit soll nun untersucht werden, welche Aufgaben einem solchen Softwarearchiv in der Emulationsstrategie zukommen, und wie diese Aufgaben erfüllt werden können.

Hierzu soll zuerst ein Überblick über die Strategie der Langzeitarchivierung durch Emulation gegeben werden. Insbesondere die verschiedenen, vom Softwarearchiv zu verwaltenden Hilfsobjekte verdienen dabei Beachtung. Mindestens ebenso wichtig ist aber auch die Frage, wie Arbeitsabläufe in der emulierten Umgebung automatisiert werden können. Im zweiten Teil erfolgt dann die eigentliche Aufgabenanalyse, sowie die Erstellung eines Konzeptes für die Struktur der zu archivierenden Daten. Darauf aufbauend wird schließlich im letzten Teil eine mögliche Implementierung in Form eines Prototypen vorgestellt, um die praktische Umsetzbarkeit der theoretischen Konzepte zu demonstrieren.

2 Langzeitarchivierung durch Emulation

Emulationstechniken in der Langzeitarchivierung einzusetzen wurde erstmals vorgeschlagen von Rothenberg in [Rothenberg, 1999], einen aktuelleren, sehr ausführlichen Überblick über die diversen Aspekte des Themas stellt [von Suchodoletz, 2009] dar. Die zentrale Idee besteht darin, ein Primärobjekt nicht in ein jeweils aktuelles Format per Migration zu überführen, sondern in seiner ursprünglichen Nutzungs- beziehungsweise Erstellungsumgebung zu nutzen.

Die Gründe für dieses Vorgehen sind zahlreich, beispielsweise ist eine Objektmigration manchmal nur mit Informationsverlust möglich[3]. Dies kann darin begründet sein, dass für manche Objekttypen gar kein adäquater, das heißt die Information vollständig erhaltender, zeitgenössischer Ersatz verfügbar ist, oder eine vollständig informationserhaltende Migration nur mit erheblichem Aufwand durchführbar ist. Als Beispiele für diese Problematik können Datenbanken dienen, oder auch sogenannte CAD-Programme[4]. Abbildung 1 zeigt CATIA, ein weit verbreitetes System dieser Gattung. Gerade diese beiden Beispiele sind aber für eine Langzeitarchivierung in besonderem Maße prädestiniert. Datenbanken sind alleine aufgrund ihrer Verbreitung von größter Wichtigkeit, wohingegen CAD-Modelle oftmals sehr langlebige Objekte, wie Flugzeuge, Schiffe oder Gebäude detailliert beschreiben.

Abbildung 1 illustriert eine mögliche Lösung des Problems, aber auch die Grenzen dieser Lösung ganz besonders plastisch: Es wäre natürlich möglich, Teilansichten des zu migrierenden

[3] Eine gute Darstellung dieser Problematik findet sich in [Mellor u. a., 2002]
[4] CAD=Computer Aided Design

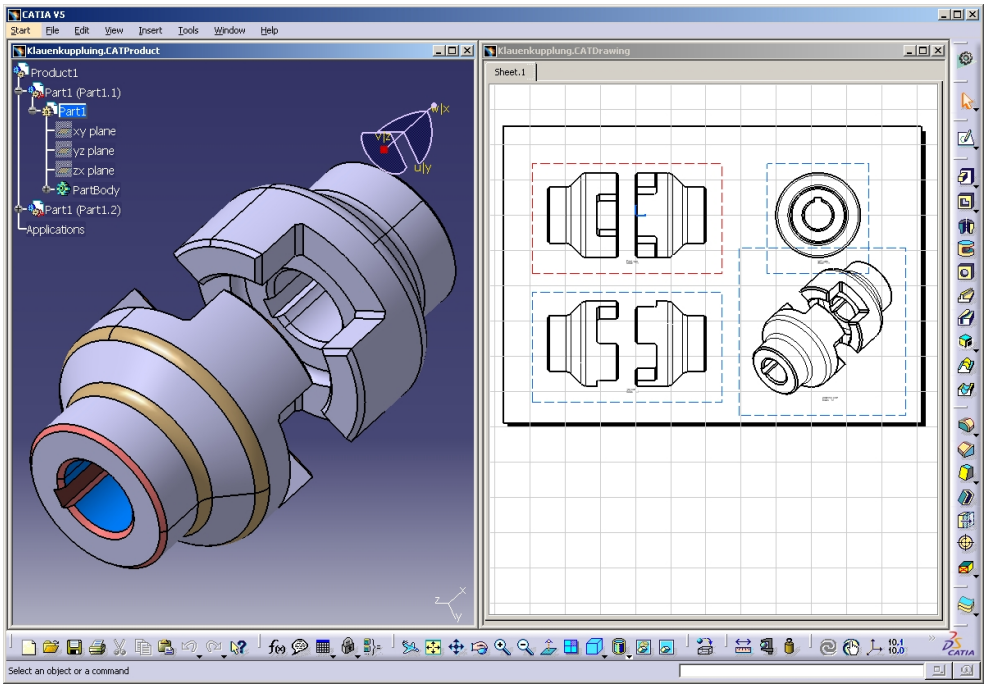


Abbildung 1: Screenshot des CAD-Programms CATIA

Primärobjektes zu erstellen, in diesem Fall zweidimensionale Bilder eines dreidimensionalen CAD-Modells aus einem bestimmten Blickwinkel. Jedoch ist in den langen Zeiträumen, die bei der Langzeitarchivierung betrachtet werden, ein zukünftiges Nutzungsinteresse kaum vorhersehbar. Es besteht also die große Gefahr, dass die gewählten Teilansichten nicht die gewünschten Informationen beinhalten.

Bei manchen Objekttypen ist ein Informationsverlust aus technischen Gründen sogar unvermeidbar. Bestes Beispiel hierfür sind Videodateien. Diese müssen allein aufgrund der Datenmenge verlustbehaftet komprimiert werden[5]. Nun sind jedoch die zur Komprimierung verwendeten Codecs[6] nicht auf unabsehbare Zeit verfügbar, so dass Videos bei einer Migration gegebenenfalls mittels aktueller Codecs umkodiert werden müssten. Hierbei summieren sich die Kompressionsverluste auf.

Schließlich ist bei manchen Objekten eine Nutzung überhaupt nur mittels Wiederherstellung der ursprünglichen Umgebung möglich. Dies gilt beispielsweise, wenn für spezielle, vor allem proprietäre, Objekttypen überhaupt keine Migrationstools verfügbar sind. Vor allem interaktive Objekte jedoch entziehen sich einer Migration prinzipbedingt. So sind Computerspiele sicherlich aufschlussreiche historische Zeugen, die, genau wie Spielfilme oder andere künstlerische

[5]Ein Spielfilm von 120 Minuten Länge umkomprimiertes Videomaterial mit 24 Vollbildern pro Sekunde im Format 720×576 Pixel entspräche rund 200 GB, eine Größenordnung, die auf absehbare Zeit eine Archivierung unmöglich macht

[6]Codec=Coder/Decoder, siehe auch Kapitel 2.1.2

Darstellungen, zum Verständnis einer Epoche erheblich beitragen können[7]. Aber auch normale Anwendungsprogramme könnten von archivarischem Interesse sein, wenn zum Beispiel die Arbeitsweise vergangener Epochen dokumentiert werden soll.

Diese Ausführungen machen deutlich, dass die Nutzung von Primärobjekten in ihrer ursprünglichen Umgebung erhebliche Vorteile bietet. Nun unterliegt aber gerade der Bereich der Informationverarbeitung einem beständigen, raschen Wandel, so dass heute aktuelle Technologien bereits nach wenigen Jahren veraltet sind. Die Produktion veralteter Hardware wird eingestellt und die entsprechenden Produktionsanlagen umgerüstet, so dass eine Produktion von Hardware vergangener Epochen, wenn überhaupt möglich, auf jeden Fall mit immensen Kosten verbunden wäre. Dies führt dazu, dass alte Hardware heutzutage nur noch in sehr geringen Stückzahlen vorhanden ist. Es wäre nun jedoch eine erhebliche Einschränkung der Verfügbarkeit, wenn die Darstellung archivierter digitaler Objekte physikalisch an diejenigen Orte gebunden wäre, an denen eines der seltenen Hardwareexemplare vorgehalten wird. Damit würde einer der großen Vorteile digitaler Dokumente, nämlich die Möglichkeit zu nahezu unbegrenzter Vervielfältigung, ad absurdum geführt.

In diesem Zusammenhang wäre es ideal, wenn die Dokumente sogar über Wide-Area-Networks, wie das Internet, zugänglich gemacht werden könnten. So könnten Bibliotheken, wie es bereits heutzutage bei Neuerscheinungen immer weitere Verbreitung findet, ihre Bestände online zum Zugriff bereitstellen. Auch für Museen, zum Beispiel das Computerspielemuseum in Berlin[8] würden sich damit ganz neue Präsentationsmöglichkeiten eröffnen.

Nicht zu vernachlässigen ist neben der begrenzten Verfügbarkeit veralteter Hardware, auch das Wissen um die Bedienung derselben. Dieses Wissen hat, ebenfalls durch den schnellen technologischen Wandel bedingt, eine extrem kurze Halbwertszeit. Mit dem Verschwinden einer speziellen Hardware vom Markt und damit rapide abnehmender Verbreitung, werden über kurz oder lang auch keine Fachkräfte mehr existieren, die diese Hardware bedienen können.

Es ist also klar, dass das reine Aufbewahren von Hardware keine Lösung im Sinne der Langzeitverfügbarkeit darstellen kann. Die Idee der Emulationsstrategie ist es nun, die zur Darstellung eines archivierten digitalen Objektes Hardware stattdessen in Software zu emulieren. Hierzu sind bereits diverse Emulatoren verfügbar, die die verschiedensten Hardwareplattformen emulieren können. Darüber hinaus können auch speziell an die Erfordernisse der Langzeitarchivierung angepasste Emulatoren entwickelt werden, wie es zum Beispiel im Rahmen des von diversen Bibliotheken und Archiven finanzierten Dioscuri-Projektes[9] geschieht.

Diese Emulatoren können auf aktueller Hardware ausgeführt werden. Somit kann auf die Hardwarekenntnisse der Archivnutzer zurückgegriffen, und das Problem des veraltenden Wissens umgangen werden. Auch der angesprochene Zugriff über ein Netzwerk wird durch manche Emulatoren vereinfacht beziehungsweise überhaupt erst ermöglicht: So bietet zum Beispiel der

[7] Ein anschauliche Schilderung der Thematik findet sich in [Baumgärtel, 2002]

[8] Homepage des Museums: `http://www.computerspielemuseum.de`

[9] Homepage des DIOSCURI-Projekts: `http://dioscuri.sourceforge.net`

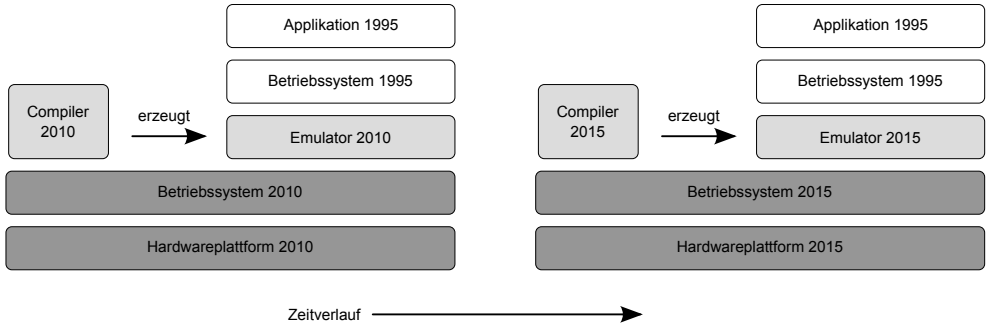


Abbildung 2: Anpassung eines Emulators im Zeitverlauf (nach [Hoeven und Wijngaarden, 2005])

Open-Source-Emulator QEMU die Möglichkeit, über die Fernwartungssoftware VNC auf die emulierte Umgebung zuzugreifen (für Details hierzu siehe Kapitel 2.4).

Ein weiterer, nicht zu unterschätzender Vorteil liegt darin, dass durch Emulatoren eine zusätzliche Abstraktionsschicht eingeführt wird. So können bei Entwicklung neuer Hardware die Emulatoren auf diese Hardware angepasst werden, ohne dass die in der emulierten Umgebung existierenden Objekte davon betroffen wären (Abbildung 2). In diesem Zusammenhang ist es übrigens bemerkenswert, dass Open-Source-Emulatoren wie QEMU klare Vorteile im Sinne einer langfristigen Perspektive bieten. Durch den frei verfügbaren Quelltext kann eine Anpassung auf neue Architekturen notfalls durch Dritte erfolgen. Bei Closed-Source Produkten wie VMWare[10] ist man hingegen darauf angewiesen, dass der Hersteller des Produktes diese Anpassung übernimmt. Hier besteht stets die Gefahr, dass der Hersteller die Unterstützung für dieses Produkt einstellt, oder er gar komplett vom Markt verschwindet. In diesem Fall ist ein Wechsel des Emulators unvermeidbar, was jedoch weitreichende Folgen für sämtliche davon abhängige Komponenten haben kann.

Alternativ zur Strategie, die Emulatoren im Laufe der Zeit an neue Architekturen anzupassen, ist eine weitere Vorgehensweise denkbar: die Schachtelung von Emulatoren. Hierbei wird, bei Veralten einer von einem bestimmten Emulator vorausgesetzten Hardwareumgebung, diese ihrerseits emuliert. Der Vorteil dieser Lösung ist klar: Eventuell aufwendige Anpassungen an neue Architekturen können entfallen, im Falle eines Closed-Source Produktes kann die Verwendbarkeit auch bei Einstellung der Weiterentwicklung durch den Hersteller gewährleistet werden. Der auf den ersten Blick bestechende Gedanke bringt jedoch einige gravierende Nachteile mit sich, die hier kurz aufgeführt werden sollen:

Erhöhte Komplexität: Durch die zusätzlichen Schichten erhöht sich die Komplexität des Gesamtsystems. Dies vergrößert zum einen die Wahrscheinlichkeit von Fehlfunktionen einzelner Komponenten, zum anderen muss der Archivnutzer mit sämtlichen Komponenten

[10] Genaugenommen handelt es sich bei VMWare nicht um einen Emulator, sondern um einen Virtualisierer, für Unterschiede siehe Kapitel 2.1.6.

des Systems umgehen können. Außerdem muss bei einer Schachtelung zusätzlich zum Wissen über die Bedienung des „äußeren“ Emulators auch Wissen über die Benutzung des „inneren“ Emulators vorhanden sein.

Erhöhter Aufwand für Objekttransport: Um archivierte Objekte innerhalb einer emulierten Umgebung nutzen zu können, müssen diese von der Hostumgebung hineintransportiert, und im Falle einer Migration das Ergebnis auch wieder heraustransportiert werden. Dieser Vorgang ist emulatorspezifisch, und muss bei einer Schachtelung gegebenenfalls mehrfach ausgeführt werden, was die Komplexität weiter erhöht.

Technische Probleme: Vor allem bei Virtualisierern tritt das Phänomen auf, dass diese aus technischen Gründen, die in der unvollständigen Isolation der emulierten Umgebung von der Hostumgebung begründet liegen, überhaupt nicht ineinander geschachtelt ausführbar sind. Details hierzu finden sich in [von Suchodoletz, 2009, Kap. 5.2].

Performanceeinbußen: Sollen die durch unvollständige Isolation auftretenden Probleme umgangen werden, so muss die gesamte Hardwareumgebung, inklusive Prozessor, emuliert werden. Dies ist jedoch mit erheblichen Performancenachteilen verbunden, die sich bei einer Schachtelung aufsummieren. Möglicherweise ist dieser Punkt aber vernachlässigbar, bedingt durch den raschen technischen Fortschritt mit einhergehender Erhöhung der Rechenleistung.

Aufgrund der genannten Nachteile ist eine Schachtelung von Emulatoren gegenüber ihrer Migration sicherlich nur zweite Wahl, und wird in dieser Arbeit auch nur eine untergeordnete Rolle spielen.

2.1 Sekundärobjekte

Bisher wurden die zu archivierenden digitalen Objekte häufig als „Primärobjekte“ bezeichnet, es handelt sich dabei um Objekte von primärem archivarischem Interesse. Die Emulationsstrategie setzt darüber hinaus jedoch eine Vielzahl von unterstützenden Komponenten voraus, die zur Darstellung des Primärobjektes nötig sind. Diese Objekte sollen im Folgenden als „Sekundärobjekte“ bezeichnet werden. Deren Verwaltung ist die zentrale Aufgabe eines Softwarearchives im Langzeitarchivierungskontext, es lohnt sich daher, die wichtigsten Typen solcher Objekte genauer zu betrachten. Beachtung verdienen dabei unter anderem die Beziehungen der einzelnen Objekte untereinander, wie zum Beispiel Abhängigkeiten[11].

2.1.1 Anwendungsprogramme

Anwendungsprogramme, oder Applikationen, sind im Sinne der Langzeitarchivierung das primäre Werkzeug zur Darstellung beziehungsweise Migration von archivierten digitalen Objekten.

[11] Der Begriff „Abhängigkeit“ bleibt hier noch etwas vage, für detailliertere Betrachtungen siehe Kapitel 3.2.1.

Es ist dabei bei weitem nicht eindeutig, welches Programm zur Darstellung eines bestimmten Objekttyps zum Einsatz kommt. Natürlich ist, wann immer möglich, bei der Auswahl eines Anwendungsprogramms dasjenige zu bevorzugen, mit welchem das Objekt erstellt wurde. Jedoch gibt es gerade für populäre Dateiformate oft mehrere Anwendungen, die mit einem bestimmten Format umgehen können. Ein bekanntes Beispiel hierfür ist die Office-Suite OpenOffice, die als Open-Source Alternative zum bisherigen Quasi-Standard Microsoft Office entstanden ist. OpenOffice ist in der Lage, die von Microsoft Office erzeugten Dokumente zu öffnen, darzustellen und Dokumente in diesem Format abzuspeichern. Gleichzeitig zeigt dieses Beispiel die oft nur unvollständige Unterstützung durch Alternativ-Applikationen. Gerade bei seltener verwendeten Optionen im Datenformat kann es zu unvollständiger oder fehlerhafter Darstellung der Objekte durch Programme von Drittherstellern kommen. Dies gilt insbesondere für proprietäre, nicht öffentlich dokumentierte Formate, wie es die Microsoft Office Dateiformate lange Zeit waren[12]. Dritthersteller oder Open-Source-Projekte können diese Formate nur durch fehlerträchtiges Reverse-Engineering entschlüsseln, was eine vollständige Unterstützung extrem erschwert. Für eine Langzeitarchivierung ist daher offen dokumentierten und frei implementierbaren[13] Datenformaten der Vorzug zu geben, und gegebenenfalls eine Migration hin zu solchen Formaten in Betracht zu ziehen.

Schließlich stellt sich bei Vorhandensein mehrer passender Alternativ-Applikationen die Frage, welche nun konkret zur Darstellung des Objekts genutzt wird. Idealerweise sollte diese Auswahl automatisch getroffen werden. Ein Archivnutzer wird sich nämlich im Allgemeinen nicht mit den Vor- und Nachteilen der angebotenen Alternativen auskennen, wobei sich dieses Problem mit zunehmendem Alter des Objekts verschärft. Die Auswahl muss also von der die Applikationen verwaltenden Komponente, dem Softwarearchiv, getroffen werden. Hierzu könnten Metriken definiert werden, die beispielsweise die Darstellungsgüte für ein gegebenes Objekttyp/Applikations-Paar abbilden. Offen bleibt dabei natürlich die Frage, wie die Darstellungsgüte in einem konkreten Fall bestimmt wird. Vorstellbar wären entweder ein Userfeedback direkt bei Benutzung, oder eine entsprechende Angabe durch den Archivverwalter beim Einstellen des Anwendungsprogramms ins Softwarearchiv.

Für das Softwarearchiv von Bedeutung ist jedoch auch noch ein weiterer Gesichtspunkt, die Frage der Installation eines Anwendungsprogrammes. Die meisten Applikationen liegen zunächst nicht in direkt benutzbarer Form vor, sondern auf Installationsdatenträgern oder in Form einer einzelnen Installationsdatei[14]. Erst durch einen in der Nutzungsumgebung durchzuführenden Installationsvorgang wird die Applikation auf das aktuelle System angepasst, und damit in einen für Objektdarstellungen nutzbaren Zustand gebracht. Mit dieser Anpassung ist jedoch eine Kopplung an die aktuelle Systemumgebung verbunden, so dass zur Archivierung auf jeden Fall der Installationsdatenträger beziehungsweise die entsprechende Datei zu bevorzugen ist. Dies

[12] Mittlerweile hat Microsoft die vor Office 2007 verwendeten Formate offengelegt, seit Office 2007 basieren die Datenformate auf „Office Open XML“, welches als ECMA- und ISO-Standard offengelegt ist.

[13] Hier spielen unter anderem Patentfragen eine Rolle.

[14] Unter Windows kann es sich dabei um sogenannte „Windows Installer“ Dateien handeln, erkennbar an der Dateiendung „.msi“. Unter Linux sind Paketverwaltungssysteme wie das „RPM Package Management“ üblich.

bedingt jedoch, dass bei der Wiederherstellung einer Nutzungsumgebung aus den archivierten Einzelkomponenten dieser Installationsvorgang erneut durchgeführt werden muss. Wie dies automatisiert geschehen kann, wird in Kapitel 2.4 dargestellt.

2.1.2 Unterstützende Komponenten für Anwendungsprogramme

Mit Anwendungsprogrammen alleine lässt sich unter Umständen ein Primärobjekt noch nicht (vollständig) darstellen. Vielmehr sind diese Programme oft auf Zusatzkomponenten angewiesen, um ihre Aufgabe zufriedenstellend zu erfüllen. Einige dieser unterstützenden Komponenten sollen hier exemplarisch vorgestellt werden, vor allem auch um mögliche gegenseitigen Abhängigkeiten zu demonstrieren:

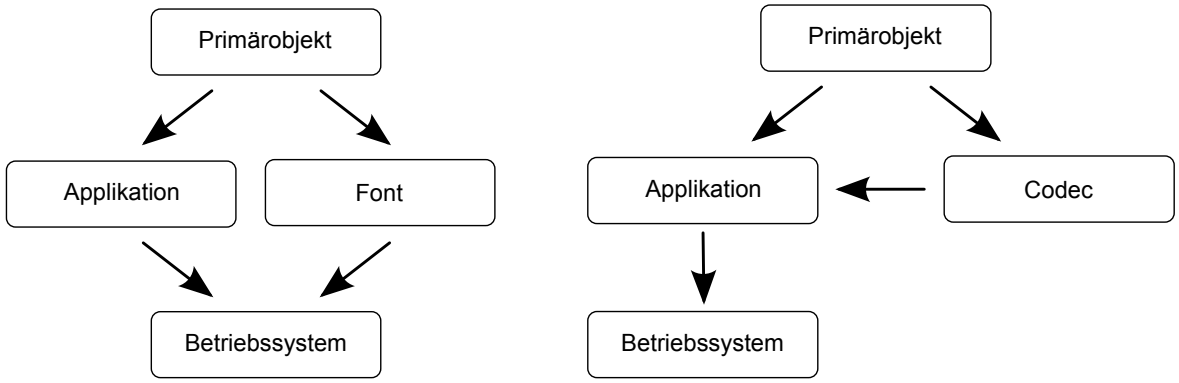
Fonts

Bei Fonts handelt es sich um digitale Objekte, meist ein oder mehrere Dateien, in denen eine Schriftart in elektronisch verarbeitbarer Form beschrieben ist. Sie sind für die Darstellung von Primärobjekten in unterschiedlichem Maße wichtig:

- Schriftarten, die auf demselben Alphabet basieren, sind meist ohne Informationsverlust untereinander austauschbar[15].
- Symbolschriftarten wie das unter Windows verbreitete Wingdings dienen dazu, Texte auf einfache Weise mit kleinen Grafikbausteinen anzureichern. Sie können bei Nichtvorhandensein bereits Verständnisschwierigkeiten auslösen.
- Dokumente, die nicht-lateinische Alphabete benutzen (z.B. kyrillisch), sind ohne entsprechende Fonts zumindest auf ASCII-basierten Systemen dann gar nicht mehr sinnvoll darstellbar.

Die meisten aktuellen Betriebssysteme implementieren eine eigene, von Applikationen unabhängige Font-Verwaltung, das heißt Fonts sind unabhängig von Anwendungsprogrammen installierbar und umgekehrt. Es besteht also keine direkte Abhängigkeit zwischen den beiden Komponenten, vielmehr ist die Darstellbarkeit des Primärobjektes abhängig einerseits vom entsprechenden Anwendungsprogramm, und andererseits den verwendeten Fonts (Abbildung 3(a)).

[15] Hier wird davon ausgegangen, dass die Information nur im Inhalt des Textes besteht und nicht in seiner Darstellung - eine Annahme die durchaus nicht immer gelten muss.



(a) Fonts (b) Codecs

Abbildung 3: verschiedene Abhängigkeitsgraphen

Codecimplementierungen

Codecs sind Verfahren zur Kodierung von Daten, meist Audio- oder Videodaten. Bekannte Beispiele für Codecs sind MPEG-2 und MPEG-4 für Videodaten, sowie MP3 und Vorbis für Audiodaten. Es ist jedoch wichtig zu verstehen, dass diese Codecs lediglich den Komprimierungsalgorithmus implementieren. Sie sind zu unterscheiden von Containerformaten wie AVI oder Ogg. Diese Formate definieren unter anderem, wie mehrere Audio- und/oder Videodatenströme in einer Datei zusammengefasst werden.

Anwendungsprogramme zur Behandlung von Audio- oder Videodaten, wie zum Beispiel die weit verbreiteten Mediaplayer, sind meist von vornherein mit Unterstützung für bestimmte Containerformate ausgestattet. Die Unterstützung von Codecs kann und muss jedoch oft durch Plugins nachgerüstet werden. Diese Plugins sind aus Sicht des Softwarearchivs eigenständige Sekundärobjekte, die auch getrennt archviert werden können. Allerdings besteht hier eine Abhängigkeit des Plugins von demjenigen Anwendungsprogramm, für das es geschrieben wurde. Der Begriff „Abhängigkeit“ ist dabei so zu verstehen, dass eine Installation des Plugins ohne vorherige Installation des Anwendungsprogramms nicht sinnvoll, und oft auch gar nicht möglich sein wird. Die Darstellbarkeit einer Audio- oder Videodatei ist hingegen abhängig sowohl von einer Applikation, die das Containerformat beherrscht, als auch den Codec-Plugins für die verwendeten Kompressionsalgorithmen (Abbildung 3(b)).

Manche Betriebssysteme, zum Beispiel aktuelle Windows-Versionen, bieten jedoch, ähnlich der Font-Verwaltung, eine applikationsunabhängige Codec-Verwaltung. Bei Windows ist dies die sogenannte DirectShow Architektur, die allgemein für Multimediaverarbeitung zuständig ist. Die Codecs werden in diesem Rahmen als sogenannte DirectShow Filter implementiert. Die Anwendungen greifen über eine standardisierte DirectShow API in generischer Weise auf die Codecs zu, und werden hierdurch von der Codecinstallation entkoppelt. Der Abhängigkeitsgraph stellt sich in diesem Fall analog zu den Abhängigkeiten bei Fonts dar: Die Darstellbarkeit des

Primärobjektes hängt ab von der Anwendung und dem Codec, die beiden Komponenten sind voneinander unabhängig.

Weitere Hilfskomponenten

Neben den bisher genannten gibt es noch eine Vielzahl weiterer Hilfskomponenten, von denen hier nur einige kurz genannt werden sollen: So gibt es Werkzeuge, die einen virtuellen Drucker im Betriebssystem installieren, den Ausdruck jedoch in eine PDF-Datei umleiten. Solche virtuellen Drucker können für Migrationen sehr hilfreich sein. Packprogramme sind ein weiteres sehr verbreitetes Werkzeug um Dateien verlustfrei zu komprimieren. Bekannte Beispiele sind diverse RAR- und ZIP-Implementierungen. Sie können für den Transport von Primärobjekten von der Hostumgebung in die emulierte Nutzungsumgebung (Kapitel 2.3) sehr hilfreich sein, da der Platz beispielsweise auf virtuellen Disketten begrenzt ist.

2.1.3 Virtuelle Maschinen und Skriptsprachen

Unter unixartigen Betriebssystemen schon seit längerer Zeit verbreitet, findet mit der Entwicklung von Java und Microsofts .NET[16] auch unter Windows eine weitere wichtige Klasse von Systemkomponenten immer größeren Zuspruch: virtuelle Laufzeitumgebungen sowie Skriptspracheninterpreter. Bekannte Beispiele sind neben den genannten Java und .NET, die den Laufzeitumgebungen zuzurechnen sind, Skriptsprachen wie Perl und Python. Diese bilden eine zusätzliche Abstraktionsschicht zwischen dem Anwendungsprogramm einerseits, und dem Betriebssystem sowie der Hardware andererseits.

Die beiden Subtypen erreichen diese Abstraktion auf unterschiedliche Weise. Während Programme in Skriptsprachen zur Laufzeit Schritt für Schritt durch einen Interpreter abgearbeitet werden, werden Java oder .NET Programme kompiliert. Das Kompilat besteht jedoch nicht aus Maschinencode für eine spezielle Prozessorarchitektur, sondern aus einem Zwischencode für eine virtuelle Maschine, sogenanntem Bytecode[17]. Die Abstraktion vom Betriebssystem erfolgt dadurch, dass vom Betriebssystem angebotene Funktionen, wie Dateioperationen, durch eigene Konstrukte der Programmiersprache ersetzt werden, die dann intern auf die Betriebssystemroutinen abgebildet werden.

Das erklärte Ziel dieser Systeme ist eine erhöhte Plattformabhängigkeit, wobei dieser Begriff von verschiedenen Herstellern durchaus unterschiedlich interpretiert wird: Während Microsoft darunter hauptsächlich die Tatsache versteht, dass .NET Programme auf unterschiedlichen Windows-Plattformen lauffähig sind[18], existieren Java-Laufzeitumgebungen unter anderem für diverse Windows-, Unix- und MacOS-Varianten.

[16] sprich: „Dotnet“

[17] bei .NET auch CIL, „Common Intermediate Language“ genannt

[18] Es existiert mit dem Mono-Projekt zwar eine .NET Implementierung für Unix, diese ist jedoch im Funktionsumfang weit vom Original entfernt

Eine solche Zusatzschicht bringt natürlich den Nachteil einer erhöhten Fehleranfälligkeit mit sich, zum Beispiel für Versionsprobleme. Dennoch macht die Plattformabhängigkeit virtuelle Laufzeitumgebungen und Skriptsprachen für die Emulationsstrategie sehr interessant: So ist nur ein Emulator für eine einzige der unterstützten Hardwarearchitekturen, und nur ein einziges der unterstützten Betriebssysteme nötig, um die gesamte Menge der für diese Laufzeitumgebung entwickelten Anwendungsprogramme zu erschließen. Damit verringert sich der Archivierungsaufwand für die Emulatoren, zumal die Nutzung eines zusätzlichen Emulators, als zentralem Teil der Emulationsstrategie, auch über die reine Archivierung hinaus erheblichen Aufwand erzeugt (siehe hierzu Kapitel 3.1.4).

Aus Sicht der Abhängigkeiten stellt sich die Situation somit wie folgt dar: Das Anwendungsprogramm besitzt nur noch eine Abhängigkeit von der Laufzeitumgebung, diese wiederum hängt ab vom Betriebssystem und der Hardwareumgebung (beziehungsweise einem entsprechenden Emulator). Eine direkte Abhängigkeit der Applikation von Hardware oder Betriebssystem existiert nicht.

2.1.4 Betriebssysteme

Auch nativ für eine bestimmte Prozessorarchitektur kompilierte Anwendungsprogramme laufen so gut wie nie direkt auf der Hardware ab. Ausnahmen bilden nur sehr frühe Großrechnersysteme und die in den 80er Jahren verbreiteten Heimcomputer. In sämtlichen heute verbreiteten Architekturen hingegen haben Betriebssysteme die Aufgabe, eine Ablaufumgebung für Programme zu bieten. Sie abstrahieren von der Hardware mit Hilfe von Gerätetreibern, und stellen den Programmen standardisierte APIs zum Zugriff auf die Hardware und darüber hinaus für Aufgaben auf höherem Abstraktionsniveau zur Verfügung, wie zum Beispiel Dateizugriff oder Netzwerkkommunikation. Ferner verwalten sie die Ausführung von Programmen.

Die bekanntesten Beispiele für Betriebssysteme der X86-Architektur sind Linux, Windows und MacOS, wobei manche davon auf mehreren Hardwarearchitekturen ablauffähig sind. Vor allem Linux tut sich hier hervor, es existieren Linux-Varianten für Embedded-Systeme ebenso wie für Großrechner. Es ist jedoch wichtig zu beachten, dass Betriebssysteme, im Gegensatz zu den bereits besprochenen Laufzeitumgebungen, die Anwendungsprogramme nicht vollständig von der Hardware isolieren. Die Programme werden weiterhin für eine konkrete Prozessorarchitektur, z.B. X86 oder Sparc, übersetzt, und sind auch nur auf dieser Architektur lauffähig.

2.1.5 Treiber für virtuelle Hardware

Wie bereits im letzten Kapitel kurz erwähnt, sprechen moderne Betriebssysteme die Hardware meist nicht direkt an. Stattdessen definieren sie allgemeine Schnittstellen, zum Beispiel für Soundausgabe, die dann von Programmen genutzt werden können. Spezielle Module, sogenannte Treiber, übernehmen dann die Implementierung dieser Schnittstellen. Sie sind speziell an die

konkrete Hardware angepasst, und werden oft von den Hardwareherstellern entwickelt und bereitgestellt.

Emulatoren simulieren meist real existierende Hardware, so dass Treiber für diese Hardware genutzt werden können. Es ist also bei Auswahl und Konfiguration eines Emulators darauf zu achten, dass möglichst weit verbreitete Hardware emuliert wird, da für diese mit hoher Wahrscheinlichkeit Treiber für viele unterschiedliche Betriebssysteme existieren.

2.1.6 Emulatoren

Emulatoren stellen die zentrale Komponente in der Emulationsstrategie dar, und verdienen daher besondere Beachtung. Zunächst soll kurz ein Überblick über verschiedene Typen von Emulatoren gegeben werden:

Betriebssystem-Emulatoren: Ein Betriebssystem-Emulator erlaubt es, für ein bestimmtes Betriebssystem geschriebene Programme unter einem anderen Betriebssystem ablaufen zu lassen. Ein Beispiel hierfür ist der verbreitete Emulator Wine[19], der die Ausführung von Windows-Programmen unter anderen unter Linux ermöglicht. Ein solcher Emulator bildet die API des emulierten Betriebssystems auf die vom Hostsystem zur Verfügung gestellten Schnittstellen ab. Dies gelingt jedoch häufig nur unvollständig, was die Menge der ausführbaren Programme stark einschränkt. Außerdem findet keinerlei Abstraktion der physikalischen Hardware statt, der Ansatz ist also für die Langzeitarchivierung eher uninteressant.

Virtualisierer: Das prominenteste Beispiel dieser Klasse von Emulatoren ist sicherlich das vom gleichnamigen Hersteller vertriebene VMware[20]. Ein Virtualisierer stellt der emulierten Umgebung virtuelle Hardware, vor allem I/O-Geräte und Speicher bereit. Der Prozessor hingegen wird nicht virtualisiert, stattdessen werden die Maschinenbefehle direkt an den physikalischen Prozessor durchgereicht. Virtualisierer ermöglichen unter anderem die Ausführung alter Betriebssysteme, für die keine Treiber für die Hardware der Hostumgebung existieren.

Hardwareemulatoren: Hardwareemulatoren stellen den für die Langzeitarchivierung am flexibelsten nutzbaren Ansatz dar. Sie simulieren eine Hardwareumgebung vollständig, inklusive des Prozessors. Daher lässt sich auch, durch Neuübersetzen und gegebenenfalls eine Portierung des Emulators auf andere Hostplattformen, ein Plattformwechsel für Objekte der Nutzungsumgebung vollständig transparent bewerkstelligen. Eine gute Darstellung dieses Prinzips findet sich in [Holdsworth und Wheatley, 2001] . Die folgenden Betrachtungen werden sich daher ausschließlich auf Hardwareemulatoren beziehen. Als gutes Beispiel für die Klasse der Hardwareemulatoren kann das Open-Source-Projekt QEMU gelten.

[19] Homepage des Emulatorprojekts: `http://www.winehq.org`
[20] Homepage des Herstellers: `http://www.vmware.com`

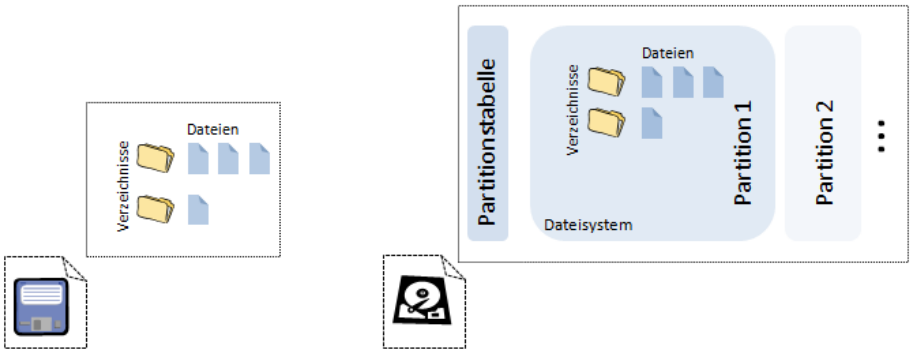


Abbildung 4: Interner Aufbau einer Diskette und einer Festplatte (aus [Welte, 2008])

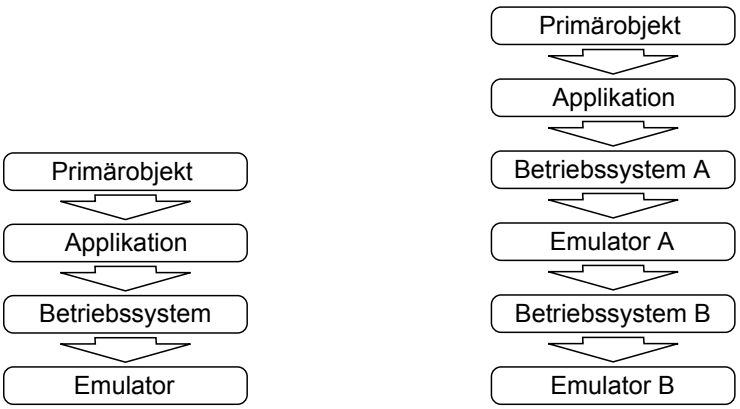
Für den Entwurf eines Softwarearchivs verdienen nun einige Aspekte besondere Beachtung:

Einige Emulatoren bieten Konfigurationsmöglichkeiten bezüglich der emulierten Hardware. So akzeptiert der Emulator QEMU beim Aufruf Kommandozeilenparameter, welche die emulierte Prozessorarchitektur und an das virtuelle System angeschlossene simulierte Disketten- und Festplattenlaufwerke festlegen. Sollen nun standardisierte, oder gar automatisierte, Abläufe innerhalb der emulierten Umgebung stattfinden, so setzen diese eine ganz bestimmte Umgebung voraus. Damit die Abläufe später wiederholt werden können, muss der Kontext, bestehend aus der Emulatorkonfiguration, in geeigneter Weise im Softwarearchiv abgelegt werden.

Ein weiterer wichtiger Gesichtspunkt ist die Frage, wie Emulatoren der simulierten Umgebung Festspeicher zur Verfügung stellen. Seit vielen Jahren haben sich blockorientierte Geräte als Standard für Festspeicher durchgesetzt. Emulatoren nutzen Containerdateien im Hostsystem, deren Inhalt der emulierten Umgebung als Blockgerät präsentiert wird. Bei virtuellen Disketten- und optischen Laufwerken wird der Dateiinhalt dabei meist eins zu eins abgebildet, es können also beispielsweise ISO-Images als virtuelles CDROM-Laufwerk eingebunden werden[21]. Aufgrund der komplexeren internen Struktur (siehe Abbildung 4), nutzen Emulatoren für virtuelle Festplatten jedoch ein spezielles, emulatorspezifisches Format. Praktische Experimente in der Vergangenheit haben nun gezeigt, dass die Installation eines Betriebssystems in einer emulierten Umgebung nicht trivial, und schon gar nicht automatisiert durchführbar ist. Ein reines Ablegen des Installationsdatenträgers eines Betriebssystems im Archiv wird also nicht ausreichen. Stattdessen wird in dieser Arbeit der Ansatz verfolgt, Containerdateien mit einem vorinstallierten Betriebssystem zu archivieren. Im Folgenden sollen diese Containerdateien „Betriebssystem-Images“ genannt werden.[22]

[21] Siehe hierzu auch Kapitel 2.3.

[22] Eine konkrete Installation eines Betriebssystems setzt im Allgemeinen eine ganz bestimmte Hardwarekonfiguration voraus. Umso wichtiger ist die weiter oben erwähnte Archivierung der Emulatorkonfiguration.



(a) einfach (b) geschachtelt

Abbildung 5: Beispiele für View-Paths

2.2 View-Paths

Im letzten Kapitel wurden einige häufig vorkommende Arten von Softwarekomponenten vorgestellt. Diese sollen nun in einen Gesamtzusammenhang gebracht werden.

Dieser Gesamtzusammenhang wird durch ein zentrales Konzept in der Emulationsstrategie hergestellt, das Konzept des „View-Path“ [van Diessen und Steenbergen, 2002, Kap. 6]. Dieses ist auch für die Konzeption eines Softwarearchivs von großer Bedeutung. Zum Verständnis sollte man sich kurz die grundlegende Problematik der Langzeitarchivierung ins Gedächtnis rufen, nämlich die Tatsache, dass archivierte digitale Objekte nicht direkt in der Anwendungsumgebung des Betrachters dargestellt werden können. Der Ansatz der Emulationsstrategie ist es, eine virtuelle Nutzungsumgebung zur Darstellung des Objektes herzustellen. Ein View-Path ist nun ein Darstellungspfad, der vom Primärobjekt zur aktuellen Anwendungsumgebung des Betrachters zeigt. Er enthält üblicherweise - in dieser Reihenfolge - eine (Betrachtungs-)Applikation, ein Betriebssystem sowie einen Hardwareemulator (Abbildung 5(a)). Diese Aufzählung zeigt bereits, dass man View-Paths durchaus als Schichtung dieser Komponententypen betrachten kann. Dies ist zur Veranschaulichung zwar sehr hilfreich, eine strenge Einteilung in vorgegebene Schichten ist jedoch nicht sinnvoll. So können manche Schichten entfallen, Homecomputer beispielsweise besitzen üblicherweise kein Betriebssystem. Ebenso gibt es Komponenten, die nicht einer der oben genannten Schichten zugeordnet werden können, etwa virtuelle Laufzeitumgebungen oder Codecs. Bei einer Schachtelung von Emulatoren können sogar unterhalb der Emulator-Schicht noch weitere Schichten eingeführt werden (Abbildung 5(b)).

Eine wichtige Unterscheidung muss jedoch getroffen werden: View-Paths beschreiben nicht zwangsläufig Abhängigkeiten. So können Abhängigkeiten durch einen allgemeinen gerichteten, azyklischen Graphen beschrieben werden; zwei Beispiele hierfür zeigt Abbildung 3 auf Seite 9. Ein View-Path hingegen ist eine geordnete Liste, die eher als Installationsreihenfolge zu ver-

stehen ist. Die Erzeugung eines View-Paths aus gegebenen Abhängigkeiten könnte durch eine topologische Sortierung des Abhängigkeitsgraphen geschehen[23].

Da die topologische Sortierung nicht eindeutig ist, werden im allgemeinen mehrere View-Paths zu einem gegebenen Objekttyp existieren, selbst wenn nur eine einzige Applikation zur Darstellung des Objekttyps vorhanden ist. Stehen mehrere solcher Applikationen zur Verfügung, und lässt man bei Abhängigkeiten Alternativen zu, so wird sich die Zahl der View-Paths für einen gegebenen Objekttyp weiter erhöhen. Dies ist einerseits wünschenswert, da so eine gewisse Redundanz gewährleistet werden kann, andererseits müssen die entstehenden Mehrdeutigkeiten in geeigneter Weise aufgelöst werden.

Damit zusammenhängend stellt sich auch die Frage nach dem Auswahlkriterium für View-Paths. In bisherigen Arbeiten, etwa [van Diessen und Steenbergen, 2002], wird als Auswahlkriterium alleine das Format des Primärobjektes herangezogen. Gerade die in Kapitel 2.1.2 dargestellte Font- und Codec-Problematik zeigt jedoch, dass eine Auswahl alleine über das Objektformat möglicherweise nicht ausreichend ist. Zumindest müsste die Definition des Objektformates in diesem Fall sämtliche im Objekt verborgenen Abhängigkeiten abbilden. Um die Komplexität nicht unnötig zu erhöhen, wird in dieser Arbeit dennoch von einer einfachen Format-Definition ausgegangen, der PRONOM-UID[24].

2.3 Datenaustausch zwischen Host- und Nutzungsumgebung

Bisher wurden die Komponenten einer emulierten Nutzungsumgebung sowie ihre Kombination in View-Paths ausführlich dargestellt. Für eine sinnvolle Verwendung dieser Komponenten müssen jedoch auch Daten zwischen Host- und Nutzungsumgebung ausgetauscht werden können. So muss zur Darstellung eines Primärobjektes dieses vom Hostsystem in die Nutzungsumgebung hineintransportiert werden. Soll eine Migration durchgeführt werden, so ist das Ergebnis der Migration außerdem auf geeignete Weise wieder zurück in die Hostumgebung zu transferieren, um es dort weiterzuverarbeiten. Außerdem muss, wenn Komponenten wie etwa Anwendungsprogramme in der Nutzungsumgebung installiert werden sollen, die Installationsquelle dem virtuellen System zugänglich gemacht werden. Je nach Emulator werden hierfür unterschiedliche Methoden angeboten, die wichtigsten sollen im Folgenden kurz vorgestellt werden:

[23] Genaueres dazu in Kapitel 3.2.2

[24] PRONOM ist eine Format-Registry des Britischen Nationalarchivs, die einer Vielzahl von Formaten unter anderem einen eindeutigen Bezeichner zuordnet: `http://www.nationalarchives.gov.uk/PRONOM/Default.aspx`

Virtuelle Blockgeräte

Blockgeräte haben sich seit vielen Jahren als Standard für Festspeicher durchgesetzt. Beispiele hierfür sind neben Festplatten vor allem optische Datenträger[25], und bis vor einigen Jahren die Diskette. Eine neuere Entwicklung sind USB-Sticks, die mittlerweile das Erbe der Diskette als einfach wiederbeschreibbares Transportmedium angetreten haben. Diese Blockgeräte existieren im Hostsystem als Datei, deren Inhalt die Blockstruktrur des Gerätes darstellt. Sie können in der Hostumgebung, entweder mit Mitteln des Betriebssystems oder externen Werkzeugen (unter Linux beispielsweise `lomount`), als virtuelles Blockgerät eingebunden, und anschließend beschrieben oder gelesen werden. Zur Erzeugung von ISO-Images für optische Datenträger existieren ebenfalls zahlreiche Werkzeuge für die unterschiedlichsten Betriebssyteme, unter Anderem das Programm `mkisofs` für Linux. Ein erfolgreicher Datenaustausch zwischen Host- und Nutzungsumgebung mittels solcher Blockgeräte setzt natürlich ein Dateisystem voraus, welches sowohl von der Host-, als auch der emulierten Nutzungsumgebung gelesen und geschrieben werden kann. Ein solches Dateisystem muss, vor allem bei zunehmendem Alter der Nutzungsumgebung, nicht zwangsläufig existieren. Es wird, der Einfachheit halber, im Folgenden dennoch als gegeben betrachtet.

Netzwerk

Die meisten Emulatoren bieten virtuelle Netzwerkadapter an; so emuliert QEMU den früher weit verbreiteten NE2000 Chipsatz, zu dem auch heute noch kompatible Chipsätze existieren. Für die Kopplung des virtuellen Adapters an das Hostsystem kommen zahlreiche unterschiedliche und emulatorspezifische Methoden, wie Kernelmodule zum Einsatz. Allen gemeinsam ist jedoch, dass das Gastsystem im Hostsystem als weiterer Netzwerkknoten erscheint, ebenso umgekehrt.

Voraussetzung für eine erfolgreiche Kommunikation sind - analog zu den Dateisystemen bei Blockgeräten - Protokolle, die sowohl vom Host- als auch vom Gastsystem unterstützt werden. Die erste Wahl ist hier das seit 1970 im Einsatz befindliche IPv4, das heutzutage eine überragende Bedeutung erlangt hat. Auf der Anwendungs-Schicht des OSI-Modells sind es vor allem Datenaustausch-Protkolle wie FTP, NFS oder SMB, die für den Objekttransport interessant sind. Diese haben gegenüber den Blockgeräten den Vorteil eines erheblich flexibleren und umkomplizierteren Datenaustausches, auch bei laufender Nutzungsumgebung. Dieser Vorteil wird jedoch durch eine erheblich komplexere Konfiguration erkauft.

Emulatorspezifische Methoden

Über die bereits genannten Methoden hinaus bieten manche Emulatoren noch spezifische Austauschmethoden an. So existiert beim Virtualisierer VMware das Konzept der „Shared Folders“.

[25] Optische Datenträger haben jedoch die Einschränkung, dass sie in der Nutzungsumgebung meist nicht beschreibbar sind.

Dadurch können Ordner des Hostsystems im Gastsystem als Netzwerkdatenträger angesprochen werden. Diese Methoden bedingen jedoch eine hohe Kopplung zwischen Hostsystem, Gastsystem und dem Emulator, und sind bei größerem technologischen Abstand zwischen Gast- und Hostsystem meist nicht mehr verfügbar [von Suchodoletz, 2009, Kap. 6.8.5].

2.4 Interaktivitätsproblem und Aufzeichnungen

Soll ein Softwarearchiv für die Emulationsstrategie umgesetzt werden, so steht man früher oder später zwangsläufig vor der Aufgabe, Sekundärobjekte automatisiert zu steuern (für Details hierzu siehe die Kapitel 3.1.1 und 3.1.2). Dies betrifft sowohl Emulatoren, als auch Objekte in der emulierten Nutzungsumgebung. Während jedoch viele Emulatoren einer Automation innerhalb der Hostumgebung, etwa über Kommandozeilenparameter, zugänglich sind, ist diese Aufgabe bei Objekten der Nutzungsumgebung, wie Anwendungsprogrammen, eine große Herausforderung.

In [Rechert u. a., 2009] werden einige mögliche Verfahren zur Lösung des dort „Interactivity Problem“ genannten Problems diskutiert (siehe auch [Rechert u. a., 2010]): Es böte sich einerseits die Möglichkeit, die Anwendungsprogramme der Nutzungsumgebung nachträglich um Automationsschnittstellen zu erweitern. Dies sei jedoch im Allgemeinen sehr komplex, und manchmal sogar unmöglich, da Quellcode und notwendiges Wissen über die internen Strukturen der Anwendung nicht mehr vorhanden seien. Als zweite Option werden Makrorecorder genannt. Dies sind spezialisierte Werkzeuge oder Funktionen von Anwendungsprogrammen oder Betriebssystemen, die in der Lage sind, Benutzerinteraktionsereignisse wie das Öffnen einer Datei, aufzuzeichnen. Dieser Ansatz ist nach Ansicht der Autoren jedoch ebenfalls ungeeignet: „However, this functionality is not standardized, differs in its usability and features. Special tools might be needed and the knowledge of the applications and operating systems is required.“

Die Autoren schlagen daher einen neuartigen Ansatz vor, der dem in dieser Arbeit vorgestellten Konzept für ein Softwarearchiv zugrunde liegt, und im Folgenden ausführlicher vorgestellt werden soll.

Die Grundidee ist es, Benutzer-Ereignisse wie Maus- und Tastatur-Events nicht innerhalb der emulierten Nutzungsumgebung aufzuzeichnen, sondern in der Hostumgebung. Damit diese Aufzeichnung später automatisiert abgespielt werden kann, muss jedes Event mit einer Vorbedingung verknüpft werden, die erfüllt sein muss, bevor das Event ausgelöst werden kann. So muss zum Beispiel mit einem simulierten Mausklick auf den OK-Button eines Dialogfensters solange gewartet werden, bis der entsprechende Dialog auf dem Bildschirm angezeigt wird. Weiterhin muss durch definierte Nachbedingungen festgestellt werden, ob das ausgelöste Ereignis den gewünschten Effekt erzielt hat, im Beispiel also das Dialogfenster geschlossen wurde.

Werden die Arbeitsabläufe innerhalb der Nutzungsumgebung interaktiv durchgeführt, so erfolgen diese Kontrollen optisch durch den Benutzer. Bei automatischer Ausführung muss diese

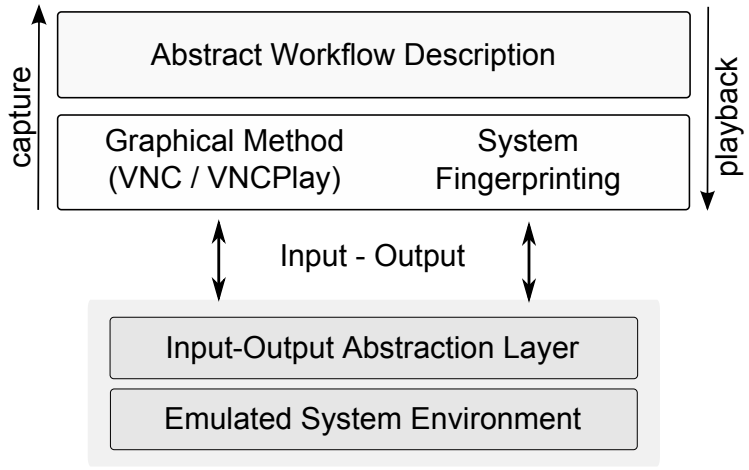


Abbildung 6: Funktionsweise der Aufzeichnung von Benutzerinteraktionen (nach [Rechert u. a., 2009])

optische Kontrolle durch andere Mechanismen ersetzt werden. Je nach Art der Nutzungsumgebung und Fähigkeiten des Emulators können hierzu unter anderem der aktuelle Bildschirminhalt, der Zustand der emulierten Prozessorregister oder des emulierten Hauptspeichers herangezogen werden.

In der Frage der technischen Basis für diese Aufzeichnungsmethode hat sich in Experimenten die auf dem RFB-Protokoll[26] aufbauende Fernwartungssoftware VNC, und hier insbesondere die Implementierung VNCPlay[27], als vielversprechende Option erwiesen. VNC erlaubt es, den Bildschirminhalt eines entfernten Rechners in einem lokalen Client darzustellen, und lokale Tastatur- und Maus-Ereignisse an den entfernen Rechner zu übertragen.

Bei VNCPlay handelt es sich um einen spezialisierten VNC-Client, mit dem interaktive VNC-Sitzungen aufgezeichnet, und später wieder abgespielt werden können (Abbildung 7). Als Vor- und Nachbedingungen, die, wie bereits beschrieben, zur Synchronisation benötigt werden, nutzt VNCPlay sogenannte Sync-Points. Dies sind rechteckige Abbilder eines kleinen Teils des Bildschirminhalts um den Mauszeiger herum. Die Aufzeichnung erfolgt bisher noch zeitbasiert, läuft also beim Abspielen nahezu in der gleichen Zeit ab. Abweichungen davon können durch eine hohe Last mit entsprechend verzögerter Reaktion der Benutzeroberfläche auftreten. In [Ruzzoli, 2010] wurde jedoch eine Erweiterung von VNCPlay implementiert, welche die Aufzeichnungen zustandsbasiert durchführt. Dies erlaubt unter anderem ein deutlich beschleunigtes Abspielen der Aufzeichnungen, sowie verbesserte Fehlerbehandlung durch direktes Springen zu bestimmten Punkten in der Aufzeichnung.

Neben dem das RFB-Protokoll implementierenden VNC-Client wird ein entsprechender Server benötigt, der den Bildschirminhalt der Nutzungsumgebung an den Client übertragen, und

[26] RFB=„Remote Framebuffer Protocol“
[27] Homepage des Open-Source-Projekts: `http://suif.stanford.edu/vncplay`

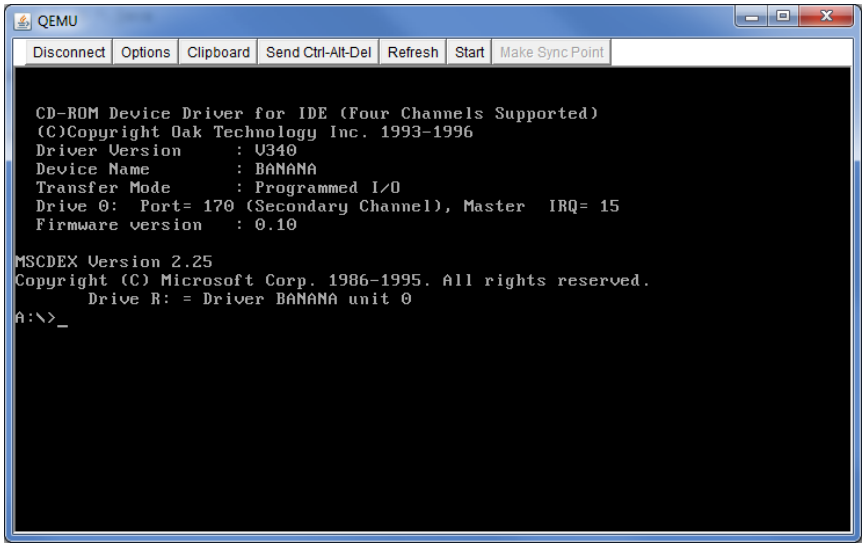


Abbildung 7: Benutzeroberfläche des VNCPlay Clients

vom Client gesendete Eingabe-Ereignisse an die Nutzungsumgebung weiterreichen muss. Die naheliegendste Lösung ist hierbei die Verwendung eines eigenständigen VNC-Servers, der mit der grafischen Ausgabe des Emulators sowie dessen Eingabeschnittstellen gekoppelt wird. Der in den Experimenten verwendete Open-Source Emulator QEMU bietet jedoch noch eine weitere, sehr elegante Möglichkeit: Er enthält bereits einen eingebauten VNC-Server, der die Nutzungsumgebung direkt für VNC-Clients wie VNCPlay zugreifbar macht.

Für die Implementierung eines Softwarearchives ist der VNCPlay Client insofern von Interesse, als für das Archiv auch eine grafische Benutzeroberfläche, oder GUI[28], bereitgestellt werden soll. Das GUI soll unter anderem die Aufzeichnung einer Softwareinstallation ermöglichen, damit diese später zusammen mit der Softwarekomponente archiviert werden kann. Da das Softwarearchiv auch remote, also über ein Netzwerk benutzbar sein soll, bietet sich eine Web-basierte Lösung an. Hier kommt nun eine weitere Fähigkeit von VNCPlay zum tragen, nämlich die Option, als Applet in Webseiten eingebettet abzulaufen. Der VNC-Server, zu dem die Verbindung aufgebaut werden soll, wird dabei als Parameter im einbettenden HTML-Code spezifiziert, dieser kann wiederum von einem Webserver dynamisch erzeugt werden. Probleme verursachen hierbei die Aufzeichnungen. So versucht derzeit das VNCPlay Applet die Aufzeichnungen als Datei im lokalen Dateisystem abzulegen[29]. Da jedoch der Browser des Benutzers sehr wahrscheinlich nicht auf demselben System wie das Softwarearchiv laufen wird, ist eine lokal vorliegende Aufzeichnungsdatei wenig hilfreich. Als Abhilfe ist eine entsprechende Modifikation des VNCPlay Applets vorstellbar, so dass die Aufzeichnung direkt an den Webserver übertragen wird.

[28] GUI=„Graphical User Interface“

[29] Dies wäre nach dem Java-Sicherheitsmodell durchaus zulässig, sofern das Applet digital signiert wurde.

3 Softwarearchiv im Kontext Langzeitarchivierung

Nach dem im letzten Kapitel gegebenen Überblick über die Emulationsstrategie in der Langzeitarchivierung, soll hier ein Konzept für ein Softwarearchiv entwickelt werden. Im nächsten Kapitel wird dann eine beispielhafte Implementierung für dieses Konzept vorgestellt.

Zur Entwicklung des Konzepts sollen zuerst die Aufgaben skizziert werden, die dem Softwarearchiv in einer Emulationsstrategie zukommen. Danach werden zwei unterschiedliche Konzepte diskutiert: Zuerst wird die dynamische Erzeugung von View-Paths diskutiert, die ein theoretisch sehr ansprechendes Konzept darstellt, jedoch einige praktische Probleme mit sich bringt. Basierend auf diesen Erfahrungen wird als Alternative das Konzept der statischen Archivierung von View-Paths vorgestellt.

3.1 Aufgaben

Für die Diskussion der Aufgaben eines Softwarearchivs bei der Langzeitarchivierung lohnt es sich zunächst, sich die in Kapitel 2.1 getroffene Unterscheidung zwischen Primär- und Sekundärobjekten in Erinnerung zu rufen: Die Primärobjekte sind Objekte von primärem archivarischem Interesse, die Sekundärobjekte sind Objekte, die im Rahmen der Emulationsstrategie lediglich unterstützenden Charakter haben, beispielsweise Anwendungsprogramme zur Darstellung der Primärobjekte. Aus dieser Überlegung lassen sich zwei Hauptaufgaben eines Softwarearchivs ableiten: Einerseits muss das Archiv Ablaufumgebungen in Form von View-Paths bereitstellen, die die Darstellung oder Migration von Primärobjekten erlauben. Ob diese View-Paths bereits fertig im Archiv vorliegen, oder bei Bedarf dynamisch erzeugt werden, ist dabei nicht festgelegt. Es erscheint jedoch in jedem Fall sinnvoll, die einzelnen Komponenten des View-Paths, also die Sekundärobjekte, in getrennter Form zu archivieren um spätere Neukombinationen zu ermöglichen. Dies stellt die zweite Hauptaufgabe des Archivs dar.

3.1.1 Bereitstellung von Ablaufumgebungen

Die zentralen Elemente einer Ablaufumgebung sind ein Emulator sowie ein vom Emulator lesbares Betriebssystem-Image. Dieses enthält sämtliche weiteren Komponenten des View-Paths, die nötig sind, um die Darstellung oder Migration eines bestimmten Primärobjekttyps durchzuführen, beispielsweise installierte Anwendungsprogramme oder Fonts.

Mit diesen beiden Elementen könnten nun bereits Primärobjekte dargestellt und migriert werden. Der Nutzer des Archivs würde hierfür den Emulator unter Verwendung des Betriebssystem-Images starten, und das Primärobjekt mit einer der in Kapitel 2.3 beschriebenen Methoden in die Nutzungsumgebung transferieren. Daraufhin würde er innerhalb der emulierten Nutzungsumgebung die zur Darstellung oder Migration des Objektes nötigen Arbeitsschritte durchführen,

also beispielsweise die Datei in einem Betrachtungsprogramm öffnen oder ein Konvertierungsprogramm starten. Im Falle einer Migration müsste danach das Migrationsergebnis noch aus der Nutzungs- in die Hostumgebung zurücktransportiert werden.

Dieses Vorgehen würde dem Benutzer allerdings ein hohes Maß an Wissen über die Bedienung der Nutzungsumgebung und des Emulators abverlangen. Je größer der zeitliche Abstand zwischen Erstellungszeitpunkt des Objektes und Zugriffszeit ist, desto unwahrscheinlicher ist es jedoch, dass er über letzteres verfügt. Auch die manuelle Bedienung der Emulatoren dürfte in ihrer Komplexität, und aufgrund des Zeitaufwandes, dem Archivnutzer nicht zuzumuten sein. Bei der Darstellung von Primärobjekten ist vielmehr wünschenswert, den User durch möglichst weitgehende Automation zu unterstützen. Bei Objektmigrationen ist ein von User-Interaktion vollständig unabhängiger Ablauf sogar unerlässlich. Nur auf diese Art können große Archivbestände mit vertretbarem Arbeitsaufwand migriert werden.

Während die Emulatoren innerhalb der Hostumgebung automatisiert gesteuert werden können, kommen für die Automatisierung der in der emulierten Umgebung nötigen Arbeitsschritte nun die in Kapitel 2.4 beschriebenen Aufzeichnungen zum Einsatz. Für jede Migration oder Darstellung eines bestimmten Primärobjekttyps muss vom Archivverwalter eine solche Aufzeichnung erzeugt, und vom Softwarearchiv archiviert werden. Sie bildet schließlich, neben dem Emulator und dem Betriebssystem-Image, den dritten Teil einer Ablaufumgebung.

3.1.2 Archivierung einzelner View-Path-Komponenten

Ganz unabhängig von der Frage, ob die vom Archiv bereitgestellten Ablaufumgebungen dynamisch erzeugt oder bereits in aggregierter Form archiviert werden, ist eine Archivierung der einzelnen View-Path-Komponenten wünschenswert. Die Komponenten sollten dabei möglichst allgemein verwendbar sein. Bei der Installation, beispielsweise von Anwendungsprogrammen, werden diese jedoch oft auf die ganz konkrete Systemumgebung angepasst. Außerdem ist es in vielen Fällen sehr schwierig bis unmöglich, sämtliche Teile einer installierten Komponente vom restlichen System zu trennen[30]. Eine Archivierung der installierten Komponenten scheidet somit aus. Vielmehr sollten die Komponenten in installierbarer Form archiviert werden. Konkret kann hierzu beispielsweise der Inhalt des Installationsdatenträgers herangezogen werden. Aber auch anderen Formen sind denkbar, wie die direkte Archivierung einer einzelnen Datei im Falle eines RPM-Archivs[31]. Allen diesen Möglichkeiten ist gemeinsam, dass sie als Bytestream darstellbar sind, daher bietet sich diese Abstraktion als zu archivierendes Objekt an. Die daraus entstehende Frage, wie das konkrete, im Bytestream verborgene Format zu archivieren ist, wird später noch genauer erörtert.

[30] Ein gutes Beispiel hierfür sind die unter Windows von Anwendungsprogrammen in großer Zahl erzeugten Registry-Einträge.

[31] Der unter Linux weit verbreitete RPM Package Manager organisiert Komponenten, wie beispielsweise Anwendungsprogramme, in einzelnen Archivdateien.

Neben dem Bytestream sind, um das Image im Nachhinein wiederfinden zu können, beschreibende Metadaten mit abzulegen. Vorstellbar wären zum Beispiel Elemente des Dublin-Core Metadatenstandards[32], wie Titel und Hersteller. Unerlässlich sind außerdem Informationen, die zur Installation nötig sind, etwa Lizenzinformationen in Form von Seriennummern, die bei der Installation eingegeben werden müssen.

Darüber hinaus vorteilhaft ist eine Archivierung des Installationsprozesses der Softwarekomponente auf einem gegebenen Betriebssystem-Image, in Form einer Aufzeichnung. Die Gründe liegen, analog zu den obigen Überlegungen, in der Unterstützung des Benutzers bei eventuell nötigen Neuinstallationen der Softwarekomponente, oder gegebenenfalls deren vollautomatischer Durchführung. Die genaue Bedeutung dieser Aufzeichnung variiert jedoch je nach Art der Erzeugung der Ablaufumgebungen. Während bei dynamischer Generierung die Aufzeichnung vor allem für die vollautomatische Installation eine zentrale Rolle spielt, so liegt der Fokus bei statischer Archivierung der Ablaufumgebungen eher auf dem Unterstützungsaspekt.

In Hinblick auf die automatische Installation kommt dem Betriebssystem, als einer der View-Path-Komponenten, eine Sonderrolle zu. Da Betriebssysteme in besonders enger Art und Weise mit der durch den Emulator bereitgestellten virtuellen Hardware interagieren, ist eine Installation eine ausgesprochen anspruchsvolle Aufgabe, die nicht automatisiert durchführbar ist. Dies haben Experimente ergeben. Sie müssen also, im Gegensatz zu den anderen Komponenten, stets in bereits installierter Form als Containerdatei („Image“) archiviert werden.

3.1.3 Kontext für archivierte Objekte

Viele der archivierte Objekte sind von bestimmten Technologien abhängig:

- Beim Archivieren einer Softwarekomponente wird, wie beschrieben, der Installationsdatenträger sowie eine Aufzeichnung der Installation archiviert. Damit die Aufzeichnung beim späteren Abspielen korrekte Ergebnisse liefert, benötigt sie die Angabe des Typs des Installationsdatenträgers, wie „Diskette“ oder „CDROM“.

- Für Migrations- und Darstellungsaufzeichnungen muss unter anderem die Art und Weise des Transports von zu migrierenden oder darzustellenden Primärobjekten zwischen realer und virtueller Umgebung spezifiziert werden.

- Betriebssystem-Images oder Software-Komponenten sind auf eine bestimmte Rechnerarchitektur, und eventuell bestimmte virtuelle Hardware angewiesen.

Bei der Übersetzung dieser Abhängigkeiten in ein Archivmodell erscheint es jedoch vorteilhaft, die Struktur der archivierten Daten möglichst allgemeingültig zu halten. Der Grund dafür

[32] Urheber dieses Standards ist die „Dublin Core Metadata Initiative“: `http://dublincore.org`

ist, dass gerade im Umfeld der Langzeitarchivierung, die Archivstruktur möglichst lange Bestand haben sollte. Die zum aktuellen Zeitpunkt eingesetzten Technologien unterliegen hingegen einem raschen Wandel. Daher sollten Abhängigkeiten von konkreten Technologien so weit wie möglich vermieden werden.

Bei dem Benutzer nahestehenden Teilen des Systems, beispielsweise dem GUI, ist die Situation eine andere. Hier sind Abhängigkeiten von konkreten Technologien, beispielsweise in Form einer Auswahlliste von Installationsdatenträgertypen, kaum vermeidbar. Eine direkte Präsentation der abstrakten Konzepte gegenüber dem Archivverwalter wäre zwar möglich, jedoch höchst unergonomisch. Diejenigen Teile des Systems, die von konkreter Technologie abhängig sind, können dann bei Aufkommen neuer Technologien, wie zum Beispiel DVDs als Installationsdatenträger, entsprechend angepasst werden. Dies gilt selbstverständlich nur so lange, wie die Anpassungen auf die in der Archivstruktur implementierte Abstraktion abbildbar sind.

Dem Entwurf der Abstraktion kommt also eine zentrale Bedeutung zu. Wichtig ist vor allem, dass alle für eine spätere Nutzung relevanten Informationen archiviert werden: Veränderungen des Frontends, zum Beispiel durch eine Anpassung an neue Technologien, dürfen die im Archiv enthaltenen Daten nicht unbrauchbar machen. Anders formuliert bedeutet dies, dass es möglichst wenig implizite Abhängigkeiten der Archivdaten von äußeren, nicht archivierten Komponenten, wie in diesem Fall dem Frontend, geben sollte.

Diese allgemeinen Betrachtungen sollen am Beispiel einer Installationsaufzeichnung veranschaulicht werden: Diese benötigt die Angabe des Typs des Installationsdatenträgers. Die naive Vorgehensweise wäre es nun, den Typ als Enumeration zu archivieren, wobei die Werte die derzeit verfügbaren Technologien, wie Disketten oder CDROMs, darstellen. Beim Entstehen neuer Technologien, wie beispielsweise DVDs, muss diese Enumeration erweitert werden, was bestehende Daten unter Umständen ungültig machen kann.

Soll die Installation nun mittels der Aufzeichnung wiederholt werden, so muss der Emulator das Installationsdatenträgerimage entsprechend seines Typs entweder als Diskettenimage, oder als CDROM-Image einbinden. Das die Aufzeichnung abspielende System muss die Enumeration also auf eine Emulatorkonfiguration abbilden. Diese Abbildung muss ebenfalls bei jeder neuen Technologie angepasst werden, außerdem sind bei einem Verlust der Abbildungsdefinition die archivierten Daten nutzlos.

Die genannten Probleme können durch ein generisches Abspeichern der Emulatorkonfiguration verhindert werden, da so von konkreten Datenträgertypen abstrahiert wird. Ein manuelles Erstellen einer solchen Konfiguration für jede zu archivierende Softwarekomponente ist aber wiederum höchst ineffizient, und verlangt vom Benutzer außerdem ein hohes Maß an Wissen über die Bedienung des Emulators. Wünschenswert wäre vielmehr die einfache Angabe des Installationsdatenträgertyps in der Benutzeroberfläche. Hier kommt nun die oben geschilderte „Misch-Abstraktion“ zum Einsatz: Der Typ wird im Frontend konkret angegeben, das Frontend bildet ihn dann auf eine Emulatorkonfiguration ab und archiviert diese. Listing 1 zeigt einen möglichen Kontext in Form einer Liste von Aufrufparametern für QEMU.

```
-M pc -m 512 -k de -hda /var/tmp/osimage.img  \
-cdrom /var/tmp/office2000.iso
```

Listing 1: Beispiel für einen Kontext in Form einer QEMU Parameterliste

Wie dieses Beispiel gezeigt hat, ist eine Speicherung der Emulatorkonfiguration gegenüber dem Ablegen einzelner Aspekte, wie dem Typ des Installationsdatenträgers, auf jeden Fall vorzuziehen. Auch bei Auftauchen völlig neuer Aspekte bietet dieses Vorgehen weit größere Flexibilität. Noch weiter verallgemeinert könnte man von der Umgebung oder dem Kontext sprechen, in der ein Aufzeichnung nutzbar ist. Dieser Kontext kann dann allgemein als Binärdatenstrom archiviert werden. Dies erlaubt einerseits die Speicherung von Emulatorkonfigurationen beliebigen Typs, zum Beispiel einer Liste von Kommandozeilenparametern im Falle des Emulators QEMU oder auch binärer Konfigurationsdateien. Andererseits können so auch Aspekte der von der Aufzeichnung erwarteten Umgebung abgebildet werden, die nicht in der Emulatorkonfiguration enthalten sind. Ein Beispiel hierfür ist der Dateiname eines Primärobjekts, welches durch die Aufzeichnung zur Darstellung gebracht werden soll.

3.1.4 Archivierung von Emulatoren

Die Speicherung des Kontextes als Binärdatenstrom ist ein wichtiger Schritt hin zur einer möglichst weitgehenden Vermeidung von externen Abhängigkeiten. Dennoch ist natürlich auch ein solcher Kontext von einem konkreten Emulator abhängig, da er ja eine nur für diesen Emulator passende Konfiguration enthält.

Hier entsteht nun ein fundamentaler Konflikt: Die Speicherung von Emulatoren im Archiv zur Laufzeit ist nur dann sinnvoll, wenn das Gesamtsystem zur Entwicklungs- beziehungsweise Übersetzungszeit von konkreten Emulatoren unabhängig ist. Andernfalls müsste das System bei jedem neu archivierten Emulator angepasst werden, was natürlich nicht sinnvoll ist. Das Archiv-GUI und die mit dem Archiv verbundenen Systeme, die beispielsweise Migrationen durchführen, müssen jedoch das Kontextformat interpretieren und den Emulator mit der im Kontext abgelegten Konfiguration starten. Sie sind also in jedem Fall von konkreten Emulatoren abhängig[33].

In dem in dieser Arbeit entwickelten Konzept werden die Emulatoren daher nicht im Archiv abgelegt. Die Abhängigkeit des Kontextes von ihnen existiert aber weiterhin. Das Archiv vergibt daher für jeden unterstützten Emulator einen eindeutigen Bezeichner, der Kontext wird dann mit diesem Bezeichner verknüpft. Im GUI und den anderen mit dem Archiv kooperierenden Systemen ist dann emulatorspezifischer Code enthalten, der Erzeugung und Interpretation des Kontextes übernimmt.

[33] Einige weitere Überlegungen zu diesem Problem finden sich in Kapitel 5.2.

3.1.5 Kontext und Referenzen

Zur Umgebung und damit dem Kontext, den eine Installationsaufzeichnung benötigt, gehört streng genommen auch das Binary des Installationsdatenträgers. Dieses Binary sollte aber auf jeden Fall getrennt archiviert werden. Es muss daher im Kontext geeignet referenziert werden. Die Art der Referenz ist dabei abhängig vom konkreten Emulator und dem zugehörigen Kontextformat. Im Falle des Emulators QEMU beispielsweise muss der Installationsdatenträger als Image-Datei vorliegen, der Name der Datei wird beim Start des Emulators als Kommandozeilenparameter übergeben. Die Auflösung der Referenz, im Beispiel also die Erzeugung einer Datei mit dem in der Parameterliste genannten Dateinamen, ist Teil des emulatorspezifischen Codes im Archiv.

Entsprechendes gilt für Migrations- und Darstellungsaufzeichnungen: Hier gehört zwar nicht das Primärobjekt selbst zum Kontext (es ist ja zum Zeitpunkt der Archivierung der Aufzeichnung nicht bekannt), wohl aber die Art und Weise des Imports beziehungsweise Exports, insbesondere der Dateiname des Objekts. Auch hier muss emulatorspezifischer Code dafür sorgen, dass das in einem konkreten Migrations- oder Darstellungsfall vorhandene Primärobjekt dem vorher definierten Kontext entspricht, etwa durch Anpassung des Dateinamens.

3.2 Dynamische Erzeugung von View-Paths

Um die genannten Aufgaben zu erfüllen, sind unterschiedliche Vorgehensweisen denkbar. Eine der Alternativen basiert auf dem zentralen Konzept der dynamischen Erzeugung von View-Paths. Hierbei liegt der Schwerpunkt auf der Archivierung der einzelnen View-Path-Komponenten. Der Archivverwalter definiert dabei Beziehungen und Abhängigkeiten zwischen den Komponenten. Beim Abrufen einer Ablaufumgebung, also eines vollständigen View-Paths, werden die nötigen Komponenten gemäß dieser Beziehungen ausgewählt und kombiniert.

3.2.1 Datenmodell

Um von der abstrakten Idee zu einem konkreten Datenmodell zu gelangen, müssen zum Einen zu archivierende Entitäten, zum Anderen die möglichen Arten von Beziehungen zwischen diesen Entäten identifiziert werden. Da Emulatoren aus den in Kapitel 3.1.4 genannten Gründen nicht archiviert werden, werden sie hierbei nicht weiter betrachtet:

View-Path-Komponente: An erster Stelle der Entitäten sind natürlich die View-Path-Komponenten zu nennen. Beispiele hierfür sind Anwendungsprogramme, Fonts oder Laufzeitumgebungen wie ein JRE. Sie liegen in Form eines Installationsdatenträgerimages vor, und sind mit Hilfe von Aufzeichnungen dynamisch und automatisch installierbar. Sie können bestimmte Migrationen oder Darstellungen von Objekttypen mit Hilfe von Aufzeichnungen unterstützen.

Betriebssystem-Image: Betriebssysteme sind, wie bereits beschrieben, nicht trivial automatisch installierbar, und können daher nicht als gewöhnliche View-Path-Komponente behandelt werden. Sie werden vielmehr durch Containerdateien repräsentiert, die ein bereits installiertes Betriebssystem enthalten.

Service: Bei dieser Entität handelt es sich um eine Hilfskomponente, die zur Modellierung von Alternativ-Abhängigkeiten dient. Ein Service bildet die Tatsache ab, dass manche Programme für eine korrekte Funktion nur allgemein einen bestimmten Dienst benötigen, beispielsweise einen SMTP-Server, jedoch auf keine konkrete Implementierung (wie beispielsweise Exim[34]), angewiesen sind.

Neben den Entitäten selbst stellen die Beziehungen zwischen ihnen den zweiten wichtigen Teil des Konzepts dar:

„depends on“: View-Path-Komponenten können voneinander abhängig sein. Dies bedeutet, dass eine abhängige Komponente eine oder mehrere andere Komponenten benötigt, um korrekt zu funktionieren. Hierdurch werden auch Installationsabhängigkeiten modelliert, also die Tatsache, dass eine Komponente ohne vorherige Installation einer anderen nicht installiert werden kann. Beim Ziel der Abhängigkeit kann es sich, neben einer anderen View-Path-Komponente, auch um einen Service handeln.

„supplied by“: Services können von einer oder mehreren View-Path-Komponenten angeboten werden.

„installable on“: Um View-Path-Komponenten zu einem kompletten View-Path zusammensetzen zu können, müssen diese auf einem oder mehreren Betriebssystem-Images installierbar sein. Dies geschieht mit Hilfe einer Installationsaufzeichnung, die also Teil der Beziehung ist.

Abbildung 8 zeigt eine grafische Darstellung des Datenmodells, welche auch noch einige weitere Entitäten, wie Metadatensätze, Aufzeichnungen und Migrationsdefinitionen enthält. Auf eine Modellierung von Darstellungsunterstützungen wurde der Einfachheit halber verzichtet, sie geschieht analog zu Migrationsunterstützungen.

3.2.2 Erzeugung von Ablaufumgebungen

Die vom Archiv bereitzustellenden Migrations- und Darstellungsumgebungen bestehen aus einem View-Path und einer passenden Aufzeichnung. Der View-Path ist dabei technisch repräsentiert durch die Angabe eines zu verwendenden Emulators, sowie durch eine Containerdatei, welche alle weiteren View-Path-Komponente in installierter Form enthält. Beim Abrufen einer solchen Umgebung muss das Archiv also

[34] Exim ist ein unter Linux/Unix weit verbreiteter Mail-Transfer-Agent: `http://www.exim.org`

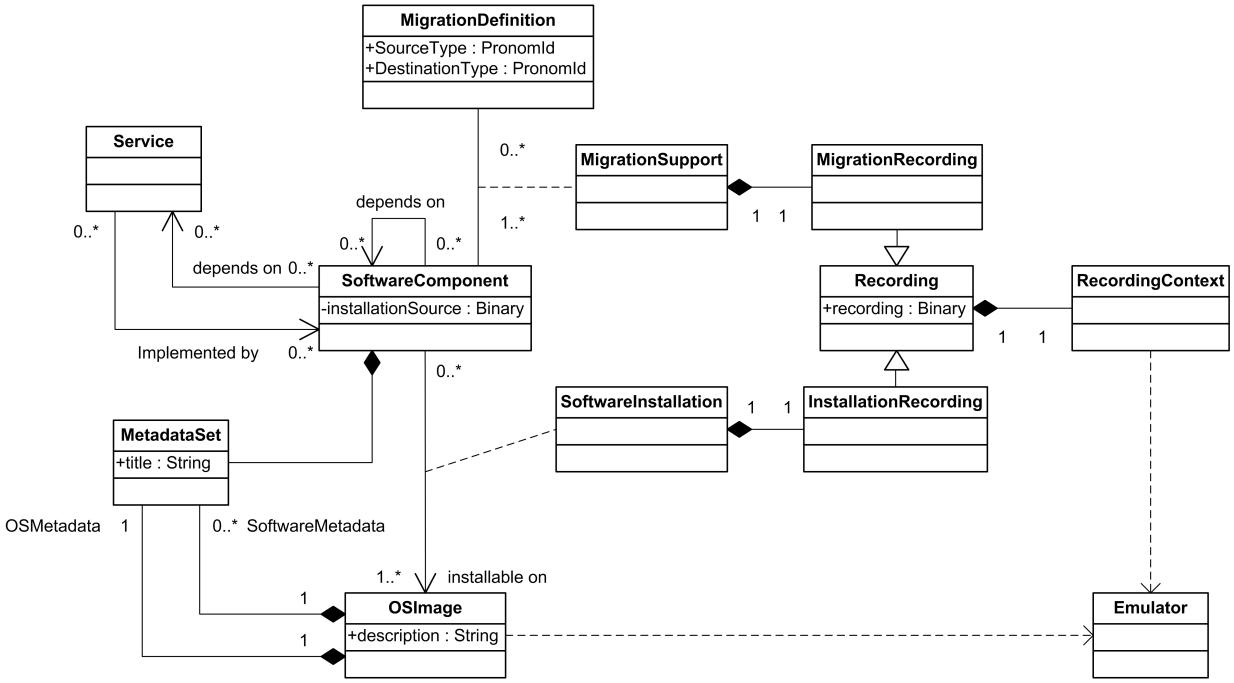


Abbildung 8: Datenmodell bei dynamischer Erzeugung der View-Paths

1. die nötigen View-Path-Komponenten bestimmen, und

2. aus den Einzelkomponenten eine Containerdatei erzeugen.

Ersteres geschieht, indem zuerst eine View-Path-Komponente, eine „Start-Komponente“, bestimmt wird, welche die angefragte Migration bzw. Darstellung unterstützt. Nachdem die Auswahl erfolgt ist, muss der Abhängigkeitsgraph der Start-Komponente aus dem globalen Beziehungsgraphen, also dem Graphen aller Entitäten und Beziehungen, abgeleitet werden. Hierzu wird nur derjenigen Teilgraph aus Komponenten, Services und Beziehungen betrachtet, der von der Start-Komponente aus per „depends on“-Beziehungen im Falle von Komponenten, oder „supplied by“-Beziehungen im Falle von Services erreichbar ist. Aus diesem Graphen werden nun die Services eliminiert, indem jeweils eine der implementierenden Komponenten ausgewählt wird.

Nach der Bestimmung der nötigen Komponenten muss aus dem resultierenden Graphen eine Containerdatei erzeugt werden. Hier wird zuerst überprüft, ob für alle Komponenten ein gemeinsames Betriebssystem-Image existiert, also ein Image, zu dem eine „installable on“-Beziehung besteht. Ist dies nicht der Fall, so müssen, beispielsweise durch Backtracking, andere Alternativen ausgewählt werden. Existiert keine Alternative, die eine gemeinsame Installationsbasis ermöglicht, so ist der View-Path nicht realisierbar.

Wurde eine gemeinsame Installationsbasis gefunden, so muss als letzter Schritt eine Installationsreihenfolge bestimmt werden. Da im Allgemeinen davon ausgegangen werden muss, dass

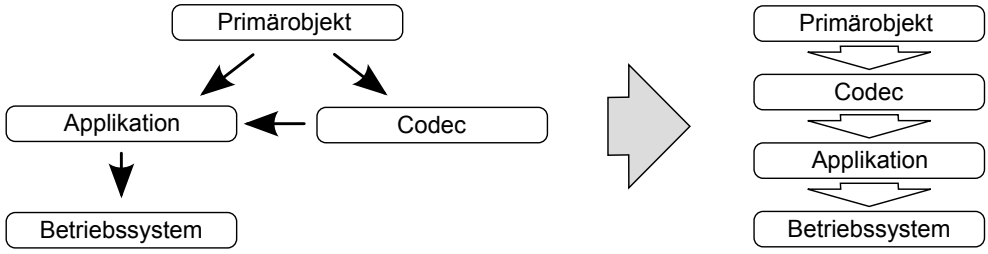


Abbildung 9: Generierung eines View-Path aus zuvor definierten Abhängigkeiten

eine Abhängigkeit zwischen View-Path-Komponenten auch zum Installationszeitpunkt besteht, können Komponenten jedoch nicht in beliebiger Reihenfolge installiert werden. Vielmehr muss die korrekte Reihenfolge durch topologische Sortierung des Abhängigkeitsgraphen bestimmt werden (Abbildung 9). Nachdem dies geschehen ist, können die einzelnen Installationen durch Abspielen der Installationsaufzeichnungen auf einem der gemeinsamen Installationsbasis-Images durchgeführt werden. Dies resultiert schließlich in der gewünschten Containerdatei, die alle View-Path-Komponenten enthält.

3.2.3 Kritik

Die Vorteile des vorgestellten Konzepts liegen auf der Hand: Es ist modular, flexibel, und bildet die in Kapitel 2.1 dargestellten Abhängigkeiten gut ab. Allerdings wurden, zwecks Konzentration auf die grundlegenden Aspekte, einige Vereinfachungen vorgenommen:

- Da die Installation einer View-Path-Komponente auf einem Basisimage erhebliche Zeit in Anspruch nehmen kann, ist die erneute Generierung eines Images für einen kompletten View-Path bei jeder Anfrage außerordentlich ineffizient. Dies ließe sich jedoch beispielsweise durch einen Caching-Mechanismus umgehen, der für häufig nachgefragte View-Paths bereits fertig generierte Images vorhält.

- Um eine topologische Sortierung und gegebenenfalls ein Backtracking zu ermöglichen, muss der Abhängigkeitsgraph zyklenfrei sein, eine wechselseitige (Laufzeit-)Abhängigkeit von zwei View-Path-Komponenten kann also nicht abgebildet werden. Diese Beschränkung ließe sich jedoch möglicherweise durch eine Unterscheidung von Laufzeit- und Installationsabhängigkeiten abmildern.

- Es ist nicht klar, welche Kriterien zur Auflösung von Mehrdeutigkeiten herangezogen werden. Beispiele für solche Mehrdeutigkeiten sind die Auswahl einer Startkomponente, oder die Komponentenauswahl bei Auflösung einer Service-Abhängigkeit. Hierzu könnten möglicherweise durch User-Interaktion unterstützte Metriken herangezogen werden, siehe hierzu auch Kapitel 5.2.

Neben diesen Vereinfachungen, deren Eliminierung grundsätzlich möglich ist, birgt das Konzept der dynamischen View-Path-Erzeugung jedoch auch prinzipielle Probleme. Insbesondere die Aufzeichnungen erweisen sich, in Kombination mit der eigentlich vorteilhaften Flexibilität des Konzepts, als kritisches Element. So ist es eine der grundlegenden Annahmen, dass Installations-, Migrations- und Darstellungsaufzeichnungen zwar an eine bestimmte View-Path-Komponente (und im Falle einer Installationsaufzeichnung auch an ein OS-Image) gebunden, jedoch von allen anderen Komponenten unabhängig sind. Dahingehende Recherchen sowie weitere Überlegungen haben jedoch ergeben, dass diese Annahme, bei derzeitigem Stand der Aufzeichnungstechnik, nicht haltbar ist. So kann eine Installationsaufzeichnung, wie in [Rechert u. a., 2009] dargestellt, bereits dann fehlschlagen, wenn vorher eine andere View-Path-Komponente installiert wurde: „For example creating the above described View-Path in the opposite order (first installing AmiPro and then the PostScript driver) the playback of the installation procedure failed at the very last step. The screen's fingerprint differed more than the allowed threshold of 5% of mismatched pixels. This was mainly due to the change of the desktop background, as the icon of the previously installed printerapplication was missing“. Auch bei Migrations- und Darstellungsaufzeichnungen sind Zweifel angebracht, ob die grafische Darstellung der zugehörigen Komponenten tatsächlich stets unabhängig davon ist, welche Komponenten zur Auflösung von eventuellen Service-Abhängigkeiten herangezogen werden.

Für eine Lösung dieser Probleme wäre eine wesentlich abstraktere und damit für Änderungen in der grafischen Darstellung weniger anfällige Aufzeichnungstechnik nötig. Eine andere mögliche Lösung wäre, die durch die Aufzeichnungen induzierten Abhängigkeiten beim Archivieren der Komponenten mit aufzunehmen. Dies würde jedoch dem Archiv letztlich jegliche Auswahlmöglichkeit nehmen, und damit das Konzept der dynamischen Erzeugung ad absurdum führen.

3.3 Statische Archivierung von View-Paths

Aufgrund der im letzten Kapitel beschriebenen Probleme wird klar, dass ein revidiertes Konzept für die Struktur der zu archivierenden Daten benötigt wird. Dieses Konzept sollte weniger Freiheitsgrade enthalten, da sich gerade diese Freiheitsgrade als Problemquelle herauskristallisiert haben. Es wird also weniger flexibel, dafür jedoch erheblich robuster sein.

3.3.1 Grobkonzept

Die Grundidee des revidierten Konzeptes besteht darin, die Aufgabe der Erzeugung von fertig zusammengestellten Images vom Archiv auf den Archivverwalter zu verlagern. Entsprechend ist die zentrale Aufgabe des Archivs nun auch nicht mehr die Verwaltung von einzelnen View-Path-Komponenten und die dynamische Erzeugung von View-Paths, sondern die statische Archivierung von vollständigen View-Paths in Form von Images.

Die einzelnen Softwarekomponenten sollen jedoch weiterhin in installierbarer Form, also beispielsweise als Installationsdatenträger, archiviert werden. Dies erweist sich in vielerlei Hinsicht als sinnvoll: Beispielsweise ist so die Installation einer bereits im Archiv vorhandenen Softwarekomponente auf einem anderen, ebenfalls im Archiv befindlichen Image möglich.

Die Basis der Archivstruktur bilden, ähnlich wie im ersten Konzept, Images, die nichts außer einem Betriebssystem in installierter Form enthalten. Auch im revidierten Konzept werden sie völlig unabhängig vom Softwarearchiv erstellt, da sich die zugrundeliegende Problematik nicht verändert hat: Die Erstellung der Images ist, wie viele praktische Tests gezeigt haben, mit zahlreichen technischen Detailproblemen verbunden, und lässt sich daher nicht automatisieren. Neu im revidierten Konzept ist die Einführung einer zweiten Art von Images. Diese können auf bereits im Archiv befindlichen Images basieren, und sollen im Folgenden als „aggregierte Images“ bezeichnet werden. Sie entsprechen jeweils ihrem Basisimage, auf dem eine zusätzliche Softwarekomponente installiert wurde.

Da die zentrale Aufgabe des Archivs nun die Verwaltung von Images ist, wird die Fähigkeit, eine Migration durchzuführen, beziehungsweise ein Objekt darzustellen, auch nicht einer einzelnen Softwarekomponente zugeordnet, sondern einem kompletten Image. Hier zeigt sich bereits ein Vorteil der reduzierten Freiheitsgrade: Es wird auf elegante Weise das Problem gelöst, dass manche Aufgaben die Kombination mehrere Softwarekomponenten erfordert: So kann beispielsweise die Migration einer Microsoft Word 97 Datei nach PDF sowohl Microsoft Office 97, als auch einen virtuellen PDF-Drucker voraussetzen.

3.3.2 Datenmodell

Die im letzten Abschnitt grob umrissene Idee soll nun in einem Datenmodell formalisiert werden. Eine Darstellung des Modells als UML-Klassendiagramm zeigt Abbildung 10. Im Folgenden werden die beteiligten Entitäten und Assoziationen vorgestellt und näher beschrieben.

Emulator

Wie bereits in Kapitel 3.1.4 dargelegt, werden die genutzten Emulatoren vorerst nicht im Archiv abgelegt. Dennoch bestehen Abhängigkeiten anderer Entitäten vom Emulator, so dass diese Entität dennoch ins Datenmodell aufgenommen wurde.

Image

Beim Image handelt es sich um die zentrale Entität im Datenmodell. Es stellt eine Containerdatei für einen bestimmten Emulator dar, die ein installiertes Betriebssystem und eventuell weitere Softwarekomponenten in installierter Form enthält.

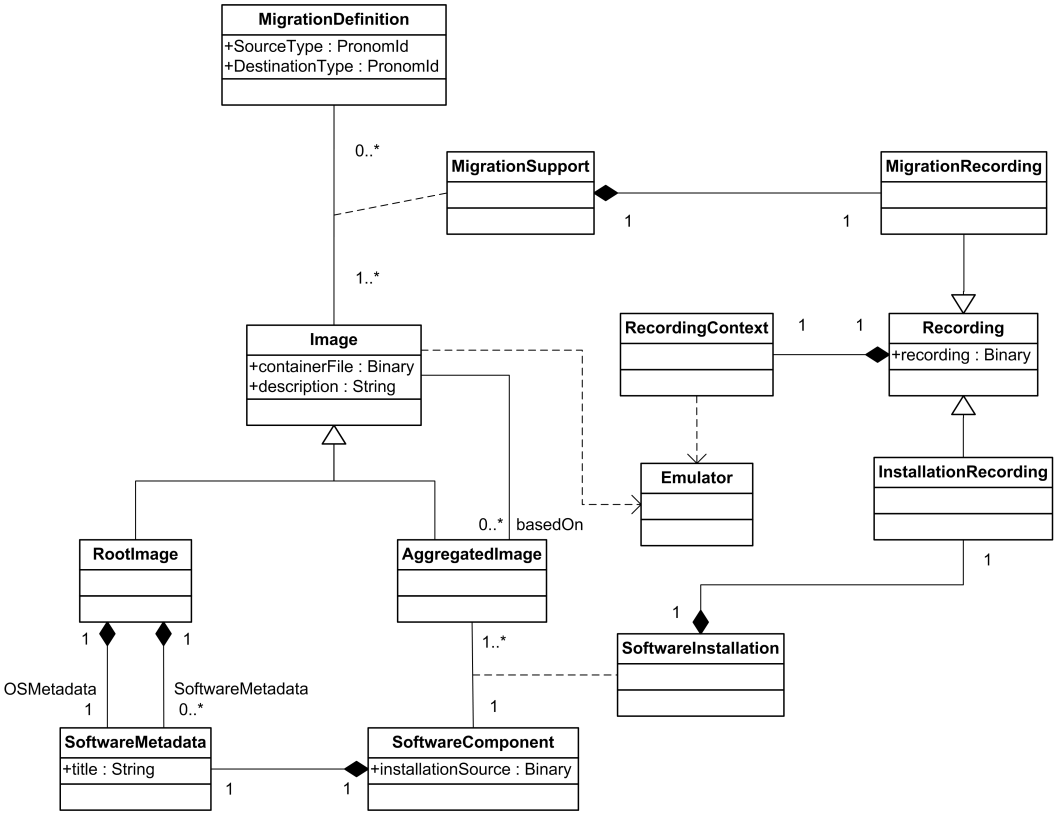


Abbildung 10: Datenmodell bei statischer Archivierung der View-Paths

Da das Format von Containerdateien im allgemeinen emulatorspezifisch ist, kann ein Image nur mit einem ganz bestimmten Emulator genutzt werden. Es besteht also eine semantische Abhängigkeit vom Emulator. Ferner kann ein Image mehrere (oder auch keine) Objektmigrationen unterstützen, dies wird im Modell durch die MigrationSupport Assoziation ausgedrückt. Schließlich müssen, um das Wiederauffinden von Images zu erleichtern, für das Betriebssystem und jede installierte Softwarekomponente jeweils ein Metadatensatz mit dem Image verknüpft werden.

Es existieren zwei Spezialisierungen des Images, das AggregatedImage und das RootImage.

AggregatedImage

Ein aggregiertes Image entsteht durch Installation einer Softwarekomponente auf einem anderen Image, das hier „Basisimage“ genannt werden soll. Es übernimmt als Spezialisierung von Image dessen Eigenschaften, die Menge der Metadatensätze setzt sich hierbei jedoch aus den Metadatensätzen des Basisimages und dem Metadatensatz der installierten Softwarekomponente zusammen. Die Beziehung zum Basisimage ist ebenfalls im Datenmodell abgebildet, wobei zu beachten ist, dass auf einem Basisimage mehrere, oder auch gar keine, aggregierte Images basieren können. Es handelt sich also um den Beziehungstyp n:1.

Neben der Beziehung zum Basisimage ist das aggregierte Image mit derjenigen Softwarekomponente verknüpft, durch deren Installation es zustandegekommen ist. Diese Verknüpfung wird durch die SoftwareInstallation Assoziation modelliert.

SoftwareInstallation

Diese Assoziation stellt die Beziehung zwischen aggregiertem Image und Softwarekomponente dar. Ein aggregiertes Image ensteht immer duch Installation genau einer Softwarekomponente. Eine Softwarekomponente kann jedoch durchaus auf mehreren Basisimages installiert werden, was jeweils zu einem neuen aggregierten Image führt. Eine Softwarekomponente alleine, ohne zugeordnetes Image, ist im vorliegenden Konzept nicht vorgesehen, es handelt sich also um den Beziehungstyp 1..*:1.

Um Softwareinstallationen später gegebenenfalls erneut, möglicherweise sogar automatisch, durchführen zu können, ist eine Aufzeichnung des Installationsvorganges nötig. Besondere Beachtung verdient hierbei die Frage nach dem Umfeld, in dem diese Aufzeichnung sinnvoll nutzbar ist. Einerseits ist sie, wie die Erfahrungen aus dem ersten Konzept gezeigt haben, im Allgemeinen nur auf genau dem Basisimage ablauffähig, auf dem sie erstellt wurde. Andererseits ist sie natürlich an die Softwarekomponente gebunden. Eine alleinige Zuordnung entweder zum aggregierten Image, oder zur Softwarekomponente ist also nicht ausreichend. Vielmehr muss die Aufzeichnung als Teil der Assoziation modelliert werden.

SoftwareComponent

Hierbei handelt es sich um eine einzelnes Sekundärobjekt im Sinne von Kapitel 2.1, welches kein Emulator oder Betriebssystem ist. Ein Beispiel hierfür wäre eine Applikation wie OpenOffice, oder ein bestimmter Codec. Eine SoftwareComponent enthält als Attribut ein Image des Installationsdatenträger als binärer Datenstrom. Darüber hinaus ist sie mit einem beschreibenden Metadatensatz verknüpft, um das spätere Wiederauffinden zu erleichern. Das Auffinden einzelner Softwarekomponenten wird unter anderem nötig sein, um eine bereits im Archiv befindliche Softwarekomponente auf einem anderen Basisimage zu installieren.

MetadataSet

Diese Entität entspricht einem Metadatensatz für Software. Es könnte sich dabei um Elemente eines vordefinierten Satzes, wie beispielsweise des Dublin Core Metadata Element Set, handeln. Im vorliegenden Modell wurde jedoch zwecks Vereinfachung ein Titel als einziges Element aufgenommen. Der Metadatensatz ist stets Teil einer Kompositionsbeziehung, da unterschiedliche Softwarekomponenten und Betriebssysteme sich aller Voraussicht nach auch in ihren Metadaten unterscheiden werden.

RootImage

Wie aus der Beschreibung des AggregatedImage hervorgeht, setzt dieses zwingend ein Basisimage voraus. Es ist also klar, dass noch ein weiterer Image-Typ benötigt wird, der kein Basisimage voraussetzt und als Senke im gerichteten Graph der Images dienen kann. Dieser Typ ist das RootImage. Es handelt sich dabei um ein Image, welches meist nur das installierte Betriebssystem beinhaltet, und wird unabhängig vom Softwarearchiv manuell erstellt. In diesem Fall enthält es nur einen Metadatensatz für das Betriebssystem.

Das RootImage kann aber auch weitere Softwarekomponenten in installierter Form enthalten. Diese werden dann vom Archiv als integraler Bestandteil des RootImage betrachtet, und nicht getrennt in installierbarer Form (also als SoftwareComponent) archiviert. Eine solche Vorinstallation ist beispielsweise in Betracht zu ziehen, wenn das Betriebssystem-Image ohne die Softwarekomponente nicht sinnvoll nutzbar, oder die Installation der Komponente nicht automatisiert durchführbar ist. Falls weitere Softwarekomponenten installiert sind, kann das RootImage für jede dieser Komponenten einen eigenen Metadatensatz enthalten.

MigrationDefinition und MigrationSupport

Eine MigrationDefinition definiert die Art einer Migration als Tupel aus Quell- und Zieldateityp. Die MigrationSupport Assoziation repräsentiert die Unterstützung eines Migrationstyps durch ein Image.

Der Assoziationstyp 1..*:0..* ergibt sich aus folgenden Überlegung: Zuerst einmal gilt, dass ein Image mehrere Migrationen unterstützen kann. Ebenso ist es möglich, dass gar keine Migrationen unterstützt werden, da ja jede Installation einer Softwarekomponente ein neues Image generiert. Ist diese Softwarekomponente alleine jedoch noch nicht ausreichend für eine Migration, so wird auch kein Migrationssupport vorhanden sein. Umgekehrt kann eine Migration von mehreren Images unterstützt werden. Gäbe es jedoch kein Image, welches eine bestimmte Migration unterstützt, so gibt es auch keinen Grund, wieso diese Migrationsdefinition dem Archiv bekannt sein sollte.

Die Unterstützung einer Migration durch ein Image benötigt als integralen Bestandteil eine Aufzeichnung, welche die Migration tatsächlich mit Hilfe der im Image installierten Software durchführt. Hierbei gelten zu den Software-Installationsaufzeichungen analoge Überlegungen: Eine Migrationsaufzeichnung ist einerseits nur auf einem bestimmten Image durchführbar, andererseits an einen konkreten Migrationstyp gebunden. Die Aufzeichnung ist also Teil der Assoziation.

Auf die Definition einer eigenen Entität samt Assoziation für Objektdarstellungen wurde der Übersichtlichkeit halber verzichtet. Sie geschieht analog zur Migration, wobei statt Quell- und Zieldobjekttyp nur eine einzige Typdefinition nötig ist.

Recording und RecordingContext

Installations- und Migrationsaufzeichnung können allgemein zu Aufzeichnungen, hier „Recording“ genannt, zusammengefasst werden. Ein Recording enthält einerseits die Aufzeichnung selbst als binärer Datenstrom. Wie jedoch in Kapitel 3.1.3 dargestellt, benötigen Aufzeichnungen grundsätzlich einen Kontext, einen „RecordingContext“. Da dieser unter anderem Emulator-Konfigurationen enthält, besteht eine semantische Abhängigkeit des Kontextes von einem konkreten Emulator.

3.3.3 Datenmodell und View-Paths

Im vorgestellten Datenmodell sind View-Paths nicht offensichtlich enthalten. Es stellt sich also die Frage, wie die vom Archiv verwalteten View-Paths durch das Datenmodell repräsentiert werden, und welche Rolle dabei die in [von Suchodoletz, 2009, Kap. 6.6] vorgestellten View-Path-Caches spielen.

Abbildung von View-Paths durch das Datenmodell

Die Images und ihre Beziehungen untereinander können als gerichteter Graph mit den Images als Knoten und der Basisimage-Relation als Menge der Kanten betrachtet werden. Aufgrund der Struktur des Datenmodells ist klar, dass es sich dabei um einen Wald mit den RootImages als Wurzelelementen handeln muss. Ein einfaches Beispiel für einen solchen Graphen ist in Abbildung 11 dargestellt. Außerdem ist jedem AggregatedImage eine Softwarekomponente zugeordnet, jedes RootImage beinhaltet implizit ein Betriebssystem. Schließlich können Images mit Migrations- oder Darstellungssupport versehen werden. Derart markierte Images dienen nun als Startpunkt eines entsprechenden View-Paths. Die weiteren Komponenten des View-Paths ergeben sich durch die Basisimage-Relation, bis hin zum durch das Root-Image repräsentierte Betriebssystem. Die letzte Komponente im View-Path, der Emulator, ist implizit durch die Abhängigkeit des Images von einem Emulator gegeben.

Festzustellen ist weiterhin, dass mehrere View-Paths für einen Objekttyp im Archiv vorgehalten werden können, indem mehrere Images mit der Fähigkeit zur Darstellung eines bestimmten Objekttyps verknüpft werden. In diesem Fall muss eine Auswahl getroffen werden. Ist jedoch einmal ein „Startimage“ ausgewählt, so sind, im Gegensatz zur dynamischen View-Path-Erzeugung, sämtliche weiteren Komponenten des View-Paths, sowie ihre Reihenfolge, eindeutig festgelegt.

Datenmodell und View-Path-Caches

Die Erzeugung eines aggregierten Images durch Installation einer Softwarekomponente ist eine sehr zeitaufwendige Operation. Beim Konzept der dynamischen View-Path-Erzeugung schien

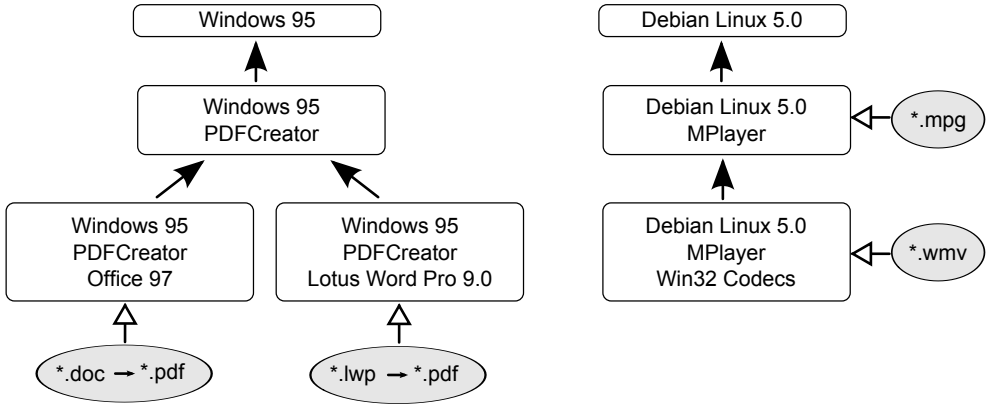


Abbildung 11: Beispiel für Images mit Migrations- und Darstellungsunterstützung

daher die zusätzliche Einführung ein Caching-Mechanismusses unumgänglich. Bei der statischen View-Path-Archivierung sind diese Caches jedoch bereits als integraler Bestandteil enthalten: Es handelt sich um diejenigen AggregatedImages, die eine Migration oder die Darstellung eines bestimmten Objekttyps erlauben. Sie enthalten eine Nutzungsumgebung, in der alle für die entsprechende Operation nötigen Komponenten bereits fertig kombiniert sind.

Zu bedenken ist jedoch Folgendes: Die Caches sollten, nach dem allgemeinen Verständnis des Begriffes „Cache“, lediglich einen einfacheren und performanteren Zugriff auf Informationen bieten, und selbst keinerlei Informationen enthalten, die nicht vollständig aus anderen Quellen wiederhergestellt werden kann. Es sollte also möglich sein, die aggregierten Images, oder zumindest die durch sie repräsentierte Containerdatei, ohne Informationsverlust zu löschen, beispielsweise um Speicherplatz im Archiv einzusparen. Auch eine eventuelle Backupstrategie könnte hiervon profitieren, so könnte man auf ein Backup der speicherplatzintensiven Caches komplett verzichten, solange die zugrundeliegenden Komponenten und Metadaten vollständig gesichert werden.

Diese Eigenschaften treffen auf die aggregierten Images auf den ersten Blick nicht zu. Bei näherer Betrachtung erweisen sich jedoch zumindest die durch sie repräsentierten Containerdateien als tatsächlich redundant. Hier zeigt es sich, warum es sinnvoll ist, eine Installationsaufzeichnung der Softwarekomponente im Archiv abzulegen: Es wäre jederzeit möglich, durch Wiederabspielen der Installationsaufzeichnung, aus dem Basisimage die Containerdatei eines AggregatedImages wiederherzustellen. Bei den grundlegenden, nicht aus anderen Informationen ableitbaren Daten handelt es sich also um die Softwarekomponente, die Installationsaufzeichnung samt Kontext, sowie die Angabe, auf welchem Basisimage die Installation erfolgte. Eine Anpassung des Datenmodells, die die Unterscheidung zwischen essentiellen und abgeleiteten Informationen berücksichtigt, zeigt Abbildung 12.

Bei einer Nutzer-Anfrage für ein aggregiertes Image, dessen Containerdatei nicht existiert, müsste das Archiv diese nun aus den oben genannten Informationen, unter anderem dem Basisimage, erzeugen. Dieser Prozess kann nun gegebenenfalls rekursiv auf das Basisimage angewen-

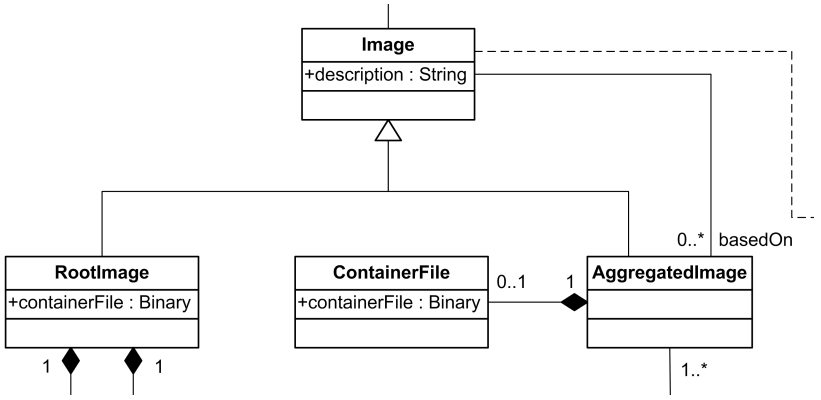


Abbildung 12: Erweiterung des Datenmodells um optionale Containerdateien

det werden, so müssten im Extremfall, bis auf das Root-Image, gar keine Containerdateien mehr vorgehalten werden.

4 Umsetzung

In diesem Kapitel soll nun eine Implementierung im Rahmen des PLANETS-Projektes vorgestellt werden, die das im letzten Kapitel vorgestellte Konzept der statischen Archivierung von View-Paths umsetzt. Da sich die Implementierung auf die Umsetzung der Kernaspekte konzentrieren soll, wurde auf die Definition und Umsetzung von Randaspekten, wie Logging oder Authentifizierung, komplett verzichtet. Auch wird auf diejenigen Use-Cases besonderen Wert gelegt, die die Problematik eines Softwarearchivs in der Emulationsstrategie besonders verdeutlichen, hier ist unter anderem das Erstellen von Aufzeichnungen zu nennen. Andere Anwendungsfälle mit eher verwaltungstechnischem Hintergrund werden hingegen nur knapp behandelt. Desweiteren muss, wie schon in Kapitel 3.1.4 beschrieben, für jeden Emulator eine Unterstützung durch das Archiv explizit implementiert werden. Aufgrund des prototypischen Charakters der Implementierung wird hier bewusst eine Beschränkung auf QEMU als einzig unterstütztem Emulator in Kauf genommen.

4.1 Anforderungsdefinition

Bevor im nächsten Kapitel die konkrete Architektur des Prototypen vorgestellt werden kann, müssen zunächst die Anforderungen erhoben werden. Dazu wird zuerst kurz das im Rahmen des PLANETS-Projekts entstandene Framework vorgestellt, welches das technische Umfeld der Implementierung bildet. Danach werden Aktoren identifiziert und deren Use-Cases vorgestellt.

4.1.1 Technisches Umfeld

Bei PLANETS („Preservation and Long-Term Access Through Networked Services“) handelt es sich um ein unter anderem von der Europäischen Union finanziertes Projekt zur Erstellung eines Frameworks zur Unterstützung digitaler Langzeitarchivierung. Es wurde am 1. Juni 2006 gestartet, und endete am 31. Mai 2010. Das Projekt, an dem diverse Bibliotheken, Archive und Universitäten aus ganz Europa beteiligt waren, war in sechs Teilbereiche untergliedert:

Preservation Planning unterstützt Archivorganisationen bei der Planung und Durchführung von Archivierungsprozessen

Characterization erlaubt die Klassifizierung digitaler Objekte

Preservation Action stellt Dienste bereit, um digitale Objekte zu migrieren beziehungsweise zur Darstellung zu bringen

Interoperability Framework bildet die nötige Integrationsplattform

Testbed stellt eine Testumgebung für die entwickelten Komponenten bereit

Dissemination and Takeup ist ein Programm, um die Akzeptanz bei Herstellern und Anwendern, beispielsweise durch Schulungen, zu fördern

Für die Implementierung eines Softwarearchivs im Rahmen einer Emulationsstrategie ist vor allem das Teilprojekt „Preservation Action“ von Bedeutung. Dieses definiert unter anderem zwei Dienste, „Migrate“ und „CreateView“. Ersterer dient dazu, digitale Objekte zu migrieren, also von einem veraltenden in ein aktuelles Format zu überführen. Den Input bildet dabei das zu migrierende Objekt, das migrierte Objekt wird als Resultat des Dienstaufrufes zurückgeliefert. Der CreateView Dienst nimmt als Eingabe ebenfalls ein digitales Objekt entgegen. Er liefert als Resultat jedoch eine URL, unter der eine visuelle Repräsentation des Objekts, beispielsweise als HTML-Seite, abrufbar ist. Das Softwarearchiv soll nun so entworfen werden, dass es von Implementierungen dieser Dienste genutzt werden kann, die Implementierung der Services selbst ist jedoch nicht Teil der Arbeit.

Die Preservation Action Services sind als Webservices spezifiziert, die Plattform bildet ein JBoss Application Server 4.2. Dieser implementiert die Java EE Spezifikation in der Version 1.4, und bietet darüber hinaus Support für Enterprise Java Beans 3.0, welche eigentlich Teil von Java EE 5 sind. Da das Archiv über ein einfaches Java-API von den Dienstimplementierungen genutzt werden soll, ist dies auch der technische Kontext, in dem das Archiv implementiert werden muss.

4.1.2 Aktoren

Bei der Analyse der Anforderungen lassen sich für die Interaktion mit dem Softwarearchiv zwei unterschiedliche Aktoren identifizieren, der Archivnutzer und der Archivverwalter:

Archivnutzer: Bei diesem Aktor handelt es sich um den primären Nutzer des Systems, seine Anforderungen bestimmen also das Verhalten des Archivs. Sein Interesse liegt darin, Primärobjekte darzustellen oder Migrationen vollautomatisch durchzuführen. Um diese Aufgaben zu erfüllen, bezieht er vom Softwarearchiv die dafür nötigen Komponenten des View-Paths. Da das Softwarearchiv in das im letzten Kapitel beschriebene PLANETS-Framework eingebettet werden soll, handelt es sich beim Archivnutzer nicht um einen menschlichen Aktor, sondern um ein Softwaresystem in Form eines PLANETS „Preservation Action"-Services. Benötigt wird also ein API[35] als Maschine-Maschine-Schnittstelle. Entsprechend werden die Use-Cases hier auch aus Maschinensicht, also bereits sehr technisch fomuliert.

Archivverwalter: Dieser Aktor ist für die Verwaltung des Archivinhaltes zuständig, wird also beispielsweise Softwarekomponenten hinzufügen oder neu kombinieren wollen. Seine Funktion ist jedoch hauptsächlich unterstützender Natur, seine Interaktionen erlauben es dem Softwarearchiv, dessen Aufgabe gegenüber dem Archivnutzer zu erfüllen. Sie leiten sich also direkt von der Struktur des Archivinhaltes ab. Da es sich beim Archivverwalter, im Gegensatz zum Archivnutzer, tatsächlich um einen menschlichen Aktor handelt, werden seine Use-Cases auch eher aus abstrakt-menschlicher Sicht beschrieben.

4.1.3 Anforderungen des Archivnutzers

Für den Archivnutzer lassen sich im einzelnen vier Use-Cases festlegen: Das Abrufen einer Umgebung zur Darstellung oder Migration eines Objektes, und das Abrufen einer Liste von unterstützten Objekttypen für die Darstellung beziehungsweise einer Liste von unterstützten Migrationen.

Abrufen einer Darstellungsumgebung

Dieser Use-Case tritt ein, wenn der PLANETS-PA-Service aufgerufen wird, um ein Primärobjekt mittels der Emulationsstrategie darzustellen. Hierzu benötigt er die Komponenten eines View-Paths, der zur Darstellung des gegebenen Objekts geeignet ist.

Um einen solchen View-Path auswählen zu können, muss der Actor zuerst den Typ des Primärobjektes auf vom Softwarearchiv unabhängige Art bestimmen. Der Typ dient dann wiederum als Eingabedatum für das Archiv, aufgrund dessen das Archiv einen geeigneten View-Path bestimmen wird. Identifiziert wird der Objekttyp, beziehungsweise das Format des Objekts, über seine PRONOM-UID (siehe auch Kapitel 2.2).

Vom Archiv erwartet der Aktor nun eine Antwort auf seine Anfrage in Form einer „Darstellungsumgebung". Diese umfasst folgende vom Archiv zu liefernde Daten:

[35] API=„Application Programming Interface"

- Zentraler Teil der Darstellungsumgebung ist eine vorbereitete Nutzungsumgebung in Form einer Containerdatei. Diese Datei muss ein installiertes Betriebssystem und sämtliche andere View-Path-Komponenten, wie zum Beispiel Awendungsprogramme, in installierter Form enthalten, die zur Darstellung des in den Eingabedaten spezifizierten Objektes nötig sind.

- Ferner ist ein Identifikator nötig um einen Emulator festzulegen, der in der Lage ist, das Format der Containerdatei zu interpretieren.

- Eine Aufzeichnungsdatei derjenigen Arbeitsschritte in der Nutzungsumgebung, die nötig sind, um das Objekt automatisiert und ohne weitere User-Interaktion zur Anzeige zu bringen.

- Die Aufzeichnung benötigt einen Kontext (Kapitel 3.1.3), bestehend unter anderem aus der Emulatorkonfiguration. Dieser Kontext muss natürlich zusammen mit der Aufzeichnung ausgeliefert werden.

Abrufen einer Umgebung für eine Migration

Dieser Use-Case ist mit dem vorigen eng verwandt. Er tritt ein, wenn der PLANETS-PA-Service aufgerufen wird, um mittels Emulationsstrategie eine Objektmigration durchzuführen. Auch hier benötigt er eine entsprechend vorbereitete Nutzungsumgebung.

Dem Archiv übergeben wird die Definition einer Migration. Dabei handelt es sich um ein Tupel bestehend aus Quell- und Zieldateityp, jeweils als PRONOM-UID spezifiziert. Als Antwort auf diese Anfrage liefert das Archiv eine „Migrationsumgebung“. Diese besteht, analog zum vorherigen Use-Case, aus folgenden Teilen:

- Einer Containerdatei, die sämtliche zur Durchführung der gewünschten Migration notwendigen View-Path-Komponenten enthält

- Einem Identifikator für den zugehörigen Emulator

- Eine Aufzeichnungsdatei, die sämtliche zur Durchführung der Migration nötigen Schritte enthält

- Einen Kontext für die Aufzeichnung.

Abfrage einer Liste von unterstützten Objekttypen zur Darstellung

Das PLANETS-Framework verlangt von den PA-CreateView-Services, dass sie auf Anfrage eine Liste der darstellbaren Objekttypen liefern können. Eine solche Anfrage wird durch diesen Use-Case abgebildet. Ein emulationsbasierter PA-Service wird diese Anfrage nämlich nicht selbst

beantworten können, er wird sie vielmehr an das Softwarearchiv weiterreichen. Schließlich ist die Menge der darstellbaren Objekte vom aktuellen Inhalt des Archivs abhängig.

Eingabewerte gibt es in diesem Fall nicht, als Ausgabe erwartet der Aktor eine Liste von Objekttypen in Form vom PRONOM-UIDs.

Ein Nachteil dieser Vorgehensweise („Pull“) ist es, dass der Service nicht seinerseits über Änderungen im Archivbestand informiert wird. Ändert sich der Archivbestand derart, dass sich auch die Menge der darstellbaren Objekttypen ändert, so wird der PA-Service davon nichts mitbekommen. Er wird daher das Archiv in regelmäßigen Abständen abfragen müssen.

Die aus Performancegesichtspunkten sauberere Lösung wäre ein Benachrichtigungsmechanismus entsprechend dem Observer-Pattern („Push“): Der Service würde sich in diesem Fall beim Archiv registrieren, und dann über alle für ihn relevanten Änderungen im Archivbestand über einen Callback-Mechanismus unterrichtet. Um die Komplexität nicht unnötig zu erhöhen, wurde jedoch auf eine genauere Ausarbeitung und praktische Umsetzung dieser Vorgehensweise verzichtet.

Abfrage einer Liste von unterstützten Objekttypen zur Darstellung

Dieser Use-Case ist fast identisch mit dem vorherigen. Der einzige Unterschied liegt darin, dass es sich bei dem Aktor in diesem Fall um einen PA-Migrate-Service handelt, der vom Archiv eine Liste der unterstützten Migrationen abfragt. Dementsprechend enthält die Antwort des Archivs eine Liste von Migrationsdefinitionen, bestehend aus Ziel- und Quell-Objekttyp.

4.1.4 Anforderungen und Aufgaben des Archivverwalters

Der Archivverwalter interagiert mit dem Softwarearchiv mittels einer grafischen Benutzeroberfläche. Dieses GUI soll webbasiert, also mit einem Browser über das Netzwerk nutzbar sein, um eine möglichst flexible, ortsunabhängige Nutzung zu ermöglichen. Die Use-Cases leiten sich nun vor allem von der Tatsache ab, dass der Archivverwalter die im Archiv gespeicherten Daten derart verwalten muss, dass das Archiv die vom Archivnutzer gestellten Anforderungen erfüllen kann. Ein Beispiel hierfür ist das Hinzufügen einer Softwarekomponente zum Archiv. Darüber hinaus wird der Archivverwalter jedoch auch eigene Anforderungen an das Archiv stellen, wie zum Beispiel die Anzeige des Archivinhaltes. Gerade letztere werden im Rahmen dieser Arbeit jedoch nicht mit Anspruch auf Vollständigkeit aufgeführt, da diese Use-Cases sehr zahlreich sein können, ohne zum Verständnis der Problematik nennenswert beizutragen. Insgesamt können, mit der genannten Einschränkung, die in Abbildung 13 zusammen mit den Anwendungsfällen des Archivnutzers gezeigten Use-Cases identifiziert werden.

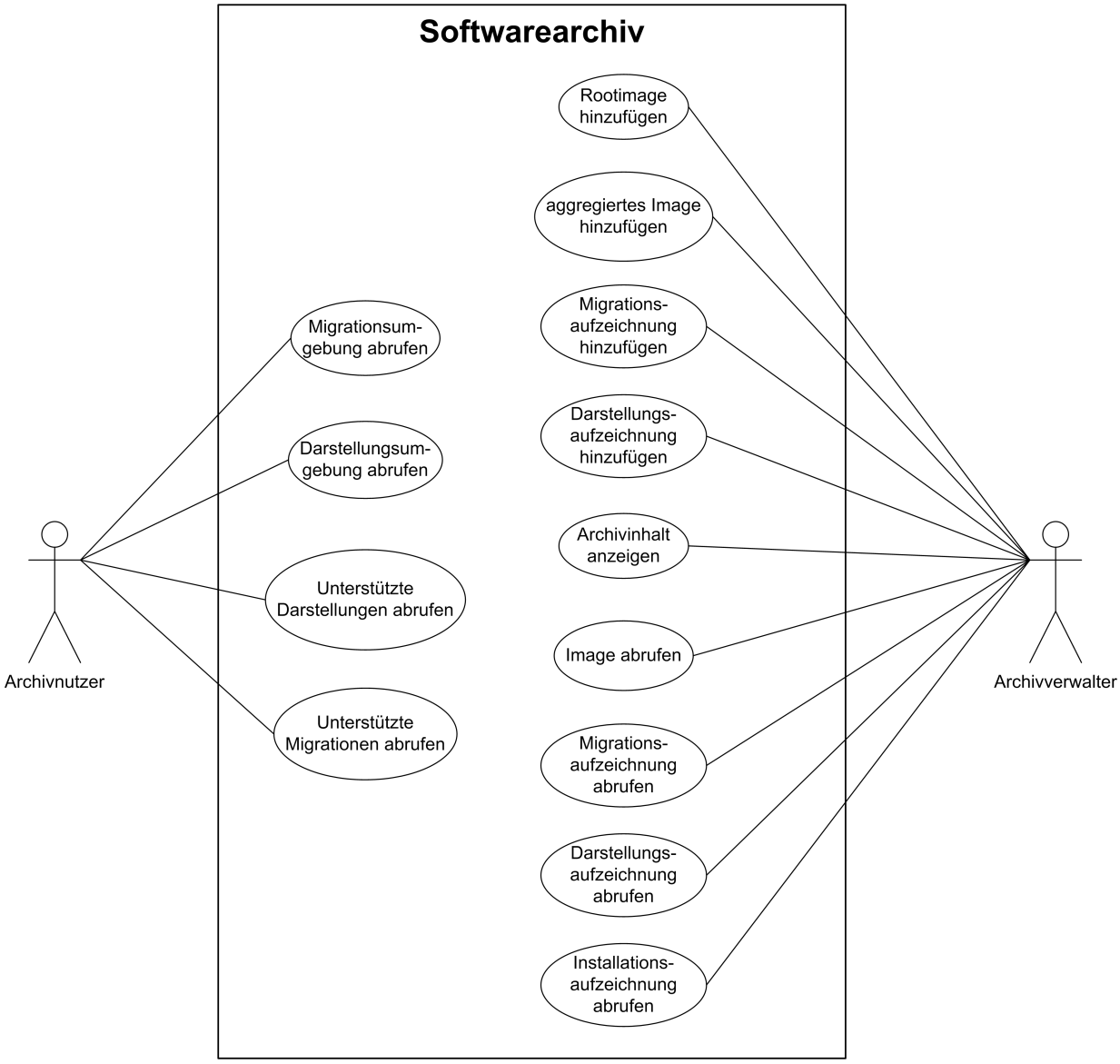


Abbildung 13: Use-Cases des Softwarearchivs

Hinzufügen eines Root-Images

Ein Root-Image ist ein Emulator-Image, welches ein Betriebssystem und eventuell vorinstallierte Zusatzkomponenten als atomare Einheit enthält. Dementsprechend einfach gestaltet sich auch der zugehörige Use-Case: Der Archivverwalter ruft die dem Use-Case zugeordnete Webseite auf, diese präsentiert ihm eine Eingabemaske, in der die benötigten Daten abgefragt werden. Im Einzelnen handelt es sich dabei um folgende Eingabefelder:

- Eine Kurzbeschreibung des Images, welche später beispielsweise in Auswahllisten angezeigt werden kann.
- Eine Auswahlbox für den Emulator, der zur Nutzung dieses Images herangezogen werden soll. In der vorliegenden Umsetzung ist als einziger Wert "QEMU"verfügbar.
- Eine Auswahlbox für eine lokale Datei, die das eigentliche Image enthält.
- Einen Titel für das Betriebssystem. Wenn später der abzulegende Metadatensatz erweitert wird, dann müssen die zusätzlichen Felder hier ebenfalls abgefragt werden.

Nach Ausfüllen aller Felder klickt der Verwalter auf „Archivieren“, wodurch die Daten ans Archiv übertragen werden.

Hinzufügen eines aggregierten Images

Dieser Use-Case stellt eine der zentralen Aufgaben des Archivverwalters dar, und ist deutlich komplexer als das Hinzufügen eines Root-Images. Er dient dazu, eine neues aggregiertes Image ins Archiv einzubringen. Dieses besteht aus einem Basisimage, auf dem eine zusätzliche Softwarekomponente installiert wurde. Ein Blick auf das Datenmodell ergibt die folgenden, vom Archivverwalter zu liefernden Daten:

- Die Angabe eines bereits im Archiv vorhandenen Images, welches als Basisimage für die Installation dienen soll.
- Eine Kurzbeschreibung des neuen, aggregierten Images.
- Die zu installierende Softwarekomponente. Hier kann entweder eine bereits im Archiv vorhandene Komponente gewählt, oder eine neue Komponente hochgeladen werden.
- Falls es sich um eine neue Komponente handelt, so muss der Verwalter dafür ein Installationsdatenträger-Image sowie den Typ des Installationsdatenträgers angeben. Außerdem müssen Metadaten, im vorliegenden vereinfachten Fall nur der Titel, abgefragt werden.
- Eine Installationsaufzeichnung, welche die Installation der Softwarekomponente auf dem Basisimage beschreibt.

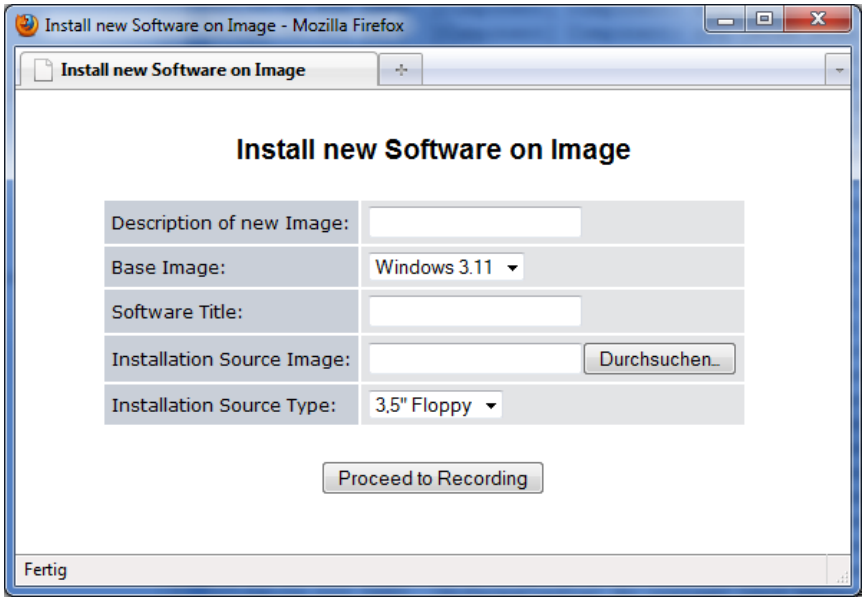


Abbildung 14: Installation einer neuen Softwarekomponente

Der Arbeitsablauf kann grob in zwei Schritte unterteilt werden:

Zuerst erfolgt die Abfrage des Basisimages, der Kurzbeschreibung und der Softwarekomponente über ein Web-Formular. Zur Auswahl des Basisimages wird hierbei eine Liste der im Archiv vorhandenen Images angezeigt. Die Spezifikation der Softwarekomponente unterscheidet sich, je nachdem ob eine bereits archivierte oder eine neue Softwarekomponente installiert werden soll. Im ersten Fall zeigt das Web-Formular eine Liste der im Archiv vorhandenen Komponenten, aus welcher der Verwalter die zu installierende Komponenten auswählt. Im zweiten Fall spezifiziert der Verwalter eine lokal vorliegende Image-Datei samt Datenträgertyp und Titel über entsprechende Felder (Abbildung 14). Die zur Verfügung stehenden Datenträgertypen sind streng genommen von dem, dem Basisimage zugeordneten, Emulator abhängig. Im Fall des im Prototypen einzig implementierten Emulators QEMU sind die Optionen „3,5-Zoll-Diskette“ und „CD-ROM“.

Hierbei ist anzumerken, dass für den vorliegenden Prototypen das Anzeigen der verfügbaren Images und Softwarekomponenten in einfachen Auswahllisten als ausreichend erachtet werden, für die Nutzung in größerem Maßstab wären sicherlich Suchfunktionen über Metadatenfelder zu implementieren.

Nach Abfrage der Daten erfolgt in einem zweiten Schritt die Aufzeichnung der Installation über das in Kapitel 2.4 beschriebene Java-Applet VNCPlay. Hierbei wird dem Archivverwalter das Applet in eine Webseite eingebettet angezeigt, wobei automatisch eine VNC-Verbindung zu einem laufenden Emulator hergestellt wird. Dieser Emulator hat das im ersten Schritt spezifizierte Basisimage gebootet, und den Installationsdatenträger der Softwarekomponente eingebunden. Der Verwalter kann nun die Aufzeichnung starten und die zur Installation nötigen Arbeitsschritte

durchführen. Zu beachten ist dabei, dass das Herunterfahren des Gastsystems auf jeden Fall in der Aufzeichnung enthalten sein muss. Ansonsten bestünde die Gefahr, dass bei einer späteren automatischen Durchführung der Installation das System nicht korrekt heruntergefahren wird, und das resultierende aggregierte Image in einem inkonsistenten Zustand verbleibt.

Hinzufügen einer Migrations- oder Darstellungsaufzeichnung

Die Images alleine sind für den Archivnutzer noch nutzlos, er möchte mit Hilfe der Images Migrationen durchführen oder Primärobjekte visuell darstellen. Der Archivverwalter muss also die Images mit diesen Fähigkeiten versehen. Hierzu benötigt das Archiv einerseits die Angabe des Typs der Migration, beziehungsweise des darzustellenden Objektes, und andererseits eine Aufzeichnung der dafür in der emulierten Umgebung nötigen Arbeitsschritte. Um letztere erstellen zu können, ist darüber hinaus ein Dummy-Objekt nötig, welches bei Erstellung der Aufzeichnung vom Archivverwalter zur Darstellung gebracht beziehungsweise migriert wird. Das Objekt selbst wird selbstverständlich nicht archiviert.

Auch dieser Use-Case gliedert sich, analog zum Hinzufügen eines aggregierten Images, in zwei Teile. Zuerst werden in einem Webformular folgende Daten abgefragt:

- Das Image, auf dem die Aufzeichnung erstellt werden soll.
- Die Angabe, ob es sich um eine Darstellungs- oder eine Migrationsaufzeichnung handeln soll.
- Der Objekt- beziehungsweise Migrationstyp, spezifiziert als PRONOM-ID beziehungsweise Kombination aus Quell- und Zielobjekttyp.
- Der Pfad zu einer lokalen Datei, welche das Dummy-Objekt enthält.

Nach dem Absenden der Daten erhält der Archivverwalter per VNCPlay Zugriff auf eine Ablaufumgebung, in der er die Aufzeichnung durchführen kann.

Im Falle einer Migrationsaufzeichnung folgt in einem dritten Schritt die Verifikation des Migrationsergebnisses. Hierzu erhält der Verwalter auf einer weiteren Webseite einen Link auf eine Ressource, welche das migrierte Objekt darstellt. Hiermit kann er das Objekt in seine Arbeitsumgebung übertragen, und Typ sowie Inhalt des Objekts überprüfen.

Weitere Use-Cases

Neben den bisherigen Anwendungsfällen, die dem Hinzufügen neuer Daten zum Archiv dienen, gibt es noch weitere, die es dem Archivverwalter erlauben, den Inhalt des Archivs abzufragen. Es handelt sich dabei um das Abfragen der archivierten Images und Softwarekomponenten, sowie

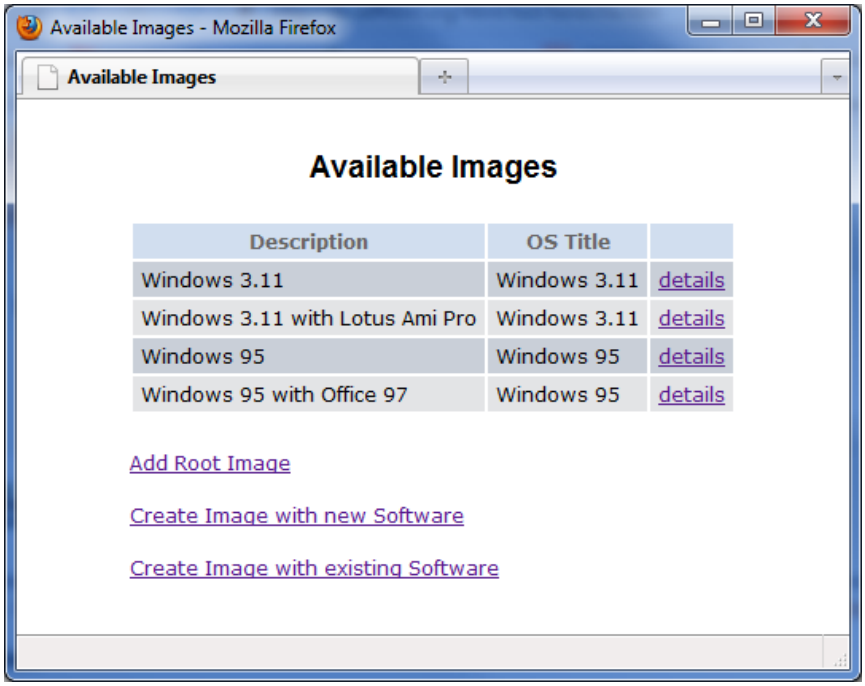


Abbildung 15: Auflistung der im Archiv abgelegten Images

das Abrufen von einzelnen Images, Softwarekomponenten und Aufzeichnungen. Da diese Use-Cases eng miteinander verknüpft, und außerdem vergleichsweise unspektakulär sind, sollen sie hier zusammengefasst dargestellt werden.

Die zwei zentralen, vom Archiv verwalteten Entitäten sind Images und Softwarekomponenten. Die Darstellung des Archivinhaltes besteht daher letztlich aus eine Liste von Images (Abbildung 15) und einer Liste von Softwarekomponenten, die im Browser jeweils in einer Tabelle dargestellt werden. Zu jedem Image wird die beim Hinzufügen angegebene textuelle Kurzbeschreibung, der Titel des Betriebssystems sowie ein Link zur Anzeige von Detailinformationen angezeigt. Ein Eintrag für eine Softwarekomponente besteht aus den definierten Metadatenfeldern (hier: dem Titel), sowie ebenfalls einem Link zu einer Detailseite.

Die Detaildarstellung eines Images (Abbildung 16) beinhaltet folgende Informationen:

- Die definierten Metadaten, im Prototyp also die Titel, aller in diesem Image enthaltenen View-Path-Komponenten, einschließlich des Betriebssystems.
- Die dem Image zugeordneten Migrationen als Quell- und Zieldateityp, sowie jeweils ein Link zum Download der Migrationsaufzeichnung (Use-Case „Abrufen einer Migrationsaufzeichnung“).
- Die dem Image zugeordneten Darstellungen als PRONOM-ID des Objekttyps, ebenfalls samt Link zur Aufzeichnung (Use-Case „Abrufen einer Darstellungsaufzeichnung“).

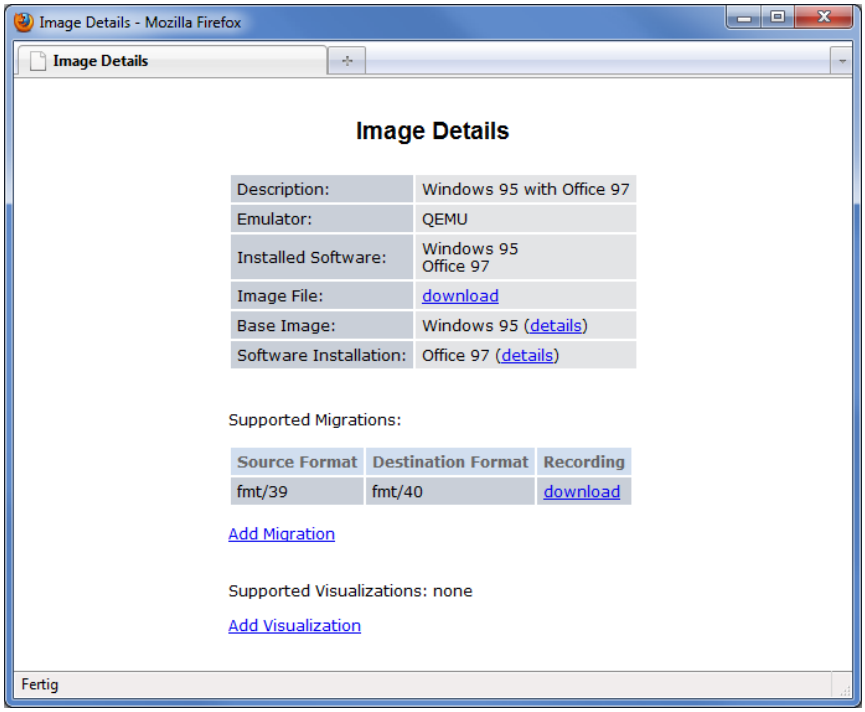


Abbildung 16: Darstellung von Detailinformationen zu einem Image

- Einen Link zum Download des Image-Binaries (Use-Case „Abrufen eines Images“).

Falls es sich bei dem Image um ein aggregiertes Image handelt, enthält die Darstellung darüber hinaus Links zur Detaildarstellung der zugehörigen Softwarekomponente und des Basisimages.

Die Detaildarstellung einer Softwarekomponente besteht aus den Metadaten, einem Link zum Download des Installationsdatenträgers („Abrufen einer Softwarekomponente“) sowie einer Liste derjenigen Images, die durch Installation der Softwarekomponente enstanden sind. Letztere enthält einen Link zum Download der Installationsaufzeichnung („Abrufen einer Installationsaufzeichnung“).

4.2 Architektur

Die Architektur des Prototypen kann grob in drei Teile, oder Schichten, eingeteilt werden:

Persistenzschicht: Hierbei handelt es sich um die Schicht, in der die Daten letztlich abgelegt werden.

Backend: Dieser Teil des Archivs dient als Adapter zwischen Frontend und der Persistenzschicht. Da die Use-Cases der Archivnutzer, also der Preservation Action Dienste des

PLANETS Frameworks, einfache Zugriffe auf die Daten der Persitenzschicht darstellen, und keine weitere Verarbeitungslogik erfordern, kann das Backend die API hierfür direkt bereitstellen. Auf eine weitere Zwischenschicht kann verzichtet werden.

Web-GUI: Das Web-GUI setzt auf dem Backend auf, und bietet dem Archivverwalter eine visuelle, Web-basierte Schnittstelle zur Verwaltung des Archivinhaltes. Hierzu ermöglicht das GUI unter anderem die Interaktion mit einer emulierten Umgebung, um aggregierte Images zu erstellen und Aufzeichnungen durchzuführen. Seine Architektur ist daher weitaus komplexer, als es bei einem reinen Datenverwaltungs-Frontend der Fall wäre, und wird in Kapitel 4.2.2 ausführlich dargestellt.

Die Unterscheidung zwischen Backend und Web-GUI ist dabei eher konzeptionell zu verstehen: Beide laufen, wie auch die PLANETS-Services, in der gleichen virtuellen Maschine, alle diese Subsysteme kommunizieren über einfache Java-Aufrufe.

4.2.1 Backend

Das Backend implementiert die eigentliche Archivierung. Zugrunde liegt das in Kapitel 3.3.2 dargestellte Datenmodell. Es bietet APIs einerseits für die PLANETS-Dienste, zum Abrufen von Migrations- und Darstellungsumgebungen, und andererseits für das Frontend, zur Verwaltung des Archivinhaltes. Die eigentliche Datenspeicherung wird jedoch nicht vom Backend selbst durchgeführt, vielmehr agiert es als Zwischenschicht zwischen dem Frontend, beziehungsweise den PLANETS-Services, und der eigentlichen Persistenzschicht.

Für die Wahl der Persistenzschicht wurden zwei Alternativen in Betracht gezogen, das Fedora Commons Repository und die Verwendung einer relationalen Datenbank. Obwohl die Wahl letztlich auf die zweite Option fiel, soll das Fedora Repository hier dennoch kurz erwähnt werden.

Fedora Commons Repository

Das Fedora Repository ist ein Ablagesystem, welches die Speicherung aller Arten von digitalen Objekten erlaubt [Lagoze u. a., 2006]. Darüber hinaus können Metadaten zu diesen Objekten, sowie Beziehungen zwischen diesen Objekten abgelegt werden.

Dazu definiert Fedora ein eigenes Objektmodell. Objekte in diesem Modell werden durch URIs indentifiziert, und bilden die Knoten eines Graphen, dessen Kanten die Beziehungen zwischen diesen Objekten beschreiben. Objekte können darüber hinaus mehrere Repräsentationen, unter anderem auch in Form von (Dublin-Core-)Metadaten, besitzen. Diese Repräsentationen bilden ebenfalls Knoten im Beziehungsgraphen, sie sind durch eine besondere Art der Beziehung ("hasRep") mit dem Objekt verbunden. Abbildung 17 zeigt ein einfaches Beispiel für einen solchen Graphen.

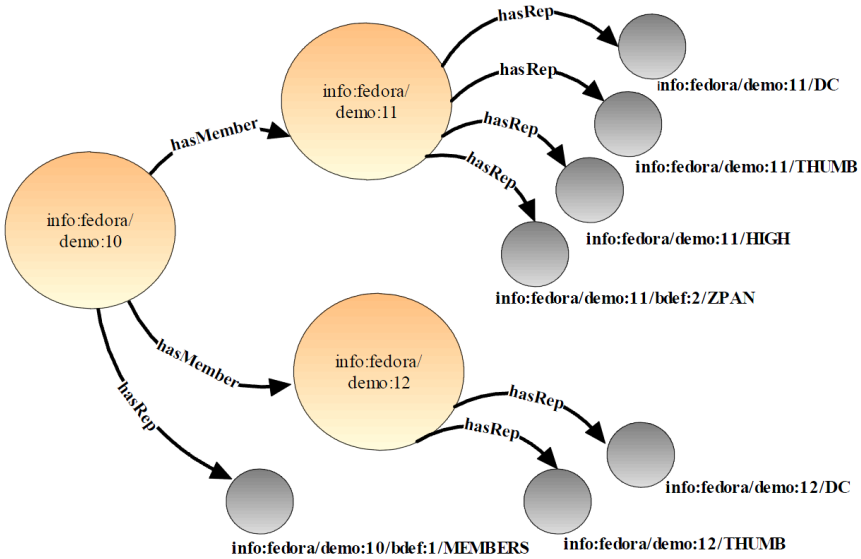


Abbildung 17: Beispiel für ein Instanz des Fedora Objektmodells (aus [Lagoze u. a., 2006])

Die Beziehungen werden in Beschreibungssprache RDF[36] ausgedrückt, was sehr allgemeine Aussagen in Form von sogenannten Tripeln ermöglicht. Jedes Tripel hat dabei die Form <Subjekt> <Prädikat> <Objekt>, wobei Subjekte und Objekte die Knoten, die Prädikate die Kanten des Objektgraphen bilden.

Der Zugriff auf einzelne Objekte, beziehungsweise ihre Repräsentationen, erfolgt über ein Webservices-API. Darüber hinaus können, mit Hilfe von RDF Anfragesprachen wie RDQL[37], Anfragen an den die Objektbeziehungen repräsentierenden RDF Graphen gestellt werden.

Gerade die Möglichkeit, beliebige, als einfacher Datenstrom repräsentierte digitale Objekte, sowie Beziehungen zwischen diesen Objekten zu archivieren, macht Fedora für die Implementierung eines Softwarearchivs interessant. Allerdings handelt es sich hierbei um ein proprietäres, in der Verwendung recht komplexes System, für das, im Gegensatz zum relationalen Modell, keine Standardwerkzeuge zur Implementierung des Datenzugriffs existieren.

Relationale Datenbank

Aufgrund der aufgeführten Nachteile des Fedora Commons Repositories wurde jedoch für den Prototyp letztlich doch auf die andere Alternative gesetzt, die Verwendung eines relationalen Datenbanksystems. Dessen Verwendung ist weitaus weniger komplex, vor allem da das relationale Modell mit SQL als Anfragesprache der Quasi-Standard für eine Persistenzschicht darstellt. Daher sind zahlreiche Standardbibliotheken verfügbar, die den Zugriff erheblich vereinfachen.

[36] RDF=„Resource Description Framework“, `http://www.w3.org/RDF`
[37] RDQL=„RDF Data Query Language“, `http://www.w3.org/Submission/2004/SUBM-RDQL-20040109`

Eine dieser Standardlösungen ist ein sogenannter O/R-Mapper: Seine Aufgabe ist die Übersetzung des objektorientierten Datenmodells in ein relationales Modell, bestehend aus Tupeln und Relationen. Für den Prototyp eingesetzt wurde JPA[38], wobei es sich dabei genaugenommen nur um eine Spezifikation im Rahmen von Java EE 5 handelt. Eine entsprechende Implementierung wird jedoch vom der im vorliegenden Fall eingesetzten Version 4.2 des JBoss Application Server bereitgestellt. Dies war letztlich auch der Grund, wieso JPA der Vorzug vor anderen Mapping-Frameworks, wie beispielsweise Hibernate, gegeben wurde. Die Funktionsweise eines O/R-Mappers kann wie folgt beschrieben werden: Zur Übersetzungszeit erzeugt der Mapper aus dem in Java implementierten objektorientierten Datenmodell ein Datenbankschema. Dabei wird im einfachsten Fall eine Klasse in eine Relation übersetzt. 1:1 oder 1:n Beziehungen werden durch Fremdschlüssel abgebildet. Die Abbildung von n:m und Vererbungsbeziehungen ist weniger trivial, hierbei sind mehrere Strategien denkbar[39]. Zur Laufzeit schließlich können Abfragen in einer eigenen Abfragesprache, in diesem Fall JPQL[40], an das Mapping-Framework gestellt werden. Daraus werden SQL-Abfragen erzeugt, und die Ergebnisse in Form von Java-Objekten zurückgegeben.

Ablage von Binaries

Neben der Abbildung des Datenmodells auf relationale Konzepte, stellt sich bei Verwendung eines RDBMS als Persistenzsystem die Frage nach der Speicherung von Binärdaten. Zwar bieten viele Datenbanksystem die Speicherung von Binärdaten in sogenannten BLOBs[41] an. Jedoch können die Binaries bei einem Softwarearchiv sehr groß werden, man denke an ein Image eines aktuellen Betriebssystems wie Microsoft Windows 7. Hier sind bei einer Speicherung direkt in der Datenbank erhebliche Performanceprobleme zu erwarten. Darüber hinaus ist auf diese Art kein Zugriff auf die Binaries „am Datenbanksystem vorbei“ möglich, was Wartung und Fehlersuche erschwert. Es wurde daher im Backend eine Trennung vorgenommen zwischen der Speicherung der Entitäten in der Datenbank, und der Speicherung der eigentlichen Binaries in einem eigenständigen System, das hier „Binary Store“ genannt werden soll. Die Referenzierung der Binaries in der Datenbank erfolgt dabei über eine URL, unter der es die Klienten des Backends, namentlich PLANETS-Services und Frontend, direkt vom Binary Store abrufen können.

Auch für den Binary Store sind verschiedene Alternativen denkbar:

- Eine Möglichkeit wäre die Speicherung im lokalen Dateisystem. Vorteile sind eine einfache Implementierung und hohe Performanz. Erkauft wird dies durch eine geringe Flexibilität bezüglich des Speicherortes: Das Rechnersystem, auf welchem die JBoss Instanz läuft, müsste in diesem Fall große Mengen von Speicherplatz vorhalten.

[38] JPA=„Java Persistence API“, `http://java.sun.com/developer/technicalArticles/J2EE/jpa`
[39] Eine gute Übersicht über die bei JPA verfügbaren Strategien findet sich in [Backschat und Rücker, 2007, Kap. 5]
[40] JPQL=„Java Persistence Query Language“
[41] BLOB=„Binary Large Object“

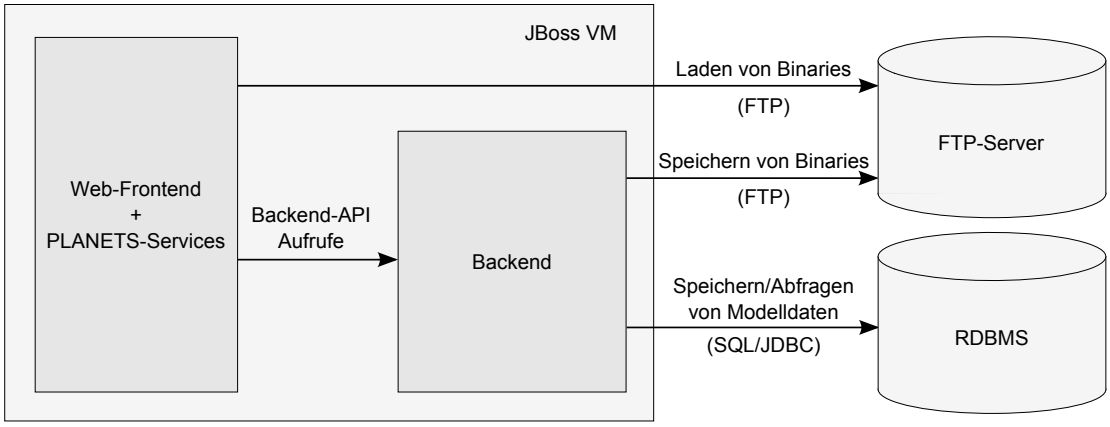


Abbildung 18: Architektur von Backend und Persistenzschicht

- Eine besser skalierbare Lösung, die letztlich auch im Prototyp umgesetzt wurde, ist die Speicherung über ein Netzwerk-Protokoll. Konkret eingesetzt wird hier FTP[42], aber auch andere Protokolle wie SMB[43] wären denkbar.

Abbildung 18 zeigt nochmals im Überblick das Zusammenspiel von Backend, RDBMS, FTP-Server, sowie Web-Frontend und PLANETS-Services.

4.2.2 Web-Frontend

Das Web-Frontend bietet dem Archivverwalter eine visuelle, Web-basierte Schnittstelle zur Verwaltung des Archivinhaltes, insbesondere zum Archivieren von Images und Softwarekomponenten sowie zur Erstellung von Installations- und Migrationsaufzeichnungen. Hierbei implementiert es spezifische Funktionen für die vom Archiv unterstützten Emulatoren, für den hier vorgestellten Prototypen also nur für QEMU. Ein Beispiel hierfür ist der Datenaustausch zwischen realer und emulierter Umgebung. Hierfür werden je nach Emulator unterschiedliche Optionen angeboten, die natürlich auch emulatorspezifisch konfiguriert werden müssen. Diese spezifischen Konfigurationen werden vom Web-Frontend auf das allgemeine Kontext-Konzept des Backends abgebildet, welches dann auch zur eigentlichen Archivierung genutzt wird.

Die Architektur des Frontends basiert in ihren Grundzügen auf der Architektur von GRATE[44]. Hierbei handelt es sich um einen am Lehrstuhl für Kommunikationssysteme der Universität Freiburg entwickelten Demonstrator, mit dem die prinzipielle Möglichkeit gezeigt wurde, emulierte Umgebungen über das Netzwerk in einem Webbrowser zur Verfügung zu stellen. Eine ausführliche Beschreibung dieses Systems findet sich in [Welte, 2008]. Gegenüber der ursprünglichen

[42] FTP=„File Transfer Protocol“

[43] SMB=„Server Message Block“, dieses Protokoll liegt unter anderem dem Dateifreigabe-Dienst von Microsoft Windows zugrunde

[44] GRATE=„Global Remote Access To Emulation-Services“

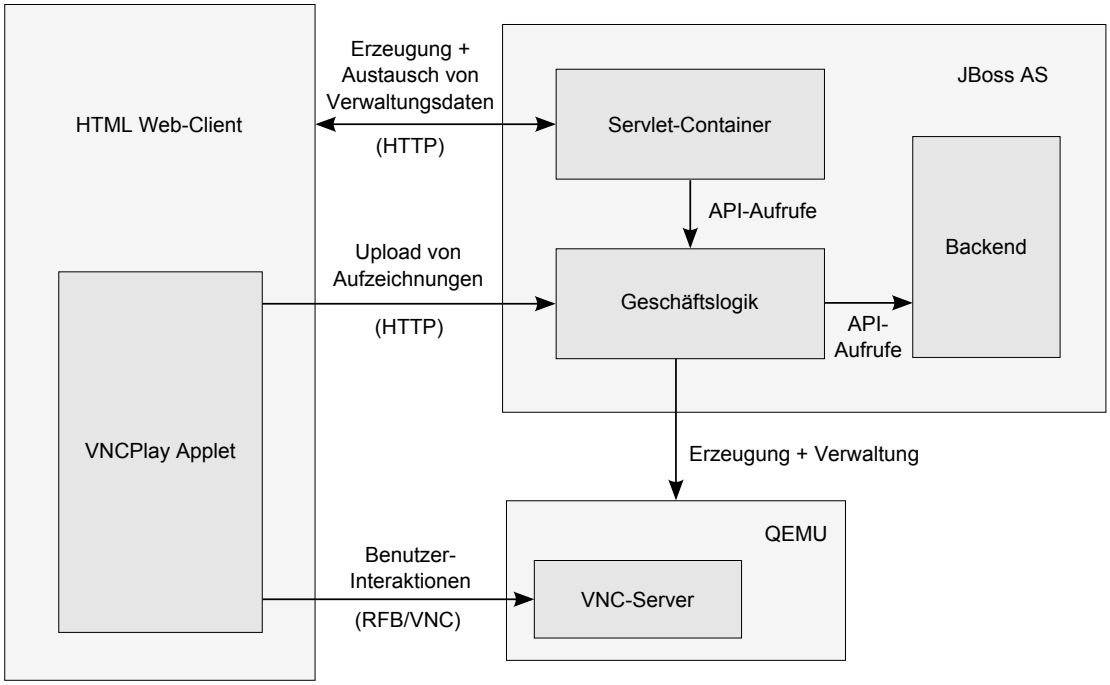


Abbildung 19: Architektur des Web-Frontends

Architektur wurden jedoch, neben den für die zusätzlichen Funktionen des Archivs nötigen Erweiterungen, auch einige Vereinfachungen vorgenommen. Dies diente einerseits der Beschränkung auf die wesentlichen Konzepte, so wurde beispielsweise auf Logging und Zugriffskontrolle verzichtet. Einige andere Vereinfachungen sind aber auch durch die fortschreitende technische Entwicklung bedingt, so bietet der QEMU-Emulator mittlerweile einen integrierten VNC-Server. Auf den in der GRATE-Architektur vorgesehenen externen Tight-VNC-Server kann daher verzichtet werden. Dies gilt natürlich nicht mehr, wenn andere Emulatoren vom Archiv unterstützt werden sollen, die dieses spezielle QEMU-Feature nicht aufweisen.

Grundlage der Architektur ist das in Kapitel 2.4 vorgestellte Konzept, auf die emulierte Umgebung per VNC-Protokoll zuzugreifen, und diese Interaktionen aufzuzeichnen. Als VNC-Server dient dabei der in QEMU integrierte Server, als Client kommt das Java-Applet VNCPlay zum Einsatz. Das Frontend setzt sich also aus folgenden Systemkomponenten zusammen (Abbildung 19):

Application-Server

Hierbei handelt es sich um den bekannten Java EE Application Server JBoss. Er stellt einerseits mit Apache Tomcat einen Servlet-Container zur Verfügung, der als Webserver die Auslieferung der dynamisch erzeugten Webseiten des Frontends übernimmt. Andererseits bietet er auch eine Laufzeitumgebung für die Geschäftslogik. Hierunter ist in diesem Fall unter anderem der

oben erwähnte emulatorspezifische Abbildungscode zu verstehen. Außerdem verwaltet die Geschäftslogik die laufenden QEMU-Instanzen anhand von Heartbeats (siehe unten): Werden keine Heartbeats für eine bestimmte Instanz mehr empfangen, so wird angenommen, dass der Webclient unsauber beendet wurde, und der QEMU-Prozess terminiert. Somit werden „Prozessleichen“ vermieden, die das System auf Dauer enorm belasten würden.

QEMU mit integriertem VNC-Server

Der Emulator übernimmt die Bereitstellung der emulierten Umgebung, in der der Archivverwalter beispielsweise die Installation von Softwarekomponenten durchführen kann. Der Prozess wird von der Geschäftslogik gestartet und verwaltet. Er stellt einen VNC-Port bereit, auf den das VNCPlay Applet zugreifen kann.

Webclient

Die Webseiten des Frontends werden vom im JBoss enthaltenen Tomcat dynamisch erzeugt. Als Webframework kommt JSF[45] zum Einsatz, da JSF Teil der Java EE Spezifikation, und somit in jedem Java EE Application Server verfügbar ist. Die Webseiten bilden, zusammen mit dem VNCPlay Applet, die eigentliche Schnittstelle zwischen Archivverwalter und Archiv, und stellen beispielsweise Eingabemasken zur Abfrage von Daten bereit. Wenn Aufzeichnungen vorgenommen werden müssen, so wird das VNCPlay Applet in die Seite integriert. In diesem Fall ist außerdem Javascript-Code enthalten, der in regelmäßigen Abständen Heartbeats in Form von HTTP-POST Nachrichten an die JBoss Instanz, und damit die Geschäftslogik sendet. Hierdurch wird der Geschäftslogik signalisiert, dass der Webclient noch aktiv ist, und die serverseitige Session samt QEMU-Prozess noch benötigt wird.

VNCPlay Applet

Die zentrale Aufgabe des VNCPlay Applets ist es, den Zugriff auf die vom QEMU bereitgestellte virtuelle Umgebung und die Erstellung von Aufzeichnungen zu ermöglichen. Hierzu wird das VNCPlay Applet bei Bedarf in die Webseiten des Archiv-GUIs eingebettet (Listing 2). Es erhält als Aufrufparameter unter anderem den Hostnamen und den Port des QEMU-VNC-Servers. Diese Parameter sind im einbettenden HTML-Code kodiert, und werden bei dessen dynamischer Erzeugung von der Geschäftslogik abgefragt. Nachdem eine Aufzeichnung erstellt wurde, wird die resultierende Aufzeichnungsdatei, ausgelöst durch den Klick des Archivverwalters auf den Stop-Button, per HTTP-PUT an eine ebenfalls als Aufrufparameter übergebene URL übertragen[46].

[45] JSF=„Java Server Faces“, `http://java.sun.com/javaee/javaserverfaces`

[46] Dieses Feature ist in der Standard-Distribution von VNCPlay nicht enthalten. Eine entsprechende Anpassung ist jedoch möglich, da VNCPlay unter einer Open-Source-Lizenz veröffentlich wurde.

```
<applet archive=vncplay-2.0.jar code="VncViewer.class"
 width=700 height=500>
    <param name="HOST" value="192.168.0.1">
    <param name="PORT" value="5900">
    <param name="autorecord" value="yes">
    <param name="traceurl"
     value="http://192.168.0.1/35312568/recording">
</applet>
```

Listing 2: HTML-Code zur Einbettung des VNCPlay Applets

4.2.3 Beispielhafte Umsetzung eines Use-Cases

Die Funktionen und das Zusammenspiel der einzelnen Systemteile lassen sich am besten anhand eines konkreten Use-Cases darstellen. Als Beispiel soll die Erstellung eines aggregierten Images dienen, wobei eine noch nicht im Archiv befindliche Softwarekomponente installiert wird. Im Sinne einer besseren Übersichtlichkeit werden hierbei Backend, Datenbank und Binary Store in einer atomaren Einheit zusammengefasst dargestellt. Das in Abbildung 20 dargestellte Sequenzdiagramm zeigt den Ablauf und die Interaktionen im Überblick.

Zuerst wird der Archivverwalter mit seinem Browser die diesem Use-Case zugeordnete Webseite aufrufen. Die Webseite enthält eine Liste von verfügbaren Images, aus denen der Verwalter ein Basisimage für die Softwareinstallation auswählen kann. Darüber hinaus bietet sie die Möglichkeit, das Image eines Installationsdatenträgers in Form einer lokalen Datei zum Upload bereitzustellen. Schließlich kann der Typ des Installationsdatenträger-Images angegeben werden, zur Auswahl stehen „Floppy“ und „CD-ROM“. Um diese Webseite dynamisch erzeugen zu können, muss der Application-Server JBoss die Liste der verfügbaren Images vom Backend abfragen.

Nachdem die Webseite dargestellt wurde, wählt der Archivverwalter Basisimage, Softwarekomponente und Datenträgertyp aus, und bestätigt diese Eingaben. Der Browser sendet die Daten daraufhin per HTTP-POST an die im Application-Server laufende Geschäftslogik. Diese erzeugt nun eine Session für diesen speziellen Vorgang, wobei folgende Daten der Session zugeordnet werden:

- Das vom Verwalter hochgeladene Binary des Installationsdatenträgers, welches im lokalen Dateisystem temporär abgespeichert wird.
- Das Binary des ausgewählten Basisimages. Dieses wird durch die Geschäftslogik vom Archiv-Backend geladen, und ebenfalls im Dateisystem abgelegt.
- Eine Prozessinstanz des QEMU-Emulators, welche nach dem Abrufen des Basisimages erzeugt wird. Ein systemweit eindeutiger TCP-Port für den integrierten VNC-Server wird

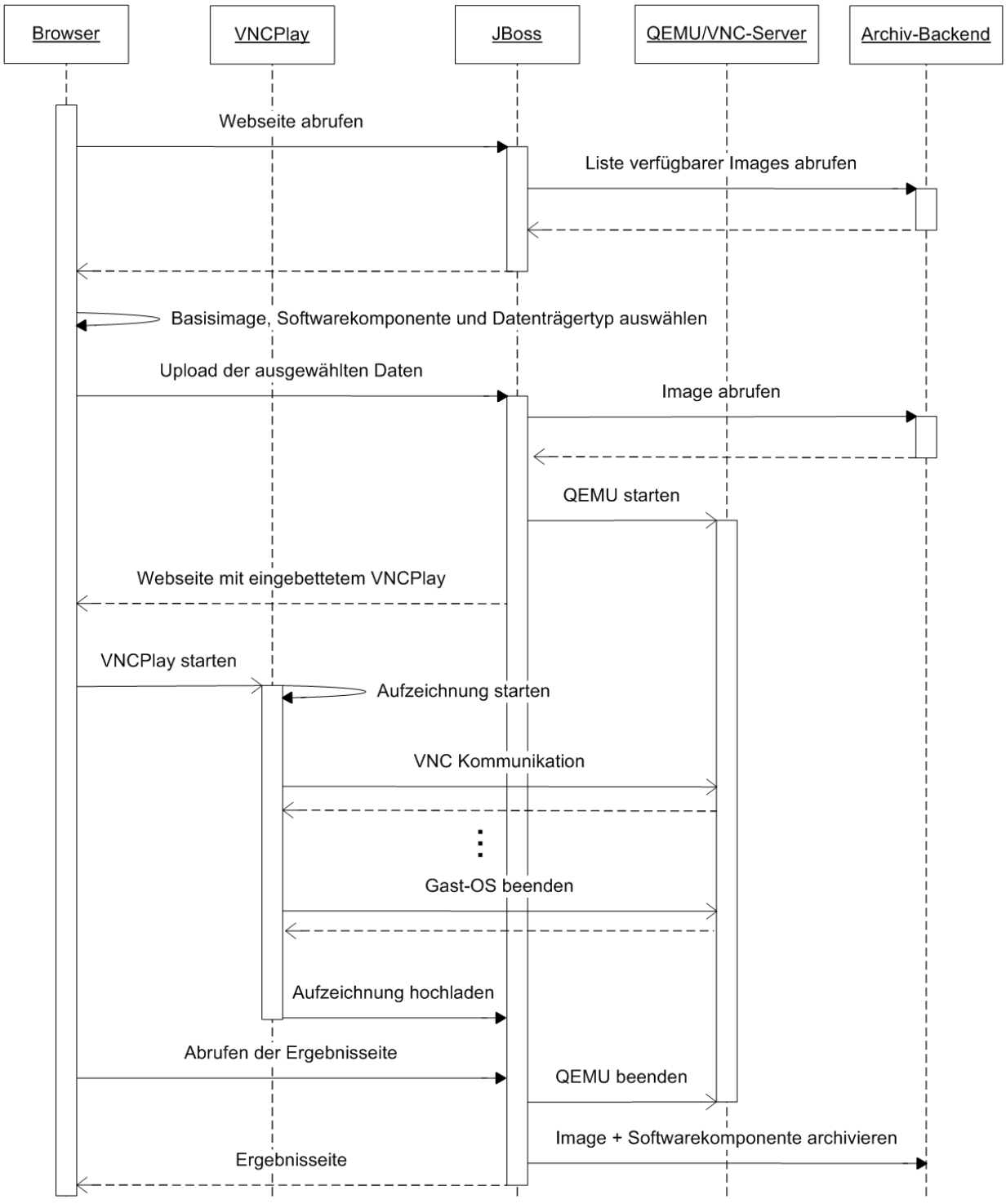


Abbildung 20: Sequenzdiagramm des Use-Cases „Archivierung einer Softwarekomponente“

dem Emulator beim Start als Aufrufparameter übergeben. Gleiches gilt für den Dateinamen des Basisimages, sowie den Namen des Installationsdatenträger-Images, welches auf diese Weise der emulierten Umgebung zur Verfügung gestellt wird.

Nach dem erfolgreichen Start des Emulators wird an den Browser des Nutzers eine Webseite zurückgeliefert, die ein eingebettetes VNCPlay-Applet enthält. Der Hostname und die generierte Portnummer werden als Aufrufparameter des Applets in HTML kodiert. Darüber hinaus wird als Parameter eine URL konfiguriert, die als Ziel-URL für die später zu übertragenende Aufzeichnung dient.

Der Browser startet nun das VNC-Applet. Der Verwalter beginnt mittels des Applet-GUI eine Aufzeichnung, und führt die zur Installation nötigen Arbeitsschritte durch. Diese werden per VNC-Protokoll an den VNC-Server des QEMU übertragen und parallel aufgezeichnet. Die letzte Interaktion muss hierbei das Herunterfahren des Gastbetriebssystems sein. Dies ist notwendig, damit ein eventuell spätere, durch Abspielen der Aufzeichnung durchgeführte Installation, das Image in einem konsistenten Zustand hinterlässt. Nach Beenden der Aufzeichnung überträgt VNCPlay die Aufzeichnung an die zuvor spezifizierte Ziel-URL per HTTP-PUT, die Geschäftslogik ordnet sie der Session zu.

Schließlich signalisiert der Archivverwalter durch Aufruf der Egebnisseite den Abschluss der Verarbeitung. Hierauf beendet die Geschäftslogik den Emulator-Prozess. Das der Session zugeordnete Basisimage wurde nun, durch die per VNC durchgeführten Interaktionen, in ein neues Image überführt, welches die zusätzlich installierte Softwarekomponente enthält. Dieses Image wird nun von der Geschäftslogik, zusammen mit dem Installationsdatenträger und der Installationsaufzeichnung, zur Archivierung an das Backend übergeben. Nach erfolgreicher Archivierung wird abschließend eine Webseite generiert, die den Archivverwalter über den erfolgreichen Abschluss des Vorganges informiert.

5 Schlussbetrachtung

Abschließend sollen die wichtigsten Ergebnisse der Arbeit noch einmal zusammengefasst, und einige Fragen aufgezeigt werden, die über die Ergebnisse der vorliegenden Arbeit hinausweisen.

5.1 Was wurde erreicht?

Grundlage der Arbeit war ein Überblick über die Emulationsstrategie, wobei unter anderem zwischen den zu archivierenden Primärobjekten und Hilfskomponenten zur Emulation, den Sekundärobjekten, unterschieden wurde. Außerdem wurden die wichtigsten Ziele der Strategie aufgezeigt, nämlich Migration und Darstellung von Primärobjekten. Davon ausgehend wurden einige

wichtige Arten von Sekundärobjekten vorgestellt. Es handelt sich dabei um Anwendungsprogramme samt unterstützender Komponenten wie Fonts oder Codecs, die Migrationen durchführen oder Primärobjekte darstellen können. Virtuelle Maschinen und Skriptspracheninterpreter bilden eine weitere wichtige Kategorie von Sekundärobjekten: Sie erlauben es für diese Umgebungen entwickelten Anwendungsprogrammen, auf einer Vielzahl von Betriebssystemen und Hardwarearchitekturen ohne Neuübersetzung ablauffähig zu sein. Betriebssysteme vermitteln, unter Zuhilfenahme von Treibern, zwischen Anwendungsprogrammen beziehungsweise virtuellen Maschinen und der Hardware. Diese ist in der Emulationsstrategie jedoch nicht physikalisch vorhanden, sondern wird durch Emulatoren simuliert.

Folgend wurde ein zentrales Konzept der Emulationsstrategie vorgestellt, der View-Path. Dieser kombiniert die einzelnen, zur Darstellung eines Primärobjekts nötigen Sekundärobjekte. Er stellt einen Pfad ausgehend vom Objekt, hin zur realen Hardware des Benutzers dar. Ein View-Path kann beispielsweise aus einem Anwendungsprogramm, einer virtuelle Maschine, einem Betriebssystem und einem Emulator bestehen.

Schließlich wurde, neben einer Darstellung von Möglichkeiten des Austauschs von Daten zwischen Host- und Gastsystem, eine Technik vorgestellt, die eine Aufzeichnung von Arbeitsabläufen innerhalb einer virtuellen Umgebung erlaubt. Diese Technik ist dabei nicht auf bestimmte Voraussetzungen innerhalb des Gastsystems, wie der Verfügbarkeit von Makrorekordern, angewiesen. Sie basiert vielmehr auf der Aufzeichnung von Benutzereingaben über Maus und Tastatur via VNC-Protokoll, und der automatischen Auswertung des Systemzustandes, beispielsweise anhand des Bildschirminhaltes.

Ausgehend von diesen Grundlagen wurden die Aufgaben eines Softwarearchivs in der Emulationsstrategie genauer betrachtet. Hierbei stellten sich vor allem die Bereitstellung von Ablaufumgebungen für die Migration oder Darstellung eines bestimmten Primärobjekttyps und die Archivierung einzelner View-Path-Komponenten als zentrale Aufgaben heraus. Bei ersterem ist vom Archiv, neben einem Emulator und einem Betriebssystem-Image, welches sämtliche weiteren View-Path-Komponenten enthält, auch noch eine Aufzeichnung zu liefern, welche die zur Darstellung oder Migration eines Primärobjekts nötigen Arbeitsschritte automatisiert ausführbar gestaltet. Die bei der Archivierung einer View-Path-Komponente abzulegenden Daten umfassen einerseits den Installationsdatenträger samt beschreibender Metadaten, andererseits aber auch eine Aufzeichnung, die den Installationsprozess der Komponente abbildet. Eine Sonderrolle spielen hierbei Betriebssysteme und Emulatoren. Erstere sind nicht automatisiert installierbar und müssen daher stets in bereits installierter Form archiviert werden. Bei Letzteren stellt sich das Problem, dass einige Teile des Archivs, insbesondere das Frontend zur Erzeugung von Aufzeichnungen, selbst Abhängigkeiten von konkreten Emulatoren enthalten. Eine generische Archivierung von Emulatoren hat sich daher als nicht sinnvoll erwiesen, vielmehr werden Emulatoren in den archivierten Daten nur per ID referenziert. Weitere Überlegungen ergaben außerdem, dass bestimmte archivierte Entitäten, wie Betriebssystem-Images oder Aufzeichnungen, für sich alleine nicht nutzbar sind. Sie benötigen einen sogenannten Kontext, der zusammen mit der Entität archiviert werden muss. Dieser Kontext kann beispielsweise eine Emulatorkonfiguration ent-

halten, aber auch darüber hinausgehende Angaben, wie den Dateinamen eines darzustellenden Primärobjektes.

Nachdem die Aufgaben des Softwarearchivs definiert waren, wurden zwei mögliche Konzepte zur Erfüllung dieser Aufgaben erarbeitet. Das erste Konzept sieht eine Archivierung von Einzelkomponenten und, auf Anfrage, die dynamische Erzeugung von View-Paths sowie einer kompletten Migrations- beziehungsweise Darstellungsumgebung vor. Hierzu wurden Beziehungen zwischen den einzelnen Komponenten definiert und dargestellt, wie die Komponenten anhand dieser Beziehungen zu einem View-Path kombiniert werden können. Vorteile dieses Konzepts sind eine hohe Modularität und Flexibilität. Es stellte sich jedoch heraus, dass diese Flexibilität, vor allem in Zusammenhang mit Aufzeichnungen, nur sehr eingeschränkt nutzbar ist. Basierend auf diesen Erkenntnissen wurde daher ein zweites Konzept entwickelt, welches die Erzeugung von fertig zusammengestellten Images vom Archiv auf den Archivverwalter verlagert. Die View-Paths werden also nicht mehr dynamisch generiert, sondern statisch archiviert. Auch für dieses Konzept wurde ein Datenmodell entwickelt, welches es dem Archiv erlaubt, alle vorher identifizierten Aufgaben abzudecken.

Zuletzt wurde dann, um die praktische Umsetzbarkeit der theoretischen Konzepte zu demonstrieren, die Implementierung eines Prototypen im Rahmen des PLANETS-Frameworks vorgestellt. Hierzu wurde zuerst eine Use-Case Analyse durchgeführt, welche die Preservation-Action-Services des Frameworks und den Archivverwalter als Aktoren definierte. Anhand dieser Use-Cases wurde dann eine Architektur entwickelt. Neben der Umsetzung des Datenmodells wurde hierbei besonderen Wert auf die Beschreibung des Web-Frontends gelegt, welches unter anderem die Erstellung von Installations-, Migrations- und Darstellungsaufzeichnungen erlauben sollte. Seine Architektur basiert auf der im Grundlagenteil vorgestellten Aufzeichnungstechnik, und setzt unter anderem auf einen als Java-Applet in Webseiten einbettbaren VNC-Client namens VNCPlay.

5.2 Ausblick

Neben den im letzten Abschnitt zusammengefassten Ergebnissen bleiben einige offene Fragen und Möglichkeiten zur Weiterentwicklung. Nur sehr rudimentär gelöst ist bisher beispielsweise die Frage, welches aggregierte Image, und damit welcher View-Path, für einen bestimmten Primärobjekttyp ausgewählt wird, wenn mehrere Images die Darstellung oder Migration dieses Objekttyps unterstützen. Derzeit ist undefiniert, wie diese Auswahl erfolgt, in der Implementierung des Prototypen wird stets das zuletzt mit der entsprechenden Unterstützung versehene Image gewählt. Dem könnte durch die Einführung einer auf View-Paths definierten Ordnungsrelation abgeholfen werden, die der Archivnutzer dem Archiv beim Abrufen einer Ablaufumgebung beispielsweise als Java-Komparator übergibt. Diese Relation könnte bestimmte Präferenzen des Benutzers, wie beispielsweise möglichst originalgetreue Darstellung oder möglichst schnelle Ausführungsgeschwindigkeit, widerspiegeln. Dies setzt jedoch voraus, dass die Metadaten der

einzelnen im Archiv enthaltenen Images mit entsprechenden Informationen, beispielsweise die Angabe der Darstellungsgüte in Prozent, angereichert werden.

Eine andere Erweiterungsmöglichkeit bietet sich bei der Archvierung von Emulatoren. Bisher werden diese in den Archivdaten nur per ID referenziert, da die Interpretation von Kontexten sowie das Web-Frontend emulatorspezifischen Code nötig machen, der bei jedem neu archivierten Emulator angepasst werden müsste. Dies ist ganz offensichtlich nicht optimal, eine vollständige Archivierung von View-Paths wäre dieser Lösung auf jeden Fall vorzuziehen. Ob dies gelingen kann, scheint aber zumindest fraglich. Ein Lösungsansatz könnte darin bestehen, die spezifischen Code-Elemente zu isolieren, und beispielsweise über einen Plugin-Mechanismus generisch behandelbar zu machen. So könnte zu jedem archivierten Emulator ein zugehöriges Plugin archiviert werden, welches den emulatorspezifischen Code enthält. Diese Lösung hat jedoch den Nachteil, dass die Schnittstellen zwischen Archiv und Plugin nur auf abwärtskompatible Weise erweitert werden kann, inkompatible Veränderungen würden die Modifikation sämtlicher im Archiv enthaltenen Plugins nötig machen.

Literaturverzeichnis

[Backschat und Rücker 2007] BACKSCHAT, Martin ; RÜCKER, Bernd: *Enterprise JavaBeans 3.0*. 2. Auflage. München : Elsevier, 2007. – ISBN 978-3-8274-1510-3

[Baumgärtel 2002] BAUMGÄRTEL, Tillmann: Computerspiele: Und weg waren sie. In: *Wochenzeitung - Die Zeit* 52 (2002), S. 60–61

[van Diessen und Steenbergen 2002] DIESSEN, Raymond van ; STEENBERGEN, Johan F.: *The Long-Term Preservation Study of the DNEP project - an overview of the results*. Amsterdam : IBM Netherlands, 2002. – ISBN 90–6259–154–X

[Hoeven und Wijngaarden 2005] HOEVEN, Jeffrey Van D. ; WIJNGAARDEN, Hilde V.: Modular emulation as a long-term preservation strategy for digital objects. In: *5th International Web Archiving Workshop (IWAW05)*, 2005

[Holdsworth und Wheatley 2001] HOLDSWORTH, David ; WHEATLEY, Paul: Emulation, Preservation and Abstraction. In: *RLG DigiNews* 5 (2001), Nr. 4. – ISSN 1093-5371

[Iraci 2005] IRACI, Joe: The Relative Stabilities of Optical Disc Formats. In: *International Journal for the Preservation of Library and Archival Material* 26 (2005), Nr. 2. – ISSN 0034-5806

[Lagoze u. a. 2006] LAGOZE, Carl ; PAYETTE, Sandy ; SHIN, Edwin ; WILPER, Chris: Fedora: an architecture for complex objects and their relationships. In: *International Journal on Digital Libraries* 6 (2006), S. 124–138. – URL `http://arxiv.org/ftp/cs/papers/0501/0501012.pdf`. – Online, letzter Zugriff 24.06.2010

[Mellor u. a. 2002] MELLOR, Phil ; WHEATLEY, Paul ; SERGEANT, Derek: *Migration on Request, a Practical Technique for Preservation*. 2002. – URL `http://www2.si.umich.edu/CAMILEON/reports/migreq.pdf`. – Online, letzter Zugriff 24.06.2010

[Rechert u. a. 2010] RECHERT, Klaus ; SUCHODOLETZ, Dirk von ; WELTE, Randolph: Emulation based services in digital preservation. In: HUNTER, Jane (Hrsg.) ; LAGOZE, Carl (Hrsg.) ; GILES, C. L. (Hrsg.) ; LI, Yuan-Fang (Hrsg.): *JCDL*, ACM, 2010, S. 365–368. – ISBN 978-1-4503-0085-8

[Rechert u. a. 2009] RECHERT, Klaus ; SUCHODOLETZ, Dirk von ; WELTE, Randolph ; DOBBELSTEEN, Maurice van den ; ROBERTS, Bill ; HOEVEN, Jeffrey van der ; SCHRODER, Jasper: *Novel Workflows for Abstract Handling of Complex Interaction Processes in Digital Preservation*. 2009. – URL `http://www.planets-project.eu/docs/papers/Rechert_NovelWorkflows_iPres2009.pdf`. – Online, letzter Zugriff 24.06.2010

[Rothenberg 1999] ROTHENBERG, Jeff: *Avoiding Technological Quicksand: Finding a Viable Technical Foundation for Digital Preservation*. 1755 Massachusetts Avenue, NW, Suite 500,

Washington, DC 20036 : Council on Library and Information Resources, 1999. – ISBN 1-887334-63-7

[Ruzzoli 2010] RUZZOLI, Felix: *Ein Framework für die zustandsbasierte Fehlererkennung und -behandlung von interaktiven Arbeitsabläufen*, Albert-Ludwigs-Universität Freiburg im Breisgau, Bachelorarbeit, 2010

[von Suchodoletz 2009] SUCHODOLETZ, Dirk von: *Funktionale Langzeitarchivierung digitaler Objekte – Erfolgsbedingungen des Einsatzes von Emulationsstrategien*. Cuvillier Verlag Göttingen, 2009. – ISBN 978-3-86727-979-6

[Welte 2008] WELTE, Randolph: *Funktionale Langzeitarchivierung digitaler Objekte - Entwicklung eines Demonstrators zur Internetnutzung emulierter Ablaufumgebungen*, Albert-Ludwigs-Universität Freiburg im Breisgau, Dissertation, 2008