Universität Konstanz
FB Informatik und Informationswissenschaft
Bachelor-Studiengang Information Engineering

Bachelorarbeit

Ein persistentes objektorientiertes Datenmodell für das Personal Information Management in ZOIL

zur Erlangung des akademischen Grades eines
Bachelor of Science (B.Sc.)

Studienfach:	Information Engineering
Schwerpunkt:	Computer Science
Themengebiet:	Angewandte Informatik

von

Michael Zöllner

Matr.-Nr.:	01/622989
Erstgutacher:	Prof. Dr. Harald Reiterer
Zweitgutachter:	Prof. Dr. Marcel Waldvogel
Betreuer:	Hans-Christian Jetter
Einreichung:	30.09.2009

Zusammenfassung

Im Informationszeitalter ist man tagtäglich mit einer immer größer werdenden Masse mehr oder weniger persönlicher Informationen konfrontiert. Aktuelle Betriebssysteme und Werkzeuge sind dabei nicht in der Lage diese Informationen effizient zu verwalten. Die Disziplin *Personal Information Management* sucht nach Lösungen, die jedem Einzelnen die effiziente Verwaltung seines *Personal Space of Information* ermöglichen. Das ZOIL Paradigma kann als eine solche Lösung angesehen werden. Der persönliche Informationsraum ist bei ZOIL in einer zoombaren Informationslandschaft repräsentiert. In dieser Bachelorarbeit wird ein Datenmodell für ZOIL vorgestellt, das die Verwaltung heterogener Informationsobjekte ermöglicht. Der Fokus des Datenmodells liegt dabei auf der flexiblen Modellierung des persönlichen Informationsraums und der Persistenz der zu verwaltenden Informationsobjekte. Für die kollaborative Nutzung von ZOIL ist es außerdem möglich, dass mehrere Geräte gleichzeitig auf einen geteilten Informationsraum zugreifen und Änderungen an diesem in Echtzeit zwischen den Geräten synchronisiert werden. Durch die Visuelle Persistenz von Informationsobjekten auf der Informationslandschaft von ZOIL erhält der Benutzer eine konsistente Benutzeroberfläche, die er nach Belieben verändern kann. Nachdem Anforderungen an das Datenmodell erhoben wurden und die Implementierung diskutiert wurde, wird das Datenmodell unter Performance Gesichtspunkten evaluiert. Die Arbeit schließt mit einem Ausblick auf zukünftige, geplante Arbeiten am Datenmodell.

Abstract

People in the information age are confronted daily with an ever-expanding mass of some sense of personal information. Current operating systems and tools are incapable of efficiently managing this information. The discipline of *personal information management* tries to find solutions to this problem that allow every single one to manage efficiently its *personal space of information*. The ZOIL paradigm can be thought of as such a solution. In ZOIL the *personal space of information* is represented in a zoomable information landscape. This bachelor thesis introduces a data model for ZOIL, which enables the management of heterogeneous information objects. The focus of the data model is on the flexible modelling of the personal space of information and the persistence of the information objects. Using ZOIL in collaborative scenarios it is possible for multiple devices to access a shared information space, whose changes are synchronized in real time between these devices. Through the visual persistence of information objects in the information landscape of ZOIL the user gets a consistent user interface, which he can change at will. After requirements of the data model have been collected and the implementation has been discussed, the data model is evaluated from a performance perspective. The thesis finishes with an outlook on future work on the data model.

Inhaltsverzeichnis

1 Einführung

"One of the effects of living with electric information is that we live habitually in a state of information overload. There's always more than you can cope with."

- Marshall McLuhan [1]

Diese Feststellung aus dem Jahre 1967 ist heutzutage aktueller und bedeutender denn je. Viele Wissensarbeiter der heutigen Zeit kämpfen tagtäglich mit irgendeiner Form von Informationsüberflutung, z.B. bei der Verwaltung von eMails [Whittaker and Sidner, 1996]. Aber nicht nur im Beruf nimmt die Informationsflut kontinuierlich zu, gerade auch im privaten Umfeld werden die Menschen mit einer immer größer werdenden Masse an Informationen konfrontiert [Lyman and Varian, 2003]. Wie Farhoomand and Drury [2002] in einer Studie feststellten, führt dies zunehmend zu Stress und Unproduktivität bei den Betroffenen. Sie können das große Potential, das in den Informationen vorhanden ist, nur unzureichend ausnützen. Das Hauptproblem dieses *Information Overload* genannten Phänomens ist das Volumen der zu verwaltenden Informationen. Außerdem können aus Zeitmangel gar nicht immer alle Informationen überhaupt überblickt und verarbeitet werden. Weitere Punkte bei denen die Personen Schwierigkeiten haben, sind die Trennung von wichtigen und unwichtigen Informationen und das Verwalten mehrerer Informationsquellen [Farhoomand and Drury, 2002]. Der letzte Punkt wird im *Personal Information Management* auch als *Information Fragmentation* bezeichnet. *Information Fragmentation* steht für die Verteilung von Informationen über Anwendungen, Verzeichnisse und Geräte hinweg und wird als eines der grundlegendsten Probleme des *Personal Information Management* betrachtet [Karger and Jones, 2006].

Die Forschungsdisziplin *Personal Information Management (PIM)* macht den täglichen Kampf mit den Informationen zum Gegenstand der Forschung. Dabei werden Verhaltensweisen und Prozesse bei der Informationsverwaltung beobachtet und analysiert und auf dieser Basis neue Strategien, Konzepte und Werkzeuge entwickelt, die die Verwaltung persönlicher Informationen erleichtern sollen.

Lansdale hat bereits 1988 die psychologischen Grundlagen für das *Personal Information Management* ausgearbeitet [Lansdale, 1988]. Unter anderem untersucht er drei unterschiedliche computergestützte Techniken auf ihre Tauglichkeit für digitales PIM. Diese drei Techniken waren das damals weit verbreitete *Direct Access*-Verfahren, damit ist das Zugreifen auf Dateien über eine genaue Verzeichnisangabe gemeint, die *Desktop-Metapher* und das *Spatial Data Management*. Keine der drei Techniken liefert ihm dabei die ideale Unterstützung der kognitiven Prozesse, die beim Klassifizieren und Wiederfinden von Informationen aus psychologischer Sicht notwendig wären.

[1] "The Best of Ideas", CBC Radio 1967

Beim *Direct Access* liegt das Problem für den Benutzer darin zu wissen, welche Informationen er sucht und wie diese im System hinterlegt sind. Die *Desktop-Metapher* hat prinzipiell dieselben konzeptionellen Fehler, versteckt diese aber hinter einer grafisch angereicherten Oberfläche, die schlichtweg einfacher zu bedienen ist als ein textbasiertes Konsolensystem. Als dritte Technik erwähnt Lansdale das *Spatial Data Management System* (SDMS), das Ende der 1970er Jahre von Bolt und Donelson am MIT entwickelt wurde [Bolt, 1979; Donelson, 1978]. Beim SDMS sind die Informationen nicht in Form von Dateien in einer Hierarchie abgelegt, sondern visuell auf einer *Informationslandschaft* verortet. Eine solche Repräsentation der Informationen wurde bereits in den späten 1960er Jahren von Miller [1968] vorgeschlagen, um die Erfahrungen, die der Mensch seit Jahrtausenden auf dem Gebiet visuell-räumlicher Navigation gesammelt hat, für die Navigation in großen Datenräumen auszunutzen:

> *"An aspect of the human use of information that has generally been overlooked in the automation of information services is the human tendency to locate information spatially. Computer-based systems do not necessarily assign any unique role to spatial tags, and so a feature of considerable importance for the organization of the user's memory seems to have been largely overlooked."*[Miller, 1968]

Die Einwände, die Lansdale gegen diese Organisationsform vorbringt, sind teilweise nicht mehr gültig:

> *"Further, it is questionable whether there is enough space (or could be) to physically hold the volume of information the average office worker maintains."*[Lansdale, 1988]

Andere behalten weiterhin ihre Gültigkeit, wie z.B. das Problem der multiplen Kategorisierung von Informationsobjekten bzw. der multiplen Verortung der Objekte auf der Landschaft:

> *"We should expect, therefore, that one of the problems of SDMS will be that users will find themselves 'hunting' around the dataworld looking for clues as to where they left particular documents."*[Lansdale, 1988]

Trotz manch offener Fragen wurde das Konzept der visuell-räumlichen Navigation in Datenräumen und speziell das der Zoomable User Interfaces (ZUI) immer wieder in verschiedenen Arbeiten aufgegriffen [Perlin and Fox, 1993; Bederson et al., 1994; Raskin, 2000] und diente letztendlich auch als Inspiration für das *ZOIL*-Paradigma [König, 2006].

Die vorliegende Arbeit befasst sich in erster Linie mit dem "O" in *ZOIL*, und zwar mit der *objektorientierten Modellierung und Persistierung von Informationsobjekten im Kontext von Personal Information Management*. In den weiteren Unterkapiteln dieser Einführung werden die einzelnen Konzepte (ZOIL, PIM) und technischen Voraussetzungen (ZOIL Framework, Media Room) im Sinne einer Motivation näher vorgestellt. In Kapitel 2 werden die Anforderungen an ein Datenmodell

aus Sicht von ZOIL, PIM und dem Media Room detailliert diskutiert, um dann im nächsten Kapitel die praktische Umsetzung des Datenmodells vorzustellen. Im Kapitel Evaluation wird diese praktische Umsetzung dann auf ihre Tauglichkeit im Umgang mit großen Datenmengen überprüft. Die Arbeit schließt mit einem kurzen Ausblick über mögliche und geplante zukünftige Arbeiten am Datenmodell.

Hinweis: Im Verlauf der Arbeit werden des Öfteren die beiden Begriffe *Datenbackend* und *Datenmodell* auftauchen. Ersterer bezeichnet dabei die Funktionalitäten, die sich primär mit der Persistierung von Informationen beschäftigen, wohingegen letzterer für die Modellierung von Informationsobjekten und des Informationsraums steht. Beide zusammen bezeichnen das Produkt der hier vorliegenden Arbeit und dessen praktischen Teil.

1.1 Personal Information Management

> *"Personal information management (PIM) is both the practice and the study of the activities people perform to acquire, organize, maintain, retrieve, use and control the distribution of information items [...] for everyday use to complete tasks and to fulfill a person's various roles."*[Jones, 2007b]

Beim *Personal Information Management* steht die Verwaltung sogenannter *Information Items* und *Information Collections* im Vordergrund. Diese grundlegenden Einheiten des PIM sind in der Regel in einem *Personal Space of Information* (PSI) integriert. Der PSI ist ein Metakonstrukt, der alle Informationsobjekte, die in irgendeiner Beziehung mit einer Person stehen, umfasst. Wichtig ist hierbei, dass es sich sowohl um digitale (virtuelle), als auch reale Informationsobjekte handeln kann, wobei in dieser Arbeit der digitale PSI im Vordergrund steht.

Die interdisziplinäre Forschung rund um das *Personal Information Management* beschäftigt sich hauptsächlich mit den Aktivitäten, die bei der Verwaltung des PSI anfallen. Dabei wurden einige Tätigkeiten als grundlegend identifiziert, wie beispielsweise das Hinzufügen, Organisieren und Wiederfinden von Objekten [Jones, 2007b; Boardman, 2004]. Neben den Aktivitäten stehen vor allem die Werkzeuge, die für diese Aktivitäten genutzt werden, im Vordergrund der Forschung. Boardman [2004] liefert hierüber eine gute Übersicht.

Jede Person muss zwangsweise in irgendeiner Form *Personal Information Management* betreiben. Es gibt daher unzählige unterschiedliche Ansätze den PSI zu verwalten. Nichtsdestotrotz konnten doch einige grundlegende Muster entdeckt werden. Dies betrifft sowohl die bereits angesprochenen Aktivitäten, wie auch oft wiederkehrende Probleme. Die Probleme beziehen sich hierbei nicht nur auf bestimmte PIM-Werkzeuge (z.B. MS Outlook) oder Techniken, sondern auch ganz grundlegend auf (gewachsene) Konzepte, die in bestimmten Situationen einfach an ihre Grenzen

kommen. Für ein effektives und effizientes *Personal Information Management* müssen die Probleme auf der Konzeptebene gelöst werden, nur dann können auch sinnvolle Techniken und Werkzeuge dafür entwickelt werden.

Im nun folgenden Unterkapitel wird ein solches Problem (die Modellierung des Informationsraums) etwas genauer analysiert.

1.1.1 Modellierung des Informationsraums

Dinge zu abstrahieren und in Klassen einzuteilen ist eine der grundlegendsten Fähigkeiten, die das menschliche Gehirn im Laufe der Evolution gelernt hat. Der Mensch erspart sich dadurch erheblichen Aufwand, z.B. bei der Kommunikation mit anderen Menschen, indem er nicht mehrere Objekte einzeln beschreiben muss, sondern diese durch eine Klasse ersetzen kann.

Objekte in der realen Welt fallen aber leider nicht immer in distinkte Kategorien. Dies kann z.B. bei der täglichen Büroarbeit beobachtet werden, wenn es darum geht, eingehende Post in ein Ordnungssystem einzusortieren. Rechnungen können z.B. nach Datum, nach Firma oder nach der Art der bestellten Waren sortiert werden. Malone [1983] berichtet von einer Studie, in der verschiedene Personen in ihrem Büroalltag beobachtet wurden, um herauszufinden, wie sie mit solchen Problemen umgehen. Dabei wurde herausgefunden, dass es zwei verschiedene Organisationstypen gibt: diejenigen, die ihre Objekte regelmäßig in ein gut organisiertes System einordnen (“filers“) und diejenigen, die die Objekte hauptsächlich auf vielen Stapeln im Büro verteilt liegen lassen (“pilers“). Der Grund für das Stapeln von Dokumenten war dabei bei vielen, dass sie ein Dokument nicht genau in eine Kategorie einteilen konnten und diese Entscheidung daher auf später vertagten oder ganz umgingen, indem sie die Dokumente auf Stapel legten.

Eine ähnliche Erkenntnis wurde auch in einer weiteren Studie gewonnen, bei der die Teilnehmer Beispielkonzepte in bereits vorhandene Kategorien einteilen mussten [Dumais and Landauer, 1983]. War unter den Kategorien eine “miscellaneous“–Kategorie dabei, so schnitten die Teilnehmer bei der Kategorisierung wesentlich schlechter ab, als ohne diese Kategorie. Ein Konzept, das beispielsweise in mehrere Kategorien passte, wurde von den Teilnehmern eher in die “miscellaneous“-Kategorie eingeordnet, als dass sich die Personen für eine, womöglich falsche, Kategorie entscheiden wollten.

Die Klassifikation von Dingen in distinkte Kategorien ist für den Menschen also eine sehr fehleranfällige Tätigkeit. Nach Dumais and Landauer [1983] gibt es dafür verschiedene Gründe: Zum einen ist die Wahl einer geeigneten Bezeichnung für eine Kategorie nicht trivial. Bezeichnungen sind oft mehrdeutig, da sie vom Kontext abhängig sind und bei verschiedenen Personen andere Bedeutungen besitzen können. Beispielsweise ist es nicht eindeutig was die Bezeichnung “Jaguar“ bedeutet, wenn kein geeigneter Kontext (Tier oder Auto) gegeben ist. Ein weiterer Grund betrifft die Kategorien selbst: Es ist nicht einfach, eindeutige Kategorien zu finden, die sich nicht über-

lappen oder mehrdeutig sind. Bei der Klassifikation eines Objekts wird außerdem meist nur eine Dimension oder ein Aspekt berücksichtigt. Betrachtet man die Objekte aus einer anderen Perspektive, ergeben sich völlig neue Kategorien. Beispielsweise können Tiere nach ihrer Art, nach Farbe oder nach Anzahl ihrer Füße kategorisiert werden.

Relevanz für das Personal Information Management Überträgt man diese Problematik in das *Personal Information Management* und speziell auf die Verwaltung von digitalen Informationsobjekten, stellt sich die Frage, wieso alle modernen Betriebssysteme Informationsobjekte in Form von Dateien in einer starren Hierarchie ablegen, die der Klassifizierung in distinkte Kategorien entspricht. Es gibt zwar die Möglichkeit Verknüpfungen auf Objekte in anderen Bereichen der Hierarchie abzulegen um dadurch eine Art multiple Klassifizierung zu ermöglichen, diesen Aufwand betreibt aber kaum ein Benutzer. Es wäre falsch, Hierarchien als ungeeignete Organisationsform für Informationsobjekte zu bezeichnen. Durch ihre charakteristische Baumstruktur erleichtern sie zwar oft den Einstieg in bestimmte Informationsräume, da der Detailgrad von der Wurzel der Hierarchie bis zu den Blättern kontinuierlich zunimmt und man so vom Allgemeinen und Abstrakten zum Detaillierten und Konkreten gelangt. Eine Hierarchie kann aber nur eine Dimension des Informationsraums aufspannen, was sich dann als problematisch herausstellt, möchte man den Informationsraum aus einer anderen Perspektive betrachten. Ein Benutzer verwaltet beispielsweise alle seine Informationsobjekte nach Projekten. Ist er nun auf der Suche nach einem bestimmten Informationsobjekt, z.B. einer Grafik, so wäre es hilfreich, wenn er sich den Informationsraum nach Informationstyp auffächern lassen könnte. Solch eine Funktionalität, auch bekannt als *Facet Browsing*, stellt auf einem aktuellen Betriebssystem, ohne den Einsatz von Zusatzsoftware, ein Ding der Unmöglichkeit dar. Abbildung 1 auf der nächsten Seite zeigt eine Erweiterung[2] für den Mail-Client Mozilla Thunderbird, die die Exploration der eMails eines Postfachs mittels *Facet Browsing* ermöglicht.

Eine einmal getroffene Designentscheidung hinsichtlich der Organisation von Informationsobjekten kann in gängigen Dateisystemen also nicht an ein plötzliches Bedürfnis angepasst werden. Dies liegt vor allem daran, dass die Hierarchie des Dateisystems zwei Funktionen zu erfüllen hat: Das *Speichern* und die *Organisation* der Objekte. Sobald ein Objekt irgendwo in einer Hierarchie abgelegt wurde, ist es an diesem Ort persistiert. Der Nutzer muss sich daher den genauen Ort merken, wenn er auf dieses Objekt zu einem späteren Zeitpunkt zugreifen möchte. Ein *Ort* wird in den meisten Betriebssystemen über eine genaue Verzeichnisangabe und einen Dateinamen repräsentiert. Das Wiederfinden des Objekts bedeutet also die genaue Wiedergabe eines Ortes, oder die Traversierung mehrerer Hierarchiestufen bis zum Ort, an dem das Objekt gespeichert ist. Während Letzteres meist mit einer Kette von Trial-and-Error Versuchen glückt, ist Ersteres ab einer gewissen Anzahl von Objekten ein Ding der Unmöglichkeit. Der Grund hierfür ist, dass Personen sich nicht an Bezeichnungen von Objekten erinnern können, die sie in einem völlig verschiede-

[2] http://code.google.com/p/simile-seek

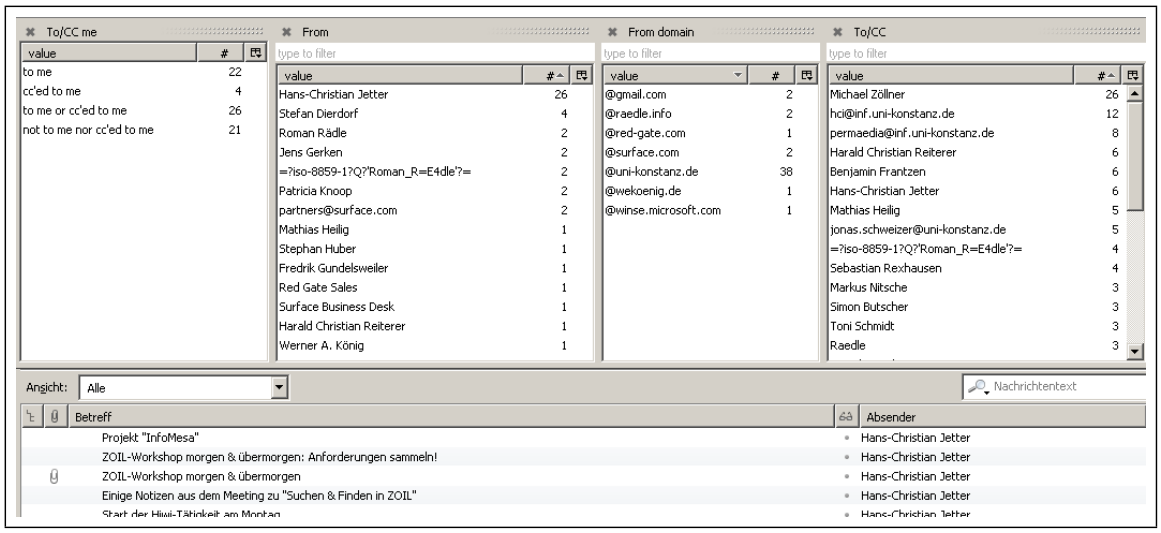

Abbildung 1: Facet Browsing Extension für Mozilla Thunderbird

nen Kontext einem Objekt vergeben haben, oder die womöglich von anderen Personen vergeben wurden. Personen erinnern sich eher an Merkmale wie Aussehen und Inhalt, als an willkürliche Bezeichnungen [Lansdale, 1988]. Die Nutzung einer Hierarchie für das Speichern von Objekten hat demnach folgende Nachteile:

- Der Nutzer muss sich auf eine Dimension festlegen, die er nicht *ad hoc* ändern kann
- Objekte müssen benannt werden
- Objekte dürfen nur an einem Ort in der Hierarchie eingehängt werden.

Eine Trennung der beiden Funktionen (*Organisation* und *Persistenz*) wäre für den Benutzer wesentlich besser. Ein Objekt könnte gespeichert werden, ohne dass der Benutzer ihm (explizit) einen Namen gibt und ohne dass er sich zu diesem Zeitpunkt festlegen muss, an welchem Ort in einer Hierarchie dieses Objekt zu stehen hat. Hierarchien könnten dann wesentlich flexibler genutzt werden, indem der Informationsraum je nach Bedürfnis des Nutzers dynamisch zu einer bestimmten Dimension aufgefächert wird. Eine bestimmte Hierarchie wäre dann nur eine bestimmte Sicht auf den Informationsraum. Von diesen Sichten könnte es unzählige geben, je nachdem welches Merkmal der Objekte gerade von Interesse ist. Eine solche Informationsmodellierung wird im Allgemeinen als *flach* bezeichnet, da die Objekte nur an einem Ort gespeichert werden [Indratmo and Vassileva, 2008].

Der Erfolg solch einer Technik hängt natürlich stark davon ab, wie viele Informationen die Objekte über sich preisgeben. Selbst wenn durch Informationsextraktion viele Metadaten der Objekte generiert werden können, sagen diese Informationen meist nicht so viel über das Objekt aus, wie

eine manuelle Klassifizierung des Benutzers es tun könnte.

Bei einer hierarchischen Modellierung ist der Nutzer dazu gezwungen seine Informationsobjekte nach einer Dimension zu klassifizieren, während bei dieser flachen Modellierung die Möglichkeit besteht, die Objekte nach beliebigen Dimensionen zu klassifizieren. Diese Freiheit führt zwangsweise zu größerem Aufwand bei der Klassifizierung. In vielen webbasierten Systemen wird die Klassifizierung beispielsweise über Schlagworte (Tags) ermöglicht. Hier wird hauptsächlich die Intelligenz der Masse [Surowiecki, 2005] ausgenutzt um eine möglichst vollständige Klassifizierung eines Objekts zu ermöglichen, der Aufwand für den Einzelnen bleibt gering. Angesichts dessen, dass sich der Benutzer in der Regel so wenig Aufwand wie möglich mit der Organisation seines Informationsraums machen möchte, ergibt sich folgendes Dilemma:

> *"The more we ask the user to do at the process of information storage, the less likely he is to do it, creating retrieval problems. On the other hand, the more we automate the process of storage and take responsibility away from the user, the less he is going to remember, and therefore the less he is going to be able to retrieve."*[Lansdale, 1988]

Diese Aussage stützt sich auf psychologische Erkenntnisse des Lernens, die jeder an sich selbst schon erfahren konnte. Inhalte, die selbst erarbeitet wurden, bleiben länger im Gedächtnis erhalten, wie Inhalte, die nur konsumiert werden. Lansdale [1988] schlägt als Lösung für dieses Dilemma Folgendes vor: Entweder muss die Forschung herausfinden, wie man den Benutzer dazu motivieren kann, notwendige Tätigkeiten, wie z.B. die Klassifizierung, durchzuführen. Wenn man sich aber auf automatische Klassifizierung verlassen will, muss herausgefunden werden, welche Merkmale von Benutzern zum Wiederfinden von Objekten im Allgemeinen benutzt werden. Zum ersten Ansatz können vor allem Techniken gerechnet werden, die dem Benutzer Vorschläge für die Klassifikationsentscheidung machen [Jäschke et al., 2008], oder die Eingabe von Schlagworten erleichtern [Hong et al., 2008]. Der zweite Ansatz war bereits Thema vieler Forschungsprojekte [Freeman and Gelernter, 1996; Nardi et al., 2002; Kaptelinin, 2003; Lansdale and Edmonds, 1992; Ringel et al., 2003; Shneiderman and Plaisant, 1994], führte aber im Allgemeinen dazu, dass ein Merkmal besonders hervorgehoben wurde und das daraus entwickelte System sich primär auf dieses Merkmal versteifte. Die Informationsobjekte wurden wiederum nur nach einer Dimension (wie in der Hierarchie) angeordnet.

Selbst wenn das gerade beschriebene (Datei)-System existieren würde, wir also unseren persönlichen Informationsraum *flach* modellieren könnten und durch die Vergabe von Schlagworten eine multiple Klassifizierung erreichen könnten, bestehen doch Zweifel, ob dieses System ausreicht um unseren Informationsraum vollständig zu modellieren und ihn auf die Aktivitäten vorzubereiten, die typischerweise im *Personal Information Management* durchgeführt werden. Ein wichtiger Aspekt wurde bis jetzt nämlich noch nicht berücksichtigt:

> *"The human mind does not work that way. It operates by association."*[Bush, 1945]

Das menschliche Gedächtnis wird in der Psychologie unter anderem als semantisches Netzwerk repräsentiert. Knoten in diesem Netzwerk repräsentieren Konzepte und Kanten zwischen den Knoten stellen die Beziehungen zwischen den Konzepten dar. Vannevar Bush zieht daraus den Schluss, dass eine ideale externe Wissensrepräsentation an diejenige des Gedächtnisses angelehnt sein muss. Auf dieser Feststellung basiert dann auch seine Wissensmaschine *Memex*, bei der gespeicherte Dokumente mit anderen (mechanisch) verknüpft werden können, damit letzten Endes das Wiederfinden (Bush spricht von *Selection*) erleichtert werden kann [Bush, 1945]. Die Idee solch einer Maschine inspirierte einige Forscher, sich mit dieser Art der Wissensrepräsentation zu beschäftigen. Allen voran begann Ted Nelson Anfang der 1960er Jahre seine Vision des Hypertexts zu entwickeln [Nelson, 1965]. Sein Xanadu-Projekt[3] sollte eine virtuelle Version des Memex von Bush realisieren, war dabei aber wesentlich mächtiger konzeptioniert, so dass es nie komplett umgesetzt wurde. Inspiriert von dieser Vision, entwickelten Forscher viele verschiedene Hypertext-Systeme[4], deren bekanntester Vertreter wohl das *World Wide Web* (WWW) ist [Berners-Lee et al., 1994][5]. Das *World Wide Web* setzt aber nur eine kleine Untermenge der ursprünglich angedachten Hypertext-Konzepte um. Unter anderem besitzen die Knoten des WWW, die durch eine eindeutige Adresse (URL) referenziert werden können, keinerlei semantischen Informationen. Diese Restriktion wurde bereits sehr bald erkannt und führte zur sog. *Semantic Web Initiative* [Berners-Lee et al., 2001]. Das *Semantic Web* stellt in erster Linie eine Weiterentwicklung des bereits vorhandenen Webs dar, in dem die Knoten (oder Ressourcen) des Netzes mittels einer formalen Beschreibungssprache semantisch detaillierter beschrieben werden, damit Maschinen aus den Informationen Schlüsse ziehen können.

Die Idee, ein semantisches Netzwerk für die Modellierung des Informationsraums zu benutzen, findet auch im *Personal Information Management* großen Anklang. Informationsobjekte im persönlichen Informationsraum sind sehr heterogen, verändern sich ständig und zwischen den einzelnen Objekten bestehen oft bestimmte Verbindungen. Mit diesen Anforderungen kommen flache Organisationsformen oder Hierarchien nicht zurecht. Beim Haystack Projekt[6] wurde versucht, die Technologien des *Semantic Web* für ein PIM-Werkzeug zu benutzen. Die Grundannahme der Entwickler von Haystack ist, dass sich der persönliche Informationsraum ständig ändert und ein Werkzeug sich diesen Veränderungen anpassen muss [Karger, 2007]. Im Gegensatz zu aktuellen PIM-Werkzeugen, bei denen der Werkzeugentwickler eine Struktur für die Verwaltung des Informationsraums vorgibt, kann der Informationsraum in Haystack nach persönlichen Vorstellungen modelliert werden. Als Grundlage für diese Flexibilität dient das *Resource Description Framework* (RDF). Informationsobjekte werden über einen eindeutigen *Uniform Resource Identifier* (URI) adressiert und können mit beliebigen anderen Informationsobjekten verbunden werden. Verbindungen zwischen Objekten sind typisiert, so dass das Konstrukt aus zwei Objekten und einer Ver-

[3] http://xanadu.com/

[4] Für eine Übersicht der Hypertext-Geschichte siehe http://de.wikipedia.org/wiki/Chronologie_der_Hypertext-Technologien

[5] Erstes Proposal des WWW: http://www.w3.org/History/1989/proposal.html

[6] http://groups.csail.mit.edu/haystack/

bindung eine gewisse Bedeutung ausdrückt. Mithilfe einer speziellen Abfragesprache kann ein so modelliertes Netzwerk dann durchsucht und analysiert werden. In der grafischen Benutzerschnittstelle (siehe Abbildung 2), die bei Haystack sehr klassisch mit Listen und Tabellen realisiert wurde, finden sich kaum Anzeichen dafür, dass der Informationsraum mit einem semantischen Netzwerk modelliert wurde. Der Benutzer arbeitet in einer ähnlichen Umgebung, wie sie z.B. MS Outlook bietet, hat aber weitaus mehr Möglichkeiten seine Informationsobjekte miteinander in Beziehung zu setzen und kann den Informationsraum von vielen Seiten betrachten.

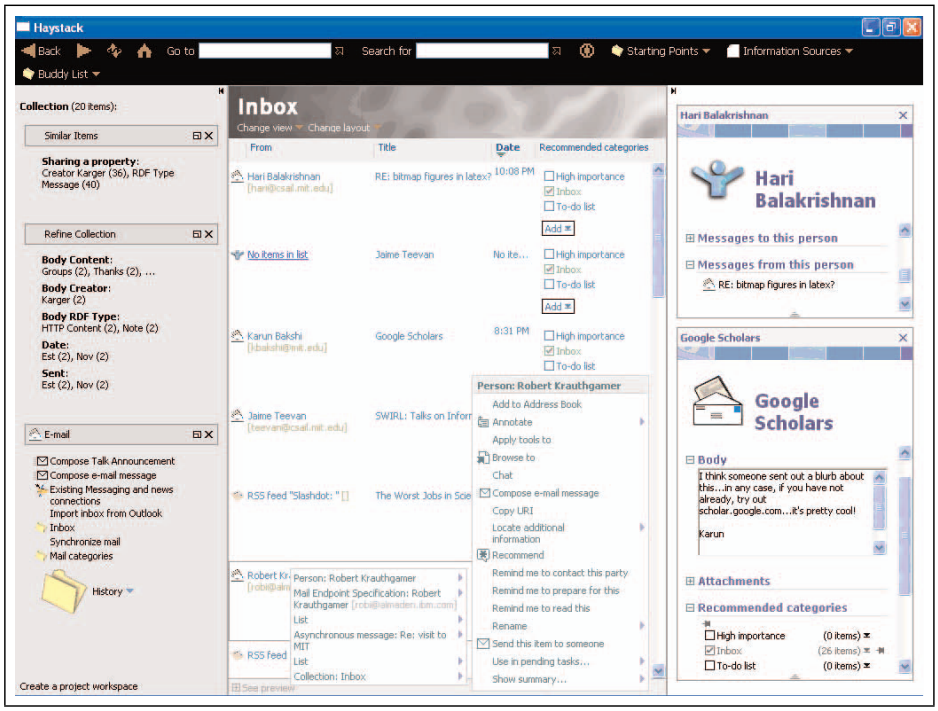

Abbildung 2: User Interface von Haystack [Karger and Jones, 2006]

Der Benutzer eines PIM-Tools möchte sich in der Regel nicht um die gerade beschriebenen Details der Informationsmodellierung kümmern. Je nach Aktivität bietet ihm eine der vielen verschiedenen Modellierungen (Hierarchie, flach, Netzwerk, ...) gewisse Vorteile gegenüber den Anderen. Ein ideales PIM-Tool sollte daher dem Benutzer alle Möglichkeiten der Informationsmodellierung anbieten, ohne ihn in ein bestimmtes Schema zu zwängen.

Eine Untersuchung gängiger Organisationsstrukturen für das *Personal Information Management* kommt zu folgendem Schluss:

> *"To take advantage of the existing organizational structures, we propose to treat them as views besides as methods to manage information items."*[Indratmo and Vassileva, 2008]

Das ideale PIM-Tool ermöglicht dem Benutzer also einen möglichst flexiblen Einstieg in seinen Informationsraum durch die Verwendung verschiedenster Organisationsstrukturen, wie z.B. Hierarchien, Semantische Netzwerke, Schlagworte, lineare und räumliche Anordnung. Mit diesen Strukturen wird er aber nur durch die Benutzeroberfläche und die Interaktionsmöglichkeiten konfrontiert; wie die Informationen intern modelliert und persistiert werden, soll ihn nicht interessieren.

1.2 Der Media Room

Der *Media Room*[7] der AG MCI der Universität Konstanz ist ein Labor für die Entwicklung innovativer Ein- und Ausgabegeräte, Interaktionstechniken und User Interfaces. Zu diesem Zweck ist er mit ausreichend aktueller und zukünftiger Hardware und Technik ausgestattet.

Große Datenmengen lassen sich auf zwei großen (65”), hochauflösenden (1920 x 1080 px) Cube-Displays, sowie einem Apple Cinema-Display mit 30” Bilddiagonale darstellen. Für die Interaktion mittels Fingern und physischen Objekten (Tokens) eignen sich mehrere Multitouch-Tische, unter anderem ein Exemplar des Microsoft Surface[8]. Als weitere innovative Eingabegeräte stehen ein Laserpointer und ein Nintendo Wii-Controller zur Verfügung. Zeichnungen und Handgeschriebenes können mittels eines Anoto-Pens[9] unmittelbar vom Papier auf den Bildschirm übertragen werden und lassen sich somit schnell in Anwendungen einbinden. Über ein Motion-Tracking System können sogar die Bewegungen der Benutzer oder bestimmter Objekte für die Interaktion mit den Systemen herangezogen werden. Die Kommunikation und Integration all dieser Geräte ermöglicht das Squidy-Toolkit, eine Eigenentwicklung der AG MCI der Universität Konstanz [König et al., 2009].

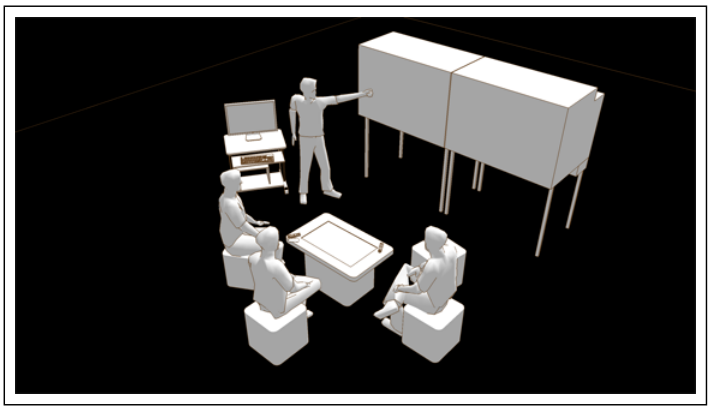

Abbildung 3: Der Media Room der Universität Konstanz als Sketch

[7] http://hci.uni-konstanz.de/index.php?a=research&b=mediaroom&lang=en
[8] http://www.surface.com
[9] http://www.anoto.com

Die besondere Infrastruktur des *Media Rooms* fördert geradezu seine Nutzung für kollaboratives Arbeiten mit digitalen und analogen Informationen. Beispielsweise kann man sich gut vorstellen, dass ein Multitouchtisch als Steuergerät für die großen hochauflösenden Displays dient. Informationsobjekte die auf dem Multitouchtisch aus Platzgründen nicht angezeigt werden können, könnten dann interaktiv auf die hochauflösenden Cubes geschoben werden, wo sie von allen Nutzern betrachtet werden können.

Dieses und andere Szenarien können natürlich nur dann funktionieren wenn die Geräte, die an der Interaktion beteiligt sind, nicht nur technisch miteinander kommunizieren (via Squidy), sondern auch auf der Datenebene eine gemeinsame Basis besitzen. Das hier vorgestellte Datenmodell soll diese Funktion übernehmen, unter anderem durch die Persistenz und Echtzeitsynchronisation der beteiligten Informationsobjekte.

1.3 ZOIL - Die Vision

> *"Das ZOIL Paradigma beschreibt keine konkrete Benutzeroberfläche oder Applikation, sondern soll als eine in sich konsistente Kombination aus Visualisierungs- und Interaktionstechniken verstanden werden, welche als variables Grundkonzept für vielfältigste Anwendungsdomänen dienen kann."* [König, 2006]

Der zentrale Bestandteil des ZOIL-Paradigmas ist die Informationslandschaft. Auf dieser theoretisch unendlichen Fläche werden Informationsobjekte, die aus beliebigen Quellen stammen, nach objekt-orientierten Gesichtspunkten dargestellt. Die Navigation in diesem Informationsraum erfolgt mittels Zooming und Panning. Neben der explorativen Navigation im Informationsraum ist auch eine eher analytisch motivierte Navigation möglich. Dazu können auf der Informationslandschaft Portale geöffnet werden, die den darunter liegenden Teil der Landschaft mit speziellen Visualisierungstechniken anzeigen.

Dieses Konzept einer *Zoomable Object-Oriented Information Landscape* wird erstmals in der Masterarbeit von Werner König erläutert und in Form einer Machbarkeitsstudie vorgestellt [König, 2006].

Gerken [2006] befasst sich in seiner Masterarbeit mit der Orientierung und Navigation in zoombaren Landschaften unter dem Aspekt kognitions-psychologischer Erkenntnisse und diskutiert in diesem Zusammenhang das ZOIL-Paradigma.

Die dritte Masterarbeit zu diesem Thema stammt von Jetter [2007]. Er beschäftigt sich hauptsächlich damit, das ZOIL-Paradigma auf eine solide theoretische Basis zu stellen. Dies geschieht durch die Formulierung von vier *Designprinzipien* für ZOIL und der Entwicklung einer visuellen Spezifikation für einen spezifischen Anwendungsfall. Später werden die vier *Designprinzipien* von

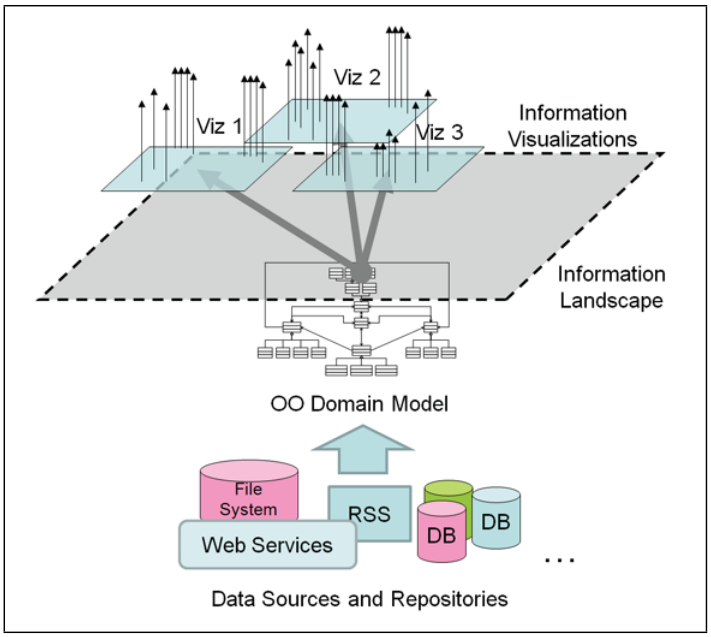

Abbildung 4: Übersicht über das ZOIL-Paradigma [Jetter et al., 2008b]

Jetter et al. [2008b] um ein weiteres Prinzip ergänzt. Die Analyse dieser Prinzipien und die Anforderungen, die sich daraus für das Datenmodell ergeben, werden im Kapitel 2.2 auf Seite 28 beschrieben.

ZOIL war ursprünglich als Ersatz für die normale Desktop-Benutzeroberfläche gedacht:

> *"ZOIL is aimed at unifying all types of local and remote information items with their connected functionality and with their mutual relations in a single visual workspace as a replacement of today's desktop metaphor."* [Jetter et al., 2008b]

Diese Definition behält weiterhin ihre Gültigkeit, jedoch entstehen durch das Erscheinen neuer Interaktionstechniken und Eingabegeräte, wie sie beispielsweise der *Media Room* bietet, neue Möglichkeiten für den Einsatz von ZOIL (siehe z.B. [Jetter, 2009]). Die nächste Generation von ZOIL wird sich daher nicht mehr ausschließlich mit "normalen" Desktop-Computern und deren grafischer Oberfläche beschäftigen. Man wird sich auch zunehmend mit Themen, wie *reality-based interaction* [Jacob et al., 2007, 2008], *embodied interaction* [Dourish, 2001] oder *instrumental interaction* [Beaudouin-Lafon, 2000] auseinander setzen und die sich daraus ergebenden Konzepte in das ZOIL-Paradigma integrieren müssen.

1.4 ZOIL - Das Framework

Neben der theoretischen Entwicklung des ZOIL-Paradigmas war die Entwicklung von Prototypen notwendig, um das ZOIL-Konzept in der Praxis zu überprüfen und zu verfeinern. Einen ersten Prototyp hat König in der ersten Arbeit über ZOIL vorgestellt [König, 2006].

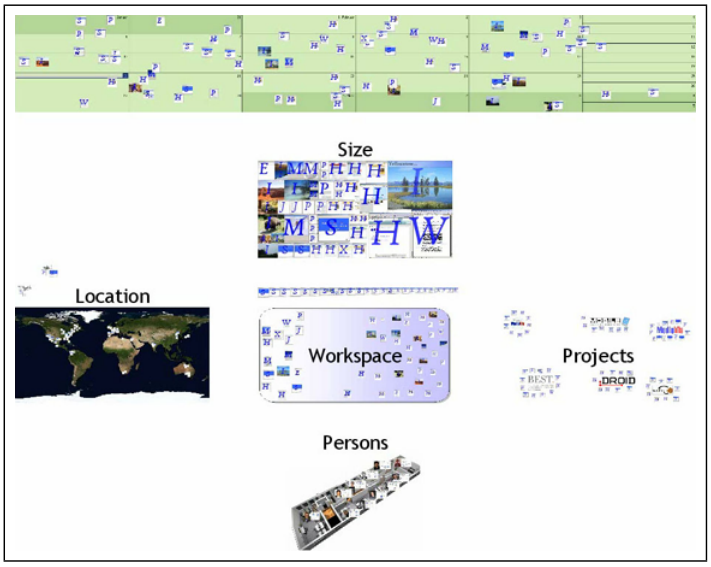

Abbildung 5: Der erste ZOIL-Prototyp von König [König, 2006]

Jetter beließ es bei einer visuellen Spezifikation eines ZOIL-Prototypen für die Anwendungsdomäne “Mediathek“ [Jetter, 2007].

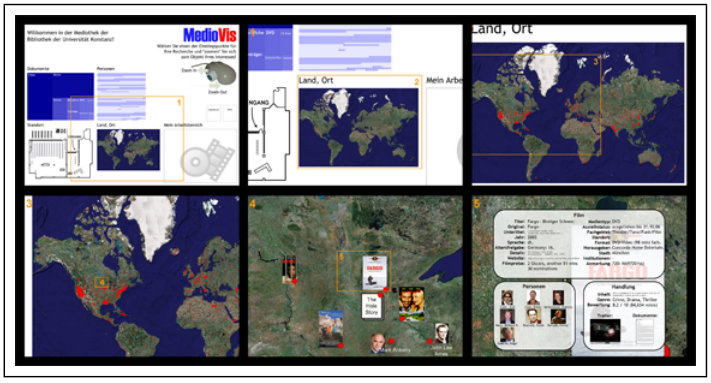

Abbildung 6: Visuelle Spezifikation eines ZOIL-Prototypen von Jetter [Jetter, 2007]

Da beim Entwickeln von Prototypen das Interaktionsdesign und der Aufbau der Landschaft klar im Vordergrund stehen, sollte sich ein Designer weniger um die technischen Aspekte kümmern müssen. Es bot sich daher an, ein Framework zu entwickeln, das zukünftigen ZOIL-Designern das strukturierte Entwerfen von Prototypen ermöglicht, ohne sich um grundlegende Dinge kümmern zu müssen.

Auf Basis von .NET/WPF wurde zuerst die visuelle Basis des Frameworks erstellt [Engl, 2008]. Mit dieser ersten Version war es bereits problemlos möglich, Informationsobjekte in einer theoretisch unendlichen, zoombaren Landschaft zu platzieren und mit dieser Landschaft und den darin liegenden Objekten zu interagieren.

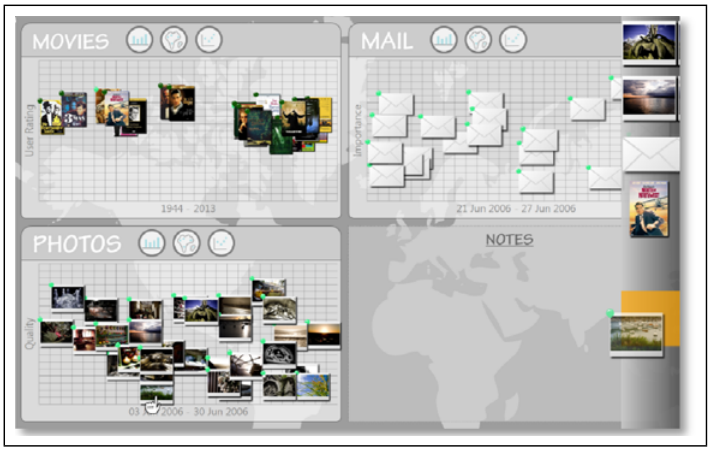

Abbildung 7: Der erste ZOIL-Prototyp auf Basis des ZOIL-Frameworks [Jetter et al., 2008a]

Recht schnell wurde jedoch deutlich, dass die rein visuelle Ausrichtung des Frameworks nicht ausreichend war. Immer wieder tauchten Fragen und Probleme auf, die sich auf die Modellierung des Informationsraums und der Informationsobjekte bezogen. Dies führte zur Entwicklung des hier vorgestellten Datenmodells.

Die zweite Version des Frameworks beinhaltet neben dem UI-Framework und dem Datenmodell mehrere Bibliotheken. In diesen befinden sich bereits vorgefertigte Komponenten für Informationsobjekte und Visualisierungen, sowie auch *Input Handler* für verschiedenste Eingabegeräte.

Abbildung 8: Ein ZOIL-Prototyp auf Basis der zweiten Version des ZOIL-Frameworks

Abbildung 9 gibt eine grobe Übersicht über das Framework in der aktuellen Version. In den weiteren Unterkapiteln werden nun die einzelnen Teile des Frameworks näher beschrieben.

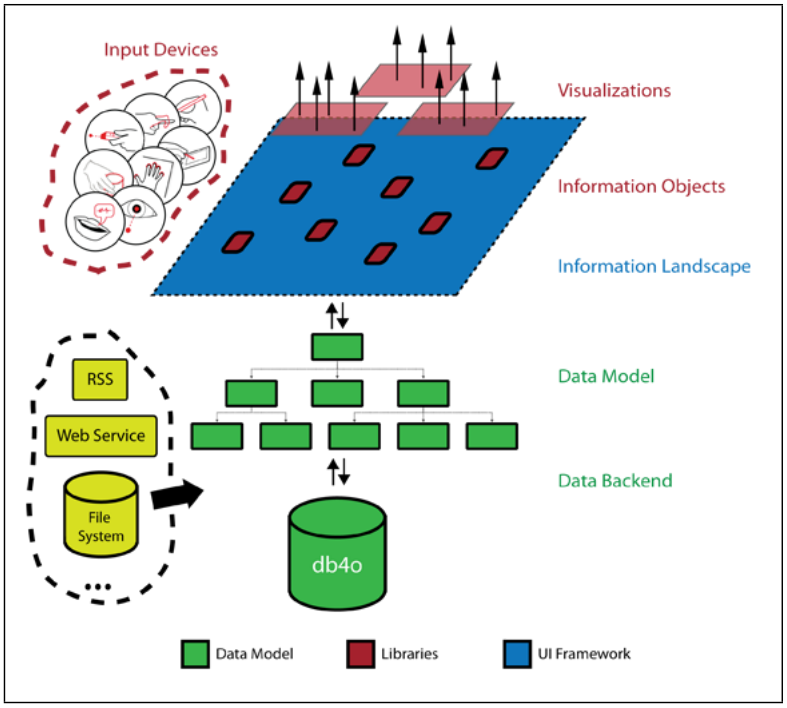

Abbildung 9: Übersicht über das ZOIL-Framework

1.4.1 UI Framework

Das ZOIL UI Framework ist der Kern des Frameworks. In ihm befindet sich die Informationslandschaft und alle Komponenten, die unmittelbar mit dieser im Zusammenhang stehen:

- **ZInformationLandscape**
  Die Informationslandschaft ist eine visuelle Komponente, die ihren Inhalt beliebig vergrößern und verkleinern (zooming) und in X- und Y-Richtung verschieben (panning) kann.
- **ZLandscapeAnimation**
  Jede Landschaft besitzt eine Animation. Diese ist für den animierten Übergang zwischen zwei Positionen auf der Landschaft zuständig.
- **ZLandscapePosition**
  Jeder Punkt auf der Landschaft ist über die so genannte *Landscape Position* auffindbar. Dieser besteht aus X- und Y-Koordinaten, sowie dem Vergrößerungsfaktor, den die Landschaft an diesem Punkt besitzt.

Des Weiteren sind hier alle Komponenten enthalten, die sich um den semantischen Zoom von Informationsobjekten kümmern:

- **ZComponent**
  Diese Komponente ist die visuelle Basisklasse für jedes Objekt, das semantisch gezoomt werden soll.
- **ZComponentFrame(s)**
  Die verschiedenen semantischen Zoomstufen eines Objekts werden über sogenannte *ZComponentFrames* definiert. Dabei repräsentiert ein Frame jeweils eine semantische Zoomstufe. Die Zoomstufen werden abhängig von der aktuell gerenderten Größe des Objekts, von der Informationslandschaft ausgetauscht.
- **ZModifiers**
  Mithilfe der *ZModifiers* kann jedem visuellen Element in der Landschaft ein individuelles Zoomverhalten übergeben werden. Beispielsweise ist es möglich, dass ein Element nicht an die aktuelle Vergrößerung der Landschaft angepasst wird, sondern seine Größe behält oder sogar invers zur Landschaft vergrößert wird.

1.4.2 Libraries

- **Library.Components**
  In dieser Bibliothek befinden sich vorgefertigte Komponenten für Informationsobjekte, z.B.

für Filme (ZMovie), Notizen (ZAnotoDrawingPad), oder Fotos (ZPhoto). Neben dem Model, der datennahen Repräsentation, befinden sich hier auch die Views, also die visuelle Repräsentation der Informationsobjekte.

- **Library.Visualizations**
  Visualisierungen sind visuelle Komponenten, die beliebige Informationsobjekte auf individuelle Art und Weise darstellen können. In der Bibliothek befinden sich zum jetzigen Zeitpunkt folgende Visualisierungen:
  - Bar Chart - Informationsobjekte werden als Balkendiagramm angezeigt.
  - Scatter Plot - Informationsobjekte werden in einem Punktdiagramm angezeigt.
  - Hyper Grid - Informationsobjekte werden in einer zoombaren Tabelle angezeigt. [Jetter et al., 2005]
- **Library.Devices**
  Die Interaktion mit der Informationslandschaft wird über verschiedenste Eingabegeräte ermöglicht. In der Bibliothek befinden sich zum jetzigen Zeitpunkt *Input Handler* für folgende Eingabegeräte
  - Maus
  - Tastatur
  - Microsoft Surface (Multitouch)
  - Nintendo Wiimote
  - OSC - Ein Netzwerkprotokoll, über das bestimmte Kommandos an die Landschaft gesendet werden können.
- **Library.Behaviors**
  Diese Bibliothek beinhaltet sogenannte *Attached Behaviors*. Das sind Verhaltensweisen, die der visuellen Repräsentation eines Informationsobjekts angeheftet werden können. Dadurch können visuelle Objekte Funktionalitäten, wie z.B. Drag & Drop, Resize & Rotate, auf einfache Art und Weise nutzen, ohne diese Funktionalitäten selber implementieren zu müssen.

1.4.3 Data Model & Data Backend

Die beiden Unterprojekte *Data Model* und *Data Backend* hängen eng miteinander zusammen. Das *Data Model* ist zuständig für die Modellierung des Informationsraums und der Informationsobjekte, und bietet hier sowohl Basisklassen für den logischen Teil (Model) eines Objekts, wie auch für den Visuellen (View + ViewModel). Das *Data Backend* ist hauptsächlich für die Persistierung

und Synchronisierung der zur Laufzeit erzeugten Informationsobjekte zuständig. Da diese beiden Unterprojekte Gegenstand der vorliegenden Arbeit sind, werden sie im Kapitel 3 auf Seite 34 detaillierter vorgestellt.

Die gerade vorgestellten Teile des Frameworks sind nicht endgültig. Das Framework in seiner jetzigen zweiten Version wird kontinuierlich weiterentwickelt, beispielsweise um neue Eingabegeräte aus dem *Media Room* und Interaktionstechniken, die sich daraus ergeben, in ZOIL zu integrieren. Es kann also durchaus vorkommen, dass bestimmte Teile irgendwann nicht mehr Bestandteil des Frameworks sind oder auch teilweise oder komplett überarbeitet werden.

Das Ziel des Frameworks muss es aber weiterhin sein, einen Baukasten für die Prototypentwicklung von ZOIL-Umgebungen darzustellen. Die einzelnen Unterprojekte und damit auch das *Data Model* und das *Data Backend* stellen (nur) einen Teil dieses Baukastens dar.

2 Anforderungen

Die ersten Anforderungen an ein Datenmodell für ZOIL sind, wie bereits erwähnt, aus der Erfahrung bei der Entwicklung von Prototypen entstanden. Diese Anforderungen waren daher sehr technisch und domänenspezifisch. ZOIL jedoch besitzt den Anspruch, ein domänenunabhängiges Paradigma bzw. Framework zu sein. Die Forschungsdisziplin *Personal Information Management* ist streng betrachtet zwar eine Domäne, jedoch sind die darin formulierten Konzepte so generisch gehalten, dass sie viele (wenn nicht die meisten) Szenarien von zukünftigen ZOIL-Anwendungen abdecken. PIM wurde daher als fundierte Ausgangsbasis für die Formulierung von Anforderungen an ein Datenmodell betrachtet.

Eine Quelle für weitere Anforderungen war verständlicherweise die theoretische Basis von ZOIL selbst, die *ZOIL-Designprinzipien*.

Zu guter Letzt entstanden durch die Möglichkeit, den *Media Room* der Universität Konstanz für ZOIL zu nutzen, weitere Anforderungen an das Backend.

Die Herleitung der Anforderungen aus ihren jeweiligen Quellen wird Gegenstand der folgenden Unterkapitel sein. Am Ende des Kapitels werden die Anforderungen dann in kompakter Form zusammengefasst, damit sie in den weiteren Teilen der Arbeit leichter referenziert werden können.

2.1 Personal Information Management

Die Anforderungen die aus dem *Personal Information Management* abgeleitet wurden, orientieren sich in erster Linie an den fundamentalen Einheiten und Aktivitäten, die in dieser Disziplin diskutiert werden. In dieser Arbeit basieren die Einheiten und Aktivitäten auf den Definitionen von Jones [2007a,b]. Eine detailliertere Diskussion der verschiedenen Definitionen wird in der Seminararbeit des Autors zu diesem Thema geliefert [Zöllner, 2008].

2.1.1 Einheiten

Information Item *information–as–thing*, im Gegensatz zu *information–as–knowledge* oder *information–as–process* bildet die Grundlage für die Einheit des *Information Items*, denn *"it is with information in this sense that information systems deal directly"* [Buckland, 1991]. Das Konzept *information–as–thing* nimmt in Bucklands Analyse auch ungewöhnliche Formen an, wie z.B. der Baum, der durch die Anzahl seiner Jahresringe bestimmte Informationen übermittelt. Für das *Personal Information Management* muss dieses Konzept deshalb pragmatisch eingeschränkt werden. Jones [2007a] bezeichnet ein *Information Item* als "packaging of information" und merkt

an: *"items encapsulate information in a persistent form [...]"*. Beispiele für *Information Items* sind demnach physische oder elektronische Dokumente, eMails, Dateien, Webseiten und weitere informationstragende Dinge aus der physischen oder digitalen Welt.

Für ZOIL stellt das *Information Item* in seiner sehr hohen Abstraktion und Heterogenität die geeignete Ausgangsbasis dar, da ZOIL den Anspruch erhebt, ein Paradigma für unterschiedlichste Anwendungsdomänen zu sein. Diese vorgegebene Abstraktion und damit einhergehende Flexibilität muss das Datenmodell mittragen. Da das *Information Item* von Jones als "packaging of information" bezeichnet wird, ist eine konkrete Ausprägung des *Information Items* also kein inhaltsleeres abstraktes Etwas, sondern ein, abhängig vom Kontext, mit entsprechenden Informationen gefülltes Objekt. Die von ZOIL vorgegebene Domänenunabhängigkeit verhindert die Vorbelegung des *Information Items* mit irgendeiner Struktur, da Menge und Art der Informationen, die in einem *Information Item* gebündelt werden sollen, stark domänenabhängig sind. Eine vorgefertigte Modellierung jeglicher *Information Items*, die irgendwann einmal in ZOIL auftauchen könnten, ist dadurch nicht möglich und auch nicht sinnvoll. Der Designer eines ZOIL-Systems sollte *Information Items*, die in seiner Domäne auftauchen, nach Belieben modellieren können.

Am Beispiel *Dokument* lässt sich dies nachvollziehen. Ein Dokument besteht hauptsächlich aus (formatiertem) Text. Weitere (Meta-)Informationen, wie beispielsweise Autoren, Titel, Entstehungsdatum, usw. beschreiben ein Dokument jedoch näher und sorgen dafür, dass das Dokument zu einem nützlichen Paket aus Informationen wird.

Die erste Anforderung an das Datenmodell ist daher die Einführung der Einheit *Information Item* in seiner abstrakten Form. Konkrete *Information Items* sollten vom Designer eines ZOIL-Systems nach Belieben modelliert werden können.

Diese Anforderung ist aber noch nicht weitreichend genug. Wie Karger [2007] zurecht bemängelt, sind Modellierungsentscheidungen in digitalen Systemen meistens in Stein gemeißelt und können vom Anwender des Systems nicht mehr geändert werden. Er schlägt daher unter anderem vor, dass der Benutzer selbst entscheiden sollte, *welche Beziehungen und Eigenschaften eines Information Items es wert sind, gespeichert zu werden um diese Informationen später wieder zu finden* [Karger, 2007]. Diese Erkenntnis ist nicht ganz neu. Bereits 1945 beschreibt Vannevar Bush in seinem visionären Artikel "As we may think" eine mechanische Maschine zur Informations- und Wissensverarbeitung und bemerkt dabei: *"A record, if it is to be useful to science, must be continuously extended"* [Bush, 1945].

Betrachtet man ein aktuelles Datei- bzw. Betriebssystem, kann man schnell feststellen, dass sich diese wichtige Funktionalität noch nicht manifestiert hat: Notizen, die beispielsweise beim Lesen eines elektronischen Dokuments gemacht werden, können nur mit speziellen Applikationen an dieses Dokument geheftet werden, es gibt keine Mechanismen, die das Erweitern von *Information Items* mit Metadaten auf Dateiebene zulassen.

Die oben stehende Anforderung wird somit ergänzt. *Information Items*, die während der Designphase modelliert wurden, sollen auch während der Laufzeit des Systems mit beliebigen Informationen erweitert werden können. Unter anderem muss es auch möglich sein, beliebige Beziehungen zwischen *Information Items* zur Laufzeit herstellen zu können.

Neben der Möglichkeit, Informationen in einem *Information Item* bündeln zu können, spielt die Tatsache, dass die Informationen in persistenter Form vorliegen, bei Buckland [1991] eine zentrale Rolle. Er unterscheidet explizit zwischen "tangibles" (information-as-thing) und "intangibles" (information-as-knowledge):

> *"If you can touch it or measure it directly, it is not knowledge, but must be some physical thing, possibly information-as-thing."* [Buckland, 1991].

Diese Unterscheidung ist auch für das *Personal Information Management* relevant, da Informationen nur in persistenter Form sinnvoll verarbeitet werden können. *Personal Information Management* beginnt erst dort, wo Informationen in persistenter Form vorliegen. Für ZOIL ergibt sich daraus als zusätzliche Anforderung, dass die modellierten und mit Informationen gefüllten *Information Items* in persistenter bzw. persistierbarer Form vorliegen müssen.

Personal Space of Information *Personal Information* besitzt nach Jones [2007a] mehrere Ausprägungen, die sich dadurch unterscheiden, dass sie unterschiedlich zu einer Person gerichtet sind. Abbildung 10 auf der nächsten Seite zeigt eine Übersicht dieser Ausprägungen. Die Vereinigung aller Ausprägungen von persönlicher Information ergibt dann den *Personal Space of Information*:

> *"Personal information, in each of its senses, combines to form a single personal space of Information (PSI) for each individual"* [Jones, 2007a].

Wie die Abbildung 10 auf der nächsten Seite zeigt, überlappen sich die verschiedenen Ausprägungen gegenseitig. Im Zentrum des PSI befinden sich demnach die Informationen, die im Besitz und unter der Kontrolle der Person stehen. Die Bereiche um diesen zentralen Bereich herum sind nicht unmittelbar unter der Kontrolle der Person und können nicht immer beeinflusst werden. Für ZOIL ist daher nur der zentrale Teil des PSI interessant, da die anderen Bereiche nicht vollständig zugänglich sind. Die Hauptfunktion dieses Bereichs ist die Persistenz von Informationen. Eine weitere Anforderung an das Datenmodell wäre daher die Möglichkeit, beliebige Informationen zu persistieren und diese Informationen dem Benutzer zugänglich zu machen.

Die Schnittstelle zwischen dem PSI und dessen Besitzer ist normalerweise sehr diffus. Informationen, die unter der Kontrolle einer Person stehen, können überall im virtuellen Raum verstreut sein. Die Person kann an vielen verschiedenen Orten Informationen persistieren und nach ihnen suchen. Diese Fragmentierung der Informationen führt oftmals zu ineffektivem und ineffizientem *Personal*

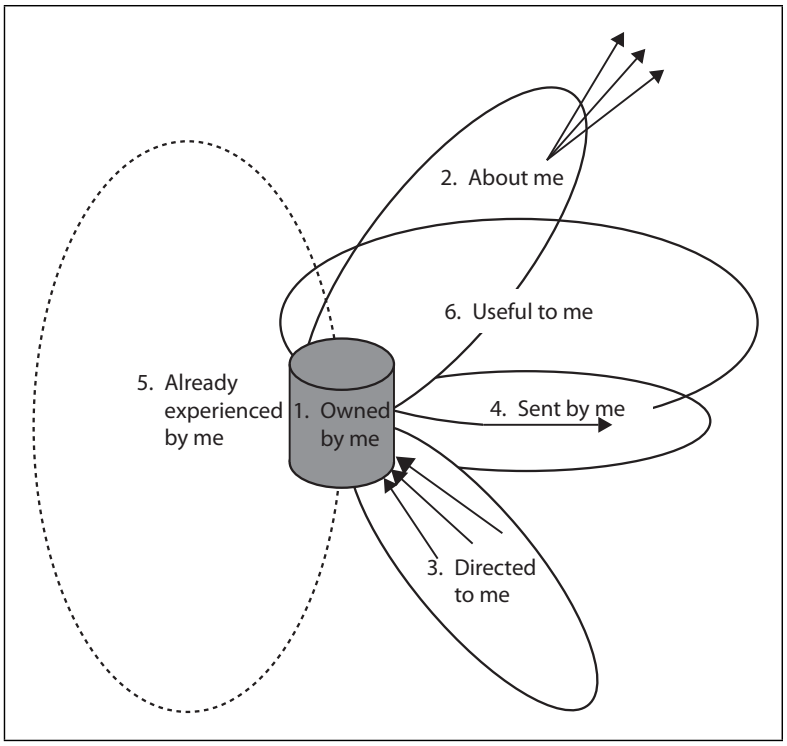

Abbildung 10: Verschiedene Ausprägungen persönlicher Information formen den PSI [Jones, 2007a]

Information Management [Karger and Jones, 2006]. Für die Person wäre es wesentlich einfacher, wenn sie sich nur an eine Instanz wenden müsste, um Informationen zu speichern oder diese zu finden. In ZOIL sollte daher jegliche Kommunikation mit dem PSI über eine einzige Schnittstelle geführt werden.

Personal Information Collection *Information Items* werden meist nicht in isolierter Form von ihrem Besitzer verwendet. Die Aktivitäten, die eine Person mit ihren Informationen durchführt, erfordern oft mehrere *Information Items*, die im aktuellen Kontext zusammen gehören. Im *Personal Information Management* bezeichnet man diese Untermengen des PSI als *Personal Information Collections* (PIC). Wie Jones [2007b] ausdrücklich betont, sind *Personal Information Collections* nicht davon abhängig, ob ihre Kinder demselben Typ angehören, oder mit einer bestimmten Anwendung bearbeitet werden können.

> *"[...] this is not a necessary feature of a PIC. People might like to place several forms of information in a PIC, even if doing so is often difficult or impossible with current software applications."* [Jones, 2007b]

Personal Information Collections werden von Jones [2007b] als überschaubare “Inseln“ des PSI betrachtet. Sie sind sehr gut dafür geeignet, eine Untermenge des PSI zu verwalten:

> *“The organization of ’everything’ in a PSI is a daunting, perhaps impossible, task. But people can imagine organizing a collection of web bookmarks, their email inbox, their laptop filing system [...], and so on.“*[Jones, 2007b]

Personal Information Collections sind also integraler Bestandteil des *Personal Information Managements*. Hieraus folgt als weitere Anforderung an das Datenmodell, dass beliebige Informationen zu *Personal Information Collections* zusammengefasst werden können. *Personal Information Collections* werden dabei nicht als Speicherort für *Information Items* angesehen, sondern als Organisationsmöglichkeit. Ein bestimmtes *Information Item* soll deshalb in mehreren *Personal Information Collections* abgelegt werden können.

2.1.2 Aktivitäten

Im Abschnitt Personal Space of Information wurde der PSI auf den Teil eingeschränkt, der sich um die Persistenz von *Information Items* kümmert und damit um alle *Information Items*, die unter der Kontrolle einer Person stehen. Aus dieser Perspektive betrachtet ist der PSI nichts anderes als ein Speicher. In solch einen Speicher können in der Regel Dinge hineingelegt und herausgenommen werden und der innere Aufbau des Speichers kann verändert werden [Jones, 2007a]. Definiert man nun *Personal Information Management* als die Interaktion mit diesem PSI, so ergeben sich aus den Operationen, die auf dem Speicher ausgeführt werden, die dazu äquivalenten grundlegenden Aktivitäten des PIM (siehe Abbildung 11) [Jones, 2007b]:

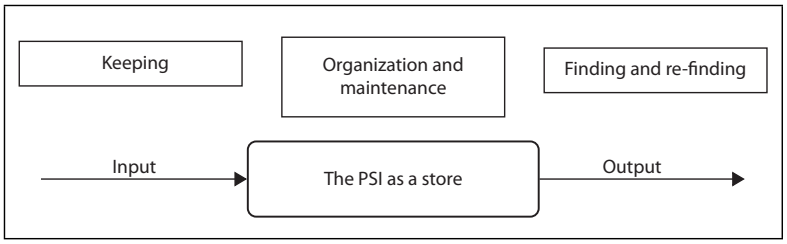

Abbildung 11: Beziehung zwischen den Operationen auf dem PSI und den PIM-Aktivitäten [Jones, 2007a]

Für Jones [2007a] sind PIM-Aktivitäten letztendlich die Mittel, die benutzt werden, um Informationen auf die Bedürfnisse der Person abzubilden:

> *“PIM activities are an effort to establish, use and maintain a mapping between information and need.“* [Jones, 2007a]

Diese Aussage rückt die Person und ihre Bedürfnisse in den Mittelpunkt der Betrachtung und löst sich von einer technischen Sichtweise des *Personal Information Managements*.

Das plötzliche Erscheinen eines *Information Items*, beispielsweise einer Visitenkarte, löst bestimmte Aktivitäten (*keeping activities*) aus, wie das Einsortieren der Visitenkarte in ein bestehendes Organisationsschema, oder das Speichern der Kontaktdaten im Mobiltelefon. Hier wird also ein Mapping für ein zukünftiges Bedürfnis hergestellt. Besteht dieses Mapping, so löst ein plötzliches Bedürfnis der Person, beispielsweise ein Telefonat mit dem Besitzer der Visitenkarte zu führen, bestimmte Aktivitäten (*finding/re-finding activities*) aus, wie die Suche nach der Telefonnummer oder den Kontaktdaten. Hier wird das zuvor erstellte Mapping benutzt um das Bedürfnis auf die Informationen abzubilden. Damit der Person dieses Mapping so lange wie möglich erhalten bleibt, muss die Person bestimmte Aktivitäten (*meta-level activities*) durchführen. Beispielsweise könnte dies bedeuten, dass die Visitenkarten in einem alphabetischen Index gespeichert werden, oder dass alte Einträge im Telefonbuch des Mobiltelefons gelöscht werden, damit aktuell benötigte Kontaktdaten schneller auffindbar sind.

Keeping Activities Jedes *Information Item*, das in den zentralen Bereich des PSI einer Person vordringt und von ihr bewusst wahrgenommen wird, erfordert eine grundlegende Entscheidung: Soll es behalten werden oder nicht ? Diese Entscheidung ist keineswegs trivial, da oft nicht klar ist, ob das *Information Item* irgendwann wieder benötigt wird. Um dies festzustellen, muss die Person ein zukünftiges Bedürfnis antizipieren [Jones, 2007a]. Fällt schlussendlich die Entscheidung, dass das *Information Item* behalten werden soll, löst das eine weitere Entscheidung aus: Es muss nun entschieden werden, wie dieses *Information Item* gespeichert wird, damit es beim Auftreten eines zukünftigen Bedürfnisses wieder gefunden werden kann:

> *"Keeping activities must address the multifaceted nature of an anticipated need."* [Jones, 2007a].

Aus psychologischer Sicht ist es nicht einfach, Informationen so zu speichern und zu kategorisieren, dass diese im Falle eines Bedürfnisses schnell wiedergefunden werden können. Speichern und Wiederfinden von Informationen sind meist zeitlich getrennte Aktionen und finden somit in unterschiedlichen Kontexten statt. Die Kontexte sind jeweils davon abhängig, was eine Person zum jeweiligen Zeitpunkt denkt, in welcher Verfassung sie sich befindet, usw. Der Kontext beim Speichern hat direkten Einfluss auf die Enkodierung der Informationen, also z.B. auf die Wahl des Namens oder eines Verzeichnisses. Der Kontext beim Wiederfinden des Objektes gibt dem Gedächtnis gewisse Vorgaben, nach denen es sich bei der Suche richten wird. Nur wenn der Kontext beim Wiederfinden, dem Kontext beim Enkodieren entspricht, ist ein erfolgreiches Wiederfinden möglich [Lansdale, 1988].

Diese Einschränkung der Leistungsfähigkeit des Gedächtnisses muss ein System, das persönliche Informationen verwaltet, berücksichtigen. Beim Enkodieren und Klassifizieren von Informationen ist dies vor allem durch das Erzeugen von vielen Meta-Informationen möglich. Wird ein *Information Item* in einem aktuellen Dateisystem in ein Verzeichnis gelegt, sind die Möglichkeiten, dieses Objekt später wieder zu finden sehr limitiert. Der Informationssuchende muss sich den Ort in der Hierarchie oder einen Dateinamen merken, während das, was er sich sowieso merkt, wie z.B. äußere Merkmale, Kontextinformationen und inhaltliche Dinge nicht bei der Suche berücksichtigt werden [Lansdale, 1988].

> *"What we are good at is being ignored, and what we are required to do is a difficult and flawed psychological process"*[Lansdale, 1988]

Ein Informationssystem, das diese Meta-Informationen, Informationen, die das Objekt näher beschreiben, mitspeichert, wird von einem Informationssuchenden effizienter genutzt werden können.

Für das Datenbackend in ZOIL ergibt sich aus dem Vorhergehenden folgende Anforderung: Meta-Informationen beliebigen Typs müssen für jedes Objekt zur Laufzeit gespeichert werden können. Des Weiteren muss es möglich sein, nach diesen Meta-Informationen zu suchen und die damit assoziierten Informationsobjekte zu finden.

Das Datenbackend muss dagegen nicht in der Lage sein, die Meta-Informationen aus den Objekten zu extrahieren, bzw. Meta-Informationen über die Objekte zu generieren. Dies sind domänenabhängige Funktionalitäten, die der jeweilige Designer des Systems implementieren muss. Das Datenbackend soll lediglich in der Lage dazu sein, Meta-Informationen zu speichern.

Finding / Re-Finding Activities Das *Finden* und *Wiederfinden* von Informationen wird, wie bereits erwähnt, meist durch das Aufkommen eines Bedürfnisses ausgelöst. Die beiden Aktivitäten unterscheiden sich dabei nur in einem Punkt: Beim *Wiederfinden* wurde die gesuchte Information von der Person bereits gesehen [Jones, 2007b]. Die Person weiß, dass die Information existiert, muss sie also nur noch lokalisieren. *Wiederfinden* ist daher meistens auf konkrete *Information Items* bezogen. Bei der Tätigkeit des *Findens* ist dies natürlich auch möglich (z.B. bei der Suche nach einer Telefonnummer in einem Telefonbuch), allerdings kann *Finden* auch bedeuten, dass keine konkrete Vorstellung vom Ergebnis vorhanden ist (z.B. bei der Suche nach einem Geburtstagsgeschenk). Die Suche nach solch schwammigen Informationen wird meist mittels Browsing erleichtert. Der Informationssuchende kann sich von einem reichhaltigen Informationsangebot inspirieren lassen und mithilfe von Verknüpfungen den Informationsraum explorativ erkunden.

Beschränkt man die Tätigkeit des Suchens von Informationen auf den zentralen Teil des PSI, so bleibt allerdings als einzige Aktivität das Wiederfinden von Informationen, da in der Regel alle *Information Items*, die im PSI persistiert sind, von der Person bereits gesehen wurden. Die Person

hat demnach eine, wenn vielleicht auch schwache, Vorstellung, welche *Information Items* sich in ihrer Kontrolle befinden und benutzt zum *Wiederfinden* Methoden, die sie schneller ans Ziel führen. Nach Lansdale [1988] sollten die Personen dabei idealerweise nach folgendem Schema vorgehen:

> *"recall-directed search, followed by recognition-based scanning"* [Lansdale, 1988]

Recall-directed search erfordert von der Person, dass sie sich zumindest an ein Merkmal des zu suchenden *Information Items* erinnern kann. Mit diesem Merkmal (oder mehreren) kann die Person dann eine Suchanfrage an das System stellen. Um *recall-directed search* zu ermöglichen, bedeutet dies für das Datenbackend, dass geeignete Suchanfragen an das System gestellt werden können. Die Antwort des Systems ist eine Liste von Objekten, die der Suchanfrage entsprechen. Beim *recall-directed search* ist es oft der Fall, dass sehr viele Objekte der Suchanfrage entsprechen. Um das Ergebnis einzugrenzen, kann die Person nun eine neu formulierte, verfeinerte Anfrage an das System senden, oder die Ergebnisse der ersten Anfrage durchscannen. Beim *recognition-based scanning* nutzt die Person ihre Fähigkeiten aus, sich an Merkmale von Objekten zu erinnern. Diese höchst visuelle Aufgabe kann mit geeigneten Visualisierungen unterstützt werden. Organisationsstrukturen, wie z.B. Hierarchien, Tags, Filtermechanismen, usw., die es ermöglichen, die Ergebnismenge Schritt für Schritt einzuschränken, können hier sehr hilfreich sein. Die Person muss dann nicht mehr die komplette Ergebnisliste durchscannen, sondern kann sich auf kleinere Teilbereiche konzentrieren und kommt somit schneller zum Ziel. Um *recognition-based scanning* zu unterstützen sollte das Datenbackend daher alle Voraussetzungen schaffen, dass die oben erwähnten Organisationsstrukturen von einem ZOIL-System benutzt werden können. Dazu zählen Funktionalitäten, wie das logische Verbinden von Objekten, multiple Klassifizierung, usw. Eine detailliertere Übersicht, über die zu unterstützenden Funktionen liefert der nächste Abschnitt, der sich mit solchen Organisationsstrukturen im Rahmen der Meta-Level Activities beschäftigt.

Meta-Level Activities Jones [2007a] zählt folgende Aktivitäten zu den Meta-Level Activities:

- Organizing
- Maintaining
- Managing privacy and the flow of information
- Measuring and evaluating
- Making sense

Von diesen fünf Aktivitäten ist die erste, *Organizing*, die relevanteste, da sie sich direkt mit der inneren Zusammensetzung des PSI beschäftigt und direkten Einfluss auf die Effektivität der *Keeping*

Activities und *Finding/re-finding Activities* hat [Jones, 2007a]. Es wird daher an dieser Stelle nur auf diese Aktivität eingegangen.

Jones [2007a] bezeichnet die *Organizing Activities* folgendermaßen:

> *"Decisions made and actions taken in the selection and implementation of a scheme to relate the information items in a collection to anticipated needs"*.

Die Beobachtungen, die Jones [2007a] hinsichtlich der *Organizing Activities* macht, sind stark abhängig von der Verwendung eines aktuellen Betriebssystems mit hierarchischem Dateisystem. In der Einleitung im Kapitel 1.1.1 auf Seite 4 wurde bereits detailliert das Problem der Modellierung des Informationsraums diskutiert. Unter anderem wurde festgestellt, dass das hierarchische Dateisystem nicht die optimale Organisationsform für einen persönlichen Informationsraum darstellt. Anstatt eine bestimmte Organisationsform vorzugeben, sollte ein PIM-Tool daher besser verschiedene Organisationsformen anbieten, die der Nutzer nach eigenem Belieben auswählen kann. Für ZOIL ist dies besonders wichtig, da die geforderte Domänenunabhängigkeit auch eine gewisse Flexibilität hinsichtlich der Organisation des Informationsraums bedingt. Es werden nun einige Organisationsformen aufgelistet, die auf alle Fälle in ZOIL zur Verfügung stehen müssen, die das Datenmodell also anbieten muss:

- **Semantisches Netzwerk** Objekte müssen beliebig miteinander in Beziehung gesetzt werden können. Die Art der Beziehung soll festgelegt werden können um bestimmte Bedeutungen auszudrücken (z.B. Ist-Teil-Von, Ist-Kind-Von, Ist-Vater-Von).
- **Multiple Klassifizierung** Durch die Vergabe von Schlagworten sollen Objekte mehrfach klassifiziert werden können.
- **Hierarchien** Hierarchien sind eine Teilmenge von semantischen Netzwerken. Sie können mittels spezieller Vater-Kind Beziehungen zwischen Objekten hergestellt werden.
- **Lineare Anordnung** Objekte müssen anhand eines bestimmten Attributs in lineare Anordnung gebracht werden können. Dies könnte beispielsweise eine chronologische Anordnung sein.
- **Räumliche Anordnung** Diese Anforderung ist bereits durch die Informationslandschaft von ZOIL erfüllt. Hier können Objekte beliebig räumlich angeordnet werden.

Wie bereits in der Einleitung erwähnt, sollen diese Organisationsstrukturen vor allem verschiedene Sichten auf den Informationsraum ermöglichen und nicht den Speicherort eines Objekts beeinflussen.

2.2 ZOIL

Das ZOIL-Paradigma basiert auf fünf *Designprinzipien*, die einen Rahmen für Anwendungen auf ZOIL-Basis vorgeben. Nachfolgend werden aus diesen Prinzipien Anforderungen für das Datenmodell und -backend abgeleitet. Nicht alle Prinzipien haben dabei direkten Einfluss auf das Datenmodell bzw. -backend, da ZOIL in erster Linie ein stark visuelles Paradigma darstellt.

2.2.1 Object-Oriented User Interface

> *"Eine ZOIL-Benutzungsschnittstelle ist eine objekt-orientierte Benutzungsschnittstelle. Ihr liegt eine objekt-orientierte Analyse des Informationsraums zugrunde, die die globale Informationsarchitektur, sowie die Sichtbarkeit und das Verhalten von Wissensobjekten auf der Benutzungsoberfläche definiert."*[Jetter, 2007]

Jetter [2007] diskutiert das Konzept des OOUI detailliert und legt dar, wieso eine solche Benutzerschnittstelle, gerade für die Wissensarbeit, Vorteile gegenüber der *Desktop Metapher* besitzt. Im Gegensatz zur *Desktop Metapher*, bei der Anwendungen dafür zuständig sind, Funktionalitäten auf Daten durchzuführen, sind die Funktionalitäten in einer objektorientierten Benutzerschnittstelle integraler Bestandteil der Objekte. Anstatt einer *Verb-Objekt* Syntax wird also eine *Objekt-Verb* Syntax bevorzugt, da diese laut [Raskin, 2000] das Einführen von Modalitäten in die Interaktion verhindert. Durch die objektorientierte Modellierung, die einem OOUI zugrunde liegt, können gewisse Verhaltensweisen und Eigenschaften, die mehrere Objekte verschiedenen Typs gemeinsam besitzen, entweder in einer gemeinsamen Basisklasse definiert werden, oder mittels Delegation oder Polymorphie einer Klasse von Objekten mitgegeben werden. Ein OOUI ist damit imstande, die selbe Aktion auf heterogenen Informationsobjekten durchzuführen, wenn diese alle die gleiche Verhaltensweise definieren.

Am Beispiel eines Computer-Strategiespiels soll dies verdeutlicht werden: Ein Strategiespiel besitzt meist verschiedene Einheiten, wie z.B. Arbeiter, Soldaten, Zivilisten, Schiffe, Tiere, usw. Diesen Einheiten kann der Spieler mittels der Objekt-Verb Syntax interaktiv Anweisungen geben. Meistens müssen dazu zuerst eine oder mehrere Instanzen dieser Einheiten selektiert werden, danach kann eine Anweisung in der Benutzerschnittstelle ausgewählt werden. Selektiert der Spieler mehrere Instanzen verschiedener Einheiten, so wird ihm nur die Schnittmenge der Anweisungen angeboten, die den selektierten Instanzen gegeben werden können. Analog dazu kann in einem OOUI bei der Selektion heterogener Objekte nur die Schnittmenge der Operationen der selektierten Objekte durchgeführt werden.

Dadurch, dass bei OOUIs größtenteils komplett auf Applikationen verzichtet wird, muss der Benutzer keinen Kontextwechsel vollziehen, wenn er verschiedene Tätigkeiten durchführen will. Der Ausgangspunkt einer Aktion ist immer ein Objekt (oder mehrere). Dadurch bleibt die Interaktion

mit dem System für alle Tätigkeiten konsistent.

Durch die Verwendung eines OOUI ist die Ausrichtung des Datenmodells natürlich schon vorgegeben. Informationsobjekte müssen in ZOIL objektorientiert modelliert werden können. Dabei sollen die gerade genannten Vorteile, die sich durch die objektorientierten Prinzipien *Vererbung*, *Delegation* und *Polymorphie* ergeben, natürlich ausführlich genutzt werden. Da gewisse Funktionalitäten, vor allem auch solche, die die Organisation des Informationsraums betreffen, von allen Informationsobjekten verlangt werden, bietet sich die Verwendung einer gemeinsamen Oberklasse für alle Informationsobjekte an. Diese Oberklasse sollte gewisse Verhaltensweisen und Eigenschaften definieren, die für alle Informationsobjekte gelten. Spezielle Informationsobjekte einer Domäne, können dann vom jeweiligen Domänenexperten erstellt werden, indem diese Oberklasse erweitert wird.

2.2.2 Semantic Zooming

Alle Objekte, die in einer Informationslandschaft von ZOIL liegen, können vom Benutzer herangezoomt werden. In ZOIL wird dafür der *Semantic Zoom* verwendet. Dies bedeutet, dass abhängig vom zur Verfügung stehenden Platz mehr oder weniger Informationen eines Objektes auf dem Bildschirm angezeigt werden. Der Zugewinn an Platz beim Zoomen wird nicht unbedingt für eine größere Ansicht genutzt, sondern für die Anzeige von mehr Informationen. Für das Datenbackend hat dies keine Auswirkungen, da die visuelle Repräsentation eines Objekts von der logischen getrennt sein sollte. Aus der Perspektive der Datenmodellierung sollte es prinzipiell egal sein, wie das Objekt visuell dargestellt wird. Dies ist auch deshalb erforderlich, damit mehrere verschiedene Repräsentationen eines Objekts erstellt und in der Landschaft verortet werden können. Die einzige Anforderung, die sich daher für das Datenmodell ergibt, ist die Trennung von *View* und *Model*, also von der logischen und der visuellen Repräsentation von Objekten.

2.2.3 Nested Information Visualization

Mithilfe von *Nested Information Visualizations* soll es in ZOIL möglich sein, eine differenzierte Sicht auf die Objekte in der Informationslandschaft zu bekommen. Visualisierungen (z.B. Balkendiagramme, Landkarten, usw.) ordnen die gewünschten Objekte dabei nach speziellen Kriterien visuell an und ermöglichen so eine eher analytischere Sichtweise auf den Datenraum. Die Visualisierungen befinden sich dabei in einer Art Container, die auch als *Portal* bezeichnet wird. Mithilfe von Filtermechanismen oder per Drag & Drop sollen sich diese *Portale* mit Informationsobjekten füllen lassen.

Als äquivalent zu den Portalen betrachten Jetter et al. [2008b] die *Personal Information Collections* aus dem *Personal Information Management*. Die Datenbasis eines Portals entspricht einer *Personal*

Information Collection, einer überschaubaren "Insel" des PSI, die mit Hilfe der Visualisierungen untersucht werden soll. Da *Personal Information Collections* bereits aus dem PIM als Anforderung hervorgegangen sind, erfolgt aus diesem Prinzip keine weitere Anforderung an das Datenbackend.

2.2.4 The Information Space as Information Landscape

Der vom Entwickler bzw. Benutzer modellierte Informationsraum wird in ZOIL auf der Informationslandschaft visualisiert. Informationsobjekte liegen dabei direkt in der Informationslandschaft und können mithilfe von *zooming* und *panning* exploriert werden. Mittels der eben erwähnten Portale bzw. Visualisierungen erhält der Nutzer einen alternativen Einstieg in den Informationsraum.

Für das Datenmodell ist dieses Prinzip weniger relevant, da hier auch die visuelle Seite des Frameworks im Vordergrund steht und weniger die logische. Eine wichtige Anforderung ergibt sich jedoch aus diesem Prinzip, nämlich die visuelle Persistenz von Objekten. Der Benutzer eines ZOIL-Systems hat die Möglichkeit die Informationsobjekte auf der Landschaft beliebig anzuordnen und nützt dabei effektiv die visuellen Fähigkeiten seines Gehirns aus um sich die relativen Positionen der Objekte zu merken. Eine einmal getätigte Anordnung auf der Informationslandschaft sollte beim nächsten Start der Anwendung nicht verloren gehen. Es ist daher wichtig, dass die visuellen Eigenschaften der Objekte, wie die Position, Größe, usw., vom Datenbackend persistiert werden, damit die Landschaft beim nächsten Start wieder hergestellt werden kann.

2.2.5 Nomadic Cross-platform User Interfaces

Im Letzten der fünf Designprinzipien wird ZOIL als universelles Paradigma für Ausgabegeräte unterschiedlichster Größe gesehen. ZOIL wäre somit in der Lage mit dem Nutzer mitzuwandern (daher der Ausdruck "nomadic"), da es sowohl auf dem Heimrechner, als auch auf dem Smartphone die gewohnte Informationslandschaft präsentiert. Damit dies möglich ist, muss der persönliche Informationsraum auf allen Geräten über das Internet zur Verfügung stehen. Das Datenbackend sollte daher eine Serverkomponente besitzen, die allen Geräten Zugriff auf den persönlichen Informationsraum ermöglicht. Diese Serverkomponente muss dafür Sorge tragen, dass mehrere Geräte zur gleichen Zeit auf den Informationsraum zugreifen können und sich dabei nicht gegenseitig stören. Die Integrität der einzelnen Objekte und des Informationsraums muss zu jeglichem Zeitpunkt gegeben sein.

2.3 Media Room

Die Ausstattung des *Media Rooms* und das damit verbundene Potential wurden bereits in der Einleitung vorgestellt. Durch die Vielzahl vorhandener Ein- und Ausgabegeräte eignet sich der *Media*

Room besonders für kollaborative Aufgaben. Eine ZOIL-Anwendung wird in solchen Szenarien daher nicht nur auf einem Rechner ausgeführt, sondern auf mehreren Rechnern gleichzeitig. Idealerweise greifen dabei alle Clients auf den selben Informationsraum zu (vgl. Abbildung 12).

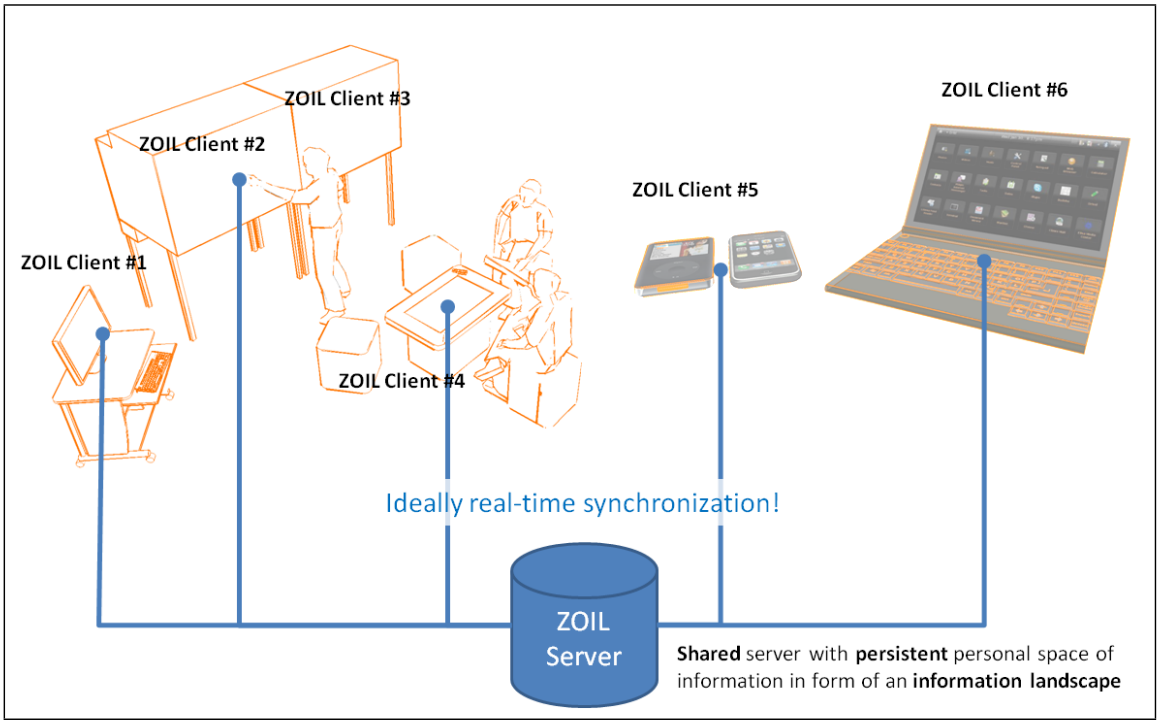

Abbildung 12: Gleichzeitiger Zugriff verschiedener ZOIL-Clients auf den Informationsraum (Skizze erstellt von Hans-Christian Jetter)

Für das Datenbackend bedeutet dies zum einen, dass alle Clients Zugriff auf den Informationsraum benötigen, dieser sollte daher idealerweise über das Internet zugänglich sein. In kollaborativen Szenarien ist es außerdem sehr wichtig, dass alle Beteiligten über Änderungen an den Informationsobjekten informiert werden, die von anderen Personen getätigt werden. Idealerweise bewirkt eine Änderung auf einem Client, die selbe Änderung auf allen anderen Clients, die auf den selben Informationsraum zugreifen. Für das Datenbackend bedeutet dies, dass Änderungen an Objekten, die auf einem Client getätigt werden, an alle anderen Clients übertragen werden müssen. Diese Anforderung gilt dabei nicht nur für die logische Repräsentation des Objekts, sondern speziell auch für die Visuelle. Beispielsweise sollten beim Verschieben eines Objekts auf der Landschaft die neuen Koordinaten direkt an alle anderen Clients gesendet werden, damit diese ihre visuelle Repräsentation anpassen können und die Benutzer sehen, dass an diesem Objekt gearbeitet wird. Die Benutzer bekommen somit ein sofortiges visuelles Feedback von den Tätigkeiten der anderen Nutzer.

2.4 Zusammenfassung

2.4.1 Funktionale Anforderungen

In den vorangehenden Abschnitten wurden Anforderungen an das Datenmodell und -backend gesammelt. Damit diese im Verlauf der Arbeit leichter referenziert werden können, werden sie hier noch mal zusammengefasst. Die einzelnen Anforderungen werden dabei aus Gründen der Übersichtlichkeit in vier übergeordnete Klassen eingeteilt.

Informationsobjekte

R11 Einführung der Einheit *Information Item* als abstrakte Basisklasse für konkrete Informationsobjekte.

R12 Basisfunktionalitäten und -daten sollten mittels objektorientierter Mechanismen (Vererbung, Delegation, Polymorphie) modelliert werden

R13 *Information Items* sind zur Design- und Laufzeit mit beliebigen Metadaten erweiterbar.

R14 Zwischen *Information Items* können beliebige Beziehungen hergestellt werden.

R15 Trennung von logischer und visueller Repräsentation eines *Information Items*.

R16 *Information Items* können in *Personal Information Collections* organisiert werden.

Informationsraum

R21 Folgende Organisationsstrukturen müssen zur Modellierung des Informationsraums realisiert werden können:

- Semantisches Netzwerk
- Multiple Klassifizierung
- Hierarchie
- Lineare Anordnung
- Räumliche Anordnung

R22 Der Informationsraum muss global verfügbar sein (Client-Server Architektur).

R23 Mehrere Clients müssen gleichzeitig auf den Informationsraum zugreifen können.

R24 Der Informationsraum muss vom Benutzer mit beliebigen Anfragen durchsucht werden können.

R25 Der Informationsraum bietet dem Benutzer eine einzige Schnittstelle um Objekte zu persistieren und Anfragen zu stellen.

Persistenz

R31 *Information Items* müssen persistent gespeichert werden können.

R32 Persistenz der Eigenschaften von visuellen Repräsentationen von *Information Items* (Visuelle Persistenz)

R33 *Information Items* sollten gelöscht und verändert werden können.

R34 Der persistierte Informationsraum sollte immer in konsistentem Zustand bleiben, auch wenn mehrere Clients gleichzeitig darauf zugreifen.

Synchronisation

R41 Jegliche Änderungen an *Information Items* müssen an alle Clients gesendet werden.

2.4.2 Nichtfunktionale Anforderungen

Zusätzlich zu den hier genannten funktionalen Anforderungen, existieren weitere nichtfunktionale Anforderungen an das Datenbackend. Diese werden nachfolgend kurz erläutert:

Performance Eine objektorientierte Benutzerschnittstelle ist auf eine gute User Experience angewiesen, da der Benutzer direkt-manipulativ mit den Informationsobjekten arbeitet und nicht die Arbeit an eine Anwendung delegiert. Er erwartet daher, dass die Bedienung flüssig ist und ihn nicht bei seiner Arbeit aufhält. Dementsprechend sollten alle Funktionalitäten, die direkten Einfluss auf die Arbeit des Benutzers mit der Benutzerschnittstelle haben, performant ausgeführt werden. Dies betrifft hauptsächlich auch das Stellen von Anfragen an den Informationsraum. Diese sollten in annehmbarer Zeit Ergebnisse liefern.

Anwendungsentwicklung Das ZOIL-Framework ist als Baukasten für die Entwicklung von Prototypen entwickelt worden. Dementsprechend sollte die Entwicklung eines ZOIL-Prototypen mithilfe des Frameworks auch gewisse Vorteile bieten. Der Entwickler sollte sich voll und ganz auf das Interaktionsdesign seines Prototyps konzentrieren können. Die Funktionalitäten des Datenbackend sollten daher ohne großen Aufwand in einen ZOIL-Prototypen eingebunden werden können.

3 Umsetzung

Das vorliegende Kapitel soll einen Einblick in die Umsetzung des Datenmodells und -backends liefern. Aus Gründen der Übersicht und des Platzes wird nicht in allen Bereichen bis auf die Ebene des Quelltexts vorgedrungen. Eine detailliertere Sicht auf die einzelnen Teile liefert dafür die Projektdokumentation des Autors [Zöllner, 2009], auf die hier verwiesen wird, sowie die Einträge im Wiki[10] zu diesem Projekt.

Bei der Entwicklung wurde zwischen *Datenbackend* und *Datenmodell unterschieden*. Wie bereits in der Einleitung beschrieben, ist das *Datenmodell* primär für die Modellierung der Informationsobjekte und des Informationsraums zuständig, während das *Datenbackend* sich um die Persistenz und die Synchronisation von Informationsobjekten kümmert.

Das ZOIL Framework, inklusive Datenmodell und -backend, wurde komplett in C# und WPF unter .NET 3.5 entwickelt. Die einzelnen Teilbereiche des Frameworks wurden dabei in mehrere Projekte aufgeteilt. Jedes dieser Projekte entspricht einer .NET Assembly. Zur Entwicklung in Visual Studio werden einzelne Projekte in einer Projektmappe (Solution) zusammengefasst. Abbildung 13 zeigt die Projekte, die bei der Entwicklung des Datenmodells und -backends zu einer Projektmappe zusammengefasst wurden.

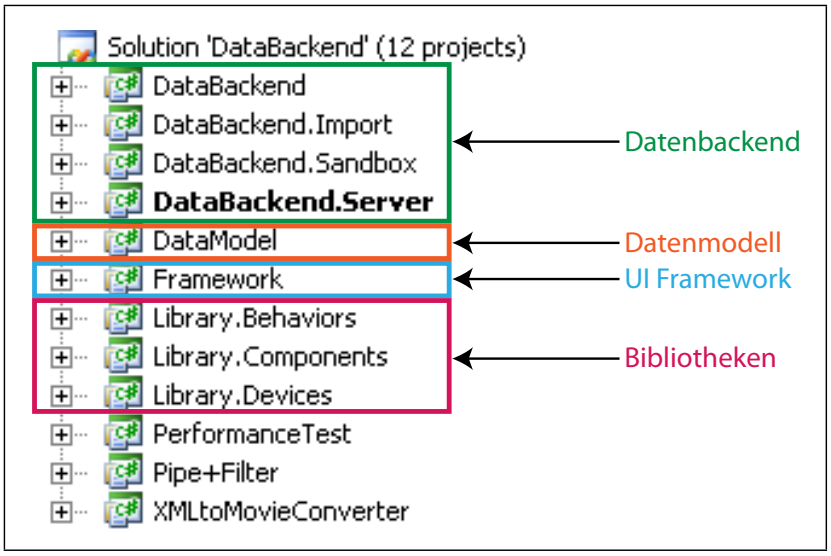

Abbildung 13: Visual Studio Solution für die Entwicklung des Datenmodells

[10] http://hci.uni-konstanz.de/permaedia/index.html

3.1 Datenmodell

3.1.1 Modellierung der Informationsobjekte

Das Konzept der Information Items wurde im Kapitel 2.1.1 auf Seite 19 detailliert vorgestellt. Hier soll nun gezeigt werden, wie Information Items in ZOIL modelliert werden (müssen), damit die zwei wichtigsten Anforderungen erfüllt werden, und zwar die Erweiterung eines Information Items zur Laufzeit mit beliebigen Daten (R13) und das Ausnützen der objektorientierten Konzepte Vererbung, Delegation und Polymorphie (R12). Ein zusätzlicher wichtiger Punkt ist die Implementierung von Organisationsstrukturen, wie z.B. einem semantischen Netzwerk (R21). Es wird gezeigt, wie sich solche Strukturen mithilfe der Information Items entwickeln lassen.

Erweiterbarkeit Erweiterbarkeit zur Laufzeit und objektorientiertes Modellieren sind auf den ersten Blick nur schwer miteinander vereinbar. Ein einmal modelliertes Objekt mit gekapselten Funktionalitäten und Daten, lässt sich zur Laufzeit in .NET nur sehr schwer verändern[11]. Dies liegt vor allem an der statischen Typisierung der Attribute des Objekts. Eine Lösung dieses Problems bietet eine Technik, die unter dem Namen *Property Bag* bzw. *Dynamic Properties* [Fowler, 1997] bekannt ist. Bei dieser Technik werden Attribute eines Objekts nicht statisch festgelegt, sondern können über sogenannte Key-Value Paare in einer Datenstruktur gespeichert werden. In .NET heißt diese Datenstruktur *Dictionary*. Als Key wird meist ein String-Objekt gewählt und als Value ein Objekt des Typs *Object*. Der Zugriff auf das Attribut erfolgt mittels spezieller Methoden. Ein Beispiel hierzu zeigt Listing 1.

```
class Movie
{
  private Dictionary<String, Object> bag;

  public Object GetValueOf(String key)
  {
    return bag[key];
  }

  public void SetValueOf(String key, Object value)
  {
    bag[key] = value;
  }
}
```

Listing 1: Öffentliche Schnittstelle einer Dynamic Properties Implementierung, angelehnt an [Fowler, 1997]

[11] Eine Veränderung zur Laufzeit erfordert den Einsatz der Common Intermediate Language (CIL) einer Art Assemblersprache für das .NET Framework

Die Vorteile einer solchen Implementierung sind offensichtlich. Zur Laufzeit kann dieses Objekt beliebige Werte speichern, wie in Listing 2 angedeutet wird.

```
Movie m = new Movie();
m.setValueOf("Title", "Batman");
...
String title = (String) m.GetValueOf("Title");
```

Listing 2: Nutzung der Dynamic Properties

Allerdings geht mit dieser Flexibilität ein wichtiger Bestandteil der objektorientierten Modellierung verloren. Die Schnittstelle, die ein Objekt durch ihre Attribute der Außenwelt anbietet, ist nicht mehr vorhanden. Bei einem statisch typisierten Objekt kann jeder Entwickler sofort sehen, welche Attribute das Objekt anbietet. Bei einer solchen Implementierung, ist das nur sehr schwer, oder gar nicht möglich. Die Verwendung von objektorientierten Konzepten, wie z.B. Vererbung oder Polymorphie ist ohne eine wohl definierte Schnittstelle nicht möglich.

In ZOIL wird nun eine Kombination der klassischen statischen Typisierung und der *Dynamic Properties* verwendet. Attribute werden statisch typisiert, ihre Werte werden allerdings mittels Key-Value-Paaren in einem *Property Bag* gespeichert. Mit dieser Implementierung ist es sowohl möglich, dem Entwickler eine wohl definierte Schnittstelle anzubieten, als auch die Objekte zur Laufzeit mit beliebigen neuen Werten zu erweitern. Listing 3 zeigt einen Ausschnitt eines Informationsobjekts aus ZOIL, bei dem dies umgesetzt wurde.

```
public class Movie : DInformationObject
{
  public String Title
  {
    get { return getValue("movie:title"); }
    set { setValue("movie:title", value); }
  }
}
```

Listing 3: Implementierung eines Properties in ZOIL

Jedes Informationsobjekt in ZOIL muss die Klasse *DInformationObject* erweitern (R11). Es bekommt dadurch automatisch Zugriff auf das *Property Bag* über die beiden Methoden *getValue* und *setValue*. Durch die statische Definition der Attribute können die Informationsobjekte die ganz normalen objektorientierten Konzepte, wie z.B. Vererbung und Polymorphie nutzen.

Es stellt sich nun natürlich die Frage, was der Vorteil dieser Implementierung ist. Attribute werden immer noch statisch definiert und der Zugriff auf das *Property Bag* ist weniger performant als z.B. die Verwendung von privaten Feldern als “Backing Store“ der Attribute. Oberflächlich betrachtet

hat man also nichts gewonnen, jedoch bietet diese Implementierung eine Möglichkeit für Entwickler, Funktionalitäten anzubieten, ohne dass dabei die Struktur der Informationsobjekte zu einem späteren Zeitpunkt verändert werden muss.

Angenommen es existiert bereits ein sehr aufwändig modellierter persönlicher Informationsraum, der in der Datenbank persistiert ist. Bei der Modellierung der Informationsobjekte wurde jedoch vergessen ein Attribut für Schlagworte (Tags) zu vergeben. Der Nutzer möchte nun aber gern seine Informationsobjekte nachträglich mit Tags versehen und auch nach diesen Tags suchen. Ein Entwickler könnte nun ein Plugin für ZOIL schreiben, das es dem Benutzer erlaubt, jedem Informationsobjekt beliebig viele Tags zuzuordnen. Anstatt dass der Entwickler nun eine eigene Klasse zum Speichern der Tags implementiert, kann er diese direkt im jeweiligen Objekt selbst speichern. Dies entspricht der objektorientierten Modellierung, da die Tags Metadaten sind, die das Objekt genauer beschreiben.

Ein weiteres Szenario das den Vorteil dieser Implementierung zeigt, ist die Visualisierung mehrerer heterogener Informationsobjekte. In diesem Fall besteht das Problem darin, dass die Visualisierung nur Attribute visualisieren kann, die in jedem der Objekte vorhanden sind. Diese Schnittmenge der Attribute muss aus den jeweiligen Objekten extrahiert werden. Da in ZOIL die Attribute über Key-Value Paare abgebildet werden, ist dies recht einfach möglich. Voraussetzung hierfür ist aber, dass Attribute, die die gleichen Informationen beschreiben, auch den gleichen Schlüssel für dieses Attribut vergeben. Es bietet sich daher an, einen gemeinsamen Metadatenstandard (wie z.B. Dublin Core[12]) für die Benennung der Attributschlüssel zu verwenden.

Listing 4 zeigt eine Beispielimplementierung für eine Methode in ZOIL, die die Schnittmenge der Attributschlüssel einer Liste von Objekten zurück liefert.

```
private List<String> getKeyIntersection(List<DInformationObject> objects)
{
  IEnumerable<String> keys = objects[0].getKeys();

  for(int i=1; i<objects.Count; i++)
    keys = keys.Intersect(objects[i].getKeys());

  return keys.ToList();
}
```

Listing 4: Generieren einer Schnittmenge der Attributschlüssel

Objektorientierung Abbildung 14 auf der nächsten Seite zeigt einen Ausschnitt der Klassen und Schnittstellen, die im Datenmodell und -backend für die Modellierung der Informationsobjekte verwendet werden. Die objektorientierte Modellierung der Information Items bedingt die Nutzung

[12] http://dublincore.org/

der Kernkonzepte der Objektorientierung. An erster Stelle wird dabei immer das Konzept der Vererbung genannt. Vererbung ist immer dann sinnvoll, wenn die Subklassen die vererbte Superklasse konzeptionell sinnvoll erweitern und dazu die Funktionalitäten und Attribute der Superklasse verwenden können. Im Gegensatz dazu werden Delegation und Polymorphie dann verwendet, wenn eine Vererbung konzeptionell keinen Sinn macht, beispielsweise wenn Funktionalitäten in beiden Klassen unterschiedlich implementiert werden müssen, oder wenn eine Mehrfachvererbung benötigt würde. Im Datenmodell ist die Klasse *DInformationObject* die Superklasse für alle Informationsobjekte. Sie implementiert die Methoden für die Dynamic Properties und bereitet das Objekt auf das Zusammenspiel mit der Datenbank *db4o* vor. In diesem Fall wird also Vererbung benutzt, da die Subklassen der Klasse *DInformationObject* dessen Funktionalität brauchen und nicht alternativ implementieren. Die Klasse *DInformationObject* implementiert die Schnittstelle *DInformationItem*. Diese Schnittstelle definiert die Properties und Methoden, die ein Information Item besitzen muss. Wie man im Klassendiagramm sehen kann wird diese Schnittstelle auch von der Klasse *DInformationCollection* implementiert. Hier wurde also das Konzept der Polymorphie ausgenutzt. Sowohl *DInformationObject*, als auch *DInformationCollection* nehmen die Gestalt der Schnittstelle *DInformationItem* an und können daher Dynamic Properties verwenden und in der Datenbank gespeichert werden. Der Grund für diese Art der Modellierung wird im nächsten Abschnitt diskutiert.

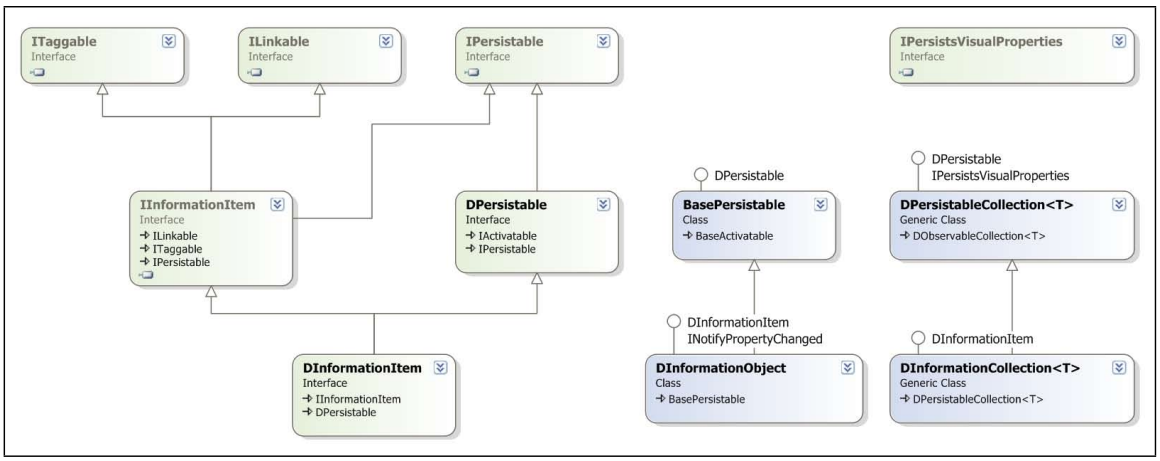

Abbildung 14: Klassen und Interfaces für die Modellierung von Information Items

Beziehungen & Organisationsstrukturen Durch die Verwendung von Dynamic Properties ist es möglich jedes Objekt zur Laufzeit zu erweitern. Diese Funktionalität kann dazu benutzt werden um beliebige Beziehungen zwischen Objekten herzustellen. Um Beispielsweise auszudrücken, dass eine Person der Autor eines Dokuments ist, kann im Dokument ein Dynamic Property *Author* erstellt werden und dabei auf das Information Item, das die Person repräsentiert, verwiesen werden (siehe Abbildung 15 auf der nächsten Seite).

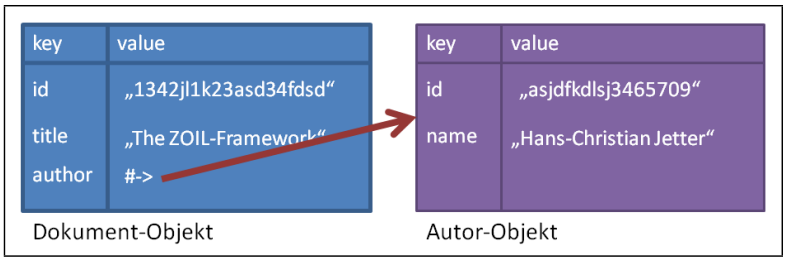

Abbildung 15: Herstellen einer Beziehung zwischen zwei Objekten

Modelliert man nun einen persönlichen Informationsraum auf diese Art und Weise, ergibt sich daraus recht bald ein Semantic Network, da jedes Objekt mit jedem anderen zur Laufzeit in Beziehung gesetzt werden kann. Durch den Einsatz von Dynamic Properties ist dabei keine Struktur vorgegeben; diese kann vom Benutzer selbst erstellt werden.

Im vorigen Abschnitt wurde bereits erwähnt, dass die Klasse *DInformationCollection* auch die Schnittstelle *DInformationItem* implementiert. Zusätzlich dazu ist die *DInformationCollection* dazu vorgesehen, Objekte des Typs *DInformationItem* zu verwalten. Eine *DInformationCollection* beinhaltet also weitere *DInformationItems* und ist selbst von diesem Typ. Mit dieser Art der Modellierung kann auf einfache Art und Weise eine Hierarchie gebildet werden, da eine *DInformationCollection* sowohl weitere *DInformationCollections* aufnehmen kann, als auch *DInformationObjects*. In Abbildung 16 wird diese Beziehung deutlich. Zusätzlich dazu kann jede *DInformationCollection* über den Dynamic Property Mechanismus mit beliebigen Attributen erweitert werden.

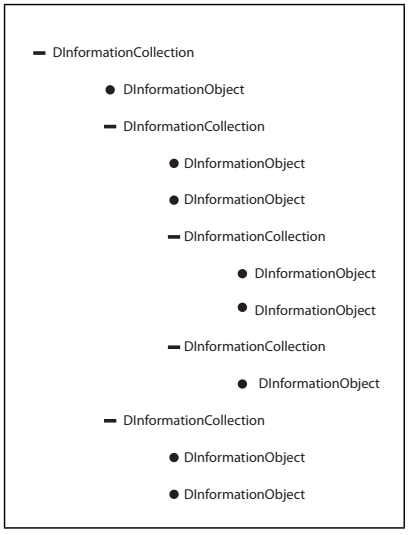

Abbildung 16: Hierarchie aus DInformationCollections und DInformationObjects

3.1.2 Trennung von visueller und logischer Repräsentation (R15)

Bei einer objektorientierten Benutzerschnittstelle werden in der Regel die Funktionalitäten und Daten der Objekte von der logischen Ebene bis in die Benutzerschnittstelle durchgereicht, da der Benutzer direkt auf den Objekten arbeitet. Es existieren nun Ansätze, wie z.B. das Naked Objects Pattern [Pawson, 2004], die aus der Klassendefinition der Objekte zur Laufzeit dynamisch eine visuelle Repräsentation (View) für jedes Objekt generieren. Eine so generierte View kann niemals die Ausdruckskraft besitzen, die eine von einem menschlichen Designer entworfene View besitzt. Sie mag vielleicht funktional und vollständig sein, ist in der Regel aber mit sehr einfachen Widgets, wie Listen oder Tabellen, realisiert. ZOIL ist per Definition ein höchst visuelles Paradigma, in dem die Repräsentation von Objekten einen hohen Stellenwert besitzt. Ein solcher Ansatz ist daher nicht tragbar. Views werden in ZOIL von Domänenexperten oder Designern entworfen. In der Windows Presentation Foundation wird hierzu die deklarative Markupsprache XAML verwendet. Jedes Element in der Benutzerschnittstelle kann mittels XAML deklariert werden. Die Funktionalität und Interaktivität der Elemente kann in einer sog. *Code-Behind*-Klasse definiert werden. Diese Trennung von visueller und logischer Ebene erleichtert gerade in professionellen Bereichen die Entwicklung von Benutzerschnittstellen enorm, da sich jeder Experte um seine Domäne kümmern kann. Es ist jedoch nicht ausreichend die Trennung nur auf der Ebene der einzelnen grafischen Elemente durchzuführen. *Separated Presentation Patterns*, wie beispielsweise Model-View-Controller (MVC) wenden diese Technik hauptsächlich auf Domänenobjekte, sog. Models, an. Jedes Model wird dabei durch eine eigene View in der Benutzerschnittstelle repräsentiert. Die View präsentiert dabei das gesamte Objekt und nicht nur Teile davon. Bei der Entwicklung von Benutzerschnittstellen mit der Windows Presentation Foundation hat sich die Verwendung des MVVM-Patterns (Model – View – ViewModel) bewährt. Dieses Pattern nutzt dabei extensiv die in WPF eingebauten Mechanismen, wie z.B. *Change Notification*, *Dependency Properties* und *Data Binding*. Der Einsatz des MVVM-Patterns im Datenmodell wird intensiv im Projektbericht zu dieser Arbeit beschrieben [Zöllner, 2009], es wird daher nur kurz auf die wichtigsten Mechanismen hingewiesen.

Im Datenmodell entspricht das *DInformationObject* dem Model des MVVM-Patterns, also der logischen Repräsentation eines Objekts. Für jeden Typ der von *DInformationObject* erbt, muss der jeweilige Domänenexperte zumindest eine View erzeugen. Mit dieser View wird das Objekt visuell auf der Landschaft repräsentiert. Im MVVM-Pattern ist nun noch eine dritte Komponente, das *ViewModel*, vorgesehen. Diese Komponente ist dafür zuständig, die Daten und Funktionalitäten aus dem Model in eine Form zu bringen, die die View darstellen kann. Das ViewModel hat zu diesem Zweck eine Referenz auf das Model und stellt der View die Daten zur Verfügung, die diese für die Visualisierung des Objekts braucht. Die View konsumiert die Daten des ViewModels mithilfe des Data Bindings von WPF. Das ViewModel dient somit als Vermittler zwischen Model und View. Daten, die in der View geändert werden, werden vom ViewModel an das Model weitergereicht und umgekehrt.

Ein Beispiel soll den Zweck des ViewModels verdeutlichen: Ein Informationsobjekt des Typs *Person* soll in der Landschaft dargestellt werden. Ein Attribut dieses Objekts ist das Geburtsdatum der Person. In der normalen Ansicht soll allerdings nicht das Geburtsdatum der Person dargestellt werden, sondern ihr momentanes Alter. Das ViewModel ist nun dafür zuständig, das Geburtsdatum, das es aus dem Model erhält in das passende Alter umzurechnen und der View zur Verfügung zu stellen. Die View stellt das Geburtsdatum als Text dar. Zur Änderung der Daten der Person, existiert nun eine andere View. Das dazugehörige ViewModel reicht hier das Geburtsdatum als Attribut vom Model in die View weiter. Die View stellt das Geburtsdatum in einer Textbox dar, in der es vom Benutzer verändert werden kann. Die Abbildung 17 verdeutlicht diesen Zusammenhang.

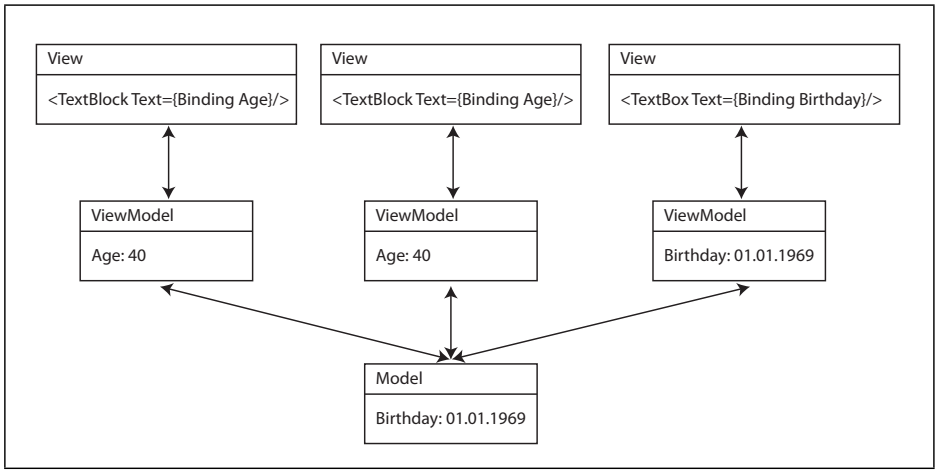

Abbildung 17: Anwendung des MVVM-Patterns

Mithilfe des MVVM-Patterns ist es möglich, dass pro Model mehrere Views in die Informationslandschaft gelegt werden können. Zwischen Model und View besteht demnach eine 1:n Beziehung, wobei zwischen View und ViewModel eine 1:1 Beziehung besteht.

Die Views (und das dazugehörige ViewModel) der Objekte werden nicht in der Datenbank persistiert, sondern zur Laufzeit erzeugt, wenn ein Objekt in der Landschaft dargestellt werden soll. Damit es trotzdem möglich ist, visuelle Eigenschaften von Objekten zu persistieren, wurden die Visual Properties entwickelt. Diese werden im nächsten Abschnitt vorgestellt. Der genaue Mechanismus, der verwendet wird, um für ein Model zur Laufzeit eine View zu erstellen, wird in [Zöllner, 2009] detailliert erklärt.

3.1.3 Visuelle Persistenz durch Visual Properties (R32)

Ein Benutzer, der Informationsobjekte auf der Informationslandschaft von ZOIL nach seinem Belieben anordnet, erwartet vom System, dass diese Arbeit nicht verloren geht, sondern bei der

nächsten Nutzung des Systems wieder hergestellt wird. Wie oben bereits erwähnt, werden die visuellen Repräsentationen von Objekten nicht in der Datenbank gespeichert. Dies ist auch nicht sinnvoll, da diese zur Laufzeit wieder hergestellt werden können, vorausgesetzt sie werden mit den richtigen Daten gefüllt. Im Datenbackend ist es möglich, die visuellen Eigenschaften einer View in speziellen Objekten, so genannten *Visual Properties*, abzuspeichern, um damit eine *visuelle Persistenz* der Informationslandschaft zu erhalten. Da es durchaus vorkommen kann und auch so gewollt ist, dass ein Informationsobjekt mehrfach in der Landschaft verortet wird, werden die *Visual Properties* nicht im Informationsobjekt selbst gespeichert, sondern in einem eigenen Objekt, das zur Laufzeit an die View gebunden wird. Die *Visual Properties* für ein Objekt beinhalten folgende Eigenschaften [Zöllner, 2009]:

- Position (X und Y Koordinaten)
- Größe (Höhe und Breite)
- Rotationswinkel
- ZIndex
- Opacity
- ViewIndex (gibt an, mit welcher View das Objekt angezeigt werden soll)
- TemplateId (gibt an, mit welchem Template die View versehen werden soll)

Wenn ein Objekt in der Landschaft angezeigt werden soll, muss der Entwickler eine Instanz des Typs *IVisualProperties* mit angeben. Die Eigenschaft *ViewIndex* wird dann dazu herangezogen, die passende View für das Objekt zu generieren. Diese View wird in die Landschaft gelegt. Die weiteren Eigenschaften aus den *Visual Properties* werden dann mittels WPF Data Binding an die *Dependency Properties* der View gebunden. Verändert sich nun eine Eigenschaft der View, z.B. die Höhe, wird diese Änderung automatisch an die *Visual Properties* gesendet. Da die *Visual Properties* in der Datenbank gespeichert werden, ist somit die *Visuelle Persistenz* der Informationslandschaft sicher gestellt. Das genaue Zusammenspiel zwischen *Visual Properties*, Informationslandschaft und Informationsobjekt wird im nächsten Abschnitt besprochen, nachdem das Konzept der *RootCollection* eingeführt wurde.

3.1.4 Personal Information Collections (R16)

Die Umsetzung von *Personal Information Collections* in ZOIL erforderte aus verschiedenen Gründen eine Eigenentwicklung. So ist es zum einen erforderlich, dass Änderungen innerhalb einer Collection direkt den Weg in die Benutzerschnittstelle finden. Andererseits muss die Collection sehr eng mit der Datenbank zusammen arbeiten, so dass Änderungen innerhalb der Collection

persistiert und aktualisiert werden. Ein weiterer Punkt, der vor allem im Zusammenspiel mit der Informationslandschaft zum tragen kommt, ist das Verwalten von Visual Properties für die Objekte der Collection. Diese Punkte werden in den folgenden Abschnitten näher betrachtet.

Änderungen an Benutzerschnittstelle und Datenbank weiterreichen Von einer View (z.B. einer ListBox), die sich an eine Collection bindet, wird erwartet, dass sie auf die Änderungen, die an der Collection getätigt werden, reagiert, indem sie die neu hinzugekommen Objekte anzeigt und die gelöschten Objekte nicht mehr anzeigt. In der Windows Presentation Foundation werden allerdings Änderungen, die an einer Collection getätigt werden, nicht automatisch der Außenwelt mitgeteilt. Damit dies möglich ist, muss eine Collection-Klasse die Schnittstelle *INotifyCollectionChanged* implementieren. Diese Schnittstelle definiert ein Ereignis *CollectionChanged* welches immer dann gefeuert wird, wenn sich die innere Zusammenstellung der Collection ändert, also beim Einfügen und Löschen von Objekten. Die Collection-Typen des Datenmodells implementieren alle die *INotifyCollectionChanged* Schnittstelle. Eine View, die sich an eine Collection des Datenmodells bindet, wird daher immer über Änderungen informiert und kann sich dementsprechend anpassen.

Im Normalfall ist es nicht notwendig, selbst Collection-Typen zu erstellen, die diese Schnittstelle implementieren, da WPF bereits eine Klasse *ObservableCollection* anbietet, die dies tut. Da die Änderungen einer Collection wegen der Synchronisierung jedoch auch immer sofort in der Datenbank gespeichert werden müssen, ist die Verwendung der *ObservableCollection* im Datenmodell nicht möglich. Für das Zusammenspiel mit der Datenbank *db4o*, die im Datenbackend für die Persistenz verwendet wird, musste daher eine neue Collection erstellt werden, die sowohl die *INotifyCollectionChanged* Schnittstelle implementiert, als auch Änderungen an der Collection direkt in die Datenbank speichert.

Verwalten von Visual Properties Setzt man das MVVM-Pattern, das bei den Informationsobjekten zum Einsatz kommt, konsequent um, so kann man sagen, dass die Informationslandschaft die View einer speziellen Collection ist [Zöllner, 2009]. Diese Collection wird in ZOIL *RootCollection* genannt. In ihr befinden sich alle Objekte, die auf der Informationslandschaft dargestellt werden sollen. Das erfolgreiche Anzeigen einer View für ein Objekt in der Landschaft erfordert das Zusammenspiel von mehreren Konzepten. Zum einen besteht die View der *RootCollection* aus der Informationslandschaft in welcher ein spezieller Canvas (*ZLandscapeCanvas*) liegt. An diesen Canvas wird die *RootCollection* gebunden, so dass der Canvas Zugriff auf die Elemente in der *RootCollection* hat. Zusätzlich zu den logischen Elementen, die in der *RootCollection* liegen, verwaltet die *RootCollection* für jedes Element eine Liste von Visual Properties. Der Canvas erzeugt aus jedem dieser Visual Properties Instanzen dynamisch eine View des Objekts und visualisiert diese. Dadurch dass die *RootCollection* eine Liste von Visual Properties verwaltet, ist es möglich, mehr als eine View pro Objekt in der Landschaft anzuzeigen. Zusätzlich dazu wird jedes Attribut der Visual Properties an das entsprechende Attribut der View gebunden. Dieses Data Binding ist essentiell, da

es sicherstellt, dass Änderungen, die an der View getätigt werden (z.B. Verschieben auf der Landschaft) direkt in den Visual Properties gespeichert werden und umgekehrt Änderungen, die von der Datenbank an die Visual Properties gesendet werden, direkt an die View weitergeleitet werden. Der Anwendungsentwickler bekommt von diesen Mechanismen im Hintergrund nichts mit. Listing 5 zeigt, welche Schritte unternommen werden müssen um ein Objekt in der Landschaft zu visualisieren.

```
//add an information object with default view to the landscape
IVisualProperties vp = VisualProperties.Default();
myRootCollection.Add(myInformationObject, vp);

//add another (modified) default View to the landscape
IVisualProperties vp2 = VisualProperties.Default();
vp2.X = 100;
vp2.Y = 100;
myRootCollection.AddVisualProperties(myInformationObject.Id, vp2);
```

Listing 5: Hinzufügen von zwei Views zur Informationslandschaft

3.2 Datenbackend

3.2.1 db4o als Datenbanklösung

Die Persistenz von Informationsobjekten kann auf verschiedene Art und Weise erreicht werden. Von der normalen SQL-Datenbank (z.B. MySQL[13]), über objektrelationale Mapper (z.B. NHibernate[14]), Graph-Datenbanken (z.B. neo4j[15]) bis zu XML-Datenbanken (z.B. BaseX[16]), existiert ein großes Spektrum an Datenhaltungslösungen.

Objektrelationale Mapper sind für das ZOIL-Datenbackend deshalb keine befriedigende Lösung, da für jede neu eingeführte Klasse von Informationsobjekten ein Mapping zwischen der Objektstruktur und der relationalen Tabellenstruktur hergestellt werden müsste. Darunter leidet letztendlich die Flexibilität des ganzen Systems. Außerdem stellt dies eine sehr große Hürde für Anwendungsentwickler dar, die ZOIL nur für die Entwicklung von Prototypen verwenden wollen.

Einen weiteren Typ von Datenbank gilt es generell im Auge zu behalten, und zwar die Graph-Datenbanken. Bei bestimmten Anwendungen, wie beispielsweise Hypertextsystemen oder Geoinformationssystemen, sind die Daten intern auf vielfältigste Art und Weise miteinander verbunden.

[13] http://www.mysql.com
[14] http://nhforge.org
[15] http://neo4j.org/
[16] http://www.inf.uni-konstanz.de/dbis/basex/

Diese den Daten inhärente Graphstruktur kann in Graph-Datenbanken modelliert werden [Angles and Gutierrez, 2008]. Der Fokus dieser Modellierung liegt dabei neben den bloßen Daten auch explizit auf den Verbindungen zwischen den Daten. So sind die Anfragesprachen mittels spezieller Graph-Algorithmen in der Lage, nicht nur einzelne Daten abzufragen, sondern auch die den Daten inhärente Struktur. Die in ZOIL geforderte Flexibilität der Modellierung des Informationsraums könnte mit Graph-Datenbanken sicherlich umgesetzt werden. Inwiefern dagegen ein Mapping zwischen den Informationsobjekten von ZOIL und den Daten des Graph-Modells notwendig wäre, ist momentan nicht abzusehen. Leider existiert zum jetzigen Zeitpunkt noch keine befriedigende Lösung für das .NET Framework, weswegen eine tiefergehende Evaluation dieses Datenbanktyps nicht möglich war.

Bei XML-Datenbanken besteht größtenteils das gleiche Problem, wie bei relationalen Datenbanken: Es muss für jeden Typ von Informationsobjekt ein Mapping zwischen der Objektstruktur und der XML-Struktur erstellt werden.

Die objektorientiert modellierten Informationsobjekte in ZOIL sind prädestiniert dafür, in einer echten Objektdatenbank persistiert zu werden. Objektdatenbanken besitzen gegenüber den meisten anderen Datenbanken den Vorteil, dass kein Mapping zwischen dem Objektmodell und dem Datenhaltungsmodell erstellt werden muss. Eine Objektdatenbank persistiert in der Regel jedes ihm übergebene Objekt, ohne irgendwelche Konvertierungen vorzunehmen. Eine sehr populäre, weil performante Objekt-Datenbank ist db4o[17], deren kostenlose Nutzung im nichtkommerziellen Bereich dank einer Open Source Lizenz sicher gestellt ist. Für das Datenbackend stellt sie eine ideale Lösung dar, da viele der Anforderungen, die im vorigen Kapitel erhoben werden, mit db4o erfüllt werden können. In den folgenden Abschnitten wird der Einsatz von db4o im Datenbackend diskutiert.

3.2.2 Der ZOIL Server

Obwohl db4o ursprünglich nur für “embedded“-Lösungen[18] konzipiert war, existiert mittlerweile eine sehr ausgereifte Client-Server Version von db4o. Diese Version macht sich auch das Datenbackend zu Nutze, war doch unter anderem eine Anforderung, dass der Informationsraum von überall her zugänglich sein muss (R22). Das Anlegen einer db4o-Datenbank wird von db4o leider nicht mit speziellen Tools ermöglicht, da db4o nur eine Objektdatenbank (ODB), und kein Objektdatenbankmanagementsystem (ODBMS) ist. Für den Server wurde daher ein spezielles Tool entwickelt, über das mittels einer grafischen Schnittstelle Datenbanken erstellt, gelöscht, gestartet, gestoppt und geleert werden können. Innerhalb dieses Tools können mehrere Datenbanken verwaltet werden. Abbildung 18 auf der nächsten Seite zeigt das GUI des ZOIL Servers.

[17] http://www.db4o.com

[18] “embedded“ bedeutet, dass die Datenbank innerhalb der Anwendung ausgeführt wird, und nicht in einem eigenen Prozess

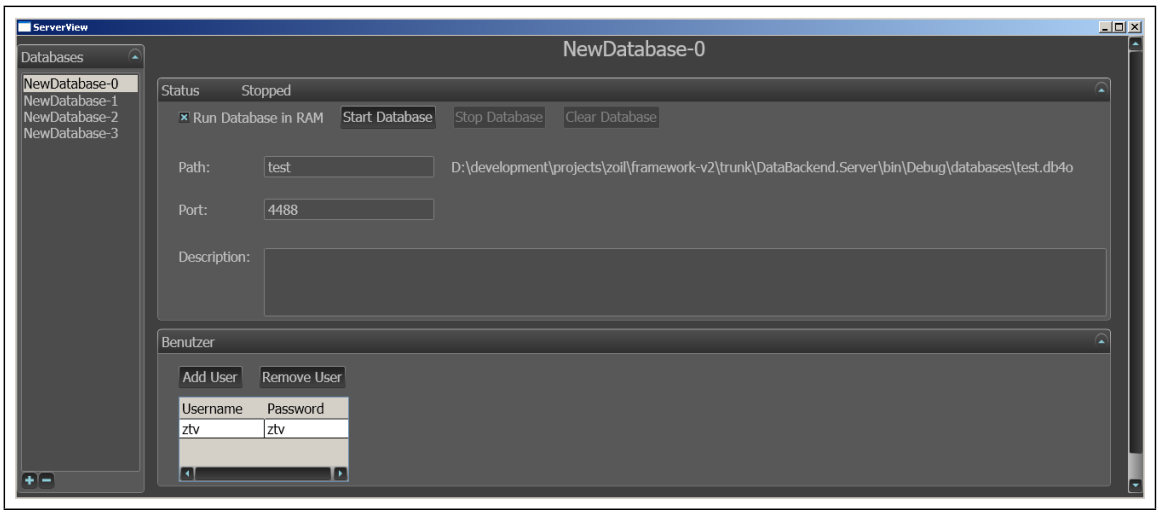

Abbildung 18: GUI des ZOIL Servers zur Verwaltung verschiedener Datenbanken

3.2.3 Der Client - Die *Database* Klasse

Auf der Client-Seite wird die Anbindung von db4o über die *Database* Klasse ermöglicht. Die Klasse wurde als Singleton implementiert, so dass zur Laufzeit nur eine Instanz dieser Klasse existiert [Gamma et al., 1995]. Ein Vorteil dieser Implementierung ist, dass keine Datenbank Handles gespeichert, oder als Parameter übergeben werden müssen, da der Zugriff auf die *Database* Klasse von überall aus möglich ist [Zöllner, 2009]. Listing 6 zeigt, wie man im Code an die *Database* Instanz gelangt. Mit dieser Art der Implementierung, wird die Anforderung umgesetzt, dass es nur eine Schnittstelle zum Informationsraum geben soll (R25). Jegliche Kommunikation mit dem PSI läuft über die *Database* Klasse.

```
Database db = Database.getInstance();
```

Listing 6: Zugriff auf die Database-Instanz

Verbindung zu db4o Bei der Verwendung von db4o ist es üblich, dass zu Beginn der Anwendung eine Verbindung zur Datenbank hergestellt wird, die erst beim Beenden der Anwendung wieder geschlossen wird. Zum einen ist die Arbeit mit der Datenbank dadurch wesentlich performanter, da nicht bei jedem Aufruf an die Datenbank erst eine Verbindung hergestellt werden muss. Wichtiger ist aber die Tatsache, dass db4o beim Herstellen einer Verbindung auf dem Client einen lokalen Cache erstellt. In diesem Cache werden diejenigen Objekte vorgehalten, die von der Anwendung aus der Datenbank angefordert wurden. Beendet man während der Laufzeit der Anwendung die Verbindung zur Datenbank, so wird der Cache gelöscht. Verändert man nun ein Objekt und öffnet

die Verbindung zur Datenbank erneut, damit diese Änderung persistiert werden kann, kann db4o keine Verbindung zwischen dem Objekt im Hauptspeicher und dem Objekt in der Datenbank herstellen; die Änderungen werden daher nicht persistiert. Damit dies verhindert werden kann, sollte beim Starten einer ZOIL-Anwendung eine Verbindung zur Datenbank hergestellt werden und erst beim Beenden wieder geschlossen werden.

Activation & Object-Cache Ein wesentlicher Punkt für das Verständnis von db4o ist die Aktivierung von Objekten. Eine Anfrage, die an den db4o-Server gestellt wird, liefert eine Liste von Objekten zurück. Die Objekte sind zu diesem Zeitpunkt noch nicht mit den Daten gefüllt. Dies geschieht erst dann, wenn das Objekt aktiviert wird. Bei der Aktivierung werden alle Attributwerte des Objekts übertragen und das Objekt kann danach ganz normal verwendet werden. Nach der Aktivierung befindet sich das Objekt in einem von db4o verwalteten Cache. Eine erneute Anfrage nach dem Objekt löst keine weitere Aktivierung des Objekts aus, da das Objekt bereits im Cache ist. Im Datenbackend wird die von db4o angebotene *Transparent Activation* benutzt, um Objekte zu aktivieren. Bei der *Transparent Activation* werden die Objekte erst dann aktiviert, wenn sie auch wirklich gebraucht werden, z.B. beim Zugriff auf ein Attribut des Objekts. Details zur *Transparent Activation* können in [Zöllner, 2009] nachgelesen werden.

3.2.4 Arbeiten mit Objekten

Transaktionen db4o ist wie viele andere Datenbanksysteme transaktionsbasiert. Lokale Änderungen, wie z.B. das Speichern, Löschen und Ändern von Objekten werden erst dann an den Server übertragen, wenn die Transaktion, in der die Änderungen durchgeführt wurden, abgeschlossen wurde. Transaktionen sind atomare Operationen, die von keiner anderen Transaktion unterbrochen werden können. Damit ist sicher gestellt, dass die Objekte der Datenbank nicht in inkonsistente Zustände gebracht werden können. Zum Abschließen einer Transaktion wird ein sogenannter *Commit* aufgerufen. Gerade für die Synchronisierung von mehreren Clients ist es wichtig, dass Änderungen an Objekten sofort an die Datenbank gesendet werden. Aus diesem Grund wird der Commit-Befehl im Datenbackend regelmäßig im Hintergrund aufgerufen. Die Zeit zwischen zwei Commits kann vom Client in der Konfiguration der Database-Klasse festgelegt werden. Ist das automatische Aufrufen der Commits deaktiviert, muss eine Transaktion über einen manuellen Commit abgeschlossen werden (siehe Listing 7).

```
Database.getInstance().CommitChanges();
```

Listing 7: Abschließen einer Transaktion, entnommen aus [Zöllner, 2009]

Persistieren eines Objekts Das Persistieren von Objekten ist die Hauptaufgabe, die db4o im Datenbackend übernehmen soll (R31). Reguläre .NET-Objekte besitzen sowohl Attribute, die aus primitiven Datentypen bestehen, als auch Attribute, die Referenzen auf andere Objekte darstellen. Ein solches Objekt zu persistieren, bedeutet, dass nicht nur die primitiven Attribute persistiert werden, sondern auch die Referenzen auf andere Objekte inklusive dieser referenzierten Objekte. Das Speichern eines einzigen Objekts kann daher das Speichern eines kompletten Objektgraphen zur Folge haben. Listing 8 zeigt, wie ein reguläres .NET Objekt mit Hilfe der Database-Klasse in der Datenbank gespeichert werden kann.

```
Movie m = new Movie();
Database.getInstance().StoreItem(m);
```

Listing 8: Speichern eines Objekts, entnommen aus [Zöllner, 2009]

Löschen eines Objekts Analog zum Speichern eines Objekts, funktioniert das Löschen von Objekten, die zuvor aus der Datenbank geholt wurden (siehe Listing 9). Wie bereits in [Zöllner, 2009] erwähnt, wird über diese Methode nicht der komplette Objektgraph gelöscht (CascadeOnDelete), da die Gefahr besteht, die referentielle Integrität der Datenbank zu zerstören. Diese Funktionalität muss vom Client selbst implementiert werden, falls dies gewünscht ist.

```
Database.getInstance().Delete(m.ID);
```

Listing 9: Löschen eines Objekts, entnommen aus [Zöllner, 2009]

Ändern eines Objekts Änderungen an lokalen Objekten werden db4o normalerweise über einen erneuten Aufruf der Store-Methode (vgl. Listing 8) mitgeteilt. Beim nächsten Commit werden diese Änderungen dann übertragen. db4o kann Änderungen an Objekten aber auch völlig transparent für den Entwickler an den Server übertragen. Dieser Mechanismus wird *Transparent Persistence* genannt. Objekte, die für diesen Mechanismus konfiguriert wurden, müssen ihre Änderungen nicht explizit über die Store-Methode bekannt geben. Durch das Setzen eines Flags werden Änderungen an diesen Objekten beim nächsten Commit automatisch übertragen. In die Klasse *DInformationObject* des Datenbackends, wurde dieser Mechanismus eingebaut. Da diese Klasse die Basis für alle Informationsobjekte bildet, funktioniert die *Transparent Persistence* automatisch für jedes Informationsobjekt. In Kombination mit dem automatischen Versenden von Commit-Befehlen, werden damit Änderungen die an einem Objekt vorgenommen werden, ohne weiteres Zutun des Entwicklers in der Datenbank persistiert [Zöllner, 2009].

Einschränkungen Beim Speichern eines Objekts wird, wie oben erwähnt, versucht, den kompletten Objektgraph des Objektes zu persistieren. Nachteile hat dieses Verhalten dann, wenn in diesem Objektgraphen Objekte enthalten sind, die db4o nicht persistieren kann. Dies trifft vor allem auf sehr betriebssystemnahe Objekte zu, beispielsweise ein Handle auf ein Fenster. db4o geht dann davon aus, dass das ursprünglich zu speichernde Objekt nicht speicherbar ist und gibt eine Fehlermeldung aus. Für das Datenbackend hat dies zur Folge, dass beim Modellieren der Objekte sorgfältig darauf geachtet werden muss, dass keine "nicht-speicherbaren" Objekte im Graphen eines Objekts vorkommen. Insbesondere ist es beispielsweise nicht möglich grafische Elemente von WPF in db4o zu speichern, da diese im Kern Referenzen auf betriebssystemnahe Objekte besitzen.

Die Objekte, die beim db4o Server zum Speichern in Auftrag gegeben werden, werden zuvor von db4o analysiert und serialisiert, damit sie in die Datenbankdatei auf der Festplatte geschrieben werden können. Die Geschwindigkeit dieses Prozesses ist mit davon abhängig, ob der db4o Server die Struktur der Objekte bereits kennt, oder nicht. Besitzt der db4o-Server eine Schablone, in Form der Klasse des Objekts, ist die Serialisierung wesentlich performanter. Ist die Klasse eines Objekts nicht vorhanden, muss db4o das Objekt nämlich mittels Reflection[19] analysieren. Da dem ZOIL-Server standardmäßig keine Klassen der Informationsobjekte übergeben werden, ist das Speichern eines Objekts in ZOIL nicht optimal schnell.

Ein ähnliches Problem tritt auf, wenn ein Client X ein neues Informationsobjekt persistiert, dessen Typ Client Y nicht kennt. Wenn beide Clients auf dem gleichen Informationsraum arbeiten, führt dies zwangsläufig zu Problemen bei Client Y, da db4o auf diesem Client das Objekt nicht erstellen kann. Dieses Problem muss zum jetzigen Zeitpunkt noch von jedem Client selbst gelöst werden. Für die Zukunft könnte man sich jedoch vorstellen, dass die Serverkomponente von ZOIL nicht nur die Objekte der Clients persistiert, sondern auch die Klassen der Objekte von den Clients anfordert und an die anderen Clients verteilt.

3.2.5 Anfragen stellen

Spezifische Anfragen an das Datenbackend bzw. die Datenbank zu stellen, wird in vielen ZOIL-Anwendungen von äußerster Wichtigkeit sein (R24). db4o bietet dem Entwickler dazu vier verschiedene Anfragetechniken an, die sich in ihrer Mächtigkeit und Syntax unterscheiden. Auf dem Server akzeptiert db4o nur eine der vier Techniken, die *SODA-Queries*. Der db4o Client muss daher Anfragen, die mit einer der drei anderen Techniken gestellt wurden, zuerst in das SODA Format konvertieren, bevor er sie an den Server sendet. SODA-Queries werden in der Regel komplett auf dem Server ausgeführt. Das Ergebnis einer solchen Query wird dann in Form einer Liste von Objekten an den Client zurück gesendet. In bestimmten Fällen ist es nicht möglich, dass die Anfrage in das SODA Format konvertiert werden kann, oder der Server kann bestimmte Teile einer

[19] Mit Reflection kann die Struktur von Objekten zur Laufzeit analysiert werden.

SODA-Query nicht verarbeiten. In diesem Fall wird nur ein kleiner Teil der Anfrage an den Server gesendet. Der Server sendet dann eine Liste aller Objekte zurück, die diesem kleinen Teil entsprechen. Der Rest der Anfrage muss auf dem Client ausgeführt werden. Dazu müssen alle Objekte, die in der Liste übertragen wurden, aktiviert werden, wenn sie nicht bereits schon im Cache des Clients liegen. Eine solche Anfrage dauert in der Regel wesentlich länger, als eine Anfrage, die direkt auf dem Server ausgeführt wird.

Bei der Nutzung des Datenbackends ist es nun enorm wichtig, dass mächtige Anfragen gestellt werden können und dass diese Anfragen in annehmbarer Zeit Ergebnisse bringen. Aus diesem Grund werden im Datenbackend alle Objekte, die sich in der Datenbank befinden, zu Beginn der Verbindung auf den Client übertragen und dort in einer Liste bereitgehalten. Die Liste wird ständig aktuell gehalten, so dass Änderungen, die andere Clients tätigen, sich dort sofort widerspiegeln. Anfragen an die Datenbank werden komplett umgangen und an diese Liste gesendet. Dadurch, dass sich die Objekte der Liste bereits im Hauptspeicher des Clients befinden, sind Anfragen auf diese Liste enorm schnell (vgl. [Zöllner, 2009]).

Im Kapitel 4 auf Seite 54 wird diskutiert, wieso dieser Ansatz für große Informationsräume zum Scheitern verurteilt ist.

LINQ-Queries Um einen groben Eindruck zu vermitteln, wie Anfragen im Datenbackend aussehen können, wird hier kurz eine der vier Anfragetechniken, LINQ (**L**anguage **IN**tegrated **Q**uery), vorgestellt. Listing 10 zeigt eine einfache LINQ-Query, die alle Schauspieler von Batman-Filmen zurück liefert.

```
var batmanActors = from Movie m in Database.getInstance().Objects
                   where m.Title.Contains(``Batman``)
                   select m.Actors
```

Listing 10: Einfache LINQ-Anfrage, entnommen aus [Zöllner, 2009]

Mit LINQ kann sowohl die Datenbank, als auch die Liste, die zu Beginn der Verbindung gebildet wird, angefragt werden, da LINQ zum einen integraler Bestandteil des .NET Frameworks ist und von db4o in das SODA Format konvertiert werden kann. Im Kapitel 4 auf Seite 54 werden LINQ-Queries hinsichtlich ihrer Geschwindigkeit evaluiert. Für einen tieferen Einblick in LINQ wird auf die Fachliteratur verwiesen, beispielsweise [Marguerie et al., 2008].

3.2.6 Synchronisierung

Die Synchronisierung verschiedener ZOIL-Clients kann auf mehreren Stufen erreicht werden (siehe Abbildung 19 auf der nächsten Seite). Die einfachste Form der Synchronisierung spielt sich auf

der Ebene von Pixeln ab. Protokolle, wie RDP oder VNC ermöglichen das Senden des Bildschirminhalts eines Servers an ein anderes Gerät. Maus- und Tastatureingaben von diesem werden an den Server zurück gesendet und dort interpretiert. Auf einer etwas tieferen Stufe wird in ZOIL die Sicht auf den Visual Tree synchronisiert. Ein gängiges Einsatzszenario ist beispielsweise die Synchronisierung der Interaktion mit der Informationslandschaft (Zooming + Panning). Hier wird das OSC-Protokoll zum Versenden von Multicast-Commands an die entsprechenden Clients verwendet. Die Synchronisierung, die im Datenbackend implementiert wurde, spielt sich auf der Ebene des Datenmodells ab. Das vom ZOIL-Server in den Arbeitsspeicher geladene Datenmodell wird ständig mit dem persistenten Datenmodell des Servers synchron gehalten. Änderungen, die an diesem Modell bei einem Client getätigt werden, werden zuerst an den Server gesendet und von diesem dann an alle anderen Clients verteilt. Der Vorteil gegenüber den anderen Methoden ist, dass auf jedem Client individuell interagiert werden kann, ohne dass die Interaktion auf einem anderen Client dadurch gestört wird. Während die Synchronisierung auf der zweiten Stufe genau dies erzwingen möchte, ist die Synchronisierung auf der ersten Stufe nicht einmal in der Lage mehrere Clients für die Interaktion zuzulassen.

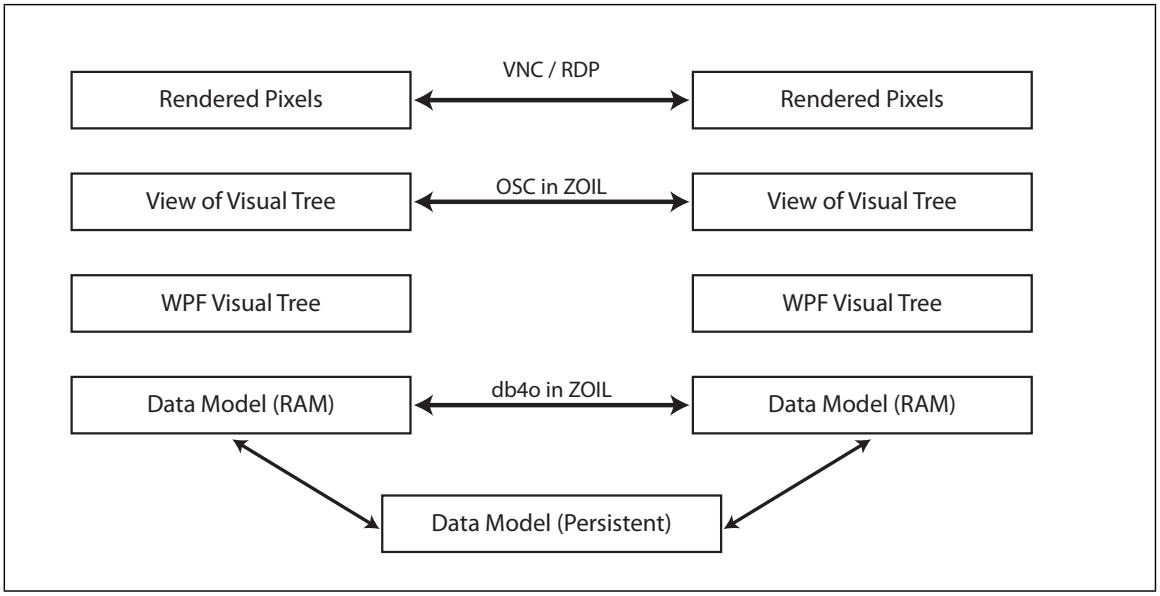

Abbildung 19: Verschiedene Stufen der Synchronisierung

Die Synchronisierung von Informationsobjekten ist eng gekoppelt mit den Transaktionen. Bei jedem *Commit* werden Änderungen an Objekten, neu erstellte Objekte oder das Löschen von Objekten automatisch an den Server übertragen. Der Server reagiert auf jeden *Commit* mit einem sog. *Callback*. Dieser *Callback* ist ein reguläres .NET Event, das auf *jedem* Client gefeuert wird und dort von der Database Klasse abgefangen wird. Als Parameter des Events wird jeweils eine Liste der veränderten, neuen und gelöschten Objekte mit übergeben. Die Objekte dieser Listen werden in

eine *Queue* gelegt, wo sie von einigen *Background Threads* asynchron weiter verarbeitet werden.

Die veränderten Objekte liegen zu diesem Zeitpunkt noch nicht in der aktuellen Version vor. Veränderte Objekte müssen explizit bei der Datenbank aufgefrischt werden, damit sie dem neuesten Zustand in der Datenbank entsprechen. Dadurch dass db4o an alle Clients einen Callback sendet, wird auch der Client, der das Objekt ursprünglich geändert hat, mit einem Callback benachrichtigt. Dieser muss das veränderte Objekt nicht auffrischen, da er bereits die aktuellste Version besitzt.

Das Auffrischen eines Objektes wird nicht automatisch von .NET bis ins User Interface durchgereicht. Damit dies möglich ist, bietet das Datenbackend die Schnittstelle *IRemoteUpdateable* an. Jedes Informationsobjekt, das diese Schnittstelle implementiert, wird über den Aufruf der Methode *OnRemoteUpdate* darüber benachrichtigt, dass es sich verändert hat. Diese Information kann das Informationsobjekt dann selber bis ans User Interface weiterreichen, damit eventuell die Repräsentation des Objekts angepasst wird.

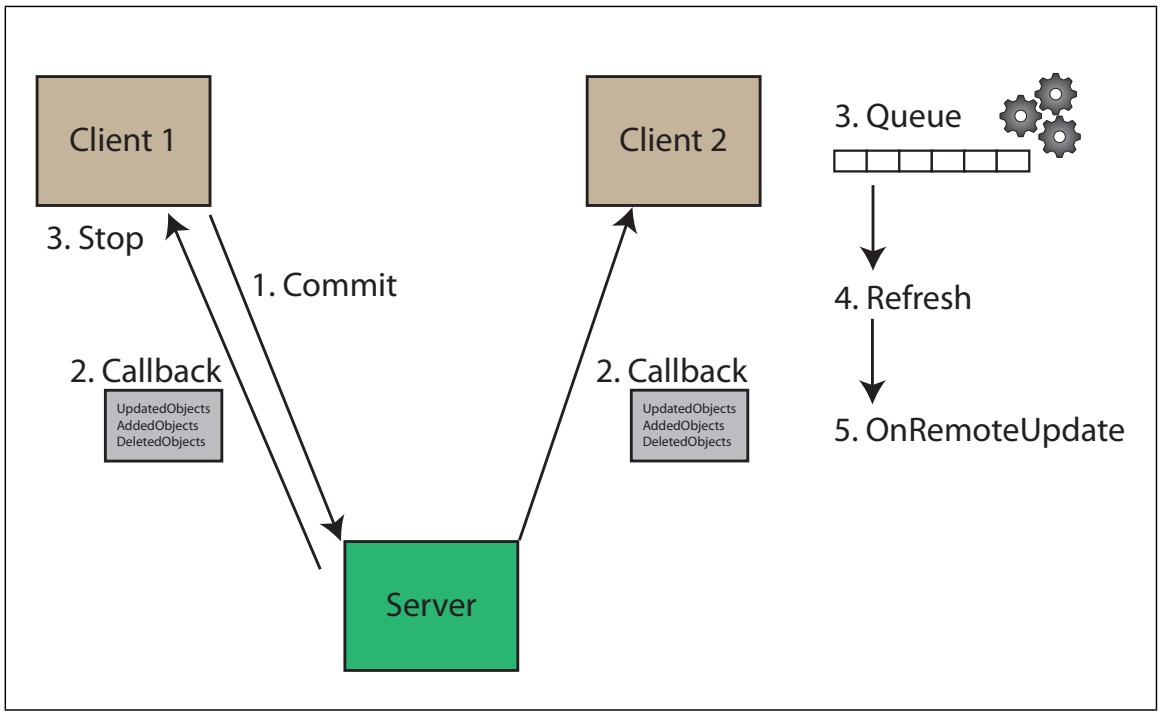

Abbildung 20: Ablauf der Synchronisierung

Synchronisierung der Visual Properties Die *Visual Properties* eines Objekts sind an *Dependency Properties* der View eines Objekts gebunden. Werden z.B. durch die Manipulation mittels Drag & Drop die X- und Y-Koordinaten der View verändert, so geht diese Änderung direkt an die Visual Properties weiter. Beim nächsten Commit werden diese Änderungen an den Visual Properties sofort an den Server übertragen und per Callback an alle Clients propagiert. Die Visual Properties

implementieren die oben erwähnte Schnittstelle *IRemoteUpdatable*, wodurch die Methode *OnRemoteUpdate* aufgerufen wird. In dieser Methode wird dann ein *PropertyChanged* Event gefeuert, woraufhin sich das Databinding zwischen den *Visual Properties* und den *Dependency Properties* der View erneuert. Bei einem genügend hohen Commit Intervall scheinen sich die Views auf den einzelnen Clients daher synchron zu bewegen. Abbildung 21 verdeutlicht den Ablauf bei der Synchronisation der *Visual Properties*.

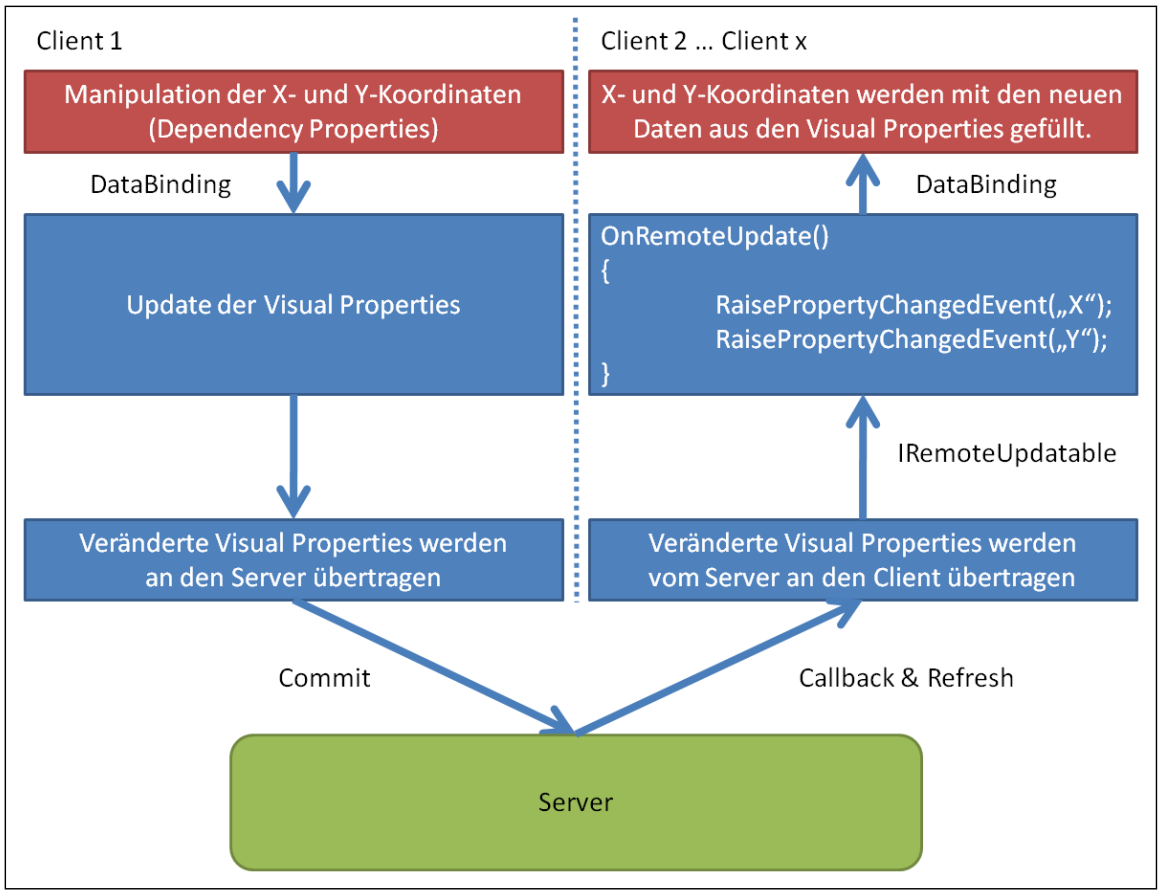

Abbildung 21: Ablauf der Synchronisierung von Visual Properties

4 Evaluation

In den vorangegangenen Kapiteln wurde ein detaillierter Einblick in das Datenmodell und -backend geliefert. Im nun vorliegenden Kapitel wird die Performance des Datenbackends evaluiert. Ein besonderer Schwerpunkt der Evaluation wird die Geschwindigkeit von Anfragen sein, da diese direkten Einfluss auf die User Experience haben. Ein etwas kleinerer Teil widmet sich der Analyse der Synchronisation.

Als Einstieg in die Evaluation wird nun ein Problem erläutert, das bei der Anwendung des Datenbackends in großen Datenräumen auftritt.

4.1 Große Datenräume

Bei einem Kurs an der Universität Konstanz sollten Studenten ein visuelles Suchsystem auf Basis des ZOIL-Frameworks entwickeln. Eine der vier Gruppen verwendete als Datengrundlage einen Ausschnitt der *ACM Digital Library*[20] mit ca. 120.000 Artikeln. Diese Artikel sollten als Informationsobjekte in die Datenbank geschrieben werden. Innerhalb des zu entwickelnden Suchsystems sollten dann Anfragen an diese Daten gestellt werden können.

Wie sich recht bald zeigte, war es nicht möglich, alle diese Artikel beim Starten des Programms in den Hauptspeicher des Clients zu laden. Die Studenten deaktivierten daher die Liste, die beim Start des Programms mit allen Objekten gefüllt wird. Mit einer spezifischen Anfrage an die Datenbank, wollten die Studenten dann eine Untermenge der Artikel in der Informationslandschaft anzeigen. Wie sich jedoch herausstellte, versuchte db4o diese Anfrage auf dem Client auszuführen. Es wurden daher wiederum alle Objekte aktiviert und der Hauptspeicher auf dem Client lief nach einiger Zeit voll, noch bevor die Anfrage ausgeführt wurde.

Der Grund für dieses Verhalten liegt in der Implementierung der *Dynamic Properties*. In der aktuellen Version des Datenbackends werden diese mit einem Dictionary implementiert, das einen *String*-Key auf ein *Object* abbildet. Anfragen, die auf ein Dictionary zugreifen, können von db4o nun aber nicht auf dem Server durchgeführt werden. Dies führt dazu, dass alle Objekte des Servers aktiviert werden müssen und die Anfrage auf dem Client ausgeführt wird.

Da es für die Implementierung von Dynamic Properties notwendig ist, dass ein Dictionary verwendet wird, kann das Problem nur mit einer etwas aufwändigeren Modellierung gelöst werden. Anstatt eines Dictionaries, das einen *String*-Key auf ein *Object* abbildet, sollte ein Dictionary zum Einsatz kommen, das einen *String*-Key auf eine spezielle Datenstruktur *IProperty* abbildet. Diese Datenstruktur wird in Listing 11 auf der nächsten Seite dargestellt.

[20] http://portal.acm.org/dl.cfm

```
public interface IProperty<T>
{
  String Key { get; set; }
  T Value { get; set; }
  IPersistable Owner { get; set; }
}

public class BaseProperty<T> : IProperty<T>
{
  String Key { get; set; }
  T Value { get; set; }
  IPersistable Owner { get; set; }
}
```

Listing 11: Neue Datenstruktur für Informationsobjekte: IProperty

Während die Methoden der Klasse *DInformationObject*, die auf das Dictionary zugreifen, auf die neue Datenstruktur umgestellt werden müssen, verändert sich bei der Implementierung der Dynamic Properties eines Informationsobjekts nichts. Listing 12 zeigt den Unterschied der aktuellen und der neuen Implementierung.

```
public class DInformationObject
{
  //Aktuelle Implementierung
  private ArrayDictionary4<String, Object> Data;

  public object GetValue(string key)
  {
    return Data[key];
  }

  public void SetValue(string key, object value)
  {
    Data[key] = value;
  }

  //Neue Implementierung
  private ArrayDictionary4<String, IProperty<object>> PropertyData;

  public object GetValue(string key)
  {
    return PropertyData[key].Value;
  }

  public void SetValue(string key, object value)
  {
```

```
    if (!PropertyData.ContainsKey(key))
      PropertyData[key] = new BaseProperty(key, value, this);
    else
      PropertyData[key].Value = value;
  }
}

//Aktuelle und neue Implementierung - keine Veränderung nötig
public class Movie : DInformationObject
{
  public String Title
  {
    get { return GetValue("title") as String;}
    set { SetValue("title", value);}
  }
}
```

Listing 12: Unterschiede zwischen aktueller und neuer Implementierung

Die Unterschiede zwischen aktueller und neuer Implementierung zeigen sich erst bei der Anfrageformulierung. Damit eine Anfrage auf dem Server ausgeführt wird, darf in ihr kein Zugriff auf das Dictionary erfolgen. Aus diesem Grund wird in der Anfrage nicht nach Informationsobjekten gesucht, sondern nur nach *Dynamic Properties*, also Instanzen des Typs *IProperty*. Über das Attribut *Owner* einer auf diese Art gefundenen *IProperty* Instanz gelangt man schlussendlich an das gesuchte Informationsobjekt. Listing 13 zeigt den Unterschied, bei der Formulierung von Anfragen mit der aktuellen und der neuen Implementierung.

```
//aktuelle LINQ-Query
var batmanMovies = from Movie m in Database.getInstance().Objects
                   where m.Title.Contains(``Batman``)
                   select m

//neue LINQ-Query
var batmanMovies = from IProperty<String> ip in Database.getInstance().Objects
                   where ip.Key=="title" && ip.Value.Contains(``Batman``)
                   select ip.Owner
```

Listing 13: Unterschied bei der Formulierung von LINQ Anfragen

Die neue Anfrage operiert in diesem Fall nur auf Objekten des Typs *IProperty* und greift nicht auf das Dictionary des Informationsobjekts zu. Es werden daher nur die Informationsobjekte aktiviert, deren Dynamic Properties der Anfrage entsprechen. Die Geschwindigkeit und der Speicherverbrauch einer solchen Anfrage verbessert sich dadurch enorm.

Zusätzlich besteht die Möglichkeit, die Datenstruktur *IProperty* auf dem Server zu indizieren, was sich vor allem bei großen Datenmengen deutlich auf die Anfragegeschwindigkeit auswirkt.

Für den Entwickler ist die Syntax der neuen Anfrage um einiges aufwändiger, als die aktuell verwendete. Im Moment wird daher an einer LINQ-Erweiterung gearbeitet, die Anfragen des alten Typs zur Laufzeit in Anfragen des neuen Typs umwandelt.

Im Kapitel 4.3 auf Seite 59 wird detailliert auf die Geschwindigkeit von Anfragen eingegangen. Unter anderem wird die gerade vorgestellte Neuimplementierung mit der aktuellen Implementierung verglichen.

4.2 Aktivierung von Informationsobjekten

Das im vorigen Kapitel beschriebene Problem wurde nur deswegen entdeckt, weil es nicht möglich war, alle Objekte, die sich in der Datenbank befanden, beim Starten des Programms zu aktivieren und in einer Liste zu speichern. Im Folgenden wird nun untersucht, ab welcher Menge von Objekten eine Aktivierung aller Objekte nicht mehr möglich oder sinnvoll ist, weil entweder der Hauptspeicher überläuft, oder die Aktivierung zu lange dauert.

Allen durchgeführten Untersuchungen lag folgender Versuchsaufbau zugrunde:

- Der Datenbankserver lief auf einem dedizierten Rechner im Netzwerk (∅ Ping: 4 ms).
- Der Rechner, auf dem die Untersuchungen durchgeführt wurden, war ein Laptop mit 3GB RAM und 2,0 GHz Dual-Core CPU
- Die Informationsobjekte wurden über ein spezielles Programm von einer XML-Datei in den Server eingelesen. In der XML-Datei war ein kleiner Ausschnitt der IMDB abgebildet, folglich repräsentierten die Informationsobjekte Filme aus der IMDB.
- Die Größe eines Informationsobjekts inkl. aller Properties betrug ca. 30 kByte.
- Die Datenbank wurde mit einer ansteigenden Zahl von Informationsobjekten im Bereich von 10 bis 50000 Objekten gefüllt. Bei 50000 Objekten lief zum ersten Mal der Speicher voll, daher wurde hier abgebrochen.
- Für jede Anzahl von Informationsobjekten (bis auf 50000) wurden 5 Durchgänge durchgeführt. Als Ergebnis wurde das arithmetische Mittel dieser 5 Durchgänge gewählt.

Die Abbildungen 22 auf der nächsten Seite und 23 auf der nächsten Seite zeigen die Dauer und den Arbeitsspeicherverbrauch bei der Aktivierung von immer mehr Objekten. Bei den Untersuchungen der Anfragen wurde festgestellt, dass es nicht notwendig ist, alle auf dem Server gespeicherten Objekte vom Typ *Object* zu aktivieren (Ansatz 1). Es reicht aus, nur die Informationsobjekte vom Typ *IPersistable* zu aktivieren (Ansatz 2). In den Abbildungen wird jeder der beiden Ansätze durch eine Kurve repräsentiert.

Die X- und Y-Achsen aller folgenden Abbildungen sind durchweg logarithmisch skaliert. Dies liegt daran, dass sowohl in X- als auch in Y-Richtung jeweils ein sehr großer Wertebereich abgedeckt werden muss, der durch “gestauchte“ Achsen besser darstellbar ist.

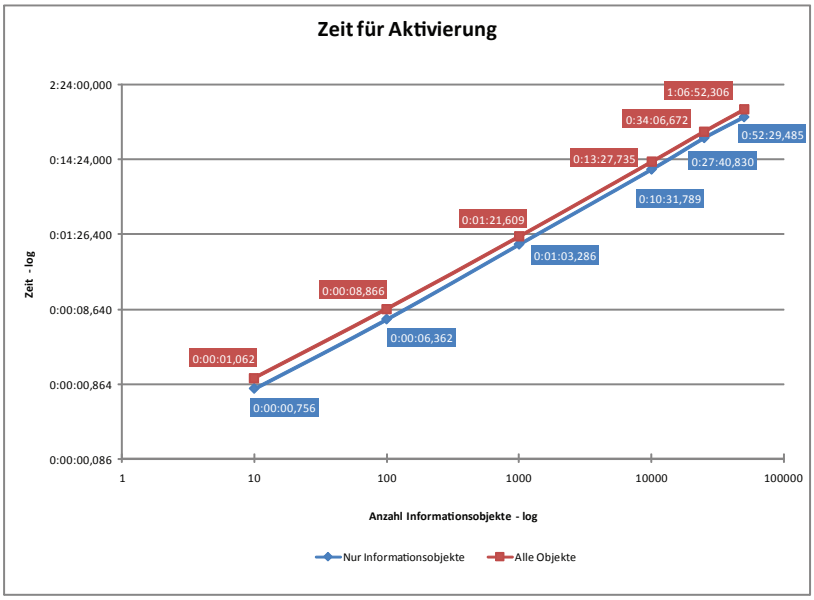

Abbildung 22: Dauer der Aktivierung von Objekten

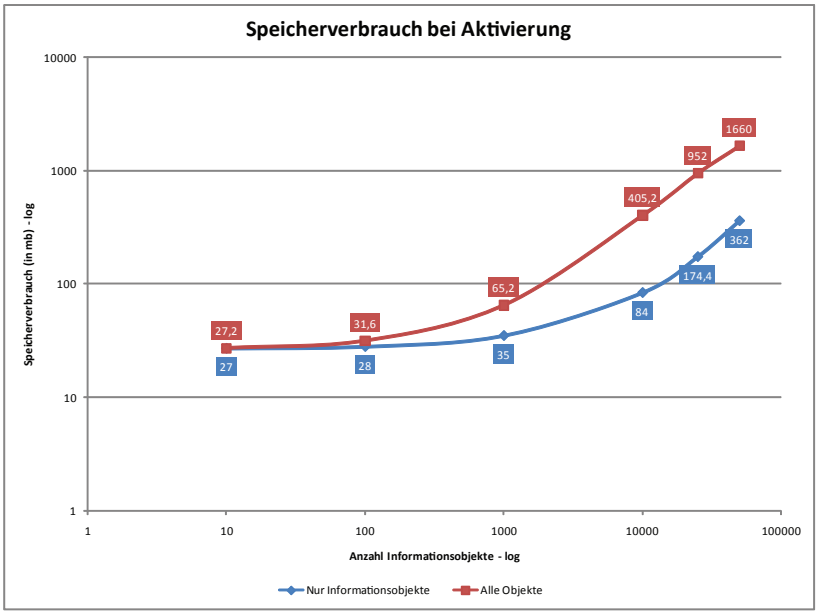

Abbildung 23: Arbeitsspeicherverbrauch beim Aktivieren von Objekten

Während bei der Dauer der Aktivierung kein großer Unterschied zwischen den beiden Ansätzen festzustellen ist, ist der Arbeitsspeicherbedarf bei Ansatz 1 ab ca. 10.000 Objekten deutlich höher, als bei Ansatz 2. Dies ist nachvollziehbar, werden doch beispielsweise bei 50.000 Informationsobjekten mit dem ersten Ansatz insgesamt ca. 880.000 “normale“ Objekte aktiviert und mit dem zweiten Ansatz nur 100.000. Betrachtet man nur den Arbeitsspeicherverbrauch ist mit dem ersten Ansatz bei 50.000 Objekten die Grenze erreicht, da der Client in diesem Fall nicht mehr Arbeitsspeicher adressieren kann. Selbst wenn man diese Grenze mit zusätzlichem Arbeitsspeicher erweitern könnte, ist die User Experience des Benutzers doch deutlich gestört, wenn er über eine Stunde warten muss, bis sein System bedienbar ist. Die Grenze für den Benutzer wird daher von der Dauer der Aktivierung vorgegeben. Nach subjektivem Empfinden des Autors liegt diese ungefähr zwischen einer und 5 Minuten bzw. zwischen 1000 bis 5000 Informationsobjekten dieser Größe. In den wenigsten Fällen wird eine solch große Menge von Objekten gleichzeitig auf der Informationslandschaft angezeigt. Dem Benutzer könnte beispielsweise eine Teilmenge der Objekte am Anfang präsentiert werden und der Rest der Objekte würde im Hintergrund von der Datenbank angefordert werden. In diesem Fall müsste der Benutzer dann allerdings mit Anfragen an den Informationsraum so lange warten, bis alle Objekte aktiviert wurden.

4.3 Geschwindigkeit von Anfragen

Eine der wichtigsten Kriterien für den Erfolg eines Suchsystems ist die Geschwindigkeit mit der eine Suchanfrage auf dem System durchgeführt wird. Im Datenbackend werden Suchanfragen idealerweise direkt auf dem Server durchgeführt und das Ergebnis dann an den Client gesendet. Wie wir bereits gesehen haben ist dies in der aktuellen Version des Datenbackends nicht möglich, da die Implementierung der Dynamic Properties nicht mit db4o harmoniert. Eine Übergangslösung ist daher die Aktivierung aller Objekte bzw. Informationsobjekte und das Ausführen der Anfrage auf dem Client. In Abbildung 24 auf der nächsten Seite ist die Geschwindigkeit einer einfachen Anfrage (siehe Listing 14) an den Client dargestellt. Wiederum wurde die Anzahl der Objekte kontinuierlich von 10 bis 50.000 erhöht.

```
var batmanMovies = from Movie m in Database.getInstance().ObjectCache.OfType<Movie>()
                   where m.Title == ''Batman''
                   select m;
```

Listing 14: Einfache LINQ-Anfrage, die auf dem Client durchgeführt wird

Selbst bei 50.000 Objekten liegt die Dauer der Anfrage noch im Bereich von unter 300ms. Da die Objekte im Hauptspeicher liegen, ist dies nicht weiter verwunderlich. Man darf jedoch nicht vergessen, dass die Aktivierung der Objekte schon sehr viel Zeit verbraucht. Die Aktivierung muss zwar nur einmal durchgeführt werden, ist aber nur für relativ kleine Informationsräume sinnvoll.

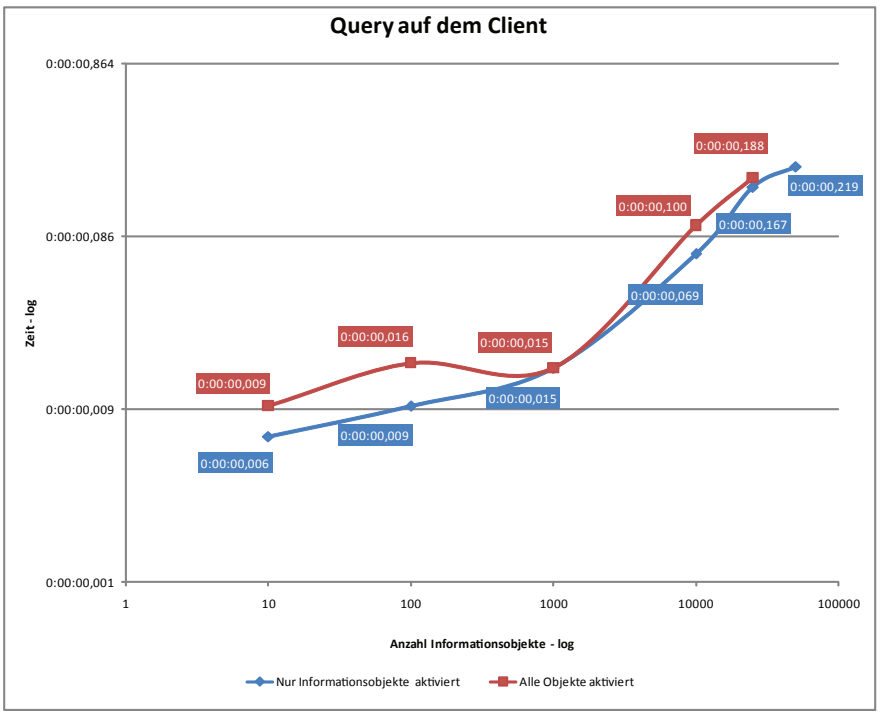

Abbildung 24: Anfragen, die auf dem Client ausgeführt werden

Aktiviert man nicht alle Objekte beim Starten der Anwendung, muss entweder die Anfrage direkt auf dem Server ausgeführt werden oder es werden bei jeder Anfrage alle Objekte aktiviert und die Anfrage wird implizit auch auf dem Client ausgeführt. Abbildung 26 auf der nächsten Seite vergleicht diese beiden Ansätze. Der erste Ansatz kann nur mit einer alternativen Implementierung der Dynamic Properties erreicht werden, und erfordert eine Änderung der Anfrage. Listing 15 und 16 zeigen die geänderte Anfrage im SODA bzw. LINQ Format.

```
IQuery query = Database.getInstance().Objects.Query();
query.Constrain(typeof(BaseProperty<Object>));
query.Descend("_key").Constrain("movie:title");
query.Descend("_value").Constrain("Batman");

var batmanMovies = query.Execute();
```

Listing 15: Einfache SODA-Anfrage, die auf dem Server durchgeführt wird

```
var batmanMovies = from BaseProperty<Object> prop in Database.getInstance().Objects
                   where prop._key == "movie:title" && m._value == ''Batman''
                   select prop.Owner;
```

Listing 16: Einfache LINQ-Anfrage mit der IProperty Implementierung

Bei der aktuellen Implementierung wächst die Dauer der Anfrage linear mit der Anzahl der Objekte, die sich auf dem Server befinden, da diese alle auf den Client übertragen werden müssen. Die Kurve ist daher nahezu identisch mit der Kurve die die Aktivierung der Objekte zeigt. Bei der neuen Implementierung ist die Anfragedauer hingegen wesentlich schneller. Selbst bei 50.000 Objekten dauert die Anfrage nicht länger als 3 Sekunden. Der Großteil der Anfragedauer fällt dabei auf die Aktivierung der Informationsobjekte, die der Anfrage entsprechen (siehe Abbildung 25). Sind diese Informationsobjekte auf Grund einer vorherigen Anfrage bereits im lokalen Cache des Clients, so liegt die Anfragedauer selbst bei 50.000 Objekten nur im Bereich von ca. einer Sekunde.

$$
\begin{aligned}
t_{query} &= t_{proc} + (n * t_{activation}) \\
t_{proc} &= \text{Zeit für das Ausführen der Anfrage auf dem Server} \\
t_{activation} &= \text{Zeit für die Aktivierung eines Informationsobjekts} \\
n &= \text{Anzahl der Informationsobjekte, die der Anfrage entsprechen}
\end{aligned}
$$

Abbildung 25: Dauer der Anfrage auf dem Server

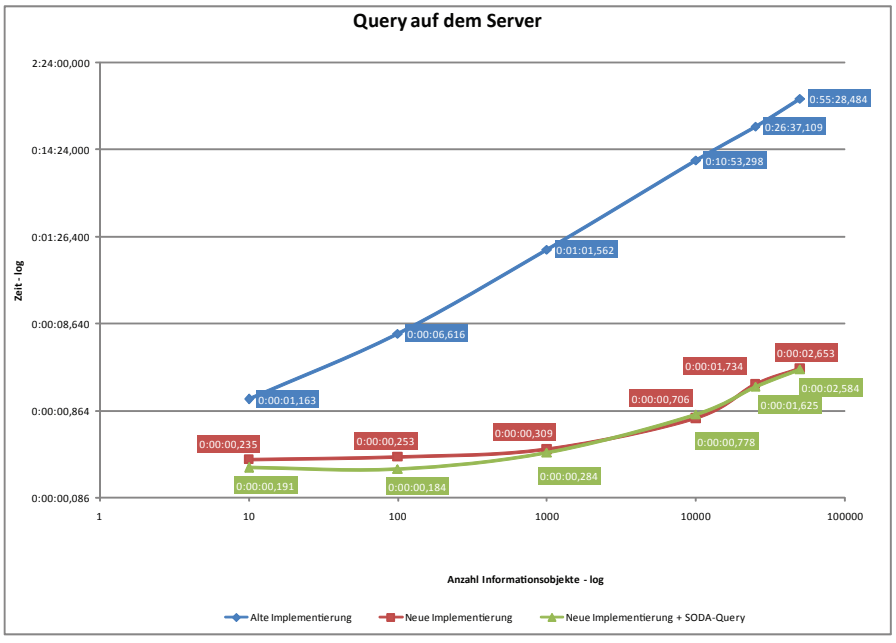

Abbildung 26: Anfragen, die auf dem Server ausgeführt werden

Abbildung 27 auf der nächsten Seite zeigt die Dauer und den Speicherverbrauch, wenn die Anfrage direkt auf der XML Quelldatei ausgeführt wird. Der Arbeitsspeicherverbrauch wächst auch hier

linear. Da die XML Repräsentation in C# etwas größer ist als die normale Objektrepräsentation, läuft hier bereits bei etwas mehr als 25.000 Objekten der Speicher voll.

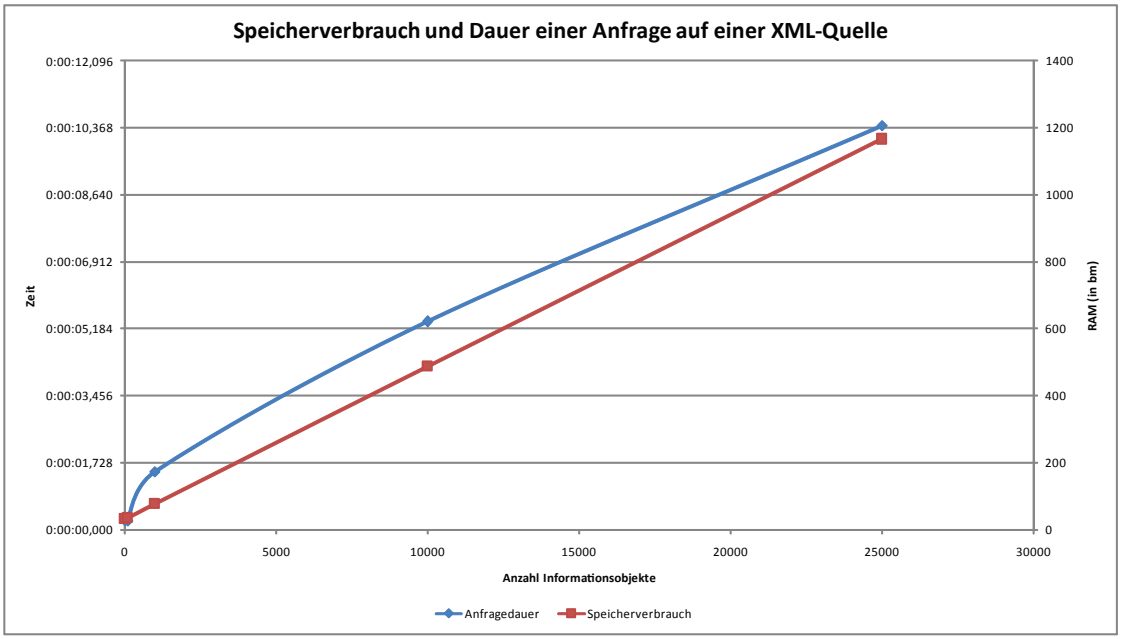

Abbildung 27: Anfragedauer und Speicherverbrauch bei einer XML Datenquelle

4.4 Analyse der Synchronisierung

Die Analyse der Synchronisierung ist eine hochkomplexe Angelegenheit. Mit jedem weiteren Client, der am Synchronisationsprozess teilnimmt, steigt die Komplexität um ein Vielfaches. An dieser Stelle soll die Thematik daher nur kurz angeschnitten werden. Die Optimierung der Synchronisierung sollte aber Bestandteil weiterer Forschungsthemen sein.

Jeder Client, der sich zum ZOIL-Server verbindet, sendet in einem regelmäßigen Intervall t_{commit} einen Commit-Aufruf an den Server. Auf jeden dieser Aufrufe reagiert der Server mit einem Callback. Dieser wird nicht nur an den Urheber des Commits, sondern an alle weiteren Clients gesendet. Bei 5 Clients, die alle zur gleichen Zeit einen Commit senden, kommen daher innerhalb kürzester Zeit jeweils 5 Callbacks an. Der Server muss dabei insgesamt 25 Callbacks an die Clients senden.

Die Round Trip Time (RTT), die für Callback und Commit benötigt wird, ist im einfachsten Fall kleiner als t_{commit}. In diesem Fall würde jeder Callback noch vor dem nächsten Commit vom Client bearbeitet, die Bearbeitung wäre in sich abgeschlossen. Um eine möglichst flüssige Synchronisie-

rung zu erhalten, liegt das ideale Commit-Intervall im Bereich von 100ms oder weniger. In vielen Fällen ist allerdings die RTT zwischen Client und Server bereits nahe bei 100ms oder darüber. In der Praxis ist es bei einem Intervall von 100ms daher eher selten der Fall, dass die Callbacks noch vor dem nächsten Commit beim Client auftauchen.

Abbildung 28 zeigt die beiden Fälle, die bei einer konstanten Differenz zwischen Commit und Callback auftreten können.

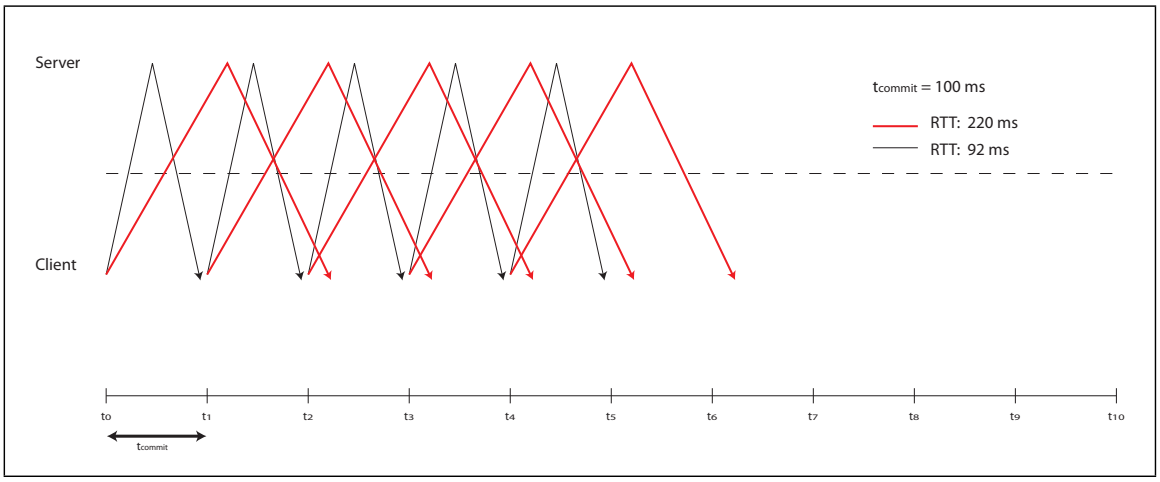

Abbildung 28: Der Commit-Callback Verlauf

Eine konstante Differenz zwischen Commit und Callback ist in der Praxis nicht vorhanden, da das Netzwerk Schwankungen unterliegt bzw. Pakete verloren gehen können, die dann neu gesendet werden müssen. Es kann daher vorkommen, dass Callbacks nicht in der richtigen Reihenfolge bei den Clients ankommen, da sie durch andere “überholt“ wurden.

Ein gängiges Synchronisationsszenario ist die Veränderung der X- und Y-Koordinaten der View eines Objekts. In diesem Fall ist das zu synchronisierende Objekt eine Instanz der Klasse Visual Properties. Die Änderung der Koordinaten wird dabei im Intervall t_{commit} an den Server gesendet. Wie bereits angesprochen kommen die Callbacks dieser Commits nicht immer in der richtigen Reihenfolge bei den anderen Clients an. Würde nun bei jedem Callback das veränderte Objekt mit gesendet, wäre es möglich, dass eine neue Änderung von einer älteren überschrieben wird. Die Callbacks beinhalten daher lediglich eine Referenz auf das veränderte Objekt und nicht das Objekt selbst. Bei der Verarbeitung des Callbacks wird immer die aktuelle Version aus der Datenbank angefordert. Es spielt daher keine Rolle, ob beispielsweise Callback #350 vor Callback #349 ankommt, da sowieso die aktuelle Version des Objekts angefordert wird. Nun kann es vorkommen, dass in relativ kurzer Zeit sehr viele Callbacks bei einem Client auftauchen, in denen immer das gleiche Objekt verändert wurde. Wenn das Objekt in diesem Fall bereits in der Warteschlange zur Bearbeitung steht, wird es hier nicht mehr eingefügt, da bei der Bearbeitung des Objekts die

aktuelle Version vom Server angefordert wird, die zusätzlichen Callbacks sind daher redundant.

Fehlerhafte Synchronisierung Die Implementierung der Synchronisation hat bereits mehrere Entwicklungszyklen durchlaufen, da immer wieder Fehler aufgetaucht sind. Das größte Problem ist das zeitweise Aussetzen der Synchronisation. Dieses Problem taucht in der aktuellen Version kaum noch auf. Dies liegt vor allem daran, dass die Verarbeitung der Callbacks auf dem Client auf mehrere Threads verteilt wird. Bei der Analyse des Synchronisationsprozesses im Rahmen dieser Arbeit wurde jedoch unter anderem festgestellt, dass die Implementierung des Timers, der dafür zuständig ist, einen Commit zu senden einen kleinen Fehler hat. Bei einem Commit-Intervall von 100ms wird erwartet, dass im Zeitraum von einer Sekunde insgesamt 10 Commits an die Datenbank gesendet werden. Das Versenden eines Commits hängt von der Anzahl der Daten ab, die sich in diesem Zeitraum verändert haben, liegt aber bei mindestens 100ms. Wenn nun der Timer im Intervall von 100ms versucht einen Commit zu senden, der vorherige Commit aber noch nicht abgeschlossen ist, überspringt der Timer diesen Commit einfach und wartet weitere 100ms. Dies wird so lange wiederholt bis der erste Commit abgeschlossen ist. In Wirklichkeit wird das Commit-Intervall von 100ms daher nicht richtig ausgenutzt, das effektive Commit-Intervall liegt bei ca. 200-400 ms. Eine Verbesserung dieser Implementierung wäre die Abarbeitung des Commits in einem eigenen Thread, so dass der Thread des Timers sofort wieder zur Verfügung steht und beim nächsten Ablauf des Intervalls wieder benutzt werden kann. Diese Verbesserung muss allerdings noch im Produktivbetrieb getestet werden, da noch nicht abzusehen ist, welche Probleme durch das asynchrone Absenden von Commits entstehen können.

5 Zusammenfassung und Ausblick

In dieser Arbeit wurde ein Datenmodell und -backend für das ZOIL Framework vorgestellt. Anforderungen, die aus dem *Personal Information Management*, dem ZOIL Paradigma und dem *Media Room* an ein flexibles und persistentes Datenmodell gestellt wurden, sind in der praktischen Implementierung umgesetzt worden. Das Datenbackend ist seit einiger Zeit integraler Bestandteil des ZOIL Frameworks.

Mit dem Datenmodell ist es möglich, sehr flexible und komplexe Informationsräume zu modellieren. Informationsobjekte sind zur Laufzeit mit beliebigen Daten erweiterbar und können so beispielsweise als Netzwerk oder Hierarchie dargestellt werden. Diese Informationsräume können dann auf der Informationslandschaft des ZOIL Frameworks abgebildet und exploriert werden.

Damit der Benutzer bei jeder Nutzung eine konsistente Informationslandschaft vorfindet, werden die visuellen Repräsentationen von Informationsobjekten mithilfe der *Visual Properties* persistiert.

Die *Visual Properties* sind auch mitverantwortlich für die Synchronisierung der visuellen Repräsentation eines Informationsobjekts. Die Synchronisierung mehrerer ZOIL-Clients ist dabei nicht auf der Pixelebene realisiert, sondern auf der Ebene von Informationsobjekten.

Die Evaluation bestimmter Teilaspekte des Datenbackends hat jedoch gezeigt, dass gerade für große Informationsräume noch sehr viel Potential für Verbesserungen vorhanden ist. Dies betrifft in erster Linie das Stellen von Anfragen an den Informationsraum.

Im Folgenden wird nun ein Ausblick auf bereits begonnene oder notwendige Arbeiten am Datenmodell und -backend gegeben.

Große Datenräume Die Evaluation hat gezeigt, dass das Verwalten sehr großer Datenräume im Datenbackend nicht effizient möglich war. Gerade bei sehr großen Informationsräumen bietet es sich an, diese auf einer extra Datenbank (z.B. SQL oder XML) zu belassen und das Datenbackend nur für die Persistenz der Objekte in der Informationslandschaft und für die Synchronisation zu nutzen. Anfragen die an den Informationsraum gestellt werden, würden von der jeweiligen Datenbank bearbeitet werden. Die Informationsobjekte, die dem Ergebnis einer Anfrage entsprechen, könnten dann im Datenbackend persistiert und in der Informationslandschaft visualisiert werden.

Erste Versuche in diese Richtung wurden bereits erfolgreich abgeschlossen. So haben beispielsweise Mitarbeiter der AG MCI die XML-Datenbank BaseX mit den Daten der IMDB gefüllt. Da BaseX alle textuellen Daten indiziert, ist die Dauer von Anfragen im Bereich von wenigen Hundert Millisekunden. Das Ergebnis dieser Anfragen wird in Informationsobjekte des Datenmodells umgewandelt. Diese werden dann in der Informationslandschaft angezeigt und im Datenbackend persistiert.

Eine weitere Erkenntnis der Evaluation war aber auch, dass durch eine neue Art der Implementierung der Dynamic Properties Anfragen in relativ kurzer Zeit auf dem Server ausgeführt werden können. Die Art und Weise wie diese Anfragen formuliert werden müssen, ist allerdings nicht entwicklerfreundlich. Es wird daher momentan an einer LINQ-Erweiterung gearbeitet, bei der normale LINQ-Anfragen in das effizientere SODA-Format konvertiert werden. Ist dies abgeschlossen, wäre es durchaus denkbar auch größere Datenräume mit dem Datenbackend effizient zu verwalten.

Pipe + Filter Eines der fünf ZOIL-Designprinzipien fordert sog. *Nested Information Visualizations*, also Visualisierungen, die ineinander geschachtelt werden können. Die Implementierung dieser Visualisierungshierarchie soll nach dem *Pipe + Filter* Design Pattern erfolgen. Bei der Anwendung dieses Patterns, dessen Ursprung in der Unix Programmierung liegt, werden die Objekte von der oberen Visualisierung an die unteren weitergeleitet. Jede der Visualisierungen ist in der Lage interaktiv Objekte auszufiltern. Diese tauchen dann in den darunter liegenden Visualisierungen nicht mehr auf.

Im Framework werden die Objekte den Visualisierungen in einer *Collection* übergeben. Für die Realisierung der *Pipe + Filter* Funktionalität müsste daher die Collection, die an die Visualisierung gebunden ist, die Filterfunktion auf der logischen Ebene übernehmen. Eine Hierarchie ineinander geschachtelter Visualisierungen könnte dann als Hierarchie von verbundenen Collections modelliert werden. So entsteht eine von den Visualisierungen unabhängige Ebene, die für die Objekte zuständig ist. Die konkreten Visualisierungen könnten sogar ausgetauscht werden.

Ein zusätzliches Feature, dass in diesem Fall mit in die Collection eingebaut werden soll, ist die *Sensitivität*. Mittels dieser kann jedem Objekt in einer Collection ein Zahlenwert zugewiesen werden. Ein Filter, der auf einer Collection angewendet wird, kann dann nicht nur ausschließlich binär arbeiten, sondern den Objekten unterschiedliche Zahlenwerte bzw. *Sensitivitäten* zuordnen. Diese *Sensitivitäten* können dann an *Visual Properties* wie Größe oder Transparenz gebunden werden. Die interaktive Änderung eines Filters wirkt sich somit direkt auf die visuelle Repräsentation eines Objekts aus.

An einer Implementierung dieser Funktionalität wird momentan gerade gearbeitet. Betroffen sind hierbei vor allem die *Personal Information Collections*.

Unabhängige Visualisierungen Die Stärke von Visualisierungen liegt in der Hervorhebung bestimmter Merkmale von Objekten. Bei einem Scatterplot können Objekte beispielsweise nach 2 Dimensionen angeordnet werden. In der aktuellen Version des Frameworks sind die Visualisierungen noch sehr stark abhängig vom Typ des Informationsobjekts. Beispielsweise wurde der Scatterplot nur für den Typ “Movie“ implementiert und optimiert. Ein anderer Typ von Informationsobjekt kann nicht dargestellt werden. Mit *unabhängigen Visualisierungen* soll es möglich gemacht werden,

dass eine Visualisierung nicht für einen bestimmten Typ von Informationsobjekt entworfen wird, sondern für heterogene Informationsobjekte. Die Visualisierung benötigt hierfür allerdings Metainformationen über die Metainformationen von Objekten. Sie muss zur Laufzeit für eine Menge von Objekten herausfinden können, welche Attribute alle Objekte gemeinsam haben und ob diese Attribute in der Visualisierung dargestellt werden können. Ein Scatterplot kann beispielsweise nur numerische Daten sinnvoll anzeigen, während eine Bar Chart Visualisierung auch mit kategorialen Daten zurecht kommt.

Um unabhängige Visualisierungen zu ermöglichen, müssten im Datenbackend in die Oberklasse der Informationsobjekte bestimmte Methoden eingebaut werden, die den Visualisierungen bei der Initialisierung helfen können.

Zusätzlich dazu müsste ein einheitlicher Metadatenstandard für die Modellierung der Informationsobjekte benutzt werden, damit sinnvolle Attributschnittmengen von Objekten erzeugt werden können.

Literatur

Angles, Renzo and Gutierrez, Claudio. Survey of graph database models. *ACM Comput. Surv.*, 40(1):1–39, 2008. ISSN 0360-0300. doi:http://doi.acm.org/10.1145/1322432.1322433.

Beaudouin-Lafon, Michel. Instrumental interaction: an interaction model for designing post-wimp user interfaces. In *CHI '00: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 446–453. ACM, New York, NY, USA, 2000. ISBN 1-58113-216-6. doi: http://doi.acm.org/10.1145/332040.332473.

Bederson, Ben B., Stead, Larry, and Hollan, James D. Pad++: advances in multiscale interfaces. In *CHI '94: Conference companion on Human factors in computing systems*, pages 315–316. ACM, New York, NY, USA, 1994. ISBN 0-89791-651-4. doi:http://doi.acm.org/10.1145/259963.260379.

Berners-Lee, Tim, Cailliau, Robert, Luotonen, Ari, Nielsen, Henrik Frystyk, and Secret, Arthur. The world-wide web. *Commun. ACM*, 37(8):76–82, 1994. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/179606.179671.

Berners-Lee, Tim, Hendler, James, and Lassila, Ora. The semantic web. *Scientific American*, 284(5):34–43, 2001.

Boardman, Richard. *Improving Tool Support for Personal Information Management*. Ph.D. thesis, Imperial College, London, 2004.

Bolt, R.A. Spatial data-management. Technical report, MIT, Architecture Machine Group, Cambridge, MA, 1979.

Buckland, Michael K. Information as thing. *JASIS*, 42(5):351–360, 1991.

Bush, Vannevar. As we may think. *The Atlantic Monthly*, 176(1):101–108, 1945.

Donelson, William C. Spatial management of information. In *SIGGRAPH '78: Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, pages 203–209. ACM, New York, NY, USA, 1978. doi:http://doi.acm.org/10.1145/800248.807391.

Dourish, Paul. *Where the action is: the foundations of embodied interaction*. MIT Press, Cambridge, MA, USA, 2001. ISBN 0-262-04196-0.

Dumais, Susan T. and Landauer, Thomas K. Using examples to describe categories. In *CHI '83: Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, pages 112–115. ACM, New York, NY, USA, 1983. ISBN 0-89791-121-0. doi:http://doi.acm.org/10.1145/800045.801592.

Engl, Andreas. A framework for an infnitely zoomable information landscape. 2008.

Farhoomand, Ali F. and Drury, Don H. Managerial information overload. *Commun. ACM*, 45(10):127–131, 2002. ISSN 0001-0782. doi:http://doi.acm.org/10.1145/570907.570909.

Fowler, M. Dealing with properties. 1997.

Freeman, Eric and Gelernter, David. Lifestreams: a storage model for personal data. *SIGMOD Rec.*, 25(1):80–86, 1996. ISSN 0163-5808. doi:http://doi.acm.org/10.1145/381854.381893.

Gamma, Erich, Helm, Richard, Johnson, Ralph, and Vlissides, John. *Design Patterns*. Addison-Wesley, Boston, MA, 1995.

Gerken, Jens. *Orientierung und Navigation in zoombaren Benutzerschnittstellen unter besonderer Berücksichtigung kognitions-psychologischer Erkenntnisse*. mastersthesis, University of Konstanz, 2006.

Hong, Lichan, Chi, Ed H., Budiu, Raluca, Pirolli, Peter, and Nelson, Les. Spartag.us: a low cost tagging system for foraging of web content. In *AVI '08: Proceedings of the working conference on Advanced visual interfaces*, pages 65–72. ACM, New York, NY, USA, 2008. ISBN 1-978-60558-141-5. doi:http://doi.acm.org/10.1145/1385569.1385582.

Indratmo, J. and Vassileva, J. A review of organizational structures of personal information management. *Journal of Digital Information*, 9(1), 2008.

Jacob, Robert J. K., Girouard, Audrey, Hirshfield, Leanne M., Horn, Michael S., Shaer, Orit, Solovey, Erin Treacy, and Zigelbaum, Jamie. Reality-based interaction: unifying the new generation of interaction styles. In *CHI '07: CHI '07 extended abstracts on Human factors in computing systems*, pages 2465–2470. ACM, New York, NY, USA, 2007. ISBN 978-1-59593-642-4. doi:http://doi.acm.org/10.1145/1240866.1241025.

Jacob, Robert J.K., Girouard, Audrey, Hirshfield, Leanne M., Horn, Michael S., Shaer, Orit, Solovey, Erin Treacy, and Zigelbaum, Jamie. Reality-based interaction: a framework for post-wimp interfaces. In *CHI '08: Proceeding of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*, pages 201–210. ACM, New York, NY, USA, 2008. ISBN 978-1-60558-011-1. doi:http://doi.acm.org/10.1145/1357054.1357089.

Jetter, Hans-Christian. *Informationsarchitektur und Informationsvisualisierung für die Post-WIMP Ära*. Master's thesis, University of Konstanz, 2007.

Jetter, Hans-Christian. Designing and implementing surface computing using the zoil paradigm. In *ACM TIS 2009*. 2009. Submitted but not accepted.

Jetter, Hans-Christian, Engl, Andreas, Schubert, Sören, and Reiterer, Harald. Zooming not zapping: Demonstrating the zoil user interface paradigm for itv applications. In *Adjunct Proceedings of European Interactive TV Conference, Salzburg, Austria, July 3-4, 2008, Jul 2008*. 2008a.

Jetter, Hans-Christian, Gerken, Jens, König, Werner A., Grün, Christian, and Reiterer, Harald. Hypergrid - accessing complex information spaces. In *HCI UK 2005: People and Computers XIX - The Bigger Picture, Proceedings of the 19th British HCI Group Annual Conference 2005*. Springer Verlag, 2005.

Jetter, Hans-Christian, König, Werner A., Gerken, Jens, and Reiterer, Harald. Zoil - a cross-platform user interface paradigm for personal information management. In *Personal Information Management 2008: The disappearing desktop (a CHI 2008 Workshop), April 5-6, 2008, Florence, Italy, Apr 2008*. 2008b.

Jones, William. *Keeping Found Things Found: The Study and Practice of Personal Information Management (Interactive Technologies) (Interactive Technologies)*. Morgan Kaufmann, 2007a. ISBN 0123708664.

Jones, William. *Personal Information Management*. University of Washington Press, 2007b. ISBN 0295987375.

Jäschke, Robert, Marinho, Leandro, Hotho, Andreas, Schmidt-Thieme, Lars, and Stumme, Gerd. Tag recommendations in social bookmarking systems. *AI Commun.*, 21(4):231–247, 2008. ISSN 0921-7126.

Kaptelinin, Victor. Umea: translating interaction histories into project contexts. In *CHI '03: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 353–360. ACM, New York, NY, USA, 2003. ISBN 1-58113-630-7. doi:http://doi.acm.org/10.1145/642611.642673.

Karger, David R. *Beyond the Desktop Metaphor: Designing Integrated Digital Work Environments*, chapter Haystack: Per-User Information Environments Based on Semistructured Data, pages 49–99. The MIT Press Cambridge, Massachusetts London, England, 2007.

Karger, David R. and Jones, William. Data unification in personal information management. *Commun. ACM*, 49(1):77–82, 2006. ISSN 0001-0782. doi:http://doi.acm.org/10.1145/1107458.1107496.

König, Werner A. *Referenzmodell und Machbarkeitsstudie für ein neues Zoomable User Interface Paradigma*. mastersthesis, University of Konstanz, 2006.

König, Werner A., Rädle, Roman, and Reiterer, Harald. Squidy: A zoomable design environment for natural user interfaces. In *CHI 2009 Extended Abstracts on Human Factors in Computing Systems, Work-In-Progress Session*. ACM Press, 2009.

Lansdale, M. The psychology of personal information management. *Applied Ergonomics*, 19(1):55–66, 1988. doi:10.1016/0003-6870(88)90199-8.

Lansdale, Mark and Edmonds, Ernest. Using memory for events in the design of personal filing systems. *Int. J. Man-Mach. Stud.*, 36(1):97–126, 1992. ISSN 0020-7373. doi:http://dx.doi.org/10.1016/0020-7373(92)90054-O.

Lyman, Peter and Varian, Hal R. How much information. Technical report, University of California, Berkeley, 2003. Retrieved from http://www.sims.berkeley.edu/how-much-info-2003 on 2009-07-11.

Malone, Thomas W. How do people organize their desks?: Implications for the design of office information systems. *ACM Trans. Inf. Syst.*, 1(1):99–112, 1983. ISSN 1046-8188. doi:http://doi.acm.org/10.1145/357423.357430.

Marguerie, Fabrice, Eichert, Steve, and Wooley, Jim. *LINQ in Action*. Manning Publications Co., 2008.

Miller, George A. Psychology and information. In *American Documentation*, volume 19, pages 286–289. Washington D.C., 1968.

Nardi, Bonnie A., Whittaker, Steve, Isaacs, Ellen, Creech, Mike, Johnson, Jeff, and Hainsworth, John. Integrating communication and information through contactmap. *Commun. ACM*, 45(4):89–95, 2002. ISSN 0001-0782. doi:http://doi.acm.org/10.1145/505248.505251.

Nelson, T. H. Complex information processing: a file structure for the complex, the changing and the indeterminate. In *Proceedings of the 1965 20th national conference*, pages 84–100. ACM, New York, NY, USA, 1965. doi:http://doi.acm.org/10.1145/800197.806036.

Pawson, Richard. *Naked Objects*. Ph.D. thesis, Trinity College, Dublin, Ireland, 2004.

Perlin, Ken and Fox, David. Pad: an alternative approach to the computer interface. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 57–64. ACM, New York, NY, USA, 1993. ISBN 0-89791-601-8. doi:http://doi.acm.org/10.1145/166117.166125.

Raskin, Jef. *The humane interface: new directions for designing interactive systems*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000. ISBN 0-201-37937-6.

Ringel, Meredith, Cutrell, Edward, Dumais, Susan T., and Horvitz, Eric. Milestones in time: The value of landmarks in retrieving information from personal stores. In *INTERACT*. 2003.

Shneiderman, Ben and Plaisant, Catherine. The future of graphic user interfaces: personal role managers. In *HCI '94: Proceedings of the conference on People and computers IX*, pages 3–8. Cambridge University Press, New York, NY, USA, 1994. ISBN 0-521-48557-6.

Surowiecki, James. *The Wisdom of Crowds*. Anchor, 2005. ISBN 0385721706.

Whittaker, Steve and Sidner, Candace. Email overload: exploring personal information management of email. In *CHI '96: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 276–283. ACM, New York, NY, USA, 1996. ISBN 0-89791-777-4. doi: http://doi.acm.org/10.1145/238386.238530.

Zöllner, Michael. Personal information management & interactive television: eine state-of-the-art analyse. Technical report, University of Konstanz, 2008.

Zöllner, Michael. Bachelor projekt: Datenbackend für zoil im kontext von pim. Technical report, University of Konstanz, 2009.

Abbildungsverzeichnis

Listings

Abkürzungsverzeichnis

LINQ	**L**anguage **IN**tegrated **Q**uery
MVVM	Model – View – ViewModel
OOUI	Object-Oriented User Interface
PIC	Personal Information Collection
PIM	Personal Information Management
PSI	Personal Space of Information
RDF	Resource Description Framework
SODA	Simple Object Database Access
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
WPF	Windows Presentation Foundation
XAML	E**X**tensible **A**pplication **M**arkup **L**anguage
ZOIL	Zoomable Object-oriented Information Landscape