Bauhaus-Universität Weimar
Fakultät Medien
Studiengang Mediensysteme

TIRA

Eine ressourcen-orientierte, übersetzungsfreie Webapplikation für Information-Retrieval-Experimente

Bachelorarbeit

Clement Benedict Welsch

1. Gutachter: Prof. Dr. Benno Maria Stein
Betreuer: Tim Gollub

Datum der Abgabe: 08. März 2010

Erklärung

Hiermit versichere ich, dass ich diese Arbeit selbständig und ohne Nutzung anderer als der angegebenen Quellen und Hilfsmittel verfasst habe. Alle Ausführungen, die dem Wortlaut oder dem Sinn nach anderen Werken entstammen, sind unter Angabe der Quelle als solche kenntlich gemacht. Selbiges gilt sinngemäß für Tabellen und Abbildungen. Diese Arbeit hat in gleicher oder ähnlicher Form noch nicht im Rahmen einer anderen Prüfung vorgelegen.

Weimar, den 8. März 2010

Kurzfassung

Eine große Problemstellungen bei der Entwicklung von Software ist der Umgang mit Komplexität und Fehlern. Durch die gezielte Auswahl der Systemkomponenten können aufwendige Vermittlungsbemühungen zwischen inkompatiblen Schnittstellen und Datenmodellen vermieden werden. Das reduziert die Komplexität und fördert somit die Qualität der Software.

Mit seinem Representional State Tranfer (ReST) zeigt Roy Thomas Fielding auf, dass eine Rückbesinnung auf die technologischen Grundlagen der Findung geeigneter Komponenten dienlich sein kann. Die Idee, ausschließlich das Hyper Text Transfer Protocoll (HTTP) und die Extensible Markup Language (XML) als technologische Grundlagen für verteilte Anwendungen einzusetzen, wird im Rahmen dieser Arbeit aufgegriffen und in einem konkreten Systementwurf umgesetzt.

Tira, eine Webapplikation für Experimente mit Algorithmen des Information Retrievals, wurde ursprünglich als klassische Drei-Schichten-Architektur konzipiert. Durch den Einsatz einer über HTTP erreichbaren nativen XML-Datenbank, der Formulierung des Datenmodells in XML und der Anwendung der Prinzipien von ReST wird im Rahmen dieser Arbeit eine exemplarische Transition zu einer Architektur für Tira vollzogen, die sich durch Flexibilität, Einfachheit und Eleganz auszeichnet.

Inhaltsverzeichnis

1. Einleitung 1

2. Motivation 3

3. Anforderungen 6

I. Modellierung und Speicherung der Daten 11

4. XML-Datenbanken 12

4.1. Evolution der Datenbanken 12
4.2. Differenzierung von XML-Datenbanken 14
4.3. Nutzungskriterien 16
4.4. Verschiedene Systeme im Überblick 17
4.5. Anfragesprachen 19

5. eXist 22

5.1. Betrieb 22
5.2. Protokolle 22
5.3. Indexierung 23
5.4. Anfragen 27
5.5. Programmierung 28

6. Datenmodell 29

6.1. XML als Datenmodell 30
6.2. Anforderungen 32
6.3. Adhoc-Struktur 34
6.4. Modellbildung für Experimentdesigns 35
6.5. Ressüme 42

II. Entwurf einer ressourcen-orientierten Architektur 44

7. ReST 45
7.1. Prinzipien . . . 46
7.2. ReST-konforme Architekturen . . . 48
7.3. Resümee . . . 48

8. Architektur 51
8.1. Drei-Schichten-Architektur . . . 51
8.2. Architektur in der Metamorphose . . . 53
8.3. Entwurf . . . 55
8.4. Resümee . . . 59

9. Spezifikation 61
9.1. Datenbank . . . 61
9.2. Dispatcher . . . 64
9.3. Worker . . . 66

10. Umfeld 67

11. Resümee 69

A. Schemata 75

Abbildungsverzeichnis

1. Usecases von Tira 6
2. Aktivitäten beim Durchführen eines Experimentes 7
3. Aktivitäten im Zuge einer Auswertung von Ergebnissen 7
4. Aktivitäten um Auswertungsergebnisse zu publizieren 8
5. Evolution der Datenbanken 13
6. Ontologie der XML-Datenbanken 15
7. W3C-Sprachfamilie für die Verarbeitung von XML 19
8. Exemplarisches Modell für Experimentdesigns 36
9. Pipeline für die Erzeugung von Design-Instanzen aus dem Modell 38
10. Ausprägungen der Drei-Schichten-Architektur 52
11. Architekturen im Vergleich 53
12. Komponenten im Vergleich 54
13. Komponenten und Datenfluss zwischen Schicht eins und drei. . 56
14. Komponenten und Datenfluss zwischen Schicht zwei und drei. . 57
15. Sequenzdiagramm 59

Tabellenverzeichnis

1. Kriterien für den Einsatz von XML-Datenbanken 16
2. Übersicht verschiedener XML-Datenbanken 17
3. Zusammenhang der Modellebenen 39
4. Die Bedeutungen der HTTP-Verben im ReST-Kontext 47

Quelltextverzeichnis

1. Beispiel für einen XPath-Ausdruck 20
2. Äquivalentes FLWOR-Statement 21
3. Verkürztes äquivalentes FLWOR-Statement 21
4. Exemplarischer XPath-Ausdruck 24
5. Vergleichsoperationen und -funktionen in XPath 25
6. Exemplarischer XPath-Ausdruck 25
7. Beispiel einer Indexkonfiguration 25
8. Abstraktes Modell in EBNF 40
9. Datenmodell in EBNF 41
10. Schema für Modell-Fragmente 75
11. Schema für Instanzdokumente. 77

1. Einleitung

Das »Testbed for Information Retrieval Algorithms«, im weiteren kurz Tira, wird seit dem Jahr 2007 an der Professur für Webtechnologien und Informationssysteme der Bauhaus-Universität Weimar entwickelt. Ziel des Projektes ist es Vergleich- und Nachvollziehbarkeit von Forschungsergebnissen aus dem Feld des Information Retrievals zu gewährleisten. Dies soll durch die Schaffung einer einheitlichen Plattform erreicht werden, welche es Wissenschaftlern weltweit erlaubt ihre Experimente in einer standardisierten Testumgebung durchzuführen. Von der interaktiven Modellierung ganzer Serien von Experimenten und ihrer automatisierten Durchführung bis hin zur Verwaltung und Auswertung der Ergebnisse soll der gesamte Arbeitsprozess unterstützt werden. Neben der Bereitstellung von Standardkollektionen und -verfahren ist die Möglichkeit der Erweiterung durch eigene Implementierungen vorgesehen.

Tira wurde als klassische Webapplikation in einer Drei-Schichten-Architektur konzipiert. Die klassische Herangehensweise bei der Umsetzung dieser Architektur führt in der Regel zu der Verwendung von mindestens zwei, bei Webanwendungen jedoch meist drei unterschiedlichen Datenmodellen. Dieser Sachverhalt ist eine unmittelbare Folge der Anwendung etablierter Technologien für die Implementierung der jeweiligen Schichten. Um dennoch die Kommunikation der Schichten untereinander zu ermöglichen, werden zusätzlich Vermittlungsschichten benötigt, die unterschiedliche Modelle ineinander überführen.

Die vorliegende Arbeit beschäftigt sich mit der Frage, wie durch die Anwendung der selben Technologie auf unterschiedlichen Schichten Transformationen zwischen inkompatiblen Modellen vermieden werden können und somit die Komplexität der Anwendung reduziert werden kann. Zu diesem Zweck werden verschiedene neuere Entwicklungen im Feld der Webanwendungen in

Augenschein genommen und hinsichtlich einer pragmatischen Umsetzung in Tira beleuchtet. Die Dissertation von Roy Thomas Fielding [Fie00], in der er den ReST-Archtitekturstil entwirft, und Dan McCrearys XRX-Architektur [McC08], die auf Fieldings Arbeit beruht und eine ressourcen-orientierte ReST-konforme Architektur beschreibt, sind die maßgeblichen Inspirationsquellen für diese Arbeit. Auf diesen fußend wird eine neue Architektur für Tira vorgestellt, die, nach einer Analyse der Struktur der Daten, mit nur einem Datenmodell und ohne Modelltransformationen sowie nur einer – minimalen – Schnittstelle zwischen ihren Komponenten auskommt.

2. Motivation

Spätestens seit Anbeginn des Zeitalters der Industrialisierung neigen Menschen dazu, Probleme oder Sachverhalte, die sich ihnen als solche darstellen, durch Technologie lösen zu wollen. Jedoch:

> »Es hat den Anschein, als ob, wie in der Wissenschaft, so auch in der Technik, mit jedem Problem, das gelöst wird, neue Probleme entstehen. Es scheint, als ob der Fortschritt mehr in dem Herausarbeiten neuer Probleme als in dem Vermindern der Probleme bestände.«[1]

Jede Technologie, die von Menschenhand erschaffen wurde, ein bestehendes Problem zu lösen, erschafft eine Fülle neuer Probleme. Das liegt in der Natur der Sache, denn Menschen machen Fehler und Fehler erzeugen Probleme. Häufig werden die Fehler jedoch nicht als solche erkannt und somit korrigiert, sondern es wird vielmehr ausschließlich das Problem wahrgenommen, welches, wie wir bereits ahnen, wiederum durch Technologie zu lösen versucht wird. Dieses Vorgehen, nämlich das Lösen von durch Technologie erzeugten Problemen, durch das Anwenden neuer Technologie, die ihrerseits wieder Probleme aufwirft, führt unweigerlich zu technologischen Systemen, deren Komplexität exponentiell anwächst und somit in absehbarer Zeit nicht mehr handhabbar sein wird. Kurz, Fehler erzeugen Komplexität, jedenfalls unter der Prämisse, dass auch Fehlentscheidungen als Fehler anzusehen sind.

„Das Grundproblem", so Andrew Tanenbaum, „besteht darin, dass Software Bugs enthält, und je mehr Software es gibt, desto mehr Bugs gibt es. Diverse Studien belegen, dass große Produktivsysteme zwischen einem und zehn Fehler

[1] Julius Goldstein (1912)

pro tausend Zeilen Code aufweisen.“[Tan09] Diese umgekehrte Aussage, dass Komplexität Fehler erzeugt, ist populärer und weithin akzeptiert, auch wenn sich immer wieder einige gallische Dörfer[2] finden, die unerbittlichen Widerstand leisten, auch und gerade in der Informatik. Prinzipien wie etwa *KISS*[3] sind althergebracht, geradezu mythisch und werden dennoch, ob ihrer banalen Aussage, selten ernst genommen.

In neuerer Zeit spricht man von Softwarequalität – ein sehr allgemeiner Begriff, der äußerst unterschiedliche Positionen in sich vereint. Zum einen die externe Perspektive des Nutzers, zum anderen eine zwiegespaltene interne Sicht auf die Dinge. Während das Augenmerk des Entwicklers ebenso wie das des Nutzers auf die Qualität des tatsächlichen Produktes gerichtet ist, ist die Rolle des Managers konträr. Sein Interesse gilt allein der Qualität des Entwicklungsprozesses – vor allem deren Messbarkeit. [FS04] So unterschiedlich diese Positionen auch sein mögen – jede von ihnen hat einen sehr individuellen Satz von Zielsetzungen – so kann man doch eine Gemeinsamkeit feststellen: *Einfachheit*.

Um komplexe Systeme trotz ihrer Komplexität handhaben und das Gebot der Einfachheit befolgen zu können, wurden Prinzipien wie das Untergliedern großer Systeme in Module, das Auftrennen nach unterschiedlichen Aspekten oder etwa die möglichst allgemeine Formulierung grundlegender Komponenten ersonnen. Insbesondere letzterer Punkt ist wichtig, denn eine Komponente, die allgemein gehalten ist, kann häufig wiederverwendet werden. Es geht hier also schlicht darum, das berühmte Rad nicht immer wieder neu zu erfinden. Genau das ist aber bei HTTP, dem Protokoll des Web, geschehen. Protokoll wurde über Protokoll gestapelt und die wunderbare Einfachheit und Vielseitigkeit unter dicken Abstraktionsschichten – einer weiteren Strategie, um Komplexität eigentlich zu vermindern – verschüttet. [RR07]

[2] Es wäre zutreffender von Metropolen zu sprechen
[3] Keep it simple, stupid!

Wie bereits in der Einleitung angeklungen, verwenden klassische Webapplikationen etablierte Technologien – eine vollkommen richtige Entscheidung, denn:

> »A complex system that works is invariably found to have evolved from a simple system that worked. The inverse proposition also appears to be true: A complex system designed from scratch never works and cannot be made to work. You have to start over, beginning with a working simple system.«[4] [Gal77]

Diese etablierten Technologien jedoch, und das wurde lange ignoriert, bedienen sich intern, jede für sich, zueinander inkompatibler Datenmodelle und sie sind, das sollte auch berücksichtigt werden, nicht unbedingt das, was man unter „einfach“ verstehen würde. Dieser Umstand führt dazu, dass zusätzliche Vermittlungsschichten eingeführt werden müssen. So entsteht unnötigerweise mehr Software und damit, wie zuvor bereits besprochen, mehr Fehler.

Beide angerissenen Probleme beruhen auf zwei gemeinsamen Ursachen: Erstens Fehlentscheidungen, die überflüssige Komplexität erzeugen, und zweitens dem Unvermögen, diese Fehlentscheidungen zu erkennen, denn Fehler, wie eingangs festgestellt, werden häufig nicht korrigiert, sondern durch neue Technologie kaschiert.

Diese Arbeit stellt keine neue Technologie vor. Sie verfolgt ein möglicherweise neues, weil wiederentdecktes Prinzip: Den Schritt zurück. Dieser Schritt ermöglicht es, sich einen Überblick zu verschaffen und so auf das Wesentliche zu reduzieren.

> »Perfektion ist nicht dann erreicht, wenn man nichts mehr hinzuzufügen hat, sondern wenn man nichts mehr weglassen kann.«[5]

Ganz im Sinne Antoine de Saint-Exupérys fühlt sich der im Rahmen dieser Arbeit erarbeitete Entwurf dem „Weniger“ verpflichtet.

[4] Gall's Law, John Gall in [Gal77]
[5] Antoine de Saint-Exupéry

3. Anforderungen

Die Anforderungen, die an ein System zu richten sind, ergeben sich zunächst aus so genannten Usecases, den Nutzungsszenarien. Usecases stellen hierbei noch nicht die Frage danach, *wie* ein System funktionieren soll, sondern *was* der Nutzer mit ihm zu tun gedenkt. Abbildung 1 zeigt diese Wünsche des Nutzers in UML-Notation. Um Erweiterungen hinzuzufügen wird es zunächst der Inanspruchnahme eines Entwicklers bedürfen, solange die Spezifikation einer zweckdienlichen Modul-Schnittstelle noch aussteht.

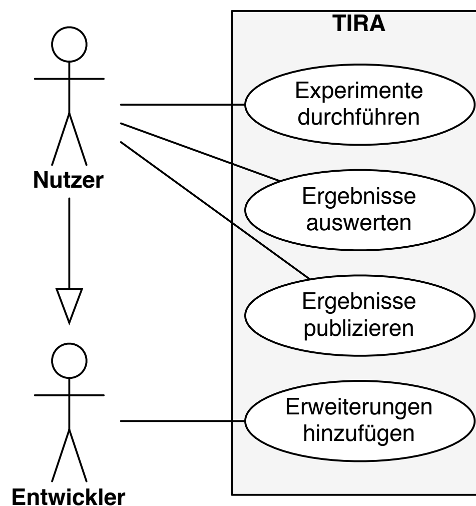

Abbildung 1: Usecases von TIRA

In einem zweiten Schritt wollen wir die einzelnen Nutzungsszenarien genauer inspizieren und uns veranschaulichen welche Aktivitäten in dem System zu verrichten sind, um die einzelnen Vorhaben des Nutzers zu realisieren. Die Frage, *wie* das System arbeitet, skizzieren die Abbildungen 2, 3 und 4. Der Aspekt der Erweiterbarkeit wurde hier vernachlässigt, da dieser ohnehin, da

dieser ohnehin aus den bereits genannten Gründen einen Sonderfall darstellt. Die Abbildungen 3 und 4 stellen einen Vorgriff auf zukünftige Weiterentwicklungen dar. Auf diese beiden darin beschriebenen Aktivitäten wird im Rahmen dieser Arbeit nicht näher eingegangen. Aufgeführt sind sie dennoch, da sie die prinzipielle Ähnlichkeit der Abläufe veranschaulichen.

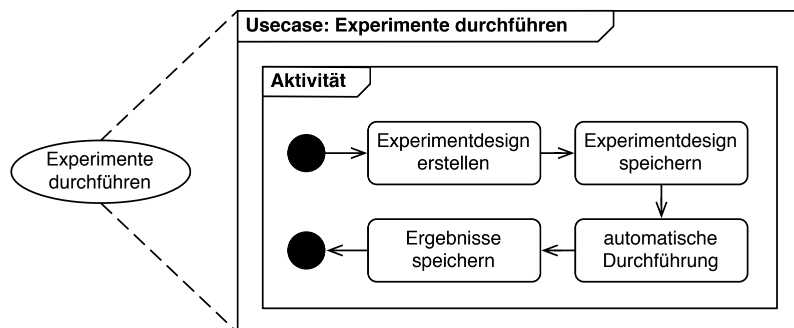

Abbildung 2: Aktivitäten beim Durchführen eines Experimentes

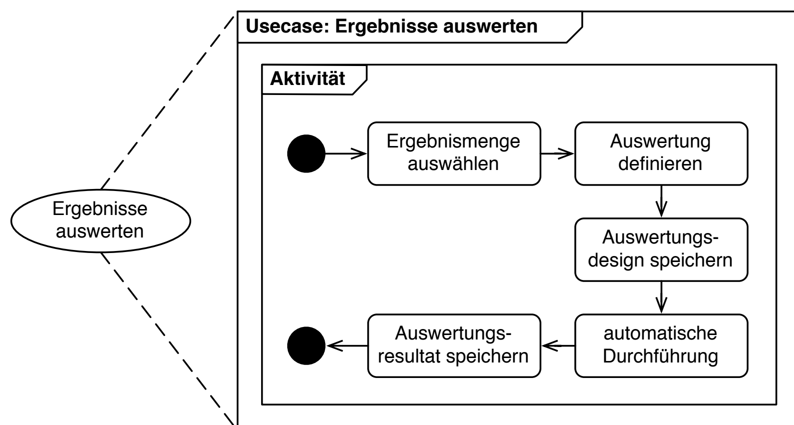

Abbildung 3: Aktivitäten im Zuge einer Auswertung von Ergebnissen

Die bereits in der Einleitung angesprochenen Zielsetzungen von Tira lassen sich fließend in konkrete Anforderungen an das zu implementierende System überführen:

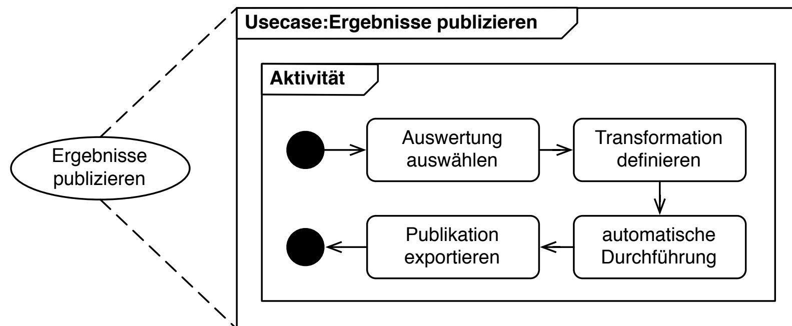

Abbildung 4: Aktivitäten um Auswertungsergebnisse zu publizieren

- Eine Implementierung als *Webapplikation* gewährleistet **weltweite Verfügbarkeit**.
- Um eine **automatische Durchführung** von Experimenten zu ermöglichen, wird die Applikationslogik als ein *WebService* bereitgestellt, der vom Client durch valide Design-Konfigurationen angesprochen werden kann und die Durchführung im weiteren autonom bewerkstelligt.
- Der Wunsch **Experimentserien** möglichst einfach und durch nur ein Design-Dokument aufsetzen zu können, stellt sich vor allem als eine Anforderung an das Datenmodell dar. Dass dieses Datenmodell natürlich durch die Applikation verarbeitet werden muss, steht hierbei zunächst auf einem anderen Blatt. Um dieser Anforderung genügen zu können soll das Datenmodell in einem einzelnen Design-Dokument *Variationen* über einzelne Parameter einbetten können. Diese Variationen müssen vor der Verarbeitung wiederum in einzelne Design-Dokumente expandiert werden. Dieses Vorgehen birgt zwei Vorteile in sich. Einerseits minimiert es den Arbeitsaufwand für den Nutzer, andererseits optimiert es die Client-Server-Kommunikation in der Form, dass eine Vielzahl von Request-Response-Zyklen zusammengefasst werden können.
- Eine **interaktive Modellierung** versteht sich als Feedback seitens des Systems. Durch eine Nutzerauswahl sollen die weiterführenden Desi-

gnmöglichkeiten plausibel eingegrenzt und entsprechend zur weiteren Auswahl bereitgestellt werden. Fehleingaben, soweit möglich, müssen dem Nutzer zur Kenntnis gebracht werden. Realisiert wird dies durch *dynamische Formulare*. Dadurch wird sichergestellt, dass, sobald ein Design vom Client zur Weiterverarbeitung an den Server übermittelt wird, keine Fehler mehr enthalten sein können.

- Ebenso wie das Erstellen von Experimenten soll auch eine **einfache Verwaltung** der selbigen gewährleistet werden. Zu diesem Zweck dient die selbe Oberfläche wie für das Experimentdesign; sie wird zusätzlich mit der Möglichkeit ausgestattet, neben der Konfiguration eines Designs simultan auch Datenbankanfragen zu generieren. Die gleiche Benutzeroberfläche ist also zugleich *Eingabe-* als auch *Anfragemaske*.

- Eine **flexible, interaktive Auswertung** der Ergebnisse wird in zukünftigen Weiterentwicklungen von TIRA Einzug halten. Hierfür erscheint es sinnvoll einen zweiten WebService bereitzustellen. Dieser sollte verschiedene zweckdienliche statistische Methoden implementieren. Wie er zu implementieren ist, ob als JAVA-Servlet, wie derzeit der Experimentier-WebService, als XQuery-Servlet, was den Vorteil direkter Verarbeitung der Ergebnisdaten hätte, oder aber über die Anbindung von prominenter Statistik-Software wie etwa SPSS, könnte Gegenstand einer anderen Arbeit sein. Denkbar wäre auch eine Bereitstellung von Exportschnittstellen über *XSL-Transformationen*, um somit eine externe Auswertung zu ermöglichen.

- Sowohl die Experimentier- als auch die Auswertungsmöglichkeiten sollten **erweiterbar** gehalten werden. Wie diese Erweiterbarkeit zu realisieren ist, ist ebenfalls nicht Gegenstand dieser Arbeit. Festgehalten werden soll an dieser Stelle jedoch bereits, dass jede Erweiterung sofortige Auswirkungen auf die Benutzerschnittstelle zeitigen wird. Die Datenbank wird mit dieser Problemstellung nicht konfrontiert sein. In jedem Fall stellt sich jedoch die Frage, wie eine *Plugin-* oder *Modulschnittstelle* zu

entwerfen ist, um auch externen Anwendern eine Nutzung ihrer Implementierungen zu ermöglichen.

Darüber hinaus werden selbstverständlich einige implizite Anforderungen an das System gerichtet sein – Anforderungen, die den prinzipiellen Softwareentwurf betreffen, und da wären solche wie:

- Modularität
- Erweiterbarkeit
- Separation der Aspekte
- Interoperabilität
- Benutzerfreundlichkeit

Im weiteren Verlauf der Arbeit werden die Konzepte und Techniken eingeführt, die das Fundament eines Systementwurfes bilden, der diese Anforderungen in Einklang bringen soll – nicht vergessend, das dieser Entwurf einer weiteren, nicht ohne Grund in der Motivation formulierten Anforderung gerecht werden soll: der *Einfachheit*.

Teil I.
Modellierung und Speicherung der Daten

4. XML-Datenbanken

4.1. Evolution der Datenbanken

XML-Datenbanken sind eine recht junge, wenn auch nicht mehr brandneue Strömung im Metier der Datenbanken. Das Aufkommen solcher Datenbanken, die sich auf die Speicherung und Abfrage von XML-Daten spezialisiert haben, ist eine folgerichtige Entwicklung im Zuge der zunehmenden Verbreitung und Durchsetzung von in XML formulierten Formaten – insbesondere im Web. Dabei stehen XML-Datenbanken in einer langen Tradition der Anpassung von Datenbanktechnologie an externe Bedürfnisse. Abbildung 5 gibt hierzu einen groben Überblick, ohne Anspruch auf Vollständigkeit.

Die wohl einzigen Datenbanken, die nur aus dem Bedürfnis nach einer Datenbank heraus sich selbst genügen mussten, waren die relationalen Datenbanken, welche seit den Siebzigern des vergangen Jahrhunderts entwickelt und erforscht werden. Diese Systeme sind in dieser Zeit in einem Maße ausgereift und theoretisch durchdrungen worden, dass sie noch bis heute die Etabliertesten sind – Und das, obwohl sich das relationale Model in keiner Programmiersprache wiederfindet.

Ein Jahrzehnt später, infolge der zunehmenden Beliebtheit des objektorientierten Programmiermodells, traten objektorientierte Datenbanken an, die sprachlichen Gräben zwischen der Welt der Anwendungsprogramme und der Datenhaltung einzuebnen – Mit mäßigem Erfolg. Objektorientierte Datenbanken fristen bis zum heutigen Tag ein Nischendasein. Gleiches gilt für objekt-relationale Datenbanken, Systeme, die intern relationale Datenbanken sind, nach außen jedoch eine objektorientierte Schnittstelle bieten. In den meisten Anwendungen

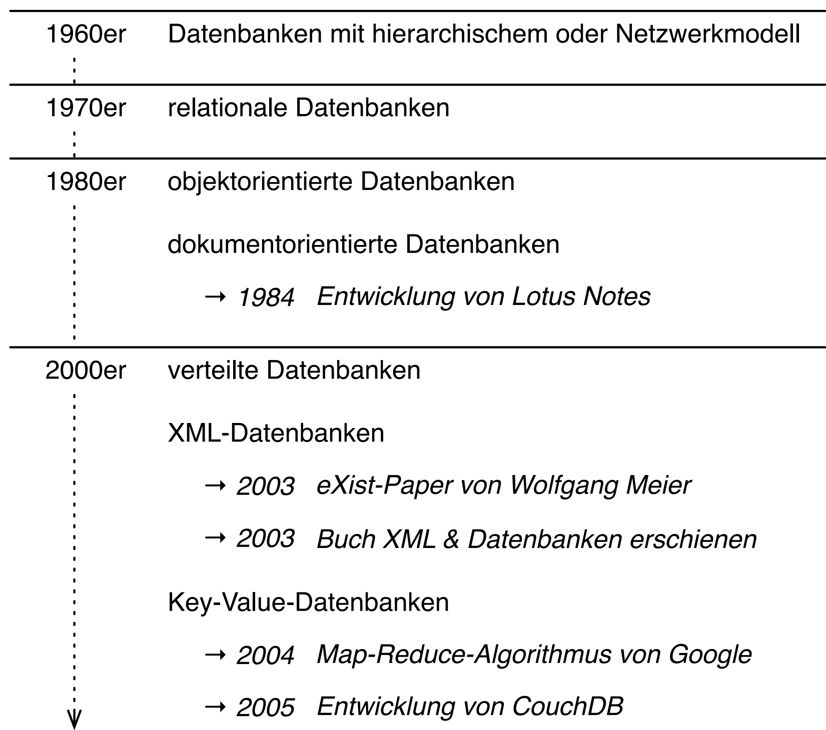

Abbildung 5: Evolution der Datenbanken

wird bis zum heutigen Tage separate Adapterlogik eingesetzt, um relationale Tabellen in Objekte zu wandeln, sogenannte OR-Mapper.

Lange boten sich keine Alternativen zu dem aufwendigen Vorgehen des OR-Mappings, bis, im gerade zu Neige gehenden Jahrzehnt, eine Vielzahl von Neu- und Wiederentdeckungen auf die Bildfläche traten. So beförderte Googles Präsentation seines Map-Reduce-Algorithmus die Entstehung verteilter Key-Value-Datenbanken, die sich, durch die Nutzung von JavaScript als Anfragesprache, besonders für das Web anbieten.
Die zunehmende Verbreitung von XML führte zunächst dazu, dass viele relationale Datenbanken um XML-Adapter erweitert wurden – eine dem OR-Mapping vergleichbare Strategie. Eine Speicherstrategie für XML, die sich die ihm eigene Baumstruktur zu Nutze macht, bieten XML-Datenbaken.

4.2. Differenzierung von XML-Datenbanken

Es haben sich im Laufe der Zeit zwei klassifizierende Bezeichnungen für XML-Datenbanken herausgebildet. Sie verhalten sich in etwa analog zu der bereits angesprochenen Differenzierung zwischen objektorientierten und objekt-relationalen Datenbanken. Man spricht von *XML-enabled* Datenbanken und *nativen* XML-Datenbanken.

XML-enabled (XED) werden Datenbanken genannt, die das Schema eines XML-Dokumentes auf ein Datenbankschema mappen. Das interne Schema der Datenbank kann relationaler, hierachischer oder auch objektorientierter Natur sein. Das Mapping erfordert *zwingend* die Existenz eines Schemas. [Bou05a]

Native Systeme (NXD) nutzen das XML-Modell direkt. Frei übersetzt nach der Definition der *XML:DB Initiative* [Bou05b] werden sie durch die folgenden Punkte charakterisiert:

- Sie definieren ein (logisches) Modell für ein XML-Dokument. Und sie speichern und durchsuchen Dokumente nach diesem Modell.
- Die fundamentale Speichereinheit ist ein XML-Dokument, wie es eine Tabellenzeile in einem RDBMS ist.
- Es ist kein bestimmtes physisches Modell zur Speicherung erforderlich.

Kimbro Staken [Sta01] hat es auf die folgenden drei, pragmatischen Punkte verdichtet:

- Die Datenbank ist auf das Speichern von XML-Daten spezialisiert und belässt alle Komponenten des XML-Modells intakt.
- In- und Output sind immer Dokumente.
- Eine NXD muss nicht immer nur eine Datenbank sein.

Native XML-Datenbanken können den dokumenten-orientierten Datenbanken zugeordnet werden. Ist also die prinzipielle Organisationseinheit im relationalen Model ein Tupel, so entspricht diesem auf Seite der nativen XML-Datenbanken ein Dokument. Dokumente, gleich welchen Schemas, werden intakt gespeichert. NXDs sind also *schemafrei* und *strukturerhaltend*. Dokumente können in *Collections* zusammengefasst werden, was in gewisser Hinsicht zu Tabellen in RDBMS vergleichbar ist, wobei es jedoch zu beachten gilt, dass diese Kollektionen hierarchisch und nicht relational strukturiert sind.

Eine genauere Differenzierung findet sich bei Klettke und Meyer [KM03]. In Abbildung 6 kann der rechte Ast auf erster Ebene als die Klasse der XML-enabled Systeme verstanden werden. Der linke Ast hingegen beschreibt eine konventionelle Speicherung im Dateisystem, wobei die Durchsuchbarkeit mittels Indizes optimiert wird. Der mittlere Ast schließlich umfasst diejenigen Systeme, die sich der Klasse der native XML-Datenbanken zuordnen lassen.

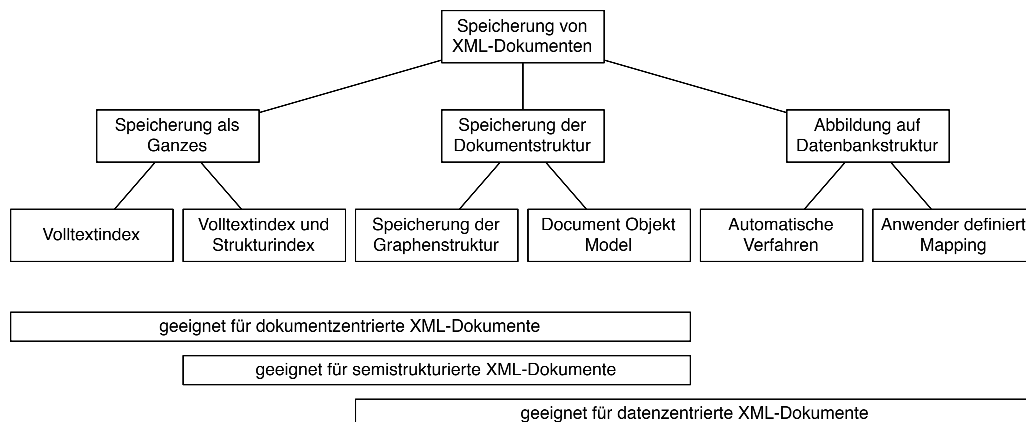

Abbildung 6: Ontologie der XML-Datenbanken [KM03]

Der untere Teil der Grafik leitet bereits zum nächsten Kapitel über; dieses soll sich im folgenden mit der Frage beschäftigen, welche Kriterien für den Einsatz eines spezifischen Typs von XML-Datenbanken sprechen und welche dagegen.

4.3. Nutzungskriterien

Aus dem Versuch der *XML-DEV*-Community einen Entscheidungsbaum zu modellieren, wann welcher Typ von Datenbank der Geeignetste sei – was allerdings nicht abschließend geklärt werden konnte – hat Leigh Dodds [Dod01] die am wichtigsten erscheinenden Kriterien destilliert, welche in Tabelle 2 zusammengefasst sind.

In Kapitel 6 wird auf die Beschaffenheit des Datenmodells von Tira genauer eingegangen. Die an dieser Stelle wichtigen Aussagen sind, dass es semistrukturiert ist und kein eindeutiges Schema existiert.

große Datenmengen	RDBMS, XED, NXD
große Dokumente	NXD
variierende Struktur (semi-strukturiert)	XED,NXD
streng strukturierte Daten	RDBMS
eindeutige Metadaten	NXD
komplexe Fragen	XED,NXD
Shared Repository	RDBMS,XED
getätigte Investitionen	k.A.
multiple oder überhaupt keine Schemata	NXD
Round-tripping (Erhalt der vollständigen Struktur)	NXD
referentielle Integrität	RDBMS
Enterprise Integration	k.A.

Tabelle 1: Kriterien für den Einsatz von XML-Datenbanken.
RDBMS: relational Database-Managementsystem
XED: XML-enabled Database
NXD: native XML-Database

Gleicht man Tabelle 2 und Abbildung 6 ab, erscheint eine Präferenz für den Einsatz einer nativen XML-Datenbank als gerechtfertigt. Diese können nicht nur mit semi-strukturierten Daten umgehen, sondern erlauben als einzige Systeme fehlende oder nicht eindeutig gefasste Schemata.

4.4. Verschiedene Systeme im Überblick

Nach der allgemeinen Einordnung der XML-Datenbanken und der Formulierung verschiedener Kriterien für deren Einsatz, ordnet Tabelle 2 die bekanntesten Vertreter in das bislang entwickelte Schema ein. Auffällig ist, dass namhafte kommerzielle Anbieter überproportional dazu neigen ihre etablierten Systeme mit dem Attribut XML-enabled für XML-lastige Anwendungen zu positionieren. Native Systeme entstammen häufig der akademischen Forschung und werden als Open Source-Projekte fortgeführt. So im Falle von BaseX und eXist.

Produkt	Hersteller	Lizenz	Typ
DB2 XML	IBM	Kommerziell	enabled
Oracle 9i	Oracle	Kommerziell	enabled
SQL Server	Microsoft	Kommerziell	enabled
Tamino	Software AG	Kommerziell	native
eXist	Wolfgang Meier	Open Source	native
Xindice	Apache	Open Source	native
Berkeley DB XML	Oracle	Open Source	native
BaseX	Uni Konstanz	Open Source	native

Tabelle 2: Übersicht verschiedener XML-Datenbanken

Bereits im vorangegangen Kapitel wurde ausgeführt, dass Tira aufgrund des entworfenen Datenmodells den Einsatz eines nativen Systems nahelegt. Die Auswahl von Systemen, die im Folgenden einer detaillierteren Betrachtung unterzogen werden, wird außerdem durch die Verfügbarkeit unter einer Open Source-Lizenz beschränkt.

BaseX wurde mit der vorrangigen Zielsetzung entwickelt XML-Daten innerhalb von Java-Anwendungen speichern und visualisieren zu können. Die Applikation ist selbst in Java implementiert und bietet XQuery für Java (XQJ) und die XML:DB-API als Schnittstellen.[6]

[6] http://www.basex.org/

Berkeley DB XML basiert auf der klassischen Berkeley DB und wurde ausschließlich für den eingebetteten Einsatz konzipiert. Sie unterstützt XQuery und XPath als Anfragesprachen.[7]

Xindice von der Apache Foundation wurde in Java implementiert. Das Projekt stellt XPath, XML:DB XUpdate und XML:DB API als Schnittstellen bereit. Des weiteren verfügt die Datenbank über eine XML-RPC-Schnittstelle.[8]

eXist ist ebenfalls in Java implementiert. Es unterstützt die Sprachen XQuery, XPath und XSLT. Mit ReST, WebDAV, SOAP, XML-RPC und Atom Publishing Protokoll bietet eXist die weitaus größte Vielfalt an Schnittstellen innnerhalb dieser Übersicht. Ebenso verhält es sich bei den Varianten des Deployments, wo neben Servlet, Stand-alone und Embedded auch die Möglichkeit der Einbindung als Cocoon-Block bereitsteht. Eine Besonderheit der Datenbank ist, dass sie die Dokumentstrukturen in einem B^+-Tree hält. Das System ist modular aufgebaut und lässt sich durch den Austausch von Modulen wie etwa XSLT-Prozessoren oder der Backend-Datenbank den jeweiligen Bedürfnissen anpassen.[9]

Mabanza und Chadwick haben in ihrem Vergleich [MC04] von nativen XML-Datenbanken die Systeme Berkeley DB XML, Xindice und eXist hinsichtlich der Performanz bei Schreib- und Lesezugriffen untersucht. Aus ihren Ergebnissen lässt sich schließen, dass eXist im Schreiben von Kollektionen am langsamsten ist, in der Anfrageverarbeitung hingegen am schnellsten. Berkeley DB XML schreibt am schnellsten und bearbeitet auch Anfragen nicht sehr viel langsamer als eXist. Apache Xindice erscheint insgesamt abgeschlagen.

Die Entscheidung für den Einsatz von eXist ist leicht zu motivieren. Es ist die einzige native XML-Datenbank im Feld, die von Hause aus über eine *ReST*-

[7] http://www.oracle.com/database/berkeley-db/xml/
[8] http://xml.apache.org/xindice/
[9] http://www.exist-db.org/

konforme HTTP-Schnittstelle verfügt. Diese Schnittstelle ist für die ins Auge gefasste Architektur von TIRA erforderlich. Da die Experimente in TIRA nur ein einziges Mal in die Datenbank geschrieben werden, im Zuge ihrer Analyse aber immer wieder umfassend durchsucht werden müssen, kann das Verhalten bei der Beantwortung von Anfragen als das entscheidende Performanzkriterium angesehen werden. Weiterhin ist das Projekt um eXIST sehr aktiv und hat durch Konzepte wie etwa Stored-XQuery, ReSTful HTTP und die Kombination mit APACHE COCOON die Entstehung der XRX-Architektur, eine der Inspirationsquellen für diese Arbeit, entscheidend befördert. Auf die Besonderheiten von eXIST, seine Konfiguration und die Wechselwirkungen mit dem Datenmodell von TIRA geht Kapitel 5 näher ein.

4.5. Anfragesprachen

Der folgende Abschnitt geht kurz auf die beiden vom W3C[10] standardisierten Anfragesprachen zur Verarbeitung von XML-Daten ein. Die von der XML:DB-INITIATIVE spezifizierten APIs spielen für TIRA keine Rolle, da diese Schnittstellen für den Einsatz innerhalb von etwa JAVA-Programmcode konzipiert sind und sich nicht für den Einsatz im Zusammenspiel mit dem HTTP-Protokoll eignen.

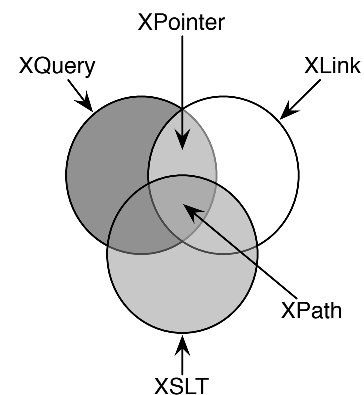

Abbildung 7: W3C-Sprachfamilie für die Verarbeitung von XML [W3S]

XQuery und XPath gehören beide dem Ökosystem von domänen-spezifischen Sprachen zur Verarbeitung von XML an. Wie auch in Abbildung 7 ersichtlich, gilt bezüglich des Sprachumfangs:

$$XPath \subset XQuery$$

[10]World Wide Web Consortium

XPath

Die Sprache dient als Hilfsmittel, eine Art Kitt für alle in Abbildung 7 aufgeführten Sprachen. Sie ermöglicht es über Pfadausdrücke Knotenmengen innerhalb von XML-Dokumenten zu selektieren. Verarbeitet werden diese Mengen dann durch eine jeweils andere Sprache. Neben der Navigation in der Dokumenthierarchie bietet XPath die Möglichkeit über *Prädikate* die Ergebnismenge zu beschränken. Der Sprachumfang von XPath genügt bereits für komplexere Anfragen an die Kollektionen einer XML-Datenbank. So würde das in Listing 1 aufgeführte Beispiel, angewendet auf eine entsprechende Kollektion, alle Konfigurationen mit einem beliebigen Element, dessen Attribut „implementation" den Wert „MajorClust" besitzt, zurückgeben.

```
//configuration[*//@implementation="MajorClust"]
```

Listing 1: Beispiel für einen XPath-Ausdruck

XQuery

Im Unterschied zu XPath handelt es sich bei XQuery nicht nur um eine Anfragesprache, sondern um eine turing-vollständige, funktionale Programmiersprache mit starker Typisierung. Die Typen entsprechen denen der XML-Schema-Sprache. Entsprechend der funktionalen Metapher ist alles Funktion beziehungsweise lässt sich als solche ausdrücken. Das gilt auch für die *FLWOR*- Statements, wie Datenbankanfragen im XQuery-Jargon bezeichnet werden. *FLWOR*, manchmal auch nur mit *FLWR* bezeichnet, wird wie „flower", die englische Blume ausgesprochen. Es ist das Äquivalent zu dem bekannten SELECT-FROM-WHERE-Statement in SQL. Die einzelnen Buchstaben des Akronyms stehen für die folgenden Ausdrücke:

for spezifiziert Teile einer Sequenz, die selektiert werden sollen.

let dient dem Initialisieren temporärer Variablen (optional).

where schränkt die Ergebnismenge ein (optional).

order sortiert die Ergebnismenge (optional).

return spezifiziert, was als Ergebnis zurückgegeben werden soll.

Das einfache Beispiel für ein *FLWOR*-Statement in Listing 2 ist äquivalent zu dem in Listing 1 aufgeführten XPath-Ausdruck, das heißt, es erzeugt die gleiche Ergebnismenge. Vergleicht man die beiden Anfragen, stellt man fest, dass sich der XPath-Ausdruck in der wesentlich umfangreicheren XQuery-Anfrage, entsprechend der vorangegangen Aussage zu dem Verhältnis der beiden Sprachen zueinander, leicht wiederfinden lässt.

```
for $conf in //configuration
where $conf//@implementaion="MajorClust"
return $conf
```

Listing 2: Äquivalentes FLWOR-Statement

Offensichtlicher wird dies, wenn der XPath-Ausdruck 1:1 in das for-Statement übernommen wird, wodurch die Einschränkung über `where` entfällt, wie in Listing 3 geschehen.

```
for $conf in //configuration[*//@implementation="MajorClust"]
return $conf
```

Listing 3: Verkürztes äquivalentes FLWOR-Statement

Der Mehrwert von XQuery findet sich somit auch nicht vordergründig in dem Formulieren von Datenbankanfragen, sondern in der sprachlichen Vollständigkeit. So ist es möglich Webapplikationen in XQuery zu implementieren. Besonders interessant in diesem Zusammenhang ist die Tatsache, dass sich auch XML in XQuery-Scripte einbetten lässt und die Ausgabe dadurch beliebig formatiert werden kann.

5. eXist

Als ein Forschungsprojekt der TU Darmstadt entstanden, wird eXist heute von einer Open Source-Gemeinde um Wolfgang Meier gepflegt und weiterentwickelt. Nach einer kurzen Zusammenfassung wesentlicher Merkmale dieser nativen XML-Datenbank werden die für den Einsatz in Tira relevanten Aspekte der Konfiguration und Programmierung eingehender untersucht. Aussagen zur Performanz und anderen technischen Details, die für die Auswahl von eXist ausschlaggebend waren, finden sich in Kapitel 4.4 auf Seite 18.

5.1. Betrieb

eXist unterstützt eine Vielzahl von Möglichkeiten die Datenbank in Betrieb zu nehmen, des so genannten *Deployments*. So kann sie, über die einschlägigen XML:DB-APIs, in Java-Applikationen eingebettet werden, als Servlet betrieben werden, wobei die ganze Fülle der implementierten Protokolle zur Verfügung steht, oder auch als Modul, als sogenannter Block, in das Apache Cocoon-Framework integriert werden. Für die anvisierte Architektur ist jedoch die stand-alone Variante von besonderem Interesse, in der lediglich ReST, XML-RPC und WebDAV als Protokolle zur Verfügung stehen.

5.2. Protokolle

Ebenfalls vielfältig sind die Protokolle, die es erlauben auf die Datenbank zuzugreifen. Dabei weist die Auswahl auf eine große Web-Affinität hin. Mit ReST, WebDAV und dem Atom-Publishing-Protokoll unterstützt die Datenbank allein drei ReST-konforme Schnittstellen, die den leichtgewichtigen, nativ auf

HTTP basierenden Protokollen zuzurechnen sind. Mit SOAP und XML-RPC stehen auch die beiden Vertreter der klassischen WebService-Protokolle zur Verfügung.

In den folgenden Abschnitten zu Programmierung und Anfragen wird ausschließlich auf die Gegebenheiten unter Verwendung der ReST-konformen HTTP-Schnittstelle Bezug genommen, da diese in TIRA zum Einsatz kommen soll.

5.3. Indexierung

Das Indizieren von Datenbankinhalten hat unbestritten maßgeblichen Einfluss auf die Performanz bei der Beantwortung von Anfragen. Dies gilt insbesondere für die eXist-Datenbank, denn diese implementiert einen eigenen, indexgetriebenen XQuery-Prozessor [Mei06]. Im Gegensatz zu anderen Implementierungen verfolgt dieser keine Traversierungsstrategie, sondern operiert mit so genannten *Struktur-Joins*. Dafür sind Indizes unentbehrlich und es ergeben sich auch einige Besonderheiten für die Formulierung von Anfragen, auf die Kapitel 5.4 näher eingeht.

Um eXist zu ermöglichen seine Stärken auszuspielen, ist zunächst die Organisation der Indizes von immenser Bedeutung. Von den zu indizierenden Daten – genauer: deren Struktur – hängt es ab, welche Strategie letztlich zum Ziel führt. Die Datenbank bietet zu diesem Zweck verschiedene Indextypen an, die entsprechend konfiguriert werden müssen. Zunächst seien die verfügbaren Indizes kurz vorgestellt. [eP09a]

Built-in-Indizes

Structural Indexes indizieren die Knotenstruktur, Elemente und Attribute der Dokumente einer Kollektion.

Range Indexes mappen spezifische Textknoten und Attribute der Dokumente einer Kollektion zu getypten Werten.

Modulare Indizes

Old Legacy Full Text Indexes mappen spezifische Textknoten und Attribute der Dokumente einer Kollektion zu Texttokens.

New Full Text Indexes integrieren Apache Lucene in die XQuery-Engine von eXist (ab Version 1.4). Dieser Volltextindex ist grundsätzlich zu bevorzugen.

NGram Indexes mappen spezifische Textknoten und Attribute der Dokumente einer Kollektion zu Teiltokens der Länge n (Default n = 3). Dies ist sehr effizient für die Suche von Substrings und in Texten, die sich nicht leicht durch Whitespaces zerteilen lassen.

Spatial Indexes sind experimentelle Indizes für Elemente zu denen Geoinformationen verfügbar sind.

Im weiteren werden die beiden Built-in-Indizes einer eingehenderen Betrachtung unterzogen, da diese für das stark typisierte Datenmodell von Tira unentbehrlich sind. Weitere Indizes werden nicht benötigt, da keine Fließtexte in den Daten vorgesehen sind.

Structural Indexes Der Strukturindex wird immer automatisch erzeugt und kann durch den Nutzer in keiner Weise beeinflusst werden. Er dient der Bearbeitung aller XQuery-/XPath-Anfragen außer „wildcard-only"-Ausdrücken wie etwa `//*`. [eP09a]

```
//book/section
```

Listing 4: Exemplarischer XPath-Ausdruck

Der XPath-Ausdruck, den Listing 4 zeigt, benötigt beispielsweise zwei Indexlookups, einen für `<book>`-Knoten und einen für `<section>`-Knoten. Ein Struktur-Join der beiden Mengen stellt im folgenden fest, welche `<section>`-Knoten Kindelemente von `<book>`-Knoten sind.

Range Indexes Dieser Index muss durch den Nutzer explizit für jeden gewünschten Knoten konfiguriert werden. Er wird für die in Listing 5 aufgeführten Vergleichsoperationen und -funktionen – etwa in XPath-Prädikaten – benötigt.

```
=, >, <
fn:contains(), fn:starts-with(), fn:ends-with(), fn:matches()
```

Listing 5: Vergleichsoperationen und -funktionen in XPath

Ist dieser Index nicht vorhanden oder nicht benutzbar, fällt eXist immer auf eine Brute-Force-Strategie zurück. Um den Index nutzen zu können, müssen die Datentypen der Anfrage mit denen des Index übereinstimmen. Der XPath-Ausdruck in Listing 6 würde beispielsweise zu einem Index-Scan, beginnend bei 100, führen. [eP09a]

```
//item[price > 100]
```

Listing 6: Exemplarischer XPath-Ausdruck

Konfiguration Die Konfiguration der Indizes erfolgt über Dateien in einer separaten Systemkollektion. Diese Konfigurationsdateien selbst sind in XML formuliert und werden in einer „Verzeichnis"-Struktur gespeichert, die die Struktur der Kollektionen innerhalb der Datenbank nachbildet. Konfigurationen gelten entsprechend für eine Kollektion und alle Unterkollektionen, sofern die Konfiguration nicht durch in der Hierarchie tiefer liegende Konfigurationen überschrieben wird. Diese Form von Vererbung ist derjenigen, die von den `htaccess`-Dateien des Apache WebServers bekannt ist, vergleichbar.

```
<collection xmlns="http://exist-db.org/collection-config/1.0">
    <index>
        <!-- Range indexes -->
        <create qname="implementation" type="xs:string"/>
        <create qname="k" type="xs:integer"/>
        <!-- "old" context-dependant path configuration -->
        <create path="//parameter@item" type="xs:string"/>
```

```
        <create path="//Collection/document" type="xs:string"/>
    </index>
</collection>
```

Listing 7: Beispiel einer Indexkonfiguration

Wie in Listing 7 ersichtlich, stehen prinzipiell zwei Varianten zur Verfügung, den zu indizierenden Element- oder Attribut-Knoten zu spezifizieren.

path -Definitionen ähneln XPath-Ausdrücken und verhalten sich auch analog zu diesen. Sie erlauben es etwa, identisch benannte, strukturell jedoch unterschiedliche Knoten einzeln zu indizieren.

qname -Definitionen demgegenüber legen einen Index über alle Knoten – sowohl Attribute, als auch Elemente – eines qualifizierten Namens an. Diese Indexdefinitionen sind grundsätzlich zu bevorzugen, da die QueryEngine Anfragen gegen diese besser optimieren kann, obgleich sie sehr groß werden können.

Wie bereits erwähnt, müssen die durch einen Range Index indizierten Felder eindeutig getypt sein. Derzeit unterstützt eXist die Typen `xs:string, xs:integer, xs:double, xs:float, xs:boolean` und `xs:dateTime`.

Indizes werden vom System automatisch verwaltet und aktualisiert – bis auf eine Ausnahme. Im Falle einer Konfigurationsänderung für eine Kollektion werden nur neu hinzugefügte Daten entsprechend der neuen Regeln indiziert. Um die gesamte Kollektion neu zu indizieren, muss dieser Prozess manuell über den *Java Admin Client* angestoßen werden.

Zusammenfassend lassen sich die folgenden Stichpunkte[Mei09] als „Best Practices" für die Indizierung festhalten:

- Niemals auf das Standardverhalten verlassen.
- Einfache Indexdefinitionen sind zu bevorzugen: „qname over path".
- Für stark typisierte Daten sollte ein Range Index angelegt werden.

- Für exakte Substringsuche und längere Textsequenzen sollte der Einsatz eines NGram Index erwogen werden.
- Der Einsatz eines Volltextindex ist sinnvoll im Falle von Texten, die durch Whitespaces in Tokens getrennt werden können.

5.4. Anfragen

Suchanfragen können über das HTTP-Protokoll wahlweise in XPath oder XQuery an die Datenbank gerichtet werden. Prinzipiell wird der entsprechende Ausdruck einem GET-Request angehängt, was insbesondere für kurze XPath-Ausdrücke sinnvoll erscheint. Dieses Vorgehen entspricht auch einem ReST-konformen Ansatz. Wird die Anfrage aber zu umfangreich, muss auf einen POST-Request zurückgegriffen werden. In beiden Fällen ist der Query Client-seitig zu generieren.

Eine weitere Möglichkeit besteht darin, lediglich Variablen mittels GET zu übermitteln und diese server-seitig in ein dort hinterlegtes Anfragescript zu integrieren. Über POST ist es darüber hinaus möglich, ganze Dokumente an den Server zu senden und dort aus diesen Querys zu erzeugen. Auf die server-seitige Programmierung geht das nachfolgende Kapitel näher ein.

Ganz gleich, für welche Variante man sich letztlich entscheidet, gilt es einige Dinge zu beachten, will man die Performanz der Datenbank ausspielen. Die wichtigsten seien im folgenden zusammengefasst:

- XPath-Prädikate sind Where-Ausdrücken vorzuziehen.
- Es sollten kurze Pfade bevorzugt werden. (`a//f` statt `a/b/c/d/e/f`)
- Das am stärksten einschränkende Prädikat sollte zuerst angewandt werden.

5.5. Programmierung

Neben den Anfragesprachen XPath 2.0 und XQuery 1.0 unterstützt eXist auch die Transformationssprache XSLT 1.0 durch den eigebundenen Prozessor Xalan bereits von Hause aus. Jedoch kann dieser auch gegen den Prozessor Saxon ausgewechselt werden, womit das Vokabular von XSLT 2.0 zur Verfügung steht.

Wie bereits in Kapitel 4.5 angeklungen, handelt es sich bei XQuery um eine vollständige Programmiersprache. eXist macht sich diesen Umstand zu Nutze und verwandelt sich von einer einfachen Datenbank zu einer Plattform für Webapplikationen. Zwei verschiedene Möglichkeiten werden dem Entwickler geboten, seine in XQuery formulierte Anwendungslogik dem System zu verabreichen.

Stored XQuerys sind Scripte, die in einer Kollektion der Datenbank hinterlegt sind. Durch die Tatsache, dass sie in einer Kollektion residieren, ist auch der Kontext im Falle einer Ausführung bereits gegeben. Ihr Verhalten bei HTTP-Anfragen ist den ReST-Prinzipien entsprechend vordefiniert. Das macht sich insbesondere bei PUT-Requests bemerkbar, die grundsätzlich zu einem Überschreiben des XQuery-Dokuments führen und nicht etwa zu dessen Ausführung.

XQuery-Servlets sind Scripte, die server-seitig im Dateisystem gespeichert werden. Sie erlauben dem Entwickler das Anfrageverhalten und den Kontext beliebig zu beeinflussen. Diese Variante der server-seitigen Programmierung ist derjenigen, die von Java Server Pages oder PHP bekannt ist, vergleichbar.

Wird eine der beiden server-seitigen Varianten angewandt, stehen dem Programmierer umfangreiche Bibliotheken zur Verfügung, wodurch die Entwicklung von Webapplikationen auf Basis von XQuery erheblich erleichtert wird.

6. Datenmodell

Eine der wichtigsten Aufgaben der Informatik ist die Strukturgebung. Ein Datenmodell beschreibt die Struktur von Daten durch Vorgabe eines Schemas. Das ist wichtig, da ohne Wissen über die Struktur von Daten eine Verarbeitung dieser unmöglich wäre. Darüber hinaus strukturiert ein Modell das Expertenwissen, das sich in einem System manifestieren soll. Datenmodelle müssen eindeutig wie auch allgemein gefasst werden. Sie sollen alle möglichen Ausprägungen entsprechender Daten erfassen können, jedoch alle anderen mit Sicherheit ausschließen. Daten gleicher Struktur können, beispielsweise für die Präsentation, unterschiedlich repräsentiert werden. Mehrere unterschiedliche Repräsentationen innerhalb eines Systems, erfordern unterschiedliche Datenmodelle und führen zu der Notwendigkeit von Transformationen.

Aus einem anderen Blickwinkel betrachtet, erfordern konkrete Technologien auch immer bestimmte Repräsentationsformen der Daten. Das bedeutet, dass sich auch die Formulierung des Datenmodells nach diesen Anforderungen zu richten hat. Datenmodelle, die sich nach technologischen Gegebenheiten richten, können sich in ihrer Ausdrucksstärke, das heißt den strukturgebenden Sprachmitteln, stark unterscheiden. Das führt spätestens bei Transformationen von einer Repräsentationsform in eine Andere zu Problemen, wenn einzelne Strukturcharakteristika nicht abgebildet werden können. Dieser „Reibungsverlust“ wird langläufig als „impedance mismatch“ bezeichnet.

Eines der Ziele dieser Arbeit ist, die Anzahl verwendeter Repräsentationsformen, besonders die zueinander inkompatibler Datenmodelle, in Tira zu minimieren. Dadurch reduziert sich die Anzahl an benötigten Datenmodellen und es lässt sich darüber hinaus der Programmcode für die sonst nötigen Transformationen einsparen. An ein einheitliches Datenmodell werden aber auch neue

Anforderungen gerichtet. So muss dieses in der Lage sein die mitunter höchst unterschiedlichen Erfordernisse der Systemkomponenten hinsichtlich Darstellung, Verarbeitung und Speicherung in Einklang zu bringen.

Daten entstehen in Tira derzeit bei der Modellierung eines Experimentes und als Resultat der Durchführung eines solchen. Die Implementierung weiterer, in den Anforderungen (Kapitel 3) bereits beschriebener Usecases, werden zukünftig zu einer Erweiterung des Datenmodells führen. Diese werden hier jedoch nur am Rande zur Sprache kommen, da deren Struktur zum gegenwärtigen Zeitpunkt noch nicht hinreichend ausgestaltet ist. Diese Arbeit konzentriert sich auf die Modellierung der Daten, welche im Zuge eines Experimentdesigns entstehen.

6.1. XML als Datenmodell

Generell ist im Umgang mit XML zwischen der Verwendung als Serialisierungssyntax für beliebige Datenmodelle und dem Einsatz als eigentliches Datenmodell zu differenzieren. XML ist in der Lage, fast jeden beliebigen Grad von Strukturierung abzubilden, was es für die Serialisierung auszeichnet. Aber wie so oft ist die Stärke zugleich die Schwäche. So kann sich die Beschreibung einer Struktur in XML nicht auf ein derart fundiertes mathematisches Modell wie etwa das relationale stützen. XML eignet sich folglich nur bedingt für die Modellierung stark strukturierter Daten. Bei der Beschreibung semi-strukturierter Daten kann die Sprache ihre Stärken jedoch voll entfalten.

Der Begriff semi-strukturiert ist nicht eindeutig definiert. Er bezeichnet einen Grad von Strukturierung, der irgendwo zwischen unstrukturiert und stark strukturiert liegt. Es gibt aber einige Indizien, die als charakteristisch gelten dürfen. Die aufgeführten Indizien implizieren, dass die Daten, wie sie in Tira entstehen, einen semi-strukturierten Charakter aufweisen. [KST02]

- Die Struktur weist Variationen auf, die sich in fehlenden oder zusätzlichen Elementen widerspiegeln.

- Die Struktur, wie sie im Rahmen dieser Arbeit definiert wird, lag zunächst nur eingebettet vor, das heißt das Schema wurde nach Vorliegen der Daten (a posteriori) formuliert.
- Die Beschreibungen der Struktur ist indikativ, d.h. nicht einschränkend. Werden neue Experimente eingeführt, können diese von der bisherigen Struktur abweichen.
- Die Struktur ist nur partiell gegeben. Teile der Daten werden nicht durch das Datenmodell strukturiert.

Die bisherigen Aussagen bezogen sich auf die Struktur der Daten. Diese haben Auswirkungen auf die Definition eines Datenmodells. Ein semi-strukturiertes Datenmodell sollte in der Lage sein die folgenden Problemstellungen zu lösen: [KST02]

- Ein umfassendes Schema wäre sehr groß und nicht mehr handhabbar. Niemand wäre in der Lage über ein umfassendes Schemawissen zu verfügen. Demnach muss das Schema ein exploratives Vorgehen unterstützen. Für diesen Zweck eignet sich eine strikte Trennung von Schema- und Instanzebene nicht.
- Ein Suche in semi-strukturierten Daten kann effizenter sein, wenn das Schema ignoriert und eine Volltextsuche durchgeführt wird.
- Schemata entwickeln sich schnell, insbesondere gilt dies natürlich für das vereinheitlichte Schema, aber auch für die Teilschemata.
- Die Typisierung von Daten ist von der Verarbeitungssituation abhängig. Ist etwa ein bool'scher Wert in der Persistenzschicht nur ein einfacher String, so ist sein Wahrheitswert für die Applikation von Bedeutung.
- Eine Unterscheidung zwischen Daten und Schemata ist nicht mehr möglich. Manche Information ist in der einen Quelle als Datum kodiert, in der anderen als Typinformation – und somit Schemaebene.

Da sich XML für die Modellierung semi-strukturierter Daten anbietet und ohnehin für die Serialisierung und Anzeige der Daten angewendet werden soll, liegt der Schritt hin zu einem in XML formulierten Datenmodell nahe. Die Verwendung einer nativen XML-Datenbank für die Speicherung der Daten ist daher folgerichtig.

6.2. Anforderungen

Da es zu den Anliegen dieser Arbeit gehört die Anzahl der Datenmodelle auf möglichst nur eines zu reduzieren, werden an dieses eine Vielzahl von Anforderungen gerichtet. Jede Komponente von TIRA, die mit den Daten arbeiten muss, und das sind bei nur einem Datenmodell alle, legt eine ihr eigene Präferenz für eine bestimmte Strukturierung der Daten an den Tag. Und das, obwohl die einzelnen Komponenten mit Bedacht ausgewählt wurden. Nahezu alle unterstützen eine native Verarbeitung von XML-Daten. Dennoch kann nur eine Vermittlung zwischen den im folgenden aufgeführten Anforderungen, keinesfalls aber eine vollständige Befriedigung aller Aspekte erreicht werden.

Anforderungen des Clients

Der TIRA-Client bewerkstelligt die Darstellung der Daten über XSL-Transformationen. Um einen interaktiven Modellierungsprozess für Experimente zu realisieren, werden im Zuge der Nutzerinteraktion einzelne Schemafragmente dynamisch geladen und in die Benutzungsoberfläche eingebunden. Während dieses *explorativen* Prozesses entsteht aus den Schemafragmenten und den Nutzereingaben ein Instanzdokument für das Design eines Experimentes. Wichtig ist in diesem Zusammenhang der folgende Punkt.

- Eine generische Benennungen der Elemente und Attribute des XML-Dokumentes ermöglichen eine einfache und wiederum generische Handhabung durch das Transformations-Stylesheet. Das führt zu einer leich-

teren Erweiterbarkeit des Modells, ohne Eingriffe in die Verarbeitungslogik des Clients vornehmen zu müssen.

Darüber hinaus ist es Aufgabe des Clients Suchanfragen an die Datenbank zu richten. Der Nutzer kann sich zu diesem Zweck derselben Benutzungsoberfläche bedienen, mit dem Unterschied, dass unvollständige Designvorgaben zu einer Wildcard-Anfrage für die betroffenen Elemente führen. Allerdings generiert der Client nun kein Instanzdokument, sondern einen XPath-Ausdruck. Zwei Aspekte, die in gleicher Form für die Datenbank gelten dürfen, sind festzuhalten:

- Eine flache Struktur in den XML-Instanzdokumenten führt zu gut lesbaren und einfachen, d.h. kurzen XPath-Ausdrücken.
- Eine prägnante Struktur, das heißt viele aussagekräftige Attribut- und Elementnamen, können zu kurzen Pfaddefinitionen führen. Allerdings würde das erfordern, dass der Client in der Lage ist eine Optimierung durchzuführen, oder aber es wird eine zweite Anfragemaske bereitgestellt, in der sich die Parameter willkürlich festlegen lassen. Beide Varianten setzen allerdings umfassendes Schemawissen voraus – die erste bei der Client-Implemetierung, letztere beim Nutzer. Der Nutzer aber könnte dadurch in die Lage versetzt werden den Datenbestand durch sukzessives Einschränken zu explorieren.

Anforderungen des Servlets

Die Gestalt des Datenmodells wird a priori durch die Struktur des Klassenmodells der AiTools maßgeblich beeinflusst. Um einen problemlosen Zugriff über Schnittstellen wie etwa DOM oder SAX zu gewährleisten, kann festgehalten werden:

- Idealerweise sind alle von einer zu instanziierenden Klasse benötigten Daten als Attribute oder Inhalt eines Elementes verfügbar.

Anforderungen der Datenbank

Die Bedingungen, unter welchen die eXist-Datenbank eine effiziente Verarbeitung der Suchanfragen gewährleistet, wurden bereits in Kapitel 5.3 besprochen. Für die konkrete Umsetzung in Tira sind einige Fakten festzuhalten:

- Suchanfragen werden vorwiegend an Attribut- oder Element-Knoten gerichtet, deren Inhalt durch den Nutzer bestimmt wird. Solche Knoten sollten indiziert werden.
- Indizes können nur über Knoten eines einzelnen Datentyps angelegt werden. Diese Felder können Attribut- oder Elementknoten sein, brauchen aber einen eindeutigen Namen, über den sie ansprechbar sind. Pfadlokalisierungen sind möglich, aber ungünstig.
- Können unterschiedliche Knoten den gleichen Kindknoten mit dem gleichen Wert enthalten, so können diese Kindknoten nur über die Struktur unterschieden werden.
- Kurz Pfadangaben ersparen aufwendige Struktur-Joins während der Verarbeitung.

6.3. Adhoc-Struktur

Aus den in Kapitel 3 benannten Anforderungen lässt sich eine erste grobe Strukturierung der Daten ableiten. Die im folgenden aufgeführten Aussagen wurden dazu herangezogen Kollektionen (siehe auch Kapitel 5) anzulegen und die Datenbank dadurch grob zu strukturieren:

Designs beschreiben die Modellierung einer durchzuführenden Verarbeitung.

Resultate fassen alle Ergebnisse einer Verarbeitung zusammen.

Experimente bestehen aus einem Design und einem Resultat.

Analysen werden auf den Resultaten von Experimenten durchgeführt (d.h. sie referenzieren diese) und bestehen ebenfalls aus einem Design und einem Resultat.

Dadurch erhöht sich die strukturelle Ähnlichkeit der Dokumente innerhalb einer Kollektion entscheidend, was die Findung eines beschreibenden Schemas erheblich erleichtert. Wie im vorangegangen Kapitel bereits genannt, ist es typisch für semi-strukturierte Daten, dass sich ein Schema nur partiell beschreiben lässt. Indem TIRA gesonderte Schemata für jede Kollektion definiert, wird diesem Umstand Rechnung getragen.

6.4. Modellbildung für Experimentdesigns

Wie bereits angekündigt, wird sich der Fokus der Modellierungsarbeit im weiteren ausschließlich auf das Design von Experimenten richten. Dieses Kapitel dient der Einführung einiger Begriffe, von denen im Weiteren intensiver Gebrauch gemacht werden wird.

Strukturanalyse

Auf Basis des gegenwärtigen Standes der Implementierung der AITOOLS durch das TIRA-Servlet entstand das in Abbildung 8 dargestellte Modell. Es beschreibt alle derzeit möglichen Experimente in Form eines gerichteten, azyklischen Graphen. Der Umstand, dass es sich nicht um einen Baum handelt, lässt bereits erahnen, dass eine vollständige Umsetzung in XML nicht ohne ein Konzept von Referenzierung auskommen wird, denn ohne ein solches würde es im Umfang nicht mehr handhabbar sein, vor allem aber Redundanzen aufweisen.

Die zum Verständnis der Grafik erforderliche Begriffssemantik lässt sich in den folgenden Punkten zusammenfassen. Gleichzeitig wird auch das aus der Analyse der Struktur gewonnene Wissen zusammengefasst.

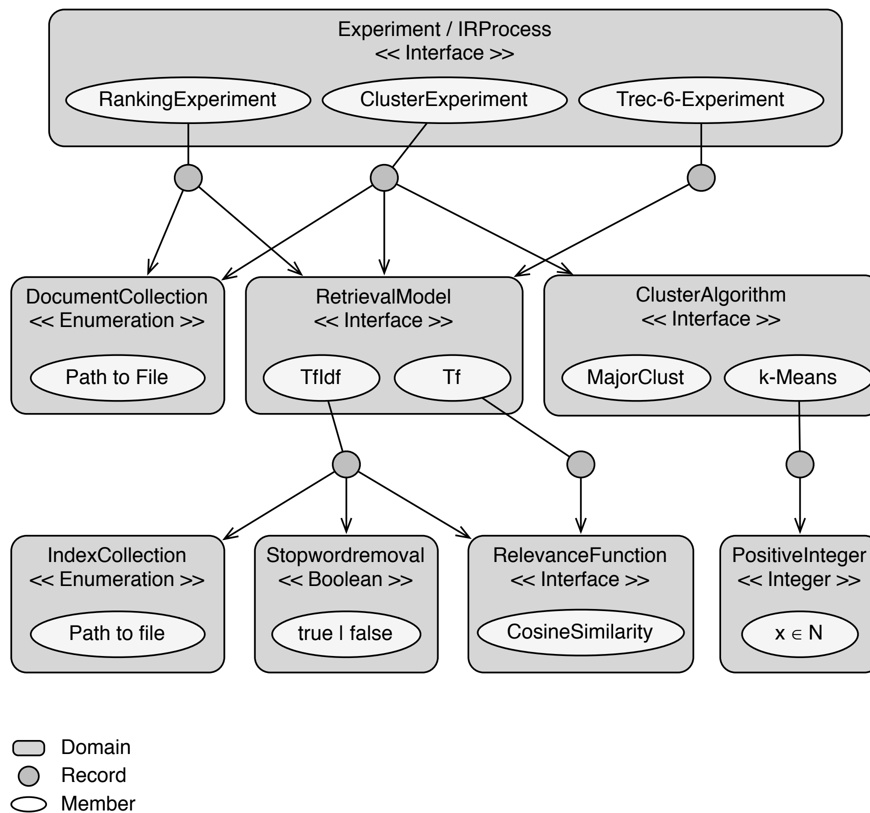

Abbildung 8: Exemplarisches Modell für Experimentdesigns

Domains sind benannte Mengen. Aus einer Domain muss immer ein Member ausgewählt werden. Domains können Interfaces, Enumerations oder primitive Datentypen beschreiben.

Member können Implementierungen, Elemente einer Aufzählung, bool'sche Werte, natürliche oder reelle Zahlen sein. Member können auf einen Record verweisen, jedoch nur, wenn es sich bei dem Member um eine Implementierung handelt.

Records beschreiben eine fixe Anzahl von Abhängigkeiten indem sie eine Menge von Verweisen auf Domains bereitstellen.

Neben der Graphenstruktur weist Abbildung 8 auch das in dem Modell strukturierte Expertenwissen aus. Beispielsweise welche Parameter zu spezifizieren sind, um ein Cluster-Experiment durchführen zu können. Da es sich um den gegenwärtigen Implementierungsstand handelt, ist das Expertenwissen dieses Modells unvollständig. Das wird es aber auch immer bleiben, denn die Zukunft bringt immer neues Wissen. Folglich strukturiert das Modell das Wissen in einem Graphen der wächst.

Erweiterung um Serienexperimente

Wie bereits in den Anforderungen in Kapitel 3 diskutiert, soll es dem Nutzer ermöglicht werden, Serien von Experimenten in einem einzigen Arbeitsschritt zu modellieren, um nicht jedes Design einer Serie einzeln definieren zu müssen. Die dahingehende Erweiterung des Datenmodells, bedeutet nicht nur mehr Komfort für den Nutzer, sondern bietet darüber hinaus die Möglichkeit eine größere Anzahl Experimente kompakt in nur einem einzigen Request in Auftrag zu geben.

Innerhalb eines Experimentdesigns sind die Member einer Domain die einzigen frei wählbaren Parameter. Die einzelnen Experimente einer Serie können sich somit lediglich in diesen unterscheiden. Es wird von *Variationen* über die Member einer Domain gesprochen. In der Benutzungsoberfläche kann dies durch das Ermöglichen einer mehrfachen Auswahl realisiert werden. Idealer Weise sollte diese Auswahl ohne Zurücklegen stattfinden, um überflüssige Wiederholungen von vornherein zu verhindern, das heißt bereits gewählte Member einer Domain stehen nicht zur erneuten Auswahl bereit. Für eine gezielt mehrfache Ausführung des selben Experimentes erscheint die Angabe der Anzahl dieser Wiederholungen zweckdienlicher als ein mehrfacher Designvorgang. In der zu Beginn von Kapitel 6.4 bereits vorgestellten Begriffssemantik ist der Wortlaut für Domains wie folgt zu erweitern.

Domains sind benannte Mengen. Aus einer Domain können Member ausgewählt werden. *Für Einzelexperimente eines, im Falle von Serienexperimenten jedes Member je ein mal.* [...]

Eine kompakte Beschreibung von Experimentierserien erfordert im Vorfeld der Durchführung und Speicherung eine Expansion in die enthaltenen Einzelexperimente. Zum einen weil das TIRA-Servlet nur einzelne Experimente durchführt, zum Anderen weil dies zu einer konsistenteren Datenbasis, die sich besser anfragen lässt führt.

Instanziierung des Modells

Das Design eines Experimentes entspricht aus Sicht des Systems einer Instanziierung des Modells. Die graphische Metapher aufnehmend, kann dieser Vorgang als das Herauslösen eines Baumes aus dem Graphen des Modells angesehen werden. Abbildung 9 skizziert die entsprechenden Abläufe.

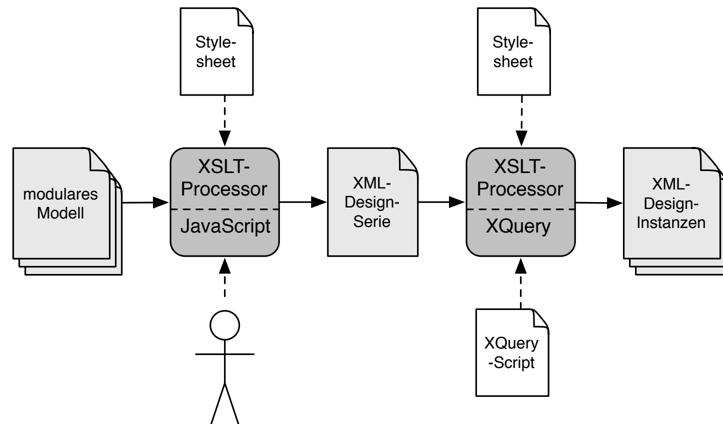

Abbildung 9: Pipeline für die Erzeugung von Design-Instanzen aus dem Modell

Zunächst wird das fragmentarische Modell, mit dem in ihm enthaltenen Expertenwissen, durch den den Einfluss des Nutzers und eines Transformations-Stylesheets exploriert und die ausgewählten Konten in einem einzelnen XML-

Dokument zusammengefasst. Dieses entstehende Dokument ist in Form eines Baumes strukturiert und kann mehrere Instanzdokumente kompakt zusammenfassen. Zu diesem Zweck können nicht nur Member, sondern auch Domains mehrere Äste besitzen. Dieses Konzept kann zwar zu Redundanzen innerhalb von Instanzserien führen, jedoch können diese leichter expandiert werden, da lediglich über die Äste einer Domain zu iterieren ist. Die Expansion wird von einem XQuery-Script, ebenfalls unter Einwirkung eines Transformations-Stylesheets, vor der Speicherung in der Datenbank automatisch vorgenommen.

Generalisierung zu einer Grammatik

Im Verlauf dieses Kapitels wird das Vorhaben in Angriff genommen, aus der in Kapitel 6.4 betrachteten Struktur der Daten, die für das Design eines Experiments erforderlich sind, eine generalisierte Grammatik zu entwickeln. Diese dient als theoretische Grundlage für die Formulierung der Schemata, die zum einen die Struktur der Instanzdokumente, zum anderen die der Modellfragmente festlegen. Zu diesem Zweck sollen zunächst die bislang verwendeten Begrifflichkeiten differenzierter betrachtet werden. Eine Unterscheidung zwischen einem abstrakten Datenmodell, welches als Metamodell verstanden werden sollte, und einem implementierungsnahen Datenmodell scheint angebracht. Tabelle 3 veranschaulicht, wie sich diese beiden Modelle zueinander verhalten, sowie deren vorzunehmende Abbildung auf die Möglichkeiten von XML.

Abstraktes Modell	Implementierungsnahes Modell	XML-Abblidung
Domain	Interface, Enumeration, Primitive	Element
Member	Implementation, Item, Integer, Real, Boolean	KindElement(e)
Record	Dependency	Struktur

Tabelle 3: Zusammenhang der Modellebenen

Wird das zur Verfügung stehende Vokabular auf das des abstrakten Modells reduziert, kann eine erste formale Definition der Struktur vorgenommen werden. Listing 8 zeigt diese in der erweiterten Bacchus-Naur-Form (EBNF).

```
<domain>    ::= { name member {<record>}? }+
<record>    ::= { <domain> }+
```

Listing 8: Abstraktes Modell in EBNF

Wie sich erkennen lässt sind die Member und der Name einer Domain die einzigen Terminalsymbole der Grammatik. Ein konkretes Design wird durch diese demnach vollständig beschrieben. Sowohl Domain-, als auch Record-Symbole dienen einzig der Strukturierung der Member. Domains strukturieren die Ausgabe durch ihr Auftreten aktiv. Records produzieren eine Rekursion zu der Domain-Produktion. Sie erzeugen dadurch ein Netzwerk aus Domain- und Member-Symbolen und strukturieren die Ausgabe passiv. Wie bereits angesprochen, kann das Datenmodell von Tira nicht durch ein einzelnes Modell beschrieben werden. Dieses wäre viel zu groß und würde eine Vielzahl von Redundanzen aufweisen. Aus diesem Grund wurde eine modulare Organisation des Modells gewählt. Die einzelnen Fragmente beschreiben jeweils immer eine Domain oder einen Record. In der Metapher des Graphen demnach dessen Knoten. Domain-Fragmente enhalten Informationen zur entsprechenden Domain und zu den enthaltenen Membern. Record-Fragmente werden von Membern referenziert und verweisen selbst wiederum auf Domains. Ihre ausschließliche Funktion besteht somit in dem Verknüpfen von Informationen.

Das Wissen, das in dem Datenmodell von Tira enthalten ist, steht im Anwendungsfall folglich immer nur partiell zur Verfügung. Globales Wissen wird zu keinem Zeitpunkt benötigt. Diese verteilte Organisation des Modells hat den Vorteil, dass es sehr einfach und jederzeit erweitert werden kann. Die Aufgabe des Clients besteht darin diese Fragmente zusammenzuführen und eine Instanz eines Design zu erzeugen. Diese Instanz beschreibt schließlich nur noch sich selbst und besitzt keinerlei Wissen über ihren Modell- oder Serienkontext.

In einem zweiten Schritt soll das abstrakte Modell in das Implementierungsnahe überführt werden (siehe Listing 9). Dadurch erweitert sich das zur Verfügung stehende Vokabular und die Struktur kann detaillierter beschrieben werden.

```
<design>          ::= <domain>
<domain>          ::= <name> <member-type>
<member-type>     ::= <enumeration> | <primitive> | <interface>
<interface>       ::= <name> { <implementation> }+
<record>          ::= { <domain> }+
<primitive>       ::= <name> integer | <name> real
                  | { <name> boolean }+ | <name> <integerrange>
<enumeration>     ::= <name> { item }+
<implementation> ::= <name> <record> | <name>
<integerrange>    ::= integer integer
<name>            ::= string
```

Listing 9: Datenmodell in EBNF

Auf Grundlage dieser Grammatik entstanden die in Anhang A abgedruckten Schemata. Diese vollziehen die Abbildung des Datenmodells auf XML. Eines beschreibt die Struktur von Instanzdokumenten; sowohl der von Einzelinstanzen, als auch der von Serien. Ein weiteres Schema, sowohl für Domain- wie auch Record-Fragmente, beschreibt die Struktur in welcher das Modell vorzuliegen hat, – nicht dessen Inhalt.

Die Schemata sind in RelaxNG[11], einer XML-Schemasprache, die eine alternative, sehr stark an die ENBF angelehnte Syntax bietet, formuliert. Sie sind dadurch leicht lesbar und kompakt. Mit dem Parser Jing[12] kann gegen diese Schemata validiert werden. Der Parser gehört auch zum Umfang der eXist-Datenbank.

[11] http://relaxng.org/

[12] http://www.thaiopensource.com/relaxng/jing.html

6.5. Ressüme

Zusammenfassend kann festgehalten werden, dass das Datenmodell von TIRA semi-strukturiert ist. Aufgrund verschiedener ausgeführter Argumente bietet sich für diesen Fall eine Formulierung in XML an. Das Modell wurde zugunsten der Erweiterbarkeit, der Vermeidung von Redundanzen und der Übersichtlichkeit modular aufgebaut. Sowohl die Struktur der Modell-Fragmente, als auch die seiner Instanzdokumente wurde in Schemata gefasst.

Obwohl die eXIST-Datenbank schemafrei ist, wurde in diesem Kapitel dem Entwurf eines Schemas sehr viel Raum gegeben. Das hat verschiedene Gründe. Zum einen weißt TIRA ein datenintensives Modell auf. Alle Knoten der XML-Dokumente haben eine funktionale Bedeutung. Es gibt keinerlei Fließtext, aber numerische Angaben. Um der Datenbank eine effiziente Bearbeitung von Anfragen zu ermöglichen, sind entsprechende Indizes erforderlich, die aber nur über getypte Element- oder Attribut-Knoten angelegt werden können. Aus diesen Gründen erscheint es angebracht ein Schema einzuführen. Ob seiner abstrakten Ausgestaltung und der dadurch erreichten Unabhängigkeit von dem im Modell enthaltenen Wissen, erlaubt das vorgestellte Schema eine flexible Erweiterung des Modells.

Experimente gliedern sich in zwei beschreibende Dokumente. Eines für das Design, ein zweites für die Ergebnisse. Diese werden jeweils der entsprechenden Kollektion zugeordnet. Diese Instanzdokumente sind vollständig selbstbeschreibend. Für ihre Darstellung ist kein erneuter Zugriff auf das Modell erforderlich.

Die abstrakte Ausgestaltung des Datenmodells provoziert jedoch eine erhöhte Komplexität der Suchanfragen. Ein weniger abstraktes, teilweise Wissen enthaltendes Datenmodell könnte zumindest den Umfang solcher Anfragen verringern. Dies ginge allerdings zu Lasten der Generik und somit der flexiblen Erweiterbarkeit des Modells. Darüber hinaus wäre auch eine Implementierung der Darstellung im Client aufwendiger. Da die Suchanfragen in TIRA in aller

Regel ohnehin von Scripten generiert werden, und diese ebenfalls von einer Generik des Datenmodells profitieren, wurde dieser Weg nicht weiter verfolgt.

Eine andere Möglichkeit die Tiefe des XML-Baums von Instanzdokumenten zu verringern und somit Anfragen zu vereinfachen, bestünde darin das Element einer Domain und das Element eines, dieser Domain zugehörigen Members, zu einem Element mit weitaus mehr Attributen zusammenzufassen. Dieses Vorgehen erzeugt jedoch, insbesondere in Instanzserien, Redundanz. Auch würde die Expansion von Instanzserien komplizierter, da nicht mehr über die Kindknoten einer Domain iteriert werden kann, sondern Knotentests durchgeführt werden müssen.

Nicht zuletzt wäre der vollständige Verzicht auf jegliches Schema denkbar. Etwa indem die Elementnamen eines Instanzdokumentes frei wählbar sind und nur bestimmte erforderliche Attribute festgelegt werden. Ein derartiges Datenmodell würde sehr viel Wissen enthalten. Wissen, über das ein Worker ohnehin verfügen muss – der Client jedoch nicht zwangsläufig. Um diesem eine Darstellung solcher Instanzdokumente, ohne Zugriff auf das dahinter liegende Modell, zu ermöglichen, müssten Metainformationen hinzugefügt werden. Die vollständige Abwesenheit eines Schemas hätte aber auch die Folge, dass eine gezielte Indizierung seitens der Datenbank unmöglich würde.

Der vorgestellte Entwurf stellt eine Blaupause dar, die einen gangbaren Kompromiss zwischen den zahlreichen Anforderungen aufzeigt. Pragmatische Erwägungen im Zuge der Implementierung können zu vereinzelten Abweichungen führen.

Teil II.

Entwurf einer ressourcen-orientierten Architektur

7. ReST

Über ReST zu sprechen heißt immer zuerst darüber zu sprechen, was es *nicht* ist. Das hilft späteren Missverständnissen vorzubeugen. Es ist keine neue Technologie, kein Protokoll und auch keine Architektur.
Es ist ein kleiner, gut überschaubarer Satz von Prinzipien, die, wendet man sie an, zu einer ebenso einfachen Architektur führen können. Kurz, man kann von einem Architekturstil für verteilte Systeme sprechen.

Das Akronym ReST steht für Representional State Tranfer. Der Begriff wurde im Jahr 2000 von Roy Thomas Fielding in Kapitel 5 seiner Dissertation geprägt und sorgt seither für einiges an Turbulenzen im Feld der WebServices.

> »Die Welt der Web Services steuert kontinuierlich auf einen großen Knall zu - schon seit die Softwarearchitekten ein weiteres Mem erzeugt haben, den Pragmatismus zu verlassen und in die Welt der Unternehmen überzulaufen.«[13] [RR07]

Diese Aussage richtet sich an die klassischen, RPC[14]-artigen WebServices, sozusagen das gegnerische Lager, namentlich XML-RPC und SOAP. Diese verfolgen die Strategie dem Web das objektorientierte Modell überstülpen zu wollen. Es handelt sich also um Technologien oder, genauer, um Protokolle. Protokolle, die das Web zwar gebrauchen, aber nicht nutzten und stattdessen seine Möglichkeiten durch dicke Abstraktionsschichten verschütten.

ReST bedeutet einen konsequenten Bruch mit den bisherigen Ansätzen, WebServices zu realisieren, denn es setzt auf eine native und intensive Nutzung des HTTP-Protokolls und seiner Möglichkeiten. Dabei müssen die sogenannten

[13]Aus dem Vorwort von David Heinemeier Hansson
[14]Remote Procedure Call

ReSTful WebServices nicht zwangsläufig auf HTTP basieren; derzeit ist es aber das einzige existierende und praktisch angewandte Protokoll. Das mag nicht zuletzt auch darin begründet liegen, dass Roy Thomas Fielding einer der Hauptautoren der HTTP-Spezifikation ist, und die Anforderungen, die er an eine ReST-konforme Architektur stellt, dadurch maßgeblich beeinflusst wurden. So erscheint es zwangsläufig sinnvoll, dass er URI (Uniform Resource Identifier), HTTP und XML als die zu wählende Technologiebasis für eine entsprechende Architektur vorschlägt. [RR07]

Was genau fordert ReST nun aber? Bevor die einzelnen Prinzipien eingehender beleuchtet werden, bietet das folgende Zitat einen Überblick in konzentrierter Dosis.

> »Hinter ReST steckt die Idee der atomaren Verarbeitung von Ressourcen über ein Protokoll, mit einem festen Satz von Operationen (Interface) und einer Repräsentation (MIME-Type) wie XML oder JSON. Eindeutige universelle Bezeichner referenzieren Ressourcen. Zentrale Operationen auf ihnen sind die üblichen „create“, „read“, „change“ und „delete“. Ein Protokoll das ReST erfüllt, muss zudem verteilt, stateless, cacheable und layered sein.«[Lei09]

7.1. Prinzipien

Adressierbarkeit. ReST führt ein neues Paradigma ein. Vergleichbar der Aussage „Alles ist ein Objekt“, die aus objekt-orientierten Programmierung bekannt sein dürfte, lautet dieses Paradigma „Alles ist eine Ressource“. Eine Ressource ist eindeutig durch eine URI adressierbar. Ressourcen können über Links verknüpft werden und besitzen einen Zustand.

Zustandslosigkeit. Wird HTTP ReST-konform angewandt erbt die Anwendung die Eigenschaften des Protokolls, wie etwa die Client-Server-Architektur oder die Zustandslosigkeit.

Der Ressourcenzustand wird durch den Server verwaltet. Der Client bringt diesen Zustand ausschließlich durch die ihm zugestellte Repräsentation der Resource in Erfahrung.

Der Anwendungszustand ist die Position des Clients innerhalb der Anwendung. Die Verwaltung dieses Zustands ist durch den Client zu realisieren.

Im Gegensatz zu RPC-artigen WebServices hält der Server zu keinem Zeitpunkt Informationen über den Anwendungszustand. Das bedeutet unter anderem, dass der beliebte und weit verbreitete Gebrauch so genannter Sessions in einer entsprechenden Architektur nicht zulässig ist.

Wohldefinierte Operationen. Die Verben des HTTP-Protokolls dienen als einheitliches Interface. ReST folgt damit prinzipiell dem CRUD-Paradigma, demzufolge nur vier Methoden, nämlich „create“, „read“, „update“ und „delete“ erforderlich sind um beliebige Operationen auf Ressourcen ausführen zu können. Die im Kontext von ReST Verwendung findenden Verben sowie deren Bedeutung fast Tabelle 4 zusammen.

HTTP-Methode	CRUD-Verb	Semantik
sichere Methoden		
GET	Read	Ressource anfordern
HEAD	—	Metainformationen anfordern
OPTIONS	—	Auslistung der verfügbaren Verben
idempotente Methoden		
PUT	Create	Ressource erzeugen
DELETE	Delete	Ressource löschen
überladbare Methoden		
POST	Update	Ressource verändern

Tabelle 4: Die Bedeutungen der HTTP-Verben im ReST-Kontext

Hypermedia. Für die Ressourcen-Repräsentationen, welche zum Client übertragen werden, nannte Roy Fielding ursprünglich XML als Datenformat, wobei dies den Sonderfall von (X)HTML einschließt. Aber auch jüngere Formate wie JSON oder Atom finden zunehmend Anwendung. Eine bestimmte Repräsentation kann durch Setzten des Accept-Headers gefordert werden.

7.2. ReST-konforme Architekturen

Häufig wird der Begriff ReST auf das HTTP-Interface verkürzt, welches im Abschnitt „Wohldefinierte Operationen“ beschrieben ist. Dieser Umstand führt dazu, dass nicht alles *ReSTful* ist, was als solches deklariert wurde. Um dem ReST-Stil folgende Architekturen von den ReSTful-HTTP-Interfaces zu differenzieren sind verschiedene Begriffe geprägt worden.
Richardson und Ruby führen in ihrem Buch [RR07] den Begriff der „ressourcen-orientierten“ Architektur (ROA) ein. Anderenorts[15] wird das Akronym WOA für sogenannte „web-orientierte“ Architekturen ins Feld geführt. Beiden gemeinsam ist die Gegenüberstellung zu den service-orientierten Architekturen (SOA). Im Rahmen dieser Arbeit wird die Metapher der Ressourcenorientierung Verwendung finden, da diese das Augenmerk sehr viel mehr auf die übrigen Aspekte neben dem HTTP-Interface lenkt, als das etwa die Metapher der Web-Orientierung vermag.

7.3. Resümee

Die von Fielding postulierten Prinzipen erfreuen sich zunehmend großer Beliebtheit und haben Einfluss auf viele Entwicklungen genommen. Große Unternehmen wie etwa Google und Amazon stellen ihre WebServices ReSTful der Öffentlichkeit zur Verfügung. Mit dem Atom Publishing Protocol (RFC 5023) wurde ein ReST-konformes Protokoll für den standardisierten Austausch von

[15] http://restpatterns.org/

Blog-Inhalten spezifiziert und nicht zuletzt nehmen viele Entwicklungsplattformen wie beispielsweise JAVA mit der JERSEY-Spezifikation entsprechende Bestandteile in ihren APIs mit auf.
Die Beliebtheit rührt von den verschiedenen Vorzügen her, die dieser Ansatz, gerade im Gegensatz zu den RPC-artigen, zu bieten hat:

- Das einheitliche und generische Interface sichert ein erwartbares Verhalten beim Zugriff auf Ressourcen zu.
- Die Verwendung etablierter und stabiler Systeme stellt breite Unterstützung sicher.
- Gute Skalierbarkeit durch die Möglichkeit des Cachings und der Verwendung von Proxys.
- Die Möglichkeit der Filterung durch eine Firewall sichert vor Missbrauch.
- Die lose, flexible Kopplung von Diensten begünstigt beispielsweise die Entwicklung von Mashups[16].
- Standardisierte Status-Codes, etwa für die Behandlung von Fehlern.
- Die Unabhängigkeit von komplexen Highlevel-Protokollen und dadurch eine Unabhängigkeit von komplexer Entwicklungssoftware.

Die wichtigste Eigenschaft und damit der größte Vorzug jedoch liegt in der *Einfachheit* begründet.

Will man die Bedeutung des Begriffs ReST zu fassen versuchen, ist die ausschließliche Betrachtung im Kontext von WebServices zu kurz gefasst. Obwohl es sich um einen Gegenentwurf zu den klassischen WebService-Techniken handeln mag, vermag es erst dieser, WebServices auch tatsächlich, und nicht nur dem Namen nach, ins Web zu bringen. Das äußert sich nicht zuletzt darin,

[16] Eine kreative Wiederverwendung von Daten in ungewohnten Kontexten.

dass ein gewöhnlicher Webbrowser in der Lage ist, ReST-konforme WebServices unmittelbar zu nutzen. Durch die zugesicherte Erwartbarkeit des Verhaltens beim Zugriff auf eine Ressource kann auch ein menschlicher Nutzer intuitiv mit dieser interagieren. Somit führt ReST ein gespaltenes Web – das der Menschen und das der Maschinen – wieder zusammen. Jede Webseite ist zugleich WebService, wie auch jeder WebService zugleich eine Webseite. Diese Erkenntnis hat weitreichende Folgen für den Umgang mit dem Web – von der Maschinenlesbarkeit bis zur Metapher der Beschreibbarkeit für jedermann.

8. Architektur

In diesem Kapitel soll, nach einigen einführenden Worten zu Drei-Schichten-Architekturen, durch eine vergleichende Gegenüberstellung des klassischen und des für diese Arbeit gewählten Ansatzes zu einem ressourcen-orientierten Entwurf für die Architektur von TIRA hingeführt werden.

Die Architektur von TIRA wurde ursprünglich als klassischer Drei-Schichten-Entwurf konzipiert. Im Laufe dieses Kapitels wird ein Architekturentwurf vorgestellt, der diesem Konzept auf eine flexible Weise folgt, jedoch mehr den Charakter einer kreativen Verknüpfung von lose interagierenden WebServices – auch Orchestrierung genannt – aufweist. Jede Komponente in TIRA weist nach außen den Charakter eines ReST-konformen HTTP-WebServices auf, bis auf den im Browser implementierten Benutzer-Client. Der vorgestellte Entwurf ist somit die Synthese einer Drei-Schichten- und einer ressourcen-orientierten Architektur. Die beiden Konzepten inhärente Client-Server-Modellierung wird durch die WebServices in zueinander unterschiedlichen Rollen ausgeübt, jedoch ohne das Schichtenkonzept zu verletzen.

8.1. Drei-Schichten-Architektur

Drei-Schichten-Architektur, oder auch englisch Three-Tier Architecture, ist eine Ausprägung der n-Tier Architecture, also schlicht des Aufbaus einer Applikation durch das Aufschichten von einzelnen Komponenten. Es wird formal zwischen strikten und flexiblen Umsetzungen solcher Architekturen unterschieden. Diese Unterscheidung ergibt sich, wie in Abbildung 10 zu sehen ist, dadurch, dass sie den Zugriff einer höheren Schicht nur auf die unmittelbar darunter liegende Schicht zulassen oder auch darüber hinaus. n-Tier-Architekturen

folgen generell dem Prinzip des Zugriffs einer höher liegenden auf eine tiefer liegende Schicht, niemals jedoch umgekehrt.

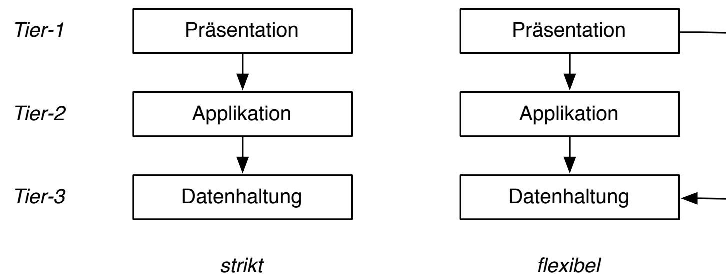

Abbildung 10: Ausprägungen der Drei-Schichten-Architektur [Bal05]

Im Falle der Drei-Schichten-Architektur haben sich eine Reihe synonym verwendeter Bezeichnungen für die einzelnen Schichten verbreitet. So spricht man bei Tier-1 auch von der Darstellung, bei Tier-2 auch von einem Fachkonzept oder Neudeutsch von BusinessLogic und Tier-3 wird auch, besonders im Zusammenspiel mit Objekten, als Persistenzschicht bezeichnet.

Klassischerweise werden, wie bereits in der Einleitung angeklungen, etablierte Systeme als Komponenten der einzelnen Schichten eingesetzt. Das bedeutet in der Praxis den Einsatz einer relationalen Datenbank für die Datenhaltung, die Verwendung einer objektorientierten Programmiersprache für die Implementierung der Applikationslogik und meist den Einsatz von XML für die Darstellung im Browser. Diese Technologieauswahl hat zur Folge, dass jede der drei Schichten ein eigenes Datenmodell erforderlich macht und zwischen diesen vermittelt werden muss. Für die Architektur bedeutet dieser Umstand die Einführung zusätzlicher Vermittlungsschichten, die inzwischen, ob ihrer häufigen Erfordernis, eigene Bezeichnungen erhalten haben. Zu nennen sind das OR-Mapping zwischen Applikationsschicht und Persistenzschicht und die Objektserialisierung zwischen Präsentation und Applikation. Somit lässt sich kaum noch von einer Drei-, sondern vielmehr von einer Fünf-Schichten-Architektur sprechen. Auf die existierenden Schwierigkeiten bei der Verwendung unterschiedlicher Datenmodelle geht Kapitel 6 kurz ein.

8.2. Architektur in der Metamorphose

Bevor der neue Architekturentwurf in seinen Details dargelegt wird, soll an dieser Stelle zunächst eine vergleichende Betrachtung angestellt werden. Dadurch sollte sich die Motivation für diese Arbeit leicht nachvollziehen lassen.

Beide Architekturen, sowohl die klassisch, als auch die hier vorgestellte, sind prinzipiell als Drei-Schichten-Architekturen konzipiert. Während das Ausgangssystem aber der strikten Umsetzung folgt, beschreitet das neue System den flexiblen Weg. Darüber hinaus wird es, durch den Wegfall des Zugriffs der Präsentations- auf die Applikationsschicht, wie in Abbildung 11 zu sehen ist, schwer, für die Existenz einer Drei-Schichten-Architektur zu argumentieren.

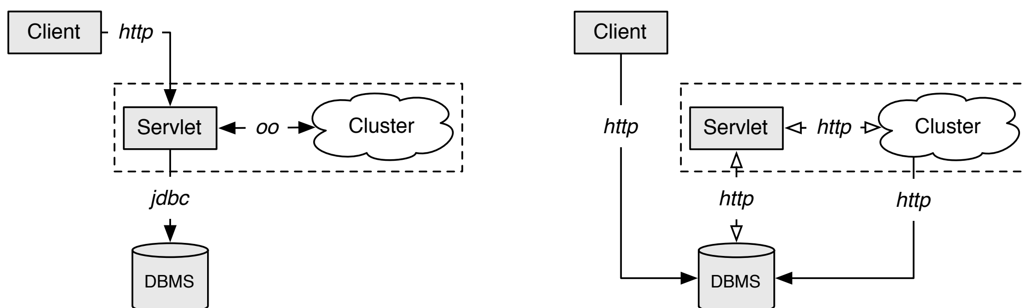

Abbildung 11: Architekturen im Vergleich
Links: Die Architektur des Ausgangssystemes
Rechts: Die Architektur nach der Transformation

Als ein Argument dafür, dass es sich auch bei dem neuen Entwurf um eine Drei-Schichten-Achitektur handelt, kann die Tatsache gelten, dass, wie in den Kapiteln 4.2 und 5.5 besprochen, eine native XML-Datenbank – und eXist im Besonderen – nicht nur eine Datenbank sein muss, sondern durchaus auch Applikationslogik zu implementieren im Stande sein kann. Weiterhin begünstigt der ausschließliche Einsatz von HTTP und XML die flexible und lose Kopplung der Komponenten. Da alle Komponenten diese Standards unterstützen, ist eine Vermittlung durch eine jeweils andere Schicht nicht erforderlich.

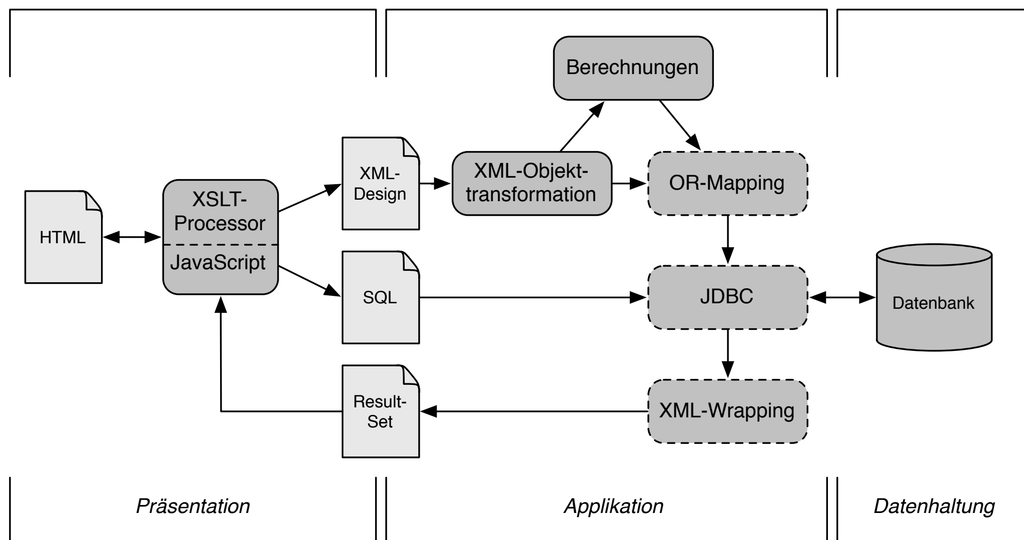

Abbildung 12: Reduzierte Ansicht des Datenflusses zwischen einzelnen Komponenten; Komponenten mit gestrichelter Umrandung entfallen durch den Austausch der Datenbank

Abbildung 12 geht detaillierter, obgleich stark vereinfacht, auf die einzelnen Komponenten der beiden Architekturen ein. Durch den Fokus der gegenüberstellenden Darstellung auf den Datenfluss und die damit verbundenen Transformationskomponenten sollte ein geschärfter Blick auf die Unterschiede und somit auf die erreichten Vereinfachungen möglich werden. Wie schnell zu erkennen ist, werden insbesondere auf der Applikationsebene die Komponenten stark ausgedünnt. Durch den Einsatz einer XML-Datenbank ist der Client nicht mehr auf die Datenbankschnittstelle der Applikationsschicht angewiesen und kann direkt auf die Datenhaltung zugreifen. Und auch die Applikationsschicht selbst ist nur noch auf eine Komponente für die Transformation sowohl eingehender wie auch ausgehender Daten angewiesen. Für die in dieser Darstellung nicht erkenntlichen Veränderungen hinsichtlich des Datenflusses zwischen den Komponenten sei auf die Abbildungen 13 und 14, die im Rahmen des folgenden Kapitels besprochen werden, verwiesen.

8.3. Entwurf

Zwei schwerwiegende Eingriffe bestimmen den hier vorgestellten Entwurf für Tira: Zum einen der Austausch der relationalen Datenbank gegen eine native XML-Datenbank, zum anderen die Kopplung aller Komponenten über ReST-konformes HTTP. Die allgemein zu erwartenden Auswirkungen dieser Eingriffe wurden in eigenen Kapiteln bereits diskutiert. Wie sich die Architektur von Tira nach den erfolgten Veränderungen darstellt, ist im Rahmen dieses Kapitels ausgeführt.

Durch die Anwendung von ReSTful HTTP lässt sich die Assemblage der Komponenten äußerst locker halten. Dieser Umstand erlaubt es, die Interaktionen der einzelnen Schichten in zwei voneinander unabhängigen Schritten nachzuvollziehen.

Nutzergetriebene Abläufe

Die derzeit implementierten Usecases umfassen das Erstellen und Speichern eines Experimentes sowie das spätere Einsehen und Durchsuchen der Resultate der durchgeführten Experimente. Diese Vorgänge werden maßgeblich durch den Nutzer beeinflusst. Wie bereits auch bei der Diskussion der Anforderungen an das Datenmodell in Kapitel 6.2 besprochen, wird die Benutzungsoberfläche in einem Webbrowser dargestellt. Neben dieser Aufgabe zeichnet der vollständig als Browser-Applikation realisierte Client für die Generierung valider wie auch plausibler Design-Instanzen und Anfrage-Querys verantwortlich. Zu diesem Zweck verfügt er über Modellwissen und Transformationslogik in Form von XSLT-Stylesheets, die die unterschiedlichen XML- und XPath- Outputs erzeugen, und mit deren Hilfe er die erhaltenen XML-Antwort-Dokumente zur Darstellung bringen kann.

Dank der ReST-konformen HTTP-Schnittstelle kann die Browser-Applikation von Schicht eins direkt mit der Datenbank, also der dritten Schicht, kommu-

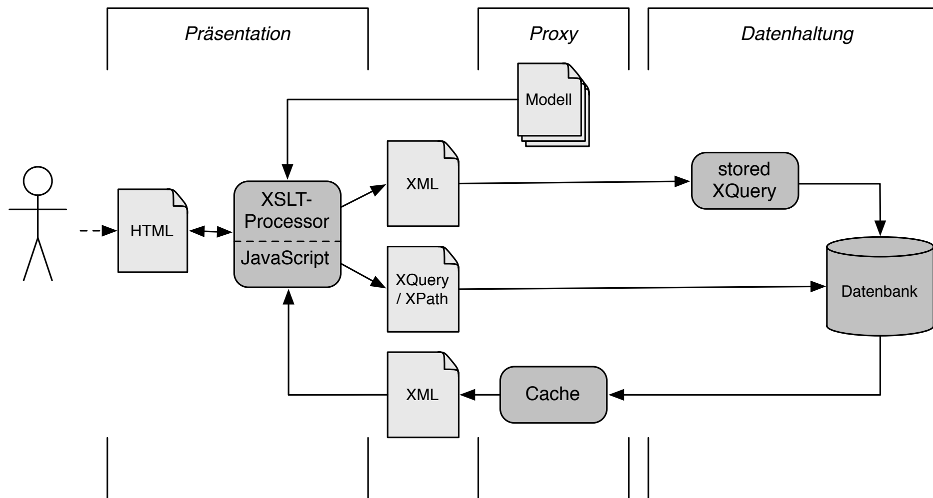

Abbildung 13: Komponenten und Datenfluss zwischen Schicht eins und drei.

nizieren. In produktiven Umgebungen ist es allerdings ratsam einen Apache-Webserver als Proxy zwischenzuschalten [eP09b]. Dieser kann auch genutzt werden, um das Modell für den Client bereitzustellen und das Caching für verschiedene statische Inhalte zu übernehmen. Beispielsweise wird sich das Design eines Experimentes nach dessen Speicherung in der Datenbank nicht mehr verändern.

Für die Speicherung von Experimentserien ist auf Seiten der Datenbank ein Script für die Expansion in einzelne Design-Instanzen erforderlich. Dieses wurde mit den von eXist im Rahmen des Einsatzes als ReST-Server gebotenen Möglichkeiten in Form eines Stored XQuery realisiert. Es ergänzt im Zuge der Expansion auch verschiedene Attribute wie den Erstellungszeitpunkt und eine ID. Abgesehen von dieser Ausnahme wird für den Zugriff des Clients auf die Datenbank ausschließlich das Standard-Interface des eXist-ReSTServers ohne Anpassung eingesetzt.

Die besprochenen Architekturschichten, deren Komponenten und der zwischen ihnen stattfindende Datenaustausch werden in Abbildung 13 zusammenfassend zur Darstellung gebracht.

Autonome Systeminteraktionen

In den vorangegangenen Ausführungen wurde die Durchführung der Experimente und die Speicherung ihrer Resultate nicht angesprochen. Diese Aktivitäten werden von Tira autonom, ohne Zutun des Nutzers, wahrgenommen. Innerhalb einer Drei-Schichten-Architektur greift, wie bereits besprochen, nur eine höher liegende Schicht auf eine tiefer liegende zu. Kenntnis von neuen Experimenten hat zunächst aber nur die Datenbank und nicht die für deren Durchführung zuständige Applikationsschicht. Folglich müsste die Applikation in regelmäßigen Intervallen die Verfügbarkeit neuer unbearbeiteter Experimente – im weiteren als Jobs bezeichnet – bei der Datenbank anfragen. Um den aus dieser Pull-Strategie resultierenden Kommunikations-Overhead zu vermeiden, erscheint die Anwendung des Observer-Patterns zweckdienlich. Da die Datenbank die Applikation somit lediglich über die Verfügbarkeit neuer Jobs informiert und diese nicht sogleich weiterleitet, ist das Konzept der Schichten-Architektur nicht durch eine etwaige Push-Strategie durchbrochen (siehe auch Abbildung 11, rechts).

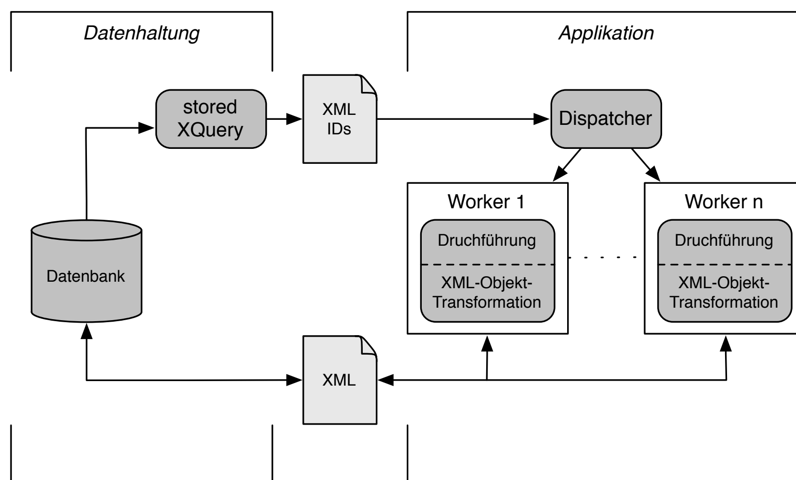

Abbildung 14: Komponenten und Datenfluss zwischen Schicht zwei und drei.

Aufgrund der hohen Anforderungen an Ressourcen für die Ausführung der Information-Retrieval-Algorithmen, mit denen in TIRA experimentiert wird, werden diese Experimente auf einem Cluster durchgeführt. Wie Abbildung 14 veranschaulicht, werden die freien Jobs zunächst, nach Benachrichtigung durch die Datenbank, von einem Verteilungsknoten – im folgenden Dispatcher genannt – abgerufen. Dieser verwaltet seinerseits eine Liste verfügbarer Rechenknoten in dem Cluster, welche im folgenden als Worker bezeichnet werden. Die Aufgabe des Dispatchers besteht ausschließlich darin, die IDs offener Jobs zu beziehen und an ihm bekannte freie Worker zu delegieren. Um die IDs offener Jobs von der Datenbank beziehen zu können, greift er auf einen Stored XQuery zu. Jeder Worker beschafft sich die zur Durchführung des ihm zugeteilten Experimentes erforderlichen Daten selbst von der Datenbank. Durch die ihm übermittelte ID des Jobs kann er die URI der zugehörigen Ressource konstruieren und die Design-Instanz direkt abrufen. Nach der Durchführung des Experimentes schreibt der Worker das die Resultate enthaltende Dokument unter gleicher ID in die Kollektion für Resultate und trägt sich wieder in die Liste des Dispatchers ein. So besteht auch zwischen Dispatcher und Workern eine Kommunikation nach dem Observer-Pattern. Diese doppelte Anwendung des Patterns ist erforderlich, da die Applikationsschicht der Datenbank keine adäquaten Mittel für die Handhabung konkurrenter Zugriffe bietet. Wohl beherrscht sie das Locking bei konkurrenten Schreibzugriffen auf die selbe Ressource, jedoch würde eine exklusiver Lesezugriff auf den Stored XQuery benötigt, um den Dispatcher in der Applikationsschicht der Datenbank implementieren zu können. Sämtliche Kommunikation zwischen Workern, Dispatcher und auch der Datenbank finden über ReST-konformes HTTP statt. Es kann somit von einem *ReSTful HTTP-Cluster* gesprochen werden.

Das in Abbildung 15 dargestellte Sequenzdiagramm zeichnet ein detailliertes Bild der zuvor besprochenen Interaktionen zwischen den WebServices. Es stellt alle Request-Response-Zyklen während der Bearbeitung eines Experimentes in ihrer zeitlichen Abfolge dar.

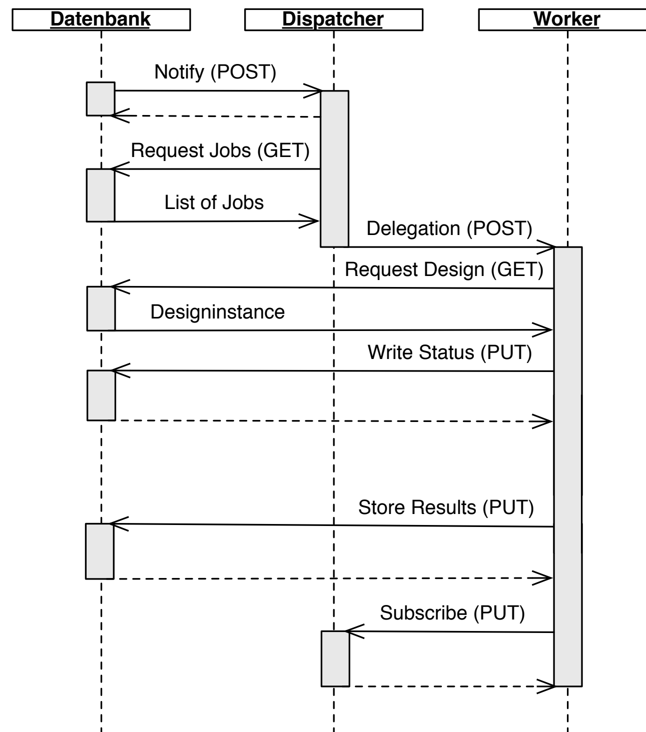

Abbildung 15: Sequenzdiagramm der Kommunikation zwischen den WebServices während der Durchführung eines Experimentes.

Als Ressourcen werden in TIRA vorrangig die Experiment-Designs und -Resultate beschreibenden XML-Dokumente verstanden, was leicht nachvollziehbar ist. Aber auch Worker sind als Ressourcen aufzufassen, wenn sie sich bei dem Dispatcher in dessen Kollektion freier Rechenknoten eintragen und dieser auf sie zugreift.

8.4. Resümee

Im Zuge dieses Kapitels wurde der Entwurf einer ressourcen-orientierten Architektur für TIRA skizziert. Es wurden die verschiedenen WebServices, deren Interaktion und Teilkomponenten beschrieben.

Obwohl dieser Entwurf weit mehr den Charakter einer Orchestrierung von WebServices aufweist, kann dennoch auch von einer Drei-Schichten-Architektur gesprochen werden, insbesondere unter Berücksichtigung der Tatsache, dass Stored XQuerys nichts anderes als eine der Datenbank angelagerte Applikationsschicht sind. Die Anwendung des Observer-Patterns stellt sicher, dass das Konzept der Schichten-Architektur nicht gebrochen wird.

Durch die einheitliche Schnittstelle der WebServices, das abstrakt gehaltene Datenmodell und die schemafrei arbeitende Datenbank lässt sich das entworfene System flexibel und einfach erweitern. Um weitere Experimente zu ermöglichen, müssten das Modell durch hinzufügen entsprechender XML-Fragmente erweitert und ein Worker, der diese Experimente durchführen kann, implementiert werden. Alle anderen Komponenten des Systems bedürfen keinerlei Anpassungen. Worker sind zu diesem Zweck als eine abstraktes Interface spezifiziert. In welcher Sprache und auf Basis welcher Bibliotheken diese implementiert werden, ist somit der pragmatischen Entscheidung des Entwicklers überlassen.

Die Spezifikation der WebServices und insbesondere ihrer Schnittstellen findet sich in Kapitel 9.

9. Spezifikation

Im folgenden werden die Schnittstellen der verschiedenen WebServices von Tira beschrieben. Diese Spezifikation ist nicht als abgeschlossen zu betrachten. Insbesondere das direkte Ansprechen von XQuery-Scripten ist, im Kontext des Anspruchs auf ReST-Konformität, zu hinterfragen. Eine Lösung dieser Fragestellung könnte darin bestehen, das URL-Rewriting von eXist oder das des Apache-WebServers zu nutzen. Dadurch würde es beispielsweise möglich, die Aktualisierung einer Ressource – wie gewünscht – unmittelbar auf dieser auszuführen, und nicht auf einem Stored XQuery oder durch das Übermitteln des XQuery-Scripts.

Die Tatsache, dass server-seitige Fehler (500er) nicht aufgeführt sind, bedeutet nicht, dass diese nicht auftreten können. Sie werden jedoch häufig durch die verwendeten Komponenten bereits implementiert. Für durch die Anfrage des Clients ausgelöste Fehler (400er) gilt, dass auch numerisch niedrigere als die aufgeführten ausgelöst werden können.

9.1. Datenbank

Auch die Datenhaltung in Tira ist als ein WebService zu begreifen. Die von diesem Persistenz-Service bereitgestellten Dienste lassen sich unter den folgenden Adressschemata erreichen:

- Im Falle eines Deployments als Servlet:

```
http://<host>:8080/eXist/rest/
```

- Im Falle eines Deployments als Standalone-Server:

```
http://<host>:8088/
```

Die in der folgenden Spezifikation angegebenen Adressfragmente der Ressourcen sind an das entsprechende Schema anzuhängen. Die derzeit unterhalb der Kollektion `/tira` befindlichen Kollektionen sind `/designs` und `/results`.

Schreiben von Design-Instanzen wird ausschließlich durch den Client (Browser) ausgeführt. Der Stored XQuery expandiert übermittelte Experimentserien und fügt den Erstellungszeitpunkt sowie Bearbeitungsstatus und eine UUID[17] als Attribut in das Top-Level-Element ein. Nach der Speicherung der einzelnen Instanzdokumente unter den jeweils zugewiesenen UUIDs gefolgt von der Dateiendung `.xml` wird dem Client ein XML-Dokument übermittelt, das die neu erstellten Ressourcen auflistet, sodass der Browser in die Lage versetzt wird, dem Nutzer eine Aufstellung dieser darzureichen.

Request	HTTP-Verb	POST
	Ressource	/tira/designs/write.xql
	Body	XML: Instanz(-Serie)
Response	Status-Code	200(OK), 400(Bad Request)
	Body	XML: Liste von angelegten Ressourcen

Queryanfragen werden vom Client (Browser) an die Datenbank gerichtet. Das Standardverhalten des ReST-Servers von eXist erlaubt die Übermittlung einer XQuery-Anfrage im POST-Body. Die Anfrage ist an die zu durchsuchende Kollektion zu richten – meist `/designs`. Die Antwort enthält eine Auflistung der URIs der in der Ergebnismenge enthaltenen Ressourcen. Es ist die Aufgabe des Clients, über weitere Leseoperationen Detailinformationen nachzuladen. Alternativ steht die Überlegung zur Disposition, das client-seitig entstehende partielle Instanzdokument, aus welchem die Anfrage generiert werden würde, an einen Stored XQuery (`search.xql`) zu übermitteln. So würde die Anfrage

[17]Universally Unique Identifier

server-seitig generiert und das Ergebnis bereits aufbereitet zurückgeben werden können. Eine solche Anfrage könnte an die Kollektion `/tira` gerichtet werden, um auszudrücken, dass die Experimente in ihrer Gesamtheit durchsucht werden. Zwar wird nach bestimmten Design-Instanzen gesucht, jedoch werden deren Results zurückgegeben.

Request	HTTP-Verb	POST
	Ressource	/tira/<collection>/
	Body	XQuery: Anfrage-Script
Response	Status-Code	200(OK), 400(Bad Request)
	Body	XML: Liste von zutreffenden Ressourcen

Lesen von Dokumenten gehört zum Standard des ReST-Servers von eXist. Benötigt wird es sowohl vom Client als auch von den Workern. Leseoperationen werden in Tira immer an eine konkrete Ressource, nicht an eine Kollektion gerichtet.

Request	HTTP-Verb	GET
	Ressource	/tira/<collection>/<UUID>.xml
	Body	—
Response	Status-Code	200(OK), 404(Not Found)
	Body	XML: Ressource/Instanzdokumente

Schreiben von Result-Dokumenten wird durch die Worker vorgenommen. Die Schnittstelle gehört zum Standard von eXist. Result-Dokumente werden unter der selben UUID gespeichert wie die zugehörigen Design-Dokumente, jedoch in der Kollektion `/results`. Um in der Datenbank einen Bearbeitungsstatus zu hinterlegen, schreibt der Worker zunächst ein leeres Resultat mit der Metainformation *progress* und überschreibt dieses nach Abschluss der Berechnungen mit den erhaltenen Resultaten und dem Status *done*.

Request	HTTP-Verb	PUT
	Ressource	/tira/results/<UUID>.xml
	Body	XML: Dokument mit Resultaten
Response	Status-Code	201(Created), 400(Bad Request)
	Body	—

Jobs müssen von dem Dispatcher abgerufen werden, um diese auf die Worker zu verteilen. Ein Stored XQuery durchsucht die gespeicherten Design-Instanzen nach solchen, deren Status mit *open* gekennzeichnet ist. Zurückgegeben wird ein XML-Dokument, welches die URIs der in der Bearbeitung noch offenen Ressourcen enthält.

Request	HTTP-Verb	GET
	Ressource	/tira/designs/jobs.xql
	Body	—
Response	Status-Code	200(OK), 404(Not Found)
	Body	XML: Liste von Ressourcen-URIs

9.2. Dispatcher

Der Dispatcher dient der Verteilung von durchzuführenden Experimenten auf freie Worker. Er ist als JAVA-Servlet implementiert. Die Inanspruchnahme einer API für ReSTful WebServices, wie etwa JERSEY, sollte erwogen werden.

Benachrichtigung des Dispatchers über die Verfügbarkeit neuer Jobs durch die Datenbank. Die Nachricht sollte keinen Inhalt besitzen. Sollte POST anderweitig ebenfalls benötigt werden, kann die URI erweitert werden.

Request	HTTP-Verb	POST
	Ressource	<Servlet-URI>
	Body	—
Response	Status-Code	202(Accepted), 400(Bad Request)
	Body	—

Anmeldung eines freien Workers beim Dispatcher unter Angabe einer eindeutigen Identifikation, beispielweise der IP-Adresse. Für den Fall, dass zukünftige Weiterentwicklungen unterschiedliche Worker erlauben, sollten Informationen bezüglich der durchführbaren Experimente bei der Anmeldung übermittelt werden.

Request	HTTP-Verb	PUT
	Ressource	<Servlet-URI>/workers/<IP-Adresse>
	Body	XML: IP-Adresse (und Worker-Typ)
Response	Status-Code	201(Created), 400(Bad Request)
	Body	—

Abmeldung eines Workers für den Fall, dass dieser abgeschaltet wird oder aus anderen Gründen nicht mehr zur Verfügung steht. Bei Zuteilung eines Jobs wird der Worker durch den Dispatcher abgemeldet und muss nicht selbst handeln.

Request	HTTP-Verb	DELETE
	Ressource	<Servlet-URI>/workers/<IP-Adresse>
	Body	—
Response	Status-Code	200(OK), 400(Bad Request)
	Body	—

9.3. Worker

Worker dienen der Durchführung von Experimenten und agieren überwiegend als Client. Die einzige Schnittstelle dient der Zuteilung von Jobs. Die Anwendung spezieller APIs ist hierfür nicht erforderlich.

Aufrufen eines Workers von Seiten des Dispatchers. Nach erfolgreicher Übermittlung der URI des dem Job zugehörigen Experimentdesigns sollte der beauftragte Worker durch den Dispatcher aus der Liste freier Worker ausgetragen werden.

Request	HTTP-Verb	POST
	Ressource	<Worker-URI>
	Body	XML: Ressourcen-URI
Response	Status-Code	202(Accepted), 400(Bad Request)
	Body	—

10. Umfeld

In diesem Kapitel sei auf zwei andere Technologie-Basen verwiesen, die alternative Ansätze für Tira bieten könnten und ebenfalls zu einer ressourcenorientierten Architektur führen würden. Es gibt verschiedene Gründe, weshalb diese Möglichkeiten im Rahmen dieser Arbeit nicht genutzt wurden. Im ersten Fall, da sich der XForms-Standard derzeit keiner großen Unterstützung seitens der Webbrowser erfreut. Im zweiten Fall, da es die Einführung eines vollständig anderen Datenmodells bedeuten würde. – Was einen zweiten vollständig anderen Entwurf bedeuten würde.

XRX-Architektur

Hierbei handelt es sich um eine im Jahr 2008 von Dan McCreary in seinem Artikel „XRX: Simple, Elegant, Disruptive“ [McC08] vorgeschlagene Architektur für Webanwendungen. Das Akronym XRX steht für XForms im Client, ReST – hier auf das Interface verkürzt – und XQuery auf Serverseite. Durch die ausschließliche Verwendung von XML-Technologien wird XRX als übersetzungsfreie Architektur, beziehungsweise im Englischen als Zero-Translation-Architectur bezeichnet. Darüber hinaus zeichnet sie einen gangbaren Weg hin zu einer rein deklarativen Programmierung. Die starke Ähnlichkeit zu der hier für Tira vorgeschlagenen Architektur liegt darin begründet, dass die Idee aus einem Artikel über XRX geboren wurde [Lei09]. Die abweichenden Design-Entscheidungen haben pragmatische Ursachen. Zum einen greift Tira auf die bestehende Java-Bibliothek AiTools zurück. Hier stellten sich Eingriffe, die über die schlichte Tarnung als WebService mit ReST-Schnittstelle hinausgingen, als indiskutabel dar. Zum anderen werden XForms derzeit von keinem Browser nativ unterstützt. Für den Internet Explorer und Firefox existieren

experimentelle Plugins. Verschiedene JavaScript-Bibliotheken erlauben die dynamische Nachrüstung solcher Fähigkeiten. Da aber der existierende Client von TIRA selbst bereits in der Lage ist das XML-Model in ein dynamisches HTML-Formular umzuwandeln, wurde von einer erneuten Implementierung Abstand genommen. Sollte sich der XForms-Standard in naher Zukunft aber zunehmend nativer Unterstützung seitens der Browser erfreuen, erscheint eine neuerliche Betrachtung angeraten. XForms erlaubt das deklarative Erstellen dynamischer Formulare in einem MVC-Pattern. Zu diesem Zweck erlaubt es das Laden eines XML-Dokumentes als Model, das Binden einzelner Elemente an Formularfelder und das Definieren von Beschränkungen auf diesen sowie die Generierung eines Output-Dokumentes. Kurz, XForms bieten in einem Standard eine vollständige Lösung aller Anforderungen, die TIRA an den Client richtet. Das wäre ein nicht von der Hand zu weisender Vorzug gegenüber dem derzeit notwendigen, aber nicht trivialen Verknüpfen so unterschiedlichster Technologien wie XSLT und JavaScript.

JSON statt XML

Ein ganz anderer Ansatz ließe sich verfolgen, indem das Datenmodel nicht in XML, sondern in JSON formuliert würde. Die JavaScript Object Notation ist eine leichtgewichtige Methode zur Serialisierung von Objektinformationen. Dieses Vorgehen würde ein rein objekt-orientiertes Datenmodell erlauben. Im Client könnte dieses Datenmodel unmittelbar durch JavaScript verarbeitet werden und auch durch das Servlet wäre eine unmittelbare Instanziierung der Objektdaten möglich. Für die Datenhaltung läge der Einsatz eines Systems aus dem recht jungen Feld der verteilten Datenbanken nahe. Diese nach einem MapReduce-Verfahren arbeitenden Datenbanken setzten häufig JSON als Datenformat ein, bieten ein ReST-konformes HTTP-Interface und erlauben es Anfragen in JavaScript zu formulieren. Ihre Stärken können diese Systeme allerdings auch erst in einem verteiltem Zustand ausspielen. Eine aktuelle Aufstellung verteilter Datenbanken findet sich in [Mül10].

11. Resümee

Eine zentrale Zielsetzung wurde für die Motivation zu dieser Arbeit formuliert: *Einfachheit*. Einfachheit, die sich vor allem in einem „Weniger“ niederschlagen sollte. Weniger Datenmodelle, weniger Transformationen, weniger Programmcode und letztlich weniger Komplexität. Um diese Zielsetzung zu erreichen wurden zwei massive Eingriffe in die Architektur von TIRA motiviert und vorgenommen.

Die Einführung einer dem ReST-Stil folgenden sogenannten ressourcen-orientierten Architektur schafft einheitliche Schnittstellen. Vermittlungsdienste zwischen Komponenten werden dadurch obsolet. Die konsequente Anwendung der ReST zugrunde liegenden Prinzipien ermöglicht eine flexible Orchestrierung der als WebServices realisierten Komponenten von TIRA.

Der Austausch der relationalen Datenbank gegen die native XML-Datenbank eXIST führt dies fort, indem auch sie über eine ReST-konforme HTTP-Schnittstelle verfügt und sich die dokumenten-orientierte Speicherung der XML-Daten nahtlos in die Ressourcen-Metapher von ReST überführen lässt. Darüber hinaus ebnet die native Speicherstrategie für XML den Weg zu einer Architektur, welche sich ausschließlich auf ein in XML formuliertes Datenmodell stützen kann. Die Freiheit von einem Schema erlaubt die flexible Erweiterung des Datenmodells.

Der Umfang der Arbeit ist trügerisch, ob ihrer umfassenden Einführung in die ihr zugrunde liegenden Konzepte und Technologien. Niemand sollte daraus schließen, dass der vorgestellte Entwurf seinem Ziel – der Einfachheit – nicht gerecht wird. Die Anzahl der Datenmodelle wurde auf eines reduziert. Einzig an der Schnittstelle zwischen TIRA und den eingesetzten Bibliotheken, wie derzeit die AITOOLS, besteht noch der Bedarf einer Übersetzung. Das geschieht

aber an den Grenzen von Tira. Somit wurde auch die Anzahl an Transformationen zwischen unterschiedlichen Datenmodellen auf praktisch Null reduziert. Dadurch, aber auch durch die Überflüssigkeit von Vermittlungsdiensten zwischen zueinander inkompatibler Schnittstellen, wird viel Programmcode obsolet. Das alles führt, wie in der Motivation in Kapitel 2 ausführlich dargelegt wurde, zu weniger Fehlern und somit zu weniger Komplexität, beziehungsweise umgekehrt.

Die im Rahmen dieser Arbeit entworfene Architektur und das zugehörige Datenmodell wurden in einer Spezifikation der Programmierschnittstellen, beziehungsweise in Schemata, fixiert. Es handelt sich somit um eine Blaupause für Tira. Diese umfasst die vollständige Transition des Bestands in die neue Architektur, wie auch die Erweiterung um Experimentserien. Erste Schritte bei der Implementierung lassen vermuten, dass die hohen Erwartungen nicht enttäuscht werden dürften.

> »Wenn wir uns klarmachen, dass der Kampf gegen Chaos, Durcheinander und unbeherrschte Kompliziertheit eine der größten Herausforderungen der Informatik ist, müssen wir zugestehen: Beauty is our Business.«[18]

Dennoch bleibt ein zwiegespaltener Eindruck zurück. Obwohl gezeigt werden konnte, dass sich XML als Datenmodell für Tira hervorragend eignet, stellt sich die Frage, ob eine andere Modellierung der Daten einen besseren Kompromiss zwischen den Anforderungen finden könnte. Etwa hinsichtlich der Formulierung von Suchanfragen, würde ein Schema mit mehr Wissen in den Elementnamen zu einfacheren XPath-Konstruktionen führen, allerdings um den Preis des Verlusts der Generik und, daraus resultierend, einem höherem Aufwand für die Darstellung.

Obgleich sich eXist und eine ressourcen-orientierte Architektur in idealer Weise ergänzen, erzwingt die Umsetzung in Tira ein Abwägen zwischen einem

[18]Edsger W. Dijkstra, 1978

Overhead an HTTP-Anfragen und einer nicht vollständig ReST-konformen Umsetzung der Schnittstellen. Um unnötig viele HTTP-Anfragen zu vermeiden, bleibt nur die Möglichkeit das einzige ReST-konform überladbare Verb POST intensiv zu nutzen. Das aber führt zu einer Untergrabung der Erwartbarkeit hinsichtlich des Verhaltens einer Ressource.

Ausblick

Viele Fragen können im Rahmen dieser Arbeit nur unbefriedigend oder gar nicht beantwortet werden. Die dem Entwurf inhärente Einfachheit und Flexibilität sollte jedoch einen guten Nährboden für zukünftige Weiterentwicklungen bereiten. Manche der im folgenden aufgeführten Punkte wurden in der Arbeit bereits angedacht, jedoch, ob der begrenzten Zeit, nicht umgesetzt. Als da wären zu nennen:

- Die Umsetzung eines URI-Rewritings, um die HTTP-Schnittstelle intuitiver zu gestalten, insbesondere um Operationen, die durch XQuery-Scripte auf Ressourcen aufgeführt werden, auf den Ressourcen und nicht auf den Scripten aufrufen zu können.
- Die Realisierung der verteilten Durchführung von Experimenten, also die konkrete Ausgestaltung des ReST-Clusters.
- Fragen des Deployments für den Produktiveinsatz mit Augenmerk auf Sicherheit und Stabilität.
- Ein Plug-In-Konzept für Erweiterungen Dritter. Ein solches müsste das Modell um weitere XML-Fragmente bereichern und einen zugehörigen Worker implementieren.
- Plattformabhängigkeit des Clients. Dieser ist derzeit auf den Browser Firefox angewiesen.
- Die Implementierung des Clients auf Basis von XForms, sobald eine Unterstützung auf breiter Basis steht.

- Die Erweiterung des Datenmodells für die Ergebnisse der Experimente. Die Struktur der Ergebnisse in ein Modell zu fassen, würde eine automatische Analyse selbiger ermöglichen.
- Die Erweiterung des Systems um automatische Analysen. In diesem Zusammenhang wäre zu überdenken, ob die Resultate einer Analyse tatsächlich gespeichert werden sollten. Es bestünde auch die Möglichkeit, die Datenbasis einer Analyse in Form des Such-Query zu speichern und das Resultat auf einer immer aktuellen Datenbasis dynamisch zu berechnen.
- Die Implementierung von Mechanismen für den Mehrbenutzerbetrieb.
- Verschiedene Mashups wären denkbar. Etwa ein Newsfeed, welcher über den Bearbeitungsstatus der Experimente eines Nutzers informiert.

Literatur

[Bal05] Heide Balzert. *Lehrbuch der Objektmodellierung*. Elsevier, Spektrum Akademischer Verlag, München, 2005.

[Bou05a] Ronald Bourret. Going Native: Use Case for native XML Databases. http://www.rpbourret.com/xml/UseCases.htm, March 2005. (Letzter Zugriff 18.11.2009).

[Bou05b] Ronald Bourret. XML and Databases. http://www.rpbourret.com/xml/XMLAndDatabases.htm, September 2005. (Letzter Zugriff 24.01.2010).

[Dod01] Leigh Dodds. XML and Databases? Follow Your Nose. http://www.xml.com/lpt/a/861, October 2001. (Letzter Zugriff 18.11.2009).

[eP09a] The eXist Project. Configuring Database Indexes. http://www.exist-db.org/indexing.html, November 2009. (Letzter Zugriff 01.02.2010).

[eP09b] The eXist Project. Production use: Good Practice. http://www.exist-db.org/production_good_practice.html, September 2009. (Letzter Zugriff 04.03.2010).

[Fie00] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[FS04] Bernd Fröhlich and Jan Springer. Vorlesung in Programmiersprachen und Softwareentwurf, Wintersemester 2003/04.

[Gal77] John Gall. *Systemantics: How Systems Work and Especially How They Fail*. Quadrangle, May 1977.

[KM03] Meike Klettke and Holger Meyer. *XML & Datenbanken*. dpunkt.verlag, 2003.

[KST02] Wassilios Kazakos, Andreas Schmidt, and Perter Tomczyk. *Datenbanken und XML*. Xpert.press. Springer, 1. edition, 2002.

[Lei09] Christoph Leisegang. Triangulär. *iX*, 5:168–172, May 2009.

[MC04] Ntima Mabanza and Jim Chadwick. A comparison of open source Native Xml Database products. Southern African Telecommunication Networks and Applications Conference, SATNAC, September 2004.

[McC08] Dan McCreary. XRX: Simple, elegant, disruptive. http://www.oreillynet.com/xml/blog/2008/05/xrx_a_simple_elegant_disruptiv_1.html, May 2008. (Letzter Zugriff 18.11.2009).

[Mei06] Wolfgang Meier. Index-driven XQuery processing in the eXist XML database. http://www.exist-db.org/xmlprague06.html, March 2006. (Letzter Zugriff 01.02.2010).

[Mei09] Wolfgang Meier. Tuning the Database. http://www.exist-db.org/tuning.html, November 2009. (Letzter Zugriff 01.02.2010).

[Mül10] Frank Müller. Es geht auch einfach: Datenbanken ohne SQL und Relationen. *iX*, 2:122–126, February 2010.

[RR07] Leonard Richardson and Sam Ruby. *WebServices mit REST*. O'Reilly, 1. edition, 2007.

[Sta01] Kimbro Staken. Introduction to Native XML Databases. http://www.xml.com/pub/a/2001/10/31/nativexmldb.html, October 2001. (Last Access 18.11.2009).

[Tan09] Andrew Stuart Tanenbaum. Editorial: Warum sind Computer so unzuverlässig? *Linux Magazin*, 2009/01, January 2009.

[W3S] W3Schools. Introduction to XQuery. http://www.w3schools.com/xquery/xquery_intro.asp. (Letzter Zugriff 01.02.2010).

A. Schemata

RelaxNG-Schema für Modell-Fragmente (Domains wie auch Records):

```
grammar {

start = (domain | record)

record =
    element record {
        attribute implementation { xsd:string },
        element domain {
            attribute reference { xsd:string },
            element label { xsd:string }
        }
    }

domain =
    element domain { (iface | enum | prim ) }

iface =
    element interface {
        attribute domain { xsd:string },
        attribute label { xsd:string },
        attribute class { xsd:string },
        element implementation {
            attribute name { xsd:string },
            attribute label { xsd:string },
            attribute class { xsd:string },
            attribute record { xsd:boolean }
        }+
    }

enum =
    element enumeration {
        attribute domain { xsd:string },
        attribute label { xsd:string },
        element item {
```

```
            attribute label { xsd:string },
            xsd:string
        }+
    }

prim =
    element primitive {
        attribute domain { xsd:string },
        attribute label { xsd:string },
        (bool | int | intrange | float)
    }

bool =
    attribute type { "boolean" },
    element boolean {
        attribute label { xsd:string },
        xsd:boolean
    }+

int =
    attribute type { "integer" },
    element integer { xsd:integer }

intrange =
    attribute type { "integerrange" },
    element integerrange {
        attribute min { xsd:integer },
        attribute max { xsd:integer }
    }

float =
    attribute type { "real" },
    element real {xsd:double}

}
```

Listing 10: Schema für Modell-Fragmente

RelaxNG-Schema für Instanzdokumente (Einzel- wie Serienexperimente):

```
grammar{

start =
    element experiment {
        attribute uuid { xsd:string },
        attribute author { xsd:string },
        attribute project { xsd:string },
        attribute timestamp { xsd:dateTime },
        attribute workertype { xsd:string }?,
        attribute status { "open" | "dispatched" | "progress" |
            "done" },
        element design { iface }
    }

record =
    (iface | enum | prim )*

iface =
    element interface {
        attribute domain { xsd:string },
        attribute label { xsd:string },
        attribute class { xsd:string },
        element implementation {
            attribute name { xsd:string },
            attribute label { xsd:string },
            attribute class { xsd:string },
            attribute record { xsd:boolean },
            record
        }+
    }

enum =
    element enumeration {
        attribute domain { xsd:string },
        attribute label { xsd:string },
        element item {
```

```
            attribute label { xsd:string },
            xsd:string
        }+
    }

prim =
    element primitive {
        attribute domain { xsd:string },
        attribute label { xsd:string },
        (bool | int | intrange | float)
    }

bool =
    attribute type { "boolean" },
    element boolean {
        attribute label { xsd:string },
        xsd:boolean
    }+

int =
    attribute type { "integer" },
    element integer { xsd:integer }

intrange =
    attribute type { "integerrange" },
    element integerrange {
        attribute min { xsd:integer },
        attribute max { xsd:integer }
    }

float =
    attribute type { "real" },
    element real {xsd:double}

}
```

Listing 11: Schema für Instanzdokumente.