Institut für Parallele und Verteilte Systeme
Abteilung Anwendersoftware
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3740

Regelbasiertes Pattern-Mapping von Mashup Plans

Baris Kalyoncu

Studiengang:	Informatik
Prüfer/in:	PD Dr.B.Mitschang
Betreuer/in:	Dipl.-Inf. Pascal Hirmer
Beginn am:	15. Februar 2016
Beendet am:	16. August 2016
CR-Nummer:	D.2.13, H.2.5, H.2.8, I.6.7

Inhaltsverzeichnis

Abbildungsverzeichnis

Verzeichnis der Listings

1 Einleitung

Der Einsatz von Mashup Applikationen hat in den letzten Jahren zunehmend an Bedeutung gewonnen. Unternehmen bedienen sich Anwendungen wie **Yahoo Pipes**, **IBM MashupHub** oder **Intel Mashmaker**, um unternehmensbezogene heterogene Daten und Anwendungen aus einer Vielzahl von Datenquellen zusammenzuführen, kombinieren, zu verarbeiten, anzureichern und das Ergebnis als Visualisierung zu präsentieren. Aufgrund der heutzutage stetig steigenden und oftmals verteilten Datenmengen (Big Data) ist eine möglichst generische, automatisierte Zusammenführung und Analyse (semi-) strukturierter und unstrukturierter Daten notwendig. Dieser Prozess der ad-hoc Zusammenführung mehrerer Datenquellen ist auch unter dem Begriff **Data Mashup** oder **Enterprise Mashup** bekannt. Ein Data Mashup kombiniert, manipuliert und verbindet unterschiedliche Datenquellen für eine einheitliche Visualisierung und erlaubt Anwendern ohne technischen Kenntnisse aus bestehenden Daten in unterschiedlichen Systemen ad-hoc eine neue Anwendung zu erstellen. Um derartige Mashups zu realisieren wurden Technologien geschaffen, die jedoch hohe technische Anforderungen erfordern und aus diesem Grund lediglich von Experten mit entsprechenden technischen Fertigkeiten verwendet werden können. Dies hat zur Folge, dass diese Technologien ausschließlich von einem stark eingegrenzten Nutzerkreis benutzt werden können. Des Weiteren sind bestehende Lösungen in ihrer Flexibilität eingeschränkt, d.h. sie unterstützen nur eine einzelne Art der Ausführung und erfüllen somit auch nur bestimmte Nutzeranforderungen (z.B. bzgl. Robustheit, Effizienz, Skalierbarkeit etc.)

Um diese Einschränkungen zu beseitigen wurde an der Universität Stuttgart das Data Mashup Tool **FlexMash** entwickelt, welches eine einfache Modellierung von Data Mashups durch Domänenexperten sowie eine flexible (d.h. anforderungsabhängige) Ausführung ermöglicht. Um eine möglichst abstrakte Modellierung von Data Mashups zu ermöglichen, werden domänenspezifische Mashup Plans verwendet. Ein Mashup Plan ist ein nicht-ausführbares Format zur abstrakten Modellierung und Verknüpfung von Datenquellen, sogenannten **Data Source Descriptions** (DSDs) und Datenoperationen, sogenannten **Data Processing Descriptions** (DPDs). Diese nicht ausführbaren Mashup Plans können anschließend, entsprechend der Nutzeranforderungen, auf verschiedene ausführbare Formate transformiert werden. So kann für eine robuste Ausführung des Mashup Plans eine BPEL Workflow Engine zum Einsatz kommen.

1.1 Zielsetzung

Die Transformation von Mashup Plans in eine ausführbare Darstellung, dem sogenannten ausführbaren Mashup Plan (z.B. ein BPEL Workflow), wurde bereits in vorangegangenen Arbeiten erörtert und gelöst. Im Rahmen dieser Diplomarbeit wird die Bereitstellung von Code-*Fragmenten*, die als ausführbare *Bausteine* vom Mashup Plan aufgerufen werden (in BPEL z.B. : Web Services) beschrieben. Dabei stellen Code-Fragemente die konkreten Implementierungen von *DSDs* und *DPDs* dar, wobei es

mehrere Implementierungen für DSDs und DPDs geben kann. DPDs und DSDs sind allgemein als *Pattern*, Muster zu betrachten, also als abstrakte Lösung eines Problems.

Patterns sind bewährte Lösungsmuster für häufig auftretende Problemfälle und ihre Verwendung ermöglicht Abstraktion und eine hohe Flexibilität. Für die konkrete Umsetzung von abstrakten Patterns existieren meist eine Vielzahl an Implementierungen. Diese sind jeweils abhängig vom Kontext, in dem das Pattern angewendet wird. Das Finden einer geeigneten Implementierung zu einem verwendeten Pattern, genannt Mapping, stellt jedoch eine große Herausforderung dar, da in den meisten Fällen mehr als eine mögliche Implementierung existiert und folglich eine Auswahl anhand geeigneter Kriterien erfolgen muss. Dieses Problem soll im Rahmen dieser Diplomarbeit mittels eines Konzeptes sowie einer prototypischen Implementierung gelöst werden.

Grundlegendes Ziel dieser Arbeit ist die automatische Bereitstellung von geeigneten Code-Fragmenten mit Hilfe eines Fragment-Verzeichnisses, welche die in Mashup Plans modellierten, abstrakten DSDs und DPDs durch konkrete Implementierungen ersetzen. Diese können anschließend von ausführbaren Mashup Plans aufgerufen werden. Die Auswahl einer passenden Implementierung soll dabei *regelbasiert* durch Parametrisierung der Patterns erfolgen. Das regelbasierte Pattern Mapping von Mashup Plans wird in den Kapiteln genauer erläutert.

Für die Entwicklung eines solchen Fragmente-Verzeichnisses zur Bereitstellung von konkreten DSD- und DPD-Implementierungen, soll in dieser Arbeit zunächst ein Konzept erstellt werden, welches später prototypisch umgesetzt werden soll. Hierbei soll untersucht werden, in welcher Form Code-Fragmente der DSDs und DPDs als Vorlagen (*Templates*) abgespeichert werden können. Das Verzeichnis soll als Repositorium eine effektive Verwaltung der Code-Fragmente anhand verschiedener Funktionen ermöglichen. Des Weiteren sollen für ein effizientes Retrieval von Codefragmenten aus dem Verzeichnis Metadaten verwendet. Diese sollen in abstrakter Weise die Funktionalitäten und weitere charakteristische (z.B. Eingabe- und Ausgabeparameter etc.) Eigenschaften der gespeicherten Fragmente beschreiben. Um einen schnellen und effizienten Ablauf der Suche im Repositorium zu gewährleisten, muss die Verwendung einer geeigneten Datenbanktechnologie wie z.B. relationale oder NoSQL-Technologien bzw. einer Kombination verschiedener Technologien ermittelt werden, wodurch die Nachteile der jeweiligen Datenbankansätze verringert bzw vermieden und die positiven Merkmale der Ansätze für eine effizientere Lösung ausgenutzt bzw. zusammengeführt werden. Ferner soll auf die Fragestellungen eingegangen werden, wie entsprechende Templates instantiiert, d.h. ausführbar gemacht und wie eine passende Implementierung für DSDs und DPDs gefunden werden kann. Hierzu kann beispielsweise eine baum-basierte Patternhierarchie herangezogen werden, die es ermöglicht, Patterns, also abstrakte Lösungsansätze auf der höchsten Hierarchieebene, in kleinere Bestandteile, sogenannte Subpatterns, bis hin zu ausführbaren Codefragmenten der niedrigsten Hierarchieebene zu zerlegen. Basierend auf dieser Hierarchie können mittels eines regelbasierten Transformationsansatzes für DSDs und DPDs passende, ausführbare Codefragmente (z.B. BPEL Web Services) bestimmt werden. Zur Veranschaulichung eines solchen Transformationsprozesses sollen ein oder mehrere konkrete Anwendungsszenarien definiert und eine prototypische Implementierung erstellt werden, welche später in das bestehende Data Mashup Tool FlexMash integriert werden soll bzw. kann.

Gliederung

Die Arbeit ist in folgender Weise gegliedert:

Kapitel 2 – Grundlagen: Hier werden werden die Grundlagen dieser Arbeit beschrieben.

Kapitel 3 – Grundkonzept einer Fragment-Repository: In diesem Kapitel wird das Konzept für das Repository-Verzeichnis vorgestellt. Dazu wird der Aufbau und die Funktion des Repository-Dienstes beschrieben.

Kapitel 4 – Patternbeispiele: Dieses Kapitel beschäftigt sich mit Patternbeispielen, die in einem Mashup Plan vorkommen können.

Kapitel 5 – Implementierung: In diesem Kapitel wird die technische Umsetzung (Implementierung)des Konzepts erläutert vorgestellt.

Kapitel 6 – Related Work: Hier werden ähnliche Ansätze von Fragmente-Repositories aufgelistet

Kapitel 7 – Zusammenfassung und Ausblick: fasst die Ergebnisse der Arbeit zusammen und gibt einen kurzen Ausblick.

2 Grundlagen

In diesem Kapitel werden grundlegende, informationstechnische Begriffe und Inhalte erklärt, die im weiteren Verlauf dieser Arbeit verwendet werden und für das Verständnis der Thematik dieser Arbeit notwendig sind.

2.1 Datenbanksysteme

Dieser Abschnitt basiert auf [RG00].

Ein *Datenbanksystem*(DBS) setzt sich aus zwei Komponenten zusammen:

- der *Datenbank* (DB), die die Menge der zu verwaltenden Daten enthält und
- dem *Datenbankmanagementsystem* (DBMS), der Software, die zur Verwaltung der DB benötigt wird

Eine **Datenbank**(DB) ist eine Sammlung von Daten, die typischerweise die Aktivitäten einer oder mehrerer verwandter Organisationen beschreibt. Eine Universitätsdatenbank kann beispielsweise folgende Informationen enthalten: Entitäten wie *Studenten*, *Fakultät*, *Kurse* und *Klassenzimmer* sowie Beziehungen zwischen diesen Entitäten (*Studenten-Kurswahl*, *angebotene Kurse der Universität*, *Nutzung der bestehenden Klassenzimmer* etc.)

Ein **Datenbankmanagementsystem**(DBMS) ist eine Software, die der Verwaltung und Verarbeitung von großen Datenansammlungen dient. Die Verwendung von Datenbankmanagementsystemen bringt viele Vorteile mit sich.

Die Menge an Daten, die heutzutage verarbeitet, extrahiert und weitergeleitet und gespeichert werden, steigt stetig. Dabei ist der Wert bzw. die Bedeutung von Daten als wirtschaftliches Gut allseits anerkannt. Ohne eine geeignete Verwaltung dieser großen Menge an Daten, kann der Mehrwert einer Information nicht ausgeschöpft werden. Zudem kann die effiziente Suche und das schnelle Auffinden nach der passenden Information in Bezug auf eine bestehende Anfrage bzw. Problemstellung bei einer gleichzeitig ansteigenden Menge an Informationen ohne ein leistungsstarkes und flexibles Datenverwaltungssystem nicht gewährleistet werden. Um den größtmöglichen Nutzen aus großen und komplexen Datensätzen ziehen zu können, müssen Unternehmen über Systeme bzw. Tools verfügen, die ein einfaches Verwalten und das effiziente Extrahieren nützlicher Informationen ermöglichen. Der Nutzen, der aus der Verwendung der extrahierten Information entsteht, sollte höher sein als die Kosten und der Aufwand zur Verwaltung und Suche dieser Daten.

Seit 1980 befestigte das relationale Model seine Position als dominierendes **Datenbankmanagemensystem**, DBMS. Die von IBM im Rahmen eines Projekts entwickelte Query-Sprache für relationale

Datenbanken, *SQL*, ist mittlerweile die führende Standard-Sprache. SQL wurde in den späten 80er Jahren standardisiert und der Standard durch das American National Standards Institute (ANSI) und International Standards Organizations (ISO) übernommen.

Seit dem Einzug ins Internetzeitalter spielen Datenbankmanagementsysteme eine bedeutende Rolle für die Speicherung von Daten, auf welche mittels eines Web Browsers zugegriffen werden kann. Sie ersetzen die bis dahin übliche Speicherung von Daten in Operating System Files.

2.1.1 Vorteile von Datenbankmanagementsystemen

Die Verwendung von Datenbankmanagementsystemen für die Verwaltung von Daten bietet zahlreiche Vorteile:

- **Datenunabhängigkeit**: Anwendungsprogramme sollten möglichst unabhängig von Details der Datenrepräsentation und -Speicherung sein. Ein DBMS bietet eine abstrakte Sicht auf Daten und trennt somit Anwendungscode von derartigen Details.
- **Effizienter Datenzugriff**: Ein DBMS bedient sich einer Vielzahl an Techniken, um Daten effizient speichern und auffinden zu können. Dieses Merkmal eignet sich hauptsächlich bei Daten, welche in externen Speichergeräten werden
- **Datenintegrität und Sicherheit**: Erfolgt der Datenzugriff stets mittels des DBMS, kann die DBMS Integritätsbeschränkungen geltend festlegen. So überprüft das DBMS beispielsweise, ob das Budget der Abteilung überschritten ist, bevor die Gehaltsinformation eines Mitarbeiters eingegeben wird. Des Weiteren kann ein DBMS bestimmen mittels eine Zugriffskontrolle, welche Daten für welchen Klasse von Benutzer sichtbar bzw. verfügbar sind.
- **Datenverwaltung**: Bei einer Vielzahl von Benutzern, die Daten teilen, bringt eine zentralisierte Datenverwaltung bedeutende Verbesserungen. Erfahrene Experten können die Verantwortung für die Organisation der Datendarstellung übernehmen. Dadurch wir zum Einen die Redundanz verringert und zum Anderen die Datenspeicherung für eine effizientere Suche verfeinert.
- **Simultaner Zugriff und Crash Recovery**: Ein DBMS verwaltet simultane Zugriffe auf Daten, so dass dem Benutzer der Zustand suggeriert wird, dass lediglich jeweils ein Benutzer auf die Daten zugreift. Zudem schützt das DBMS Benutzer vor den Auswirkungen eines Systemausfalls.
- **Verringerte Entwicklungszeit für Anwendungen**: Datenbankmanagementsysteme unterstützen wichtige Funktionen, welche gebräuchlich sind für zahlreiche Anwendungen, die auf die Daten eines DBMSs zugreifen. Dies erlaubt in Verbindung mit einem High-Level Interface für Daten eine vereinfachte, schnellere Entwicklung von Anwendungen. Diese sind überdies mit höherer Wahrscheinlichkeit robuster als Anwendungen, welche von Grund auf erstellt werden, da wichtige Aufgaben, anstelle der Anwendungen selbst, vom DBMS bewältigt werden

Neben den genannten Vorteilen gibt es jedoch auch Anwendungsfälle, in denen sich der Einsatz von DBMS nicht eignen. DBMS sind optimiert auf ein bestimmtes Arbeitspensum und ihre Leistung kann bei einigen spezialisierten Anwendungen nicht adäquat sein. Dies können Anwendungen mit strengen Echtzeitbeschränkungen sein oder Anwendungen mit genau festgelegten kritischen Operationen,

für die speziell angefertigte effiziente Codes geschrieben werden müssen. Ein weiterer Grund dafür, ein DBMS nicht zu verwenden, kann eine Anwendung sein, welche Daten in einer von der Query Language eines DBMS nicht unterstützten Weise manipulieren muss. Hier kann die abstrakte Sicht eines DBMS auf Daten den Anforderungen der Anwendung nicht gerecht werden und hinderlich sein.

Ein Großteil der Datenbankmanagementsysteme basieren auf dem **relationalen Datenmodell**.

2.1.2 Relationales Datenmodell

Das Zentrale Konstrukt beim relationalen Datenmodell zur Bechreibung von Daten ist eine Relation, welches man sich bildlich in Form einer Tabelle vorstellen kann. Das relationale Datenmodell beschreibt Tabellen und ihre Beziehung zu anderen Tabellen.

Eine relationale Datenbank beschreibt eine Sammlung von Tabellen (den **Relationen**), in welchen Datensätze abgespeichert werden. Dabei entspricht jede Zeile (**Tupel**) in einer Tabelle einem Datensatz (**Record**). Des Weiteren setzt sich jedes Tupel aus einer Reihe von Attributwerten (Eigenschaften) zusammen, welche die Spalten der Tabelle darstellen. Eine Relation kann als eine Menge von Tupeln (Records) verstanden werden. Im Bereich der Datenmodellierung wird die Beschreibung von Daten Schema genannt. Ein Schema für eine Relation bestimmt in einem relationalen Modell den Namen der Relation, den Namen jedes einzelnen Felds (Attribut oder Spalte) und den Typ des Felds. Den Attributen einer Tabelle können Werte aus einer festgelegten Domäne zugewiesen werden. Dementsprechend kann die Information bezüglich eines Studenten beispielsweise in einer Universitäts-Datenbank in einer Relation wie folgt als Schema gespeichert werden:

Students(sid: string, Name: string, login: string, Alter: integer, gpa: real)

Aus dem obigen Schema kann bestimmt werden, dass jeder Tupel in der Tabelle *Student* aus fünf Spalten besteht und jedem Attribut der Datentyp mitgegeben wird. In der folgenden Abbildung wird eine Instanz der Tabelle *Student* dargestellt.

Jede Zeile in der Relation Student ist ein Datensatz (*Record*), welches einen Studenten beschreibt. Jede Zeile ist hierbei nach dem festgelegten Schema angelegt, welches auch als ein Template betrachtet werden kann. Diese kann durch Integritätsbedingungen weiter verfeinert werden, die jedes Tupel der Tabelle erfüllen muss. So kann festgelegt werden, dass jeder Student eine eindeutige ID-Nummer (*sid*)besitzt.

Tabellen werden mithilfe von Primär- und Fremdschlüsseln miteinander verknüpft. Wie bereits erwähnt, identifizieren Primärschlüssel, also eine Menge von Attributen einer Relation, die Tupel einer Relation eindeutig. Im Gegensatz dazu stellen Fremdschlüssel eine Menge von Attributen innerhalb einer Relation dar, welche die Primärschlüssel derselben oder einer anderen Relation referenzieren. Mithilfe dieser Fremdschlüssel-Primärschlüssel-Beziehungen können mengenorientierte Operationen auf Relationen ausgeführt werden. Dementsprechend lassen sich Datensätze durch Schnitt-, Selektion- und Vereinigungsoperationen auf mehrere Datensätze, abfragen bzw. erstellen. Für die Formulierung dieser Abfragen wird eine spezielle Abfragesprache benötigt. Die dominierende Abfragesprache für relationale Datenmodelle ist SQL, Structured Query Language.

Zu den Stärken des relationalen Modells gehört zum Einen ihre einfache Handhabung und zum Anderen die Möglichkeit für die Verwendung von simplen High-Level-Sprachen für die Abfrage von Daten. Zudem erlaubt die einfache tabellarische Darstellung selbst Anfängern beim besseren Verständnis der Inhalte von Datenbanken. Zusammengefasst sind die wesentlichen Vorteile des relationalen Datenmodells ihre verständliche und simple Datendarstellung sowie die leichte Formulierbarkeit von Abfragen.

2.1.3 SQL

Structured Query Language, SQL ist eine deklarative Sprache für die Verwaltung und Bearbeitung von Daten in relationalen Tabellen. Das American National Standards Institute (ANSI) veröffentlichte 1986 den ersten Standard für SQL, welches mit der Zeit mit zusätzlichen Features verfeinert wurde (z.B. objektorientierte Funktionalität). Ferner konzentriert man sich seit 2006 auf die Integration von SQL und **XML**. Hierzu wurde **XQuery**definiert, welches die Abfrage von Daten aus XML-Dokumenten ermöglicht. Das Ergebnis einer SQL-Abfrage ist eine Tabelle(Ergebnismenge), wobei diese wiederum als permanente Tabelle in einer relationalen Datenbank angelegt werden oder als Eingabe anderer Abfragen genutzt werden. SQL ist eine nicht-prozedurale Sprache. Diese definieren die gewünschten Ergebnisse, überlassen jedoch den Prozess, über den diese Ergebnisse generiert werden, einer externe Instanz. Dementsprechend lassen sich mit SQL keine vollständigen Anwendungen schreiben [Bea09].

SQL-Anweisungen

SQL besteht aus mehreren getrennten Teilen :

- SQL-Schemaanweisungen, mit welchen die in der Datenbank gespeicherten Datenstrukturen definiert werden
- SQL-Datenanweisungen, welche die Bearbeitung von zuvor angelegten Datenstrukuren ermöglichen
- SQL-Transaktionsanweisungen, mit denen Transaktionen gestartet, beendet oder *zurückgerollt* werden können

Eine SQL-Datenanweisung sieht wie folgt aus:

SELECTcustom ID **FROM** customers **WHERE** lastname = Kalyoncu

Die **FROM**-Klausel bestimmt hierbei, aus welcher Tabelle Daten benötigt werden. Mithilfe der **SELECT**-Klausel wird festgelegt, welche Spalten aus der Tabelle abgefragt werden sollen und anschließend mit der **WHERE**-Klausel Bedingungen hinzugefügt, um bestimmte Daten zu filtern. Im Beispiel oben werden aus der Tabelle *customers* diejenigen *customID*-Nummern entnommen, welche den Nachnamen *Kalyoncu* besitzen.

Neben dem relationalen Datenmodell, existieren weitere Modelle wie das hierarchische Modell (z.B. verwendet in IBMs IMS DBMS), das Network Modell, das objektorientierte Modell, sowie das objekt-relationale Modell. Das relationale Datenmodell ist jedoch eine der dominantesten Modelle.

Datenbankmanagementsysteme können anhand von sogenannten **Queries** nach bestimmten Daten abgefragt werden. Dazu stellt ein DBMS eine spezialisierte Sprache, die **Query Language** bereit. Das relationale Modell unterstützt leistungsstarke Query Languages.

Mithilfe einer Datenmanipulationssprache (**Data Manipulation Language**, DML) kann ein Benutzer die Daten eines DBMS modifizieren, neue Daten erstellen und die Datenbank nach bestimmten Daten abfragen. Die Query Language ist eines von mehreren Komponeten einer DML, welches Konstrukte für das Hinzufügen, Entfernen und Modifzieren von Daten anbietet.

2.1.4 Transaktionsmanagement

Greifen mehrere Benutzer gleichzeitig auf eine Datenbank zugreifen, kann es zu Konflikten kommen. Als Beispiel kann ein Szenario aus dem Bankwesen betrachtet werden. Während die Anwendung eines Benutzers die Deposit-Beträge berechnet, kann eine andere Anwendung einen bestimmten Geldbetrag von einem Konto auf ein anderes Konto transferieren, welches die Deposit-Anwendung nicht nicht registriert hat. Daraus ergibt sich schließlich ein Gesamtdeposit-Betrag, der deutlich höher ausfällt, als es eigentlich sein sollte [RG00]. Das DBMS muss Benutzer vor den Auswirkungen eines Systemausfalls schützen, indem gewährleistet wird, dass alle Daten, sowie der Status aller aktiven Anwendungen, bei einem System-Neustart auf einen konsistenten Zustand zurückgesetzt werden.

Demnach ist es erwünscht, dass bei einer durch das DBMS bereits bestätigten Reisebuchung, im Falle eines Systemausfalls, die Reservierung nicht verlorengeht. Falls das DBMS die Bestätigung bzw. die Antwort auf eine Benutzeranfrage jedoch noch nicht verschickt hat, und zum Zeitpunkt eines Ausfalls, mitten im Prozess war, die notwendigen Änderungen am Datenbestand durchzuführen, sollten diese bei einem System-Neustart rückgängig gemacht werden.

Ein wichtiger Teil für die Datensicherheit ist das **Transaktionskonzept**. Eine Transaktion entspricht hierbei einer Sequenz von Programmschritten, die auf die Daten der DB ausgeführt werden. Um die Konsistenz der Datenbank zu gewährleisten, werden sämtliche Aktionsschritte einer Transaktion als eine logische Einheit betrachtet. Demnach wird eine Transaktion entweder vollständig und fehlerfrei oder gar nicht ausgeführt. Nach erfolgreicher Ausführung der Transaktion wird der Datenbestand in einem konsistenten Zustand hinterlassen. Greifen mehrere Anwender gleichzeitig auf eine Datenbank zu und modifizieren dabei möglicherweise die darin befindlichen Daten, kann es zu Konflikten kommen, welche Anomalien in der Datenbank verursachen können. So kann es vorkommen, dass ein Benutzer die Modifikationen an Daten wieder zurücksetzt. Diese sogenannte Dirty data kann jedoch, bevor sie vom Benutzer zurückgesetzt werden, von anderen Benutzern bzw. Prozessen bereits gelesen und verwendet worden sein. Daraus resultieren Anomalien, die zu schwerwiegenderen Problemen führen können. Um das Auftreten von Anomalien zu vermeiden, muss das DBMS die jeweiligen Anfragen der Benutzer sorgfältig anordnen. Dadurch wird verhindert, dass mehrere Benutzer gleichzeitig Daten ändern können [HR83].

Bei der Ausführung von Transaktionen müssen die sogenannten **ACID**-Eigenschaften eingehalten werden. Jede Transaktion muss folgende Eigenschaften erfüllen:

- **Atomicity**: Eine Transaktion wird entweder vollständig oder gar nicht ausgeführt.

- **Consistency**: Nach erfolgreicher Ausführung der Transaktion muss sich der Datenbestand in einem konsisten Zustand befinden. Diese Eigenschaft ist Voraussetzung für die Durability-Eigenschaft
- **Isolation**: Bei gleichzeitiger Ausführung mehrerer Transaktionen dürfen diese sich nicht gegenseitig beeinflussen.
- **Durability**: Die Auswirkungen einer Transaktion auf einen Datenbestand sind dauerhaft.

Gemäß dem ACID-Paradigma ist eine Datenbank nur dann konsistent, wenn es Daten beinhaltet, die aus erfolgreichen Transaktionen entstanden sind.

2.2 Patterns

Patterns sind ein bekanntes und oft verwendetes Konzept im Bereich der Computerwissenschaften. Sie beschreiben bewährte Lösungsansätze bzw. Lösungswege für häufig auftretende Problemfälle in einem spezifischen Kontext und in generischer Art und Weise. Dementsprechend sind Patterns vielseitig einsetzbar in unterschiedlichsten, spezifischen Anwendungsfällen. Das wesentliche Ziel des Konzepts der Patterns ist die Generalisierung und Abstraktion von Wissen zur Lösung von Problemfällen.

2.2.1 Definition

Patterns und Patternbasierte Sprachen sind etablierte Konzepte in verschiedenen Anwendungsbereichen der Informatik und Informationstechnologie. Ursprünglich in der Architektur eingeführt, setzte sich das Konzept der Patterns mit zunehmender Beliebtheit auch in vielen anderen Bereichen (Bildung, Design, Cloudanwendungen etc.) durch.

Patterns sind menschenlesbare Artefakte, welche Problemwissen mit generischen Lösungsansätzen kombinieren. Das Muster, welches ein Pattern beschreibt, enthält Lösungsbereiche, die Solution Knowledg in Textform darstellen [Ale77] . Diese Form der Wissensdarstellung beinhaltet den wesentlichen Kern der Lösung in abstrakter Weise. Allgemeine Lösungsbereiche des Pattern verkörpern keine konkreten Lösungsinstanzen des Pattern. Sie dienen dem Leser lediglich als Anleitung für die Implementierung einer Lösung, welche seinen Anforderungen genügt [FBB+14].

Ansätze für eine iterative Pattern-Formulierung von [Rei12]und Falkenthal et al. [FJZ+12]ermöglichen es, konkretes Lösungswissen bei der Erstellung bzw Formulierung von Patterns zu verwenden. Pattern sind nicht nur finale Artefakte, sondern werden, basierend auf initialen Ideen, innerhalb eines iterativen Prozesses formuliert, um den Status eines Pattern zu erlangen. In diesen Ansätzen unterstützt konkretes Lösungswissen lediglich den Formulierungprozess von Patterns, wird jedoch nicht explizit gespeichert, um im Falle eines Einsatzes des Pattern erneut verwendet werden zu können. (Porter et al.) belegen, dass die Auswahl von Patterns aus einer Patternsprache eine Frage der zeitlichen Reihenfolge der gewählten Patterns ist. Demnach ist die Kombination und Aggregation von Patterns abhängig von der Reihenfolge, in der die Patterns eingesetzt werden müssen. Diese Erkenntnis führt zu den sogenannten Patternsequenzen, welche teilweise geordnete Pattermengen darstellen und die

temporale Reihenfolge, in der die Patterns eingesetzt werden veranschaulichen. Der Fokus hierbei liegt jedoch in der Kombinierbarkeit von Patterns und nicht in der Kombinierbarkeit von konkreten Lösungen.

Zahlreiche Patternsammlungen und Patternsprachen werden in digitalen Patternverzeichnissen gespeichert. Obgleich diese den Benutzer bei der Navigation durch das Verzeichnis behilflich sind, verknüpfen sie Patterns nicht mit konkreten Lösungsansätzen. Folglich sind Benutzer gezwungen konkrete Lösungen manuell wiederherzustellen, wann immer ein Pattern verwendet wird.

2.2.2 Solution Implementations

Solution Implementations sind Buildings Blocks für den Einsatz und die Aggregation konkreter Lösungen aus Patterns.

Wie bereits oben erwähnt, gibt es keine Ansätze, die sich mit der Aggregation von konkreten Lösungen auseinandersetzen, wenn mehrere Patterns zusammen angewendet werden. Zudem sind konkrete Lösungsansätze weder mit Patterns verknüpft noch werden sie mit den Patterns zusammen abgespeichert. Um diese Problemstellung aufzugreifen und zu entgegnen schlagen Falkenthal, Michael, et al. vor, konkretes, implementiertes Lösungswissen als wiederverwendbare Bausteine (Building Blocks) zu definieren, die konkrete Lösungen mit Patterns verknüpft und die Komposition dieser ermöglicht.

2.2.3 Patternhierarchien

Der folgende Abschnitt basiert auf [HM16]. Damit Patterns strukturiert und mit entsprechenden Implementierungen verknüpft werden können, werden sogenannte *Pattern Graphen* verwendet. Ein **Pattern Graph** ist ein baum-basierter, gerichteter Graph, welcher sich aus Knoten und Kanten zusammensetzt. Die Knoten eines solchen Graphen repräsentieren entweder ein Pattern oder eine Implementierung. Die Kanten, welche die Knoten miteinander verbinden, stellt eine sogenannte Spezialisierung dar. Es gibt zwei verschiedene Formen von Kanten:

- **consist of-Kanten** : Diese Kanten verbinden Patterns miteinander und deuten darauf hin, dass ein Pattern aus mehreren Sub-Patterns zusammengesetzt ist. Dementsprechend kann die Problemstellung, welches das Pattern beschreibt, nur dann gelöst werden, falls alle Sub-Patterns dieses Pattern-Knoten ausgeführt werden.
- **implemented by-Kanten**: Diese Kanten werden zur Verknüpfung von Implementierungsknoten Kanten verwendet. Ist ein Pattern-Knoten mit mehr als einem Implementierungsknoten verbunden, bedeutet dies, dass diese durch eine diese Implementierungen realisiert werden kann. Welche dieser Implementierungen schließlich verwendet wird, kann entweder manuell oder automatisch bestimmt werden.

In Abbildung 2.1 ist ein Pattern-Graph dargestellt. Im Wesentlichen wird ein generisches Pattern, welches dem Wurzelknoten des Pattern-Baumes entspricht, immer weiter konkretisiert, indem es in *Sub-Patterns* aufgeteilt wird, bis diese wiederum nicht mehr weiter in weitere Sub-Patterns unterteilt

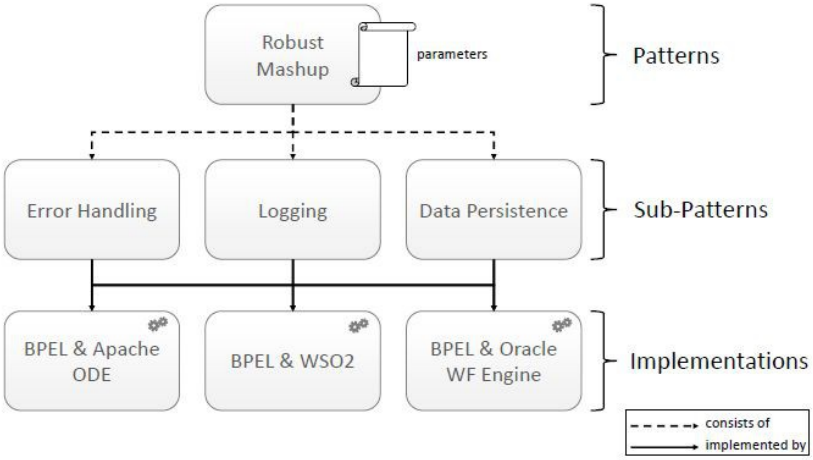


Abbildung 2.1: Der Pattern-Graph [HM16]

werden können und schließlich durch *Implementierungsfragmente* ersetzt werden. Folglich kann ein Pattern mithilfe von unterschiedlichen Abstraktionsgraden hierarchisch strukturiert werden. Zudem kann ein einziges Pattern durch mehrere Implementierungen realisiert werden. Demnach entspricht der Wurzelknoten des Baumes dem Pattern mit dem höchsten Abstraktionsgrad. Dies ist der Pattern, der im Patternkatalog beschrieben wird. Somit existiert für jeden Eintrag im Patternkatalog ein jeweils anderer Pattern Graph.

Die *Parameter* des Patterns im Wurzelknoten bestimmen maßgeblich, welcher Pfad im Pattern-Graphen durchlaufen wird, um die Implementierungen in den Blattknoten zu erreichen. Dies Auswahl des Pfades ist abhängig von Regeln, welche beim Durchlaufen des Pattern Graphen angewendet werden. Sie vergleichen die Parameter des Patterns mit vordefinierten Eigenschaften der Implementierungen, um die geeignetste Implementierung bestimmen zu können. Der Ansatz von [HM16] geht dabei davon aus, dass stets eine Implementierung gefunden werden kann, auch wenn dabei nicht alle Benutzeranforderungen erfüllt werden können. In diesem Fall obliegt es dem Benutzer darüber zu entscheiden, ob die gewählte Implementierung auch angewendet werden soll oder nicht.

Zu beachten ist, dass dieser regelbasierte Transformationsansatz für ein einziges Pattern ohne Weiteres durchgeführt werden kann. Werden jedoch mehrere Patterns kombiniert, ist die Bestimmung einer geeigneten Pattern Implementierung weitaus komplexer.

Sobald eine passende Implementierung gefunden wird, kann die Transformation des Mashup Plans zu einer geeigneten ausführbaren Darstellung beginnen. Für die Erstellung des ausführbaren Modells werden vordefinierte modularisierte Implementierungsfragmente verwendet. Soll beispielsweise die Ausführung mithilfe einer Workflow Engine erfolgen, wird der ausführbare Workflow automatisch erzeugt. So werden die Operationen, die im Mashup Plan definiert werden, anhand von BPEL Invoke-Knoten ausgeführt. Die Programmier- Logik der DSDs und DPDs wird in Codefragmenten (z.B. als Java Web Services) abgespeichert, welche vom Workflow ausgeführt werden. Wird anstelle einer Workflow Engine die Node-Red Engine verwendet, verläuft der Transformationsprozess ähnlich ab-Hier werden vordefinierte JavaScript-Codefragmente miteinander verknüpft.

2.3 Workflows

Ein **Workflow** (Arbeitsablauf) beschreibt eine definierte Abfolge von Arbeitsschritten in einem Arbeitssystem. Konkret ausgedrückt bezeichnet ein Workflow mehrere dynamische abteilungsübergreifende Aktivitäten, welche in zeitlicher oder logischer Abhängigkeit zueinander stehen. Demnach ist ein Workflow die informationstechnische Realisierung eines Geschäftsprozesses. Dabei sind die einzelnen Arbeitsschritte als Aktivitäten zu verstehen, welche oftmals zu größeren Komponenten zusammengesetzt werden können. Daraus resultiert eine erhöhte Wiederverwendbarkeit und Flexibilität. Wird der definierte Arbeitsablauf einer neuen Situation angepasst und verändert, können die Komponenten neu zusammengesetzt werden. Ziel ist die (Teil-)Automatisierung von Workflow. Ursprünglich wurden Workflows im unternehmerischem Umfeld eingesetzt, finden jedoch mittlerweile auch Einsatz im wissenschaftlichem Bereich.

2.3.1 Die Workflow-Technologie

Die rechnergestützte Ausführung von Arbeitsabläufen kann anhand von drei Dimensionen beschrieben werden: **WHO**, **WHAT** und **WITH**:

- **WHO** : Diese Dimension legt fest, welche Mitarbeiter oder Abteilungen einer Organisation eine konkrete Aktivität ausführen dürfen. Hierzu können mithilfe von Anfragen bestimmt werden, welcher Mitarbeiter bzw. welche Abteilung für die Ausführung geeignet ist. Durch die Vergabe von Rollen an Abteilungen und Mitarbeiter , kann die Struktur eine Organisation umfassender beschrieben werden.
- **WHAT**: In der WHAT-Dimension wird definiert, welche Aktivitäten in welcher Reihenfolge ausgeführt werden. Sowohl die parallele als auch die sequentielle Ausführung von Aktivitäten ist möglich.
- **WITH**: Mithilfe der WITH-Dimension wird beschrieben, welche Ressourcen aus der IT-Infrastruktur verwendet werden, um die Aktivitäten ausführen zu können.

Abbildung 2.2 stellt diese drei Dimensionen im dreidimensionalen Raum dar. Die Ausführung eines Workflows setzt sich hierbei aus einer Abfolge von Punkten zusammen. Der Treffpunkt dieser drei

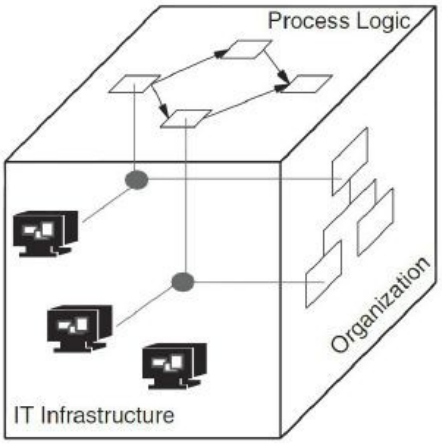


Abbildung 2.2: Workflow-Dimensionen (leyman et. al)

Dimensionen beschreibt, welche konkrete Aktivität von welcher Abteilung bzw. welchem Programm unter Verwendung welcher Ressource ausgeführt wird.

2.3.2 Workflow Management

Nach [Mül06] umfasst das **Workflow Management** alle Aufgaben, welche bei der Analyse, der Modellierung, der Simulation, der Reorganisation sowie bei der Ausführung und Steuerung von Workflows benötigt werden. Es stellt die einzelnen obligatorschen Arbeitssschritte und Abläufe zur Verfügung. Diese entsprechen einem Lebenszyklus (Lifecycle) eines Workflow. Ein **Workflow Management System** (WfMS) beschreibt ein System, welches die Phasen des Prozess-Lifecycles, durch IT-Werkzeuge unterstützt. Diese entsprechen Software, welche Komponenten für die Analyse, Modellierung, die Steuerung, die Administration, die Simulation und das Monitoring von Workflows enthalten.

Architektur eines WfMS

Nach [LR00] setzt sich die Architektur eines **WfMS** gemäß WfMC aus folgendenen drei funktionalen Bereichen zusammen. Diese Bereiche orientieren sich an der Erstellung, dem Betrieb und der Kontrolle

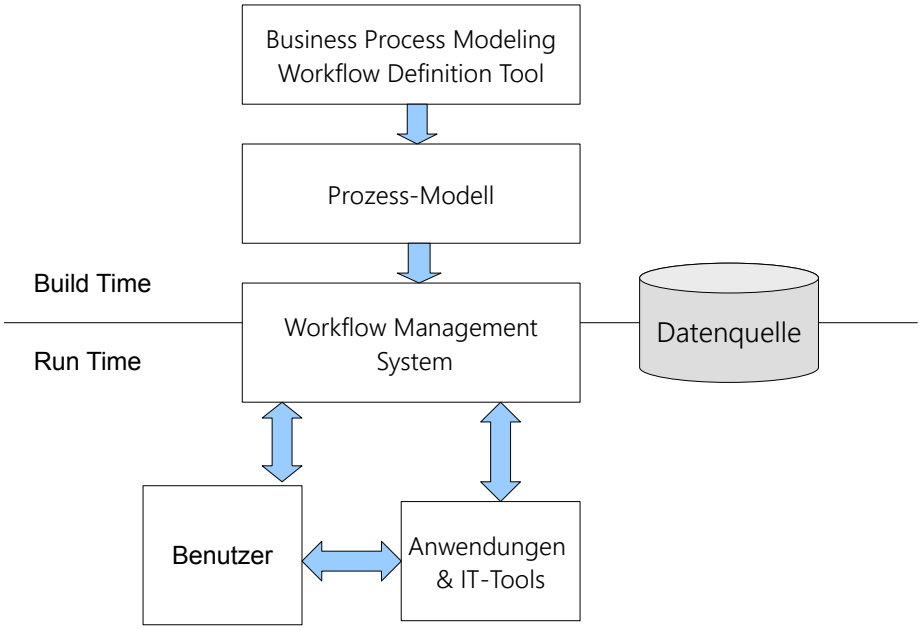


Abbildung 2.3: Funktionsbereiche innerhalb eines WfMS

eines Prozesses. Abbildung 2.3 stellt die *Funktionsbereiche* innerhalb der Architektur eines WfMS dar.

- **Build Time** : dieser Funktionsbereich umfasst alle Komponenten, die zur Verwaltung von Ressourcen, der Erstellung und Modellierung von Workflows dienen.
- **Run Time**: enthält alle Komponenten, die für die Ausführung von Workflows zuständig sind.
- **Data Base**: beinhaltet alle Daten, welche zur Build und Run Time abgelegt werden.
- **Metamodell**: [LR00] fügt einen weiteren Funktionsbereich hinzu, welche alle Strukturen umfassen, die von einem WfMS unterstützt werden.

Workflow Management Coalition (WfMC)

In Abbildung 2.4 wird das **Workflow Referenzmodell** veranschaulicht, welches von der WfMC entwickelt wurde, mit dem Ziel herstellerunabhängige Module eines WfMS miteinander verknüpfen

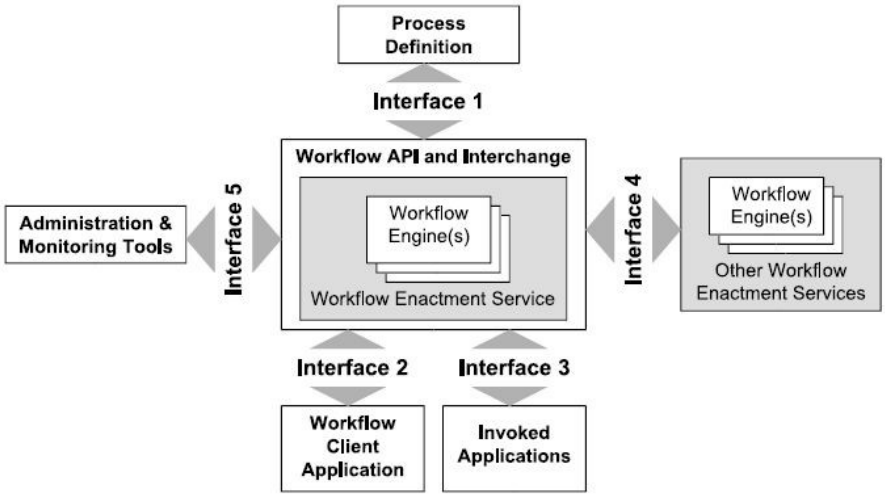


Abbildung 2.4: Workflow-Referenzmodell [Mül06]

und betreiben zu können. Dabei beschreibt das Referenzmodell die Architektur mitsamt den Hauptkomponenten und Standardschnittstellen. Diese dienen dazu, zwischen den WfMS und den einzelnen Komponenten und Werkzeugtools zu kommunizieren. Ziel ist es größtmögliche Systemunabhängigkeit und Interoperabilität zu ermöglichen.

Workflow Reference Model

Im *Workflow Reference Model* wird der Aufbau eines WfMS beschrieben. Das Referenzmodel setzt sich aus den folgenden Komponenten zusammen : Process Definition, Workflow Engine, Workflow Client Application, Invoked Applications, Other Workflow Enactment Services, Administration and Monitoring Tool.Die genauere Beschreibung dieser Komponenten können in [Mül06] nachgeschlagen werden.

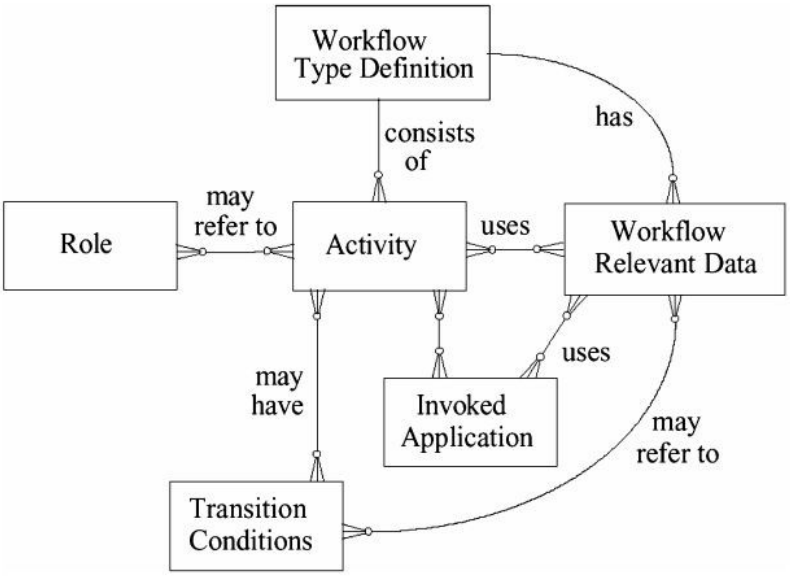


Abbildung 2.5: Workflow Metamodell [Mül06]

Metamodell eines Workflows

Das **Metamodell** eines Workflows, welches von WfMC entwickelt wurde, beschreibt die Grundstruktur eines Workflows. Wie in Abbildung 2.5 veranschaulicht wird, zeigt es die Zusammenhänge zwischen den Objekten und die Mindestanforderungen der Objektbeschreibungen des Prozesses. Dabei ist das Metamodell an die Software-Architektur eines WfMS angelehnt. Die Attribute beschreiben, welche Daten zwischen den WfMS-Komponenten ausgetauscht werden.

2.3.3 Workflow Sprachen

Workflowsprachen dienen zur Beschreibung der Struktur von Workflows. Dabei unterscheidet man zwischen den **Kontrollflussorientierten** und den **Datenflussorientierten** Sprachen

Kontrollflussorientierte Sprachen : Kontrollflussorientierte Sprachen beschreiben den Kontrollfluss. Dieser entspricht der logischen Ausführungsreihenfolge der einzelnen Aktivitäten und kann in Form eines azyklischen, gerichteten Graphen dargestellt werden. Dabei sind die Knoten des Graphen

mit den einzelnen Aktivitäten und die Kanten mit kausalen Abhängigkeiten zwischen diesen Aktivitäten gleichzusetzen. Es ist zu beachten, dass die Aktivität eines Knotens lediglich nach vollständiger und erfolgreicher Ausführung der Aktivitäten seiner Vorgänger-Knoten im Graphen ausgeführt werden kann. Des Weiteren kann eine prozedurale Logik, wie z.B. eine while-Schleife, realisiert werden, indem ein Knoten einen zusätzlichen Graphen als Subgraph enthält.

Datenflussorientierte Sprachen: Datenflussorientierte Sprachen beschreiben den Datenfluss eines Workflows. Dabei entspricht der Datenfluss den Datenabhängigkeiten zwischen den einzelnen Aktivitäten. Aktivitäten besitzen sogenannte Input-Queues, die mit eingehenden Daten befüllt werden. Diese Daten werden von jeder Aktivität entsprechend ihrer Aufgabe im Workflow bearbeitet und die Ausgabedaten anschließend an die Input-Queues der Nachfolger-Knoten weitergeleitet. Aufgrund ihrer Fokussierung auf den Datenfluss eines Workflows eignen sich datenflussorientierte Sprachen besonders für datenintensive Workflow-Sprachen.

2.3.4 Workflow-Klassen

Die folgenden Abschnitte basieren, soweit nicht anders angegeben, auf [Wag11] und[RSM11]. Abbildung 2.6 stellt unterschiedliche Workflow-Klassen in einem Diagramm dar. Dabei werden Workflows nach Kriterien, wie Datenintensität und ihrer Funktionalität in Klassen unterteilt. **Datenintensive Workflows**verarbeiten zumeist große Datenmengen, welche verteilt vorliegen können. Zur Klasse der datenintensiven Workflows zählen **ETL Workflows**, **Data Modeling Workflows**und **Data Analysis Workflows**. **Orchestration Workflows**hingegen verbinden heterogene Anwendungen. Das Ziel hierbei ist es, Geschäftsprozesse zu realisieren bzw. zu automatisieren. Sowohl **Business Workflows** als auch **Simulations Management Workflows** gehören zu den Orchestration Workflows. *Simulations Management Workflows*, *Data Analysis Workflows* sowie *Data Modeling Workflows* bilden die Komponenten einer weiteren Klasse: die Klasse der *Scientific Workflows*.

2.3.5 Business Process Execution Language

BPEL, welches auch unter dem Namen WS-BPEL bekannt ist, ist eine XML-basierte Sprache zur Beschreibung von Workflows [JEA+07]. Sie dient als Standard zur Steuerung und Koordination von geschäftsbasierten Web Services. Ziel ist die Standardisierung des Automatisierungsprozesses zwischen Web Services. BPEL erlaubt das Definieren von Business-Prozessen, die andere Dienst- und Business-Prozesse integrieren, welche ihre Funktionalität als Dienste anbieten.

WS-BPEL repräsentiert die Konvergenz der Workflow-Sprachen **WSFL**(Web Services Flow Language) und **XLANG**. WSFL basiert auf dem Konzept direkter Graphen, während XLANG eine blockstrukturierte Sprache ist. BPEL kombiniert beide Ansätze und stellt ein reichhaltiges Vokabular für die Beschreibung von Business-Prozessen zur Verfügung.

WS-BPEL ist eine kontrollflussorientierte Sprache, welches die Orchestrierung von Web Services ermöglicht. Dabei können Web Services in Workflows eingebunden werden, indem für die Knoten des Workflows Web Service-Implementierungen erstellt werden. Aus diesem Prozess resultiert ein Workflow, der selbst als ein Web Service betrachtet werden kann und eventuell als eine Komponente eines größeren Workflows fungieren kann. Dementsprechend können Workflows aus bereits bestehenden

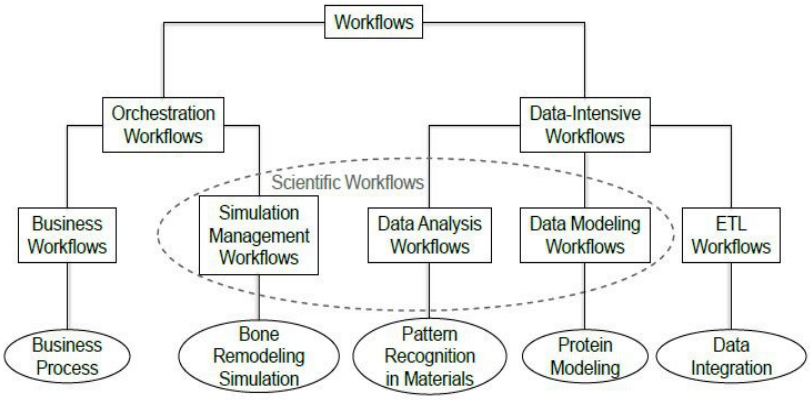


Abbildung 2.6: Klassifizierung von Workflows [RSM11]

Workflows zusammengesetzt werden. Dabei ist es möglich einzelne Komponenten des Workflows durch andere zu ersetzen oder die Anordnung dieser zu ändern. Daraus resultiert eine größerer Wiederverwendbarkeitswert des Workflows sowie eine höhere Flexibilität. So können Workflows flexibler auf Veränderungen reagieren.

BPEL basiert auf **WSDL**, *XML Schema* und **XPath**. WSDL dient hierbei der Beschreibung der Schnittstellen zu den Web Services. Daten werden mithilfe von Variablen abgespeichert, welche unterschiedliche Typen besitzen können. Dabei kann eine Variable entweder vom Typ *XML Schema Element*, *simpleType* oder *complexType* sein.

Die Hauptbestandteile in WS-BPEL sind Aktivitäten. Diese entsprechen den einzelnen Arbeitsschritten in einem Workflow, d.h. Workflows setzen sich aus einzelnen Aktivitäten zusammen. Es werden zwischen zwei Typen von Aktivitäten unterschieden:

- **Strukturierte Aktivitäten**
- **Basisaktivitäten**

Strukturierte Aktivitäten:

Strukturierte Aktivitäten sind Aktivitäten, welche sich aus anderen Aktivitäten zusammensetzen. Durch die Kombination mehrere Aktivitäten können komplexere Aktivitäten erstellt werden, welche die den Kontrollfluss von Prozessen bschreiben. Folgende Aktivitäten gehören zu dieser Kategorie:

- **Sequence** : Die Sequence-Aktivität ermöglicht die sequentielle Ausführung von Aktivitäten eines Workflows
- **For Each** : Bei einer For Each-Aktivität wird der Rumpf einer Schleife solange ausgeführt, bis eine zuvor bestimmte Anzahl an Durchläufen erreicht wird.
- **While** : Die While-Aktivität ist der For Each-Aktivität sehr ähnlich, unterscheidet sich jedoch dadurch, dass eine Aktivität solange ausgeführt bis eine Bedingung nicht mehr erfüllt ist.
- **If** : Bei einer If-Aktivität wird die Aktivität nur dann ausgeführt, wenn einen Bedingung erfüllt ist.
- **Flow** : Die Flow-Aktivität ermöglicht die parallele Ausführung von Aktivitäten bzw. die Ausführung von Aktivitäten abhängig von einem definiertem Ablaufgraph
- **ONALARM/Wait** : Hier wird der Prozessablauf solange angehalten, bis ein gewisser Zeitraum überschritten wird oder ein Ereignis eintritt.
- **Repeat Until** : Hier wird lediglich nach dem Durchlauf des Schleifenrumpfes evaluiert. Der Schleifenrumpf wird daher mindestens einmal ausgeführt.

Basisaktivitäten: Basisaktivitäten sind atomare Operationen, die sich nicht aus mehreren Aktivitäten zusammensetzen, Folgende Aktivitäten gehören zu dieser Kategorie

- **Assign** : Die Assign-Aktivität weist einer oder mehreren Variablen Werte zu
- **Receive**: Die Receive-Aktivität ermöglicht es, dass ein Prozess den Start oder die Fortführung eines Prozess hinauszögern kann, bis eine Antwort des Web Services eintrifft.
- **Reply** : Mithilfe einer Reply-Aktivität kann eine Nachricht an einen Empfänger geschickt werden.
- **Invoke** : Eine Invoke-Aktivität ermöglicht den asynchronen bzw. synchronen Aufruf eines Web Services.

Ein BPEL-Prozess setzt sich aus unterschiedlichen Komponenten zusammen. Web Services, die von einem BPEL-Prozess aufgerufen werden können mithilfe von *Partner Link*seingebunden werden. Jede Aktivität, die einen Web Service aufruft, verfügt über einen Partner Link. BPEL nutzt das Konzept von sogenannten **Roles** und **Partner Link Types**. Partner Link Types beschreiben, wie zwei über WSDL definierte Partner miteinander agieren können und was diese zur Verfügung stellen. Partner Link Types sind WSDL-Erweiterungen, die sich aus einem Typ, einen Namen und einen oder mehreren Roles zusammensetzen. Dabei kann jede Role bestimmte Operationen unterstützen. Diese werden durch Port-Types dargestellt.

WS-BPEL trennt dabei abstrakte Informationen von konkreten technischen Details. Dies wird durch abstrakte WSDL-Schnittstellen erreicht, welche keinerlei Informationen darüber ausgeben, wie die

Bindings konkret aussehen und welche konkreten Services seitens der Prozessinstanzen verwendet werden.

2.4 Service Oriented Architecture

Der Begriff **Service oriented Architecture** wurde erstmals im Jahr 1996 vom Marktforschungsunternehmen Gartner erwähnt bzw. verwendet [SN96]. Service-orientierte Architekturen, bezeichnen keine konkrete Architektur oder Technologie, sondern ein abstraktes Konzept einer Software-Architektur, in welchen das Anbieten, Suchen und Nutzen von Diensten (**Services**) über ein Netzwerk im Vordergrund steht. Dienste werden dabei hauptsächlich von Anwendungen oder anderen Diensten in Anspruch genommen. Hierbei ist es unerheblich, welche Hard- oder Software, Programmiersprache oder Betriebssystem die einzelnen Beteiligten verwenden. Services sind kleine, lose gekoppelte und eigenständige Softwarekomponenten, welche zu einem Anwendungssystem kombiniert werden können. Dieses Anwendungssystem ist wiederum leicht anpassbar und änderbar. Einheitliche Standards erlauben es, Dienste durch entsprechende Suchfunktionen zu finden, welche von Anbietern gleichermaßen problemlos publiziert werden können.

2.4.1 SOA Definition

Da es eine Vielzahl von Definitionen für Service-orientierte Architekturen gibt, ist es schwierig sich auf eine Standard-Definition festzulegen. Es bestehen bei diesen Definitionen Ueberlappungen, jedoch fehlen allerdings häufig Aspekte, die von einer anderen Definition als entscheidend betrachtet werden. Es existiert keine allgemein einheitliche Definition einer SOA. [Mel10]definiert SOA wie folgt:

„Unter einer SOA versteht man eine Systemarchitektur, die vielfältige, verschiedene und eventuell inkompatible Methoden oder Applikationen als wiederverwendbare und offen zugreifbare Dienste repräsentiert und dadurch eine von Plattformen und Programmiersprachen unabhängige Nutzung und Wiederverwendbarkeit ermöglicht“

Eine weitere Definition von OASIS aus dem Jahr 2006 [MLM+06]:

„SOA ist ein Paradigma für die Strukturierung und Nutzung verteilter Funktionalität, die von unterschiedlichen Besitzern verantwortet wird“

2.4.2 Grundlegende Merkmale einer SOA

Im Vergleich zu Ansätzen wie **Remote Method Invocation** und **Remote Procedure Call** repräsentiert eine Service-orientierte Architektur keine konkrete Technik, sondern abstrahiert unwesentliche Aspekte und stellt widerum wesentliche Apekte in den Vordergrund.

Ein wesentlicher Vorteil von SOA-Architekturen ist die **Unabhängigkeit** von den Details der jeweiligen Implementierung. SOA beschreibt losgelöst von konkreten Implementierungsdetails ein Architekturstil. Dadurch ist eine prozessorientierte Betrachtungsweise sowie eine funktionale Zerlegung der Anwendungen möglich. Dienste stellen ihre Funktionen über öffentliche Schnittstellen zur

Verfügung und können selbst Funktionen anderer Dienste über das Netzwerk in Anspruch nehmen. Gegebenenfalls ist auch die Integration ganzer Anwendungen möglich.

Dienste sind kleine, loose gekoppelte (**Loose Coupling**) und eigenständige Softwarekomponenten, welche zu größeren Anwendungssystemen kombiniert werden können. Diese sind flexibel und anpassbar. Dienste werden von Anwendungen oder anderen Diensten bei Bedarf dynamisch, d.h. zur Laufzeit gesucht, gefunden und eingebunden. Diese lose Kopplung der Dienste hat zur Folge, dass zum Zeitpunkt der Übersetzung des Programms zumeist nicht bekannt ist, wer oder was zur Laufzeit aufgerufen wird. Ferner erlaubt das dynamische Einbinden von Diensten, dass Dienste miteinander ausgewechselt werden können. Des Weiteren können mehrere Dienste miteinander kombiniert und somit größere Dienste wie z.B. Geschäftsprozesse aufgebaut werden (Orchestrierung), was die Wiederverwendbarkeit von Diensten erhöht. Derart gekapselte Dienste können in verschiedenen Umgebungen mehrfach und ohne Aufwand wiederverwendet werden.

Damit ein Benutzer geeignete Anwendungen und Dienste finden und verwenden kann, wird das Prinzip der gelben Seiten bei der Umsetzung einer SOA angewandt. Alle verfügbaren und publizierten Dienste sind in einem **Verzeichnisdienst** oder Repository (Service Registry) registriert. Anwendungen können bei Bedarf im Verzeichnisdienst nach Diensten suchen und diese mithilfe der Informationen, die von der Repository zurückgeliefert werden, dynamisch einbinden. Nachdem ein geeigneter Dienst nach erfolgreicher Suche gefunden wurde, sollte der Aufrufer in der Lage sein, sich mit diesem zu unterhalten. Dies setzt jedoch voraus, dass alle Schnittstellen in maschinenlesbarer Form beschrieben sind und offene Standards verwendet werden, damit der Nutzer den Dienst eines unbekannten Anbieters auch verstehen kann.

2.4.3 Das SOA Dreieck

Die Abbildung 2.7 stellt alle Komponenten einer Software-orientierten Architektur und die Beziehungsstruktur zwischen diesen dar. Das **SOA-Dreieck** setzt sich aus den drei Komponenten Dienstverzeichnis(**Service Registry**), Dienstanbieter (**Service Provider**) und Dienstnutzer (**Service Consumer**) zusammen.

Hierbei wird ein Dienst (*Service*) als ein eigenständiges und über ein Netzwerk durch nachrichtenbasierte Kommunikation nutzbares Softwareelement bezeichnet, das Funktionen nach außen anbietet und dessen exportierte Schnittstelle durch eine eindeutige Spezifikation beschrieben ist, die öffentlich oder für eine Zielgruppe zugänglich ist.

Der *Dienstanbieter* beschreibt seinen angebotenen Dienst in einer maschinenlesbaren Service-Beschreibung und registriert diesen im Verzeichnisdienst. Dieser Vorgang wird auch *publish*-Prozess genannt. Zusätzlich werden Metadaten hinzugefügt, welche Informationen, wie z.B. die IP-Adresse, die angebotene, öffentliche Schnittstelle und eine abstrakte Beschreibung für das Auffinden und Aufrufen des angebotenen Dienstes, beinhalten. Der Dienstanbieter ist verantwortlich für die Bereitstellung der notwendigen Infrastruktur, das Deployment, die Sicherung des Quality-of-Service und Datensicherheit. Diese Informationen können als Bedienungsanleitung betrachtet werden, welche dem Dienstnutzer beschreiben, wie der Dienst in Anspruch genommen werden kann.

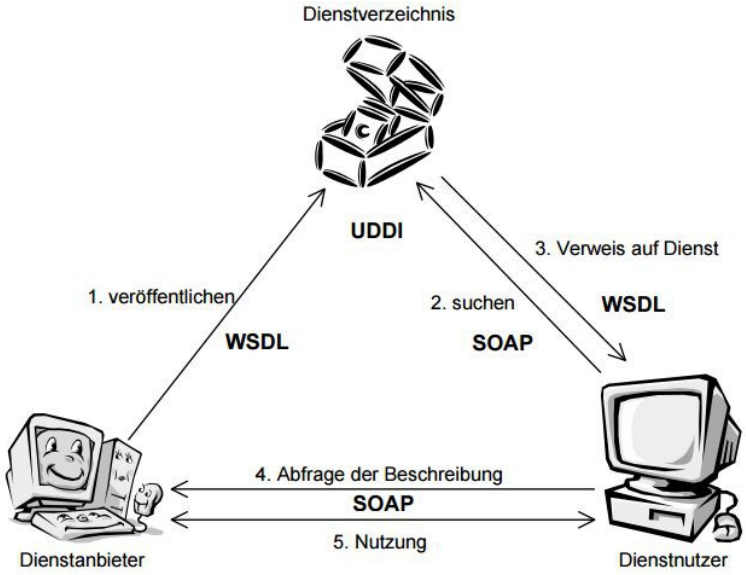


Abbildung 2.7: Das SOA-Dreieck

Der *Dienstnutzer* kann nun im Verzeichnis nach einem geeigneten Dienst suchen, indem dieser eine Beschreibung des angeforderten bzw. gesuchten Dienstes an das Verzeichnisdienst schickt. Der Verzeichnisdienst sucht daraufhin nach einem geeigneten Dienst und übergibt bei erfolgreicher Suche die ID des Dienstanbieters an den Dienstnutzer. Dieser fordert anschließend die Schnittstellenbeschreibung des Dienstanbieters an und kann mithilfe der darin enthalten Informationen den Dienst verwenden.

Ein Anbieter, der einen Dienst in Form eines Web Service anbieten möchte, erstellt von diesem zunächst eine Schnittstellenbeschreibung in Form eines entsprechenden XML-Dokuments. Dieses so genannte *WSDL*(Web Service Definition Language)-Dokument wird veröffentlicht, indem es ganz oder in definierten Teilen zu einem UDDI-basierten Verzeichnisdienst transferiert wird.(siehe Abbildung 2.7, Schritt 1). Anschließend wartet der Dienstanbieter bis ein Dienstnutzer einen entsprechenden Dienst sucht (Schritt 2). Laut Spezifikation müssen UDDI-Implementierungen zu diesem Zweck eine SOAP-Schnittstelle zur Verfügung stellen, die vom UDDI-Gremium mittels WSDL-Dokumenten beschrieben ist. Hat der Dienstnutzer einen für sich geeigneten Web Service gefunden, fordert er das WSDL-Dokument an. Der Verzeichnisdienst liefert hierzu eine Referenz (*URI*) auf das WSDL-Dokument, das der Dienstnutzer in einem weiteren Schritt anfordert (Schritt 4). Anschließend werden mit Hilfe der WSDL-Beschreibung die Programmteile erzeugt, welche die Anwendung des Dienstnutzers in

die Lage versetzen mit der Anwendung des Dienstanbieters mit Hilfe von SOAP zu kommunizieren (Schritt 5).

2.4.4 Web Services

Web Services sind ein möglicher Implementierungsansatz der Konzepte einer Service-orientierten Architektur. Wie auch beim Begriff SOA-Architektur existiert keine einheitliche, standardisierte und konsistente Begriffsdefinition für Web Services. Web Services können als Softwarekomponenten oder unabhängige, modulare Dienste aufgefasst werden, welche wohldefinierte Funktionen ueber standardisierte Schnittstellen anderen Softwarekomponenten oder Anwendungen zur Verfuegung stellen [Bet01]. Nach [ABFG04]lassen sich Web Services wie folgt beschreiben:

"A Web service is a software application identified by a URI, whose interface and bindings are capable of being defined, described, and discovered as XML artifacts. A Web service supports direct interactions with other software agents using XML-based Messages exchanged via internet-based protocols. (October 2002)"

Die Gartner Forschungsgruppe definiert Web Services folgendermaßen:

"Web services are software technologies, making it possible to build bridges between IT systems that otherwise would require extensive development efforts."

Die Standardisierung der Schnittstellen ermöglicht die lose Kopplung von Web Services untereinander beziehungsweise von Web Services und anderen Anwendungen. Falls sich die Schnittstellensignatur der Dienste nicht unterscheidet, kann der Nutzer die Softwarekomponenten problemlos austauschen.

2.4.5 Die Bestandteile von Web Services

Der folgende Abschnitt basiert auf [Lan03].

Das **Simple Object Access Protocol** (SOAP) ist ein Netzwerkprotokoll, welches das Format festlegt, mit der Nachrichten zwischen Web Services ausgetauscht werden. Es basiert auf der XML-Syntax und kann daher von Standard-Parsern eingelesen werden, ohne dass hierfür eigene Lösungen definiert werden müssen. Aufgrund seiner Unabhängigkeit von Transportprotokollen kann SOAP mit anderen Transportprotokollen wie SMTP oder HTML eingesetzt werden. SOAP umfasst lediglich die Formatierung von Daten, nicht jedoch, wie die Datenübertragung von A nach B technisch realisiert werden kann.

Eine SOAP-Nachricht entspricht einem gemäß dem SOAP-Format codierte Information und ähnelt bildlich dargestellt einem Brief. Der *Envelope* enthält als Briefumschlag die zu versendende Nachricht. Diese Nachricht entspricht einem Container, der aus einem oder mehreren *Headern* (Kopfzeile) und einem *Body*-Teil (Nachrichtentext) besteht. Im Header sind Absenderinformationen untergebracht und im Body befindet sich der eigentliche Nachrichtentext, der die Informationen über die zu versendenden Daten enthält. Dabei sind die Header optional und enthalten Informationen für den Empfänger oder wichtige Informationen, die für die Weiterverarbeitung durch Zwischenstationen auf

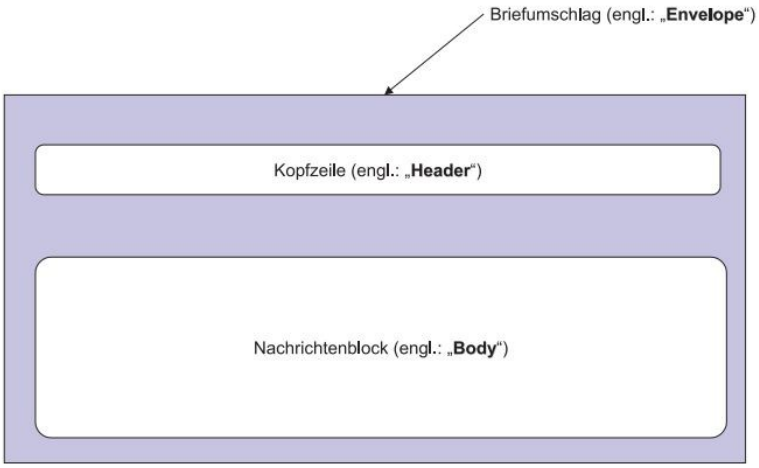


Abbildung 2.8: Struktur einer SOAP-Nachricht

dem Transportweg zwischen Absender und dem Empfänger der Nachricht notwendig sind. Demnach kann mithilfe des Attributs *mustUnderstand*, welches auf true gesetzt wurde, festgelegt werden, dass der Empfängerknoten das erhaltende Header-Element verarbeiten muss, bevor die Nachricht weitergeleitet beziehungsweise weiterverarbeitet werden kann. Ferner können durch Attribute Konditionen für die Authentifizierung oder Datenverschlüsselung definiert werden. Im Gegensatz zum Header ist der Body einer SOAP-Nachricht nicht optional. Abbildung 2.8 stellt die Analogie zwischen einer SOAP-Nachricht und einem Brief dar.

WSDL ist die Kurzform für *Web Services Description Language* und umfasst eine auf XML basierende, programmiersprachen-, protokoll- und plattformunabhängige Sprache, mit der das Dienstangebot eines Servers beschrieben werden kann. Mit WSDL kann definiert werden, welche Operationen bzw. Methoden der Serverkomponente vom Client ausgeführt werden können, sowie welche Parameter übergeben und welchen Rückgabewert die einzelnen Methoden liefern. Ein Web Service wird hierbei auf zwei Ebenen betrachtet und beschrieben. Folglich kann ein WSDL-Dokument in zwei Teile aufgeteilt werden. Zum Einen existiert eine abstrakte und wiederverwendbare Definition, welche die Funktionalitäten, die vom Web Service bereitgestellt werden, beschreiben und zum anderen eine implementationsabhängige konkrete Definition, welches alle technischen Details umfasst, mit deren Hilfe ein Web Service zur Verfügung gestellt wird.

Ein WSDL-Dokument beschreibt anhand von folgenden Elementen einen Web Service:

- **types** für die Definition von Datentypen, die verwendet werden

- **message** abstrakte Definition der zu übertragenden Daten
- **port type** (Schnittstellentypen)abstrakte Menge von Operationen, die von einem oder mehreren Endpunkten (ports) unterstützt werden
- **binding** eine konkrete Protokoll- und Formatspezifikation für einen bestimmten port type
- **port** ein einzelner end point definiert mithilfe einer Kombination von einem binding und einer Netzwerkadresse
- **service** umfasst alle ports eines port types

Diese Elemente lassen sich unterteilen in eine Menge von abstrakten Definitionen und eine weitere Menge von konkreten Definitionen. Zu den abstrakten Definitionen werden *types*, *messages* und *port type* gezählt, während konkrete Definitionen sich zusammensetzen aus *bindings*, *endpoints* (ports) und *services*. Der abstrakte Teil eines WSDL-Dokuments ist unabhängig vom jeweils verwendeten Transportprotokoll. Dieser wird innerhalb eines Binding-Elements untergebracht, wodurch der Web Service als solches technologieunabhängig ist.

UDDI steht für *Universal Description, Discovery* and Integration und ist ein standardisierter Verzeichnisdienst für sämtliche Web Services. Nachdem ein Web Service mittels WSDL definiert wurde, muss es veröffentlicht werden. WSDL enthält bereits alle Informationen, mit denen ein Dienstnutzer die Dienstleistung in Anspruch nehmen kann. UDDI ist ein Standard, der diese Informationen zugänglich macht. Ein Dienstanbieter kann über eine WSDL-Information hinaus weitere Informationen innerhalb einer UDDI Registry ablegen. So können zusätzlich noch Business-, Service- und Technikinformationen untergebracht werden, welche auch mit den Namen White Pages, Yellow Pages und Green Pages bezeichnet werden.

Die *White Pages* ähneln einem Telefonbuch und enthalten Informationen über den Dienstanbieter. Dazu gehören Angaben über den Geschäftsbereich, Kontaktdaten und eine eindeutige Unternehmens-ID-Nummer. Die *Yellow Pages* dagegen beschreiben den eigentlichen Dienst, der zur Verfügung gestellt wird. Diese entspricht einem Branchenverzeichnis, in der alle registrierten Dienste gemäß internationaler Standards kategorisiert werden. Die Schnittstellenbeschreibungen der Web Services werden in den *Green Pages* bereitgestellt.

2.5 Pipes And Filters-Architektur

Der *Pipes and Filter* Architekturstil ist ein Architekturmuster, welches ein System als eine Reihe von Filteroperationen auf Eingabedaten betrachtet. Es eignet sich demnach für Systeme, die Datenströme verarbeiten. Jeder Verarbeitungsschritt ist in einem Filter gekapselt. Daten werden über Komponenten weitergeleitet und erreichen einen Endpunkt. Filter sind über Kanäle (*Pipes*) miteinander verbunden, welche den Datentransfer zwischen zwei Komponenten ermöglicht.

Die Filter lassen sich beliebig neu anordnen, hintereinander schalten und austauschen. Dies ermöglicht es Familien von verwandten Systemen zu erzeugen.Da mehrere Filter auch parallel Daten verarbeiten können und jede Filterkomponente stufenweise Daten konsumiert und weiterleitet, steigt dadurch der

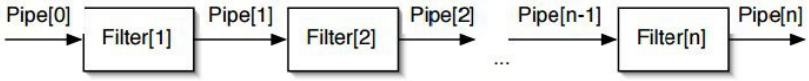


Abbildung 2.9: lineares Pipes and Filter Architekturmodell [RHJN04]

Durchsatz einer einzelnen Komponente. Pipes agieren als Zwischenspeicher zwischen benachbarten Filterkomponenten.

Eine Form dieses Architekturstils ist die lineare Pipeline. Hier verfügt eine Filterkomponente über genau eine Eingangs-Pipe und eine Ausgangs-Pipe. In Abbildung 2.9 wird ein lineares Pipes and Filter Architekturmodell veranschaulicht. Ausgereiftere Formen des Pipes and Filter Architekturmodells können im Gegensatz dazu mit datenzentierten Architekturen wie *Shared Repository*, *Blackboard* oder *Active Repository* kombiniert werden, um den Datenaustausch zwischen Filtern zu gewährleisten [AZ05].

Der Pipes and Filter Architekturstil bietet zwei wesentliche Vorteile :

- **Modularität**: jeder Filter kann modifiziert oder neu platziert werden, ohne dabei alle anderen Filter zu beeinträchtigen
- **Wiederverwendbarkeit**: zahlreiche Filter existieren bereits und können wiederverwendet werden

Weitere Vorteile sind, dass Rapid Prototyping von Pipeline Prototypen ermöglicht wird und Zwischendateien nicht notwendig sind aber so gewünscht ermöglicht werden. Nachteile sind, dass die Kosten der Datenübertragung zwischen den Filtern je nach Pipe sehr hoch sein können und dass häufig überflüssige Datentransformationen zwischen den einzelnen Filterstufen notwendig sind. Zudem ist die Fehlerbehandlung über Filterstufen hinweg teilweise schwierig.

In einer Pipes and Filters Architektur wird eine komplexe Aufgabe in mehrere sequentielle Teilaufgaben aufgeteilt. Diese werden von einer separaten, unabhängigen Komponente, der Filter-Komponente, implementiert, welches sich nur auf diese eine Teilaufgabe konzentriert. Darüber hinaus besitzt jede Filter-Komponente eine Reihe von Input-Eingängen und Output-Ausgängen. Filter können flexibel mithilfe von sogenannten Pipes verbunden werden, über welche Daten transportiert werden. Eine Pipe realisiert demnach den Datenstrom zwischen zwei Komponenten. Da mehrere Filter auch

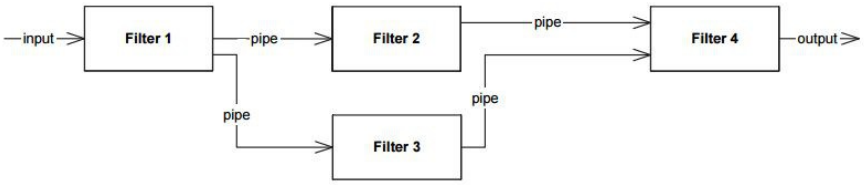


Abbildung 2.10: Pipes and Filters Beispiel [AZ05]

parallel Daten verarbeiten werden können und jede Filterkomponente stufenweise Daten konsumiert und weiterleitet, steigt dadurch der Durchsatz einer einzelnen Komponente. Pipes agieren als Zwischenspeicher zwischen benachbarten Filterkomponenten.

Die Anwendung des Pipes and Filters Architekturmodells eignet sich besonders in Anwendungsfällen, in denen zwischen Filterkomponenten geringe Kontextinformationen bewahrt bzw.ausgetauscht werden und Filter keine Statuszustände zwischen den Aufrufen speichern müssen. Pipes und Filter können flexibel miteinander kombiniert werden. Der Austausch von Daten zwischen den einzelnen Komponenten kann jedoch teuer und unflexibel sein. Ferner gibt es Leistungsüberlastungen bei der Übertragung von Daten in Pipes und Datentransformationen und die Fehlerbehandlung ist relativ schwierig.

Abbildung 2.10 stellt ein Pipes and Filter Beispiel grafisch dar.

Im Gegensatz zu Batch Sequential, in welcher es keine explizite Abstraktion für Konnektoren gibt, kommen beim Pipes and Filter Architekturstil dem Pipe-Konnektor höchste Bedeutung für die Übertragung von Datenströmen zugute. Das Hauptmerkmal ist die Flexibilität bei der Verknüpfung von Filtern mithilfe von Pipes. Dadurch lassen sich anwendungsspezifische Konfigurationen erstellen, die spezifische Problemfälle lösen. In der puren Form können lediglich zwei adjazent angeordnete Filter-Komponenten Daten untereinander austauschen.

Ausgereifte Formen des Pipes and Filter Architekturmodells können im Gegensatz dazu mit datenzentierten Architekturen wie Shared Repository, Blackboard oder Active Repository kombiniert werden, um den Datenaustausch zwischen Filtern zu gewährleisten [AZ05].

2.6 Data Mashups

In diesem Kapitel wird der Begriff *Mashup* erläutert und die Funktion von Mashup-Anwendungen beschrieben. Mashup ist eine relativ neuer Ansatz, welche es dem Nutzer ermöglicht, mehrere Dienste zu kombinieren, um daraus einen vollständig neuen Dienst mit einer anderen Funktionalität erstellen zu können. Eine Form von Mashups sind sogenannte **Data Mashups**. Da diese Arbeit hauptsächlich auf Data Mashup fokussiert, wird der Begriff Data Mashup vorgestellt und einige Beispiele für Data Mashup Tools aufgelistet.

2.6.1 Mashups

Der Begriff **Mashup** umfasst eine Web-Technologie, welches die einfache Erstellung von web-basierten Anwendungen durch Endnutzer ermöglicht. Mashup sind Datenaggregations-Anwendungen, welche Daten von verrschiedenen Datenquellen kombinieren, um verwertbare Informationen zu erstellen. [DVXB+09]

Mashup-Anwendungen werden aus mehreren User-Interface-Komponenten oder Artefakten und dem Inhalt multipler Datenquellen zusammengesetzt. Im Bereich des Software Engineering beschreibt ein Mashup die Kombination von bestehenden User Interface-Artefakten- Prozessen, Diensten und Daten für die Erzeugung von neuen Web-Seiten, Anwendungen, Prozessen und Datensätzen. In einer Mashup-Umgebung können Benutzer durch die Wiederverwendung von Artefakten bereits bestehender User Interfaces(UI), neue UIs mithilfe von High-Level Scripting-Sprachen wie HTML oder JavaScript erstellen.

2.6.2 Eigenschaften von Data Mashups

Data Mashups, welche auch unter dem Namen *Enterprise Mashups* bekannt sind, sind Technologien, die von Unternehmen verwendet werden, um auf Daten (semi-) strukturierter und unstrukturierter Datenquellen zugreifen, extrahieren und integrieren zu können.

[HSSJS08] beschreibt ein Enterprise Mashup wie folgt:

„An enterprise mashup is a Web-based resource that combines existing resources, be it content, data or application functionality, from more than one resource by empowering end users to create individual information centric and situational applications“

Data Mashups automatisieren die Extraktion von Web-Daten und ermöglichen die Strukturierung von unstrukturierten Daten, welche schließlich mit unternehmensbezogenen Daten verknüpft werden können. Im Gegensatz zu Web Mashups sind Enterprise Mashups informations- und datenzentriert. Dabei werden Transformationen und Semantiken verwendet, um unstrukturierte Daten zu strukturieren und mit anderen Datenquellen in Verbindung zu bringen. Dadurch unterstützen Enterprise Mashups die Integration in Unternehmen und End-User Mashups.

Ziel von Data Mashup Plattformen ist die flexible ad-hoc Integration von heterogenen Datenquellen. Im Gegensatz zu Applicaton Mashups, liegt ihr Fokus auf dem Data-Layer. Data Mashups werden durch

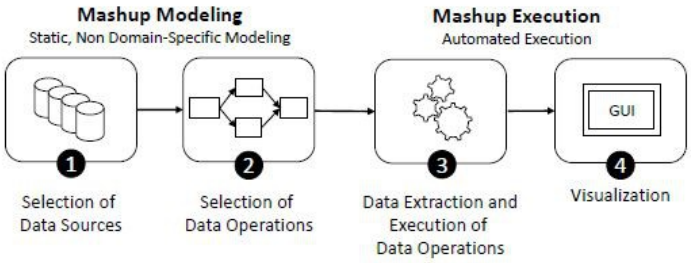


Abbildung 2.11: Data Mashup

mithilfe von Datenoperationen wie *Filter* und *Join* definiert. Das Ergebnis ist eine Datenquelle, welche alle integrierten Daten enthält. Der Benutzer legt fest welche Daten extrahiert und zusammengesetzt werde, bevor der Mashup ausgeführt wird.

In Abbildung 2.11 werden die einzelnen Stufen der Mashup Modellierung dargestellt.

Zunächst bestimmt der Benutzer die Datenquellen und welche Daten extrahiert werden sollen. Anschließend werden die Datenoperationen und ihre Reihenfolge bestimmt. Danach werden die automatisierten Schritte des Data Mashups ausgeführt. In Schritt drei erhält die Mashup Applikation die Benutzereingaben, extrahiert die Daten und führt die Operationen in der definierten Reihenfolge aus. Danach wird das Ergebnis visualisiert oder abgespeichert [HRWM15].

2.6.3 Data Mashup Tools

Es existieren zahlreiche Data Mashup Tools. Zu den beliebtesten gehören folgende [DLHPB09]:

- **Damia**: Damia wurde von IBM erstellt und erlaubt dem Benutzer Data Feeds aus unterschiedlichen Quellen, wie Internet und aus dem Geschäftsbereich zu sammeln. Es legt den Fokus auf Data Feed Aggregation und Transformation in Unternehmensumgebungen. Zusätzliche Tools wie beispielsweis QEDWiki und Feed Reader, welche Atom und RSS verwenden, können in der Präsentationsschicht für den Data Feed verwendet werden, der von Damia bereitgestellt wird.

- **Yahoo Pipes**: Yahoo Pipes ist eine web-basierter Tool von Yahoo, welcher die Erstellung von Mashup-Anwendungen durch die Aggregation und Manipulation von Daten aus Web Feeds, Web-Seiten und anderen Diensten ermöglicht. Dabei setzt sich eine Pipe aus mindestens einem Modul zusammen, wobei jedes Modul eine einzelne Aufgabe ausführt. Der Output einer Pipe kann anschließend entweder von einem Client mithilfe einer eindeutigen URL als RSS or JSON aufgerufen werden oder wird in YahooMap visualisiert.
- **MashMaker**: Intel MashMaker ist ein weiteres web-basiertes Tool für die das Abfragen und die Manipulation von Web-Daten. Im Gegensatz zu anderen Tools arbeitet MashMaker jedoch direkt auf den Web-Seiten, d.h. es ermöglicht Nutzern Mashups durch das Browsen und Kombinieren von unterschiedlichen Web-Seiten zu erstellen. Ziel dabei ist es, Nutzern eventuelle Verbesserungen in Form von Mashups oder Widgets für besuchte Web-Seiten bereitzustellen.
- **Google Mashup Editor**: Google Mashup Editor, GME, ist eine Umgebung von Google für die Entwicklung, den Einsatz und die Verteilung von Mashups. Dabei kann ein Mashup mithilfe von Technologien wie HTML, JavaScript, CSS verknüpft mit GM XML Tags und Java Script API erstellt werden, welche zudem Nutzern erlauben die Darstellung von Mashup Outputs individuell zu gestalten.

2.6.4 Vorteile und Nachteile von Data Mashups

Auch wenn Mashup-Ansätze neue Möglichkeiten für Daten- bzw. Service-Nutzer eröffnen, erfordert der Entwicklungsprozess jedoch nicht nur, dass Nutzer technisches Wissen mitbringen,um mithilfe von Programmiersprachen Code schreiben zu können, sondern auch die Fähigkeit und das Wissen darüber, wie die unterschiedlichen Web APIs der zahlreichen Dienste zu verwenden sind. Tools können hierbei bis zu einem gewissen Grad behilflich sein, da sie mit dem Ziel entwickelt worden sind, jene Benutzer mit wenig Programmierkenntnissen im Bereich der Entwicklung von Mashup-Anwendungen zu unterstützen.

Mashup Tools verarbeiten hauptsächlich Web-Daten. Dies ist vorteilhaft, da dadurch der Zugriff und die Verwaltung von Daten, die nur über das Web zugänglich sind, ermöglicht wird. Es ist jedoch zu beachten, dass Daten, die sich auf den einzelnen Desktops befinden nicht gleichermaßen verwendet werden können. Lokale Daten, welche eventuell auch eine wichtige Bedeutung haben können, werden dadurch nicht beachtet.

Der Großteil der angebotenen Tools besitzen ein internes Datenmodell, welches auf XML basiert. Dies ist bedingt durch die Tatsache, dass Daten im Web hauptsächlich im XML-Format vorliegen. Ferner verwenden Kommunikations-Protokolle für den Datenaustausch zumeist XML-Nachrichten. Neben dem XML-basierten internen Datenmodell übernimmt das objekt-basierte Datenmodell eine wichtige Rolle. Für die Verwaltung von Daten (Daten-Integration und -Manipulation) stellen Tools lediglich eine kleine Menge von Operationen zur Verfügung. Dabei wurden diese Operatoren mit dem Fokus auf die Zielfunktion des Tools entwickelt. Dadurch sind diese nicht leicht verwendbar und können beispielsweise keine komplexeren Anfragen realisieren. Mashup Tools sind erweiterbar, wobei neue Operatoren, Datenschemata entwickelt und aufgerufen bzw. in das Tool integriert werden. Die meisten Tools unterstützen jedoch nicht die Wiederverwendung von bereits erstellten Mashups. Ferner sind dies angebotenen Tools in vielen Bereichen eingeschränkt und Nutzer können ihre

Anforderungen durch die Verwendung eines einzelnen Tools nicht realisieren. Eine der größten Nachteile von Mashup Tools ist jedoch, dass obwohl Nutzer mit geringen technischen Kenntnissen diese nutzen können, gewisse Programmierkenntnisse vorausgesetzt werden. Benutzern werden grafische Benutzeroberflächen zur Verfügung gestellt, um Operationen ausführen zu können.

2.7 Mashup Plans

In diesem Abschnitt wird der Ansatz von Mashup Plans beschrieben. Dieser Ansatz basiert auf dem Pipes and Filter-Ansatz aus Kapitel2.5. Auf diesem Mashup Plan basiert das Mashup Tool FlexMash. Im Folgenden werden Mashup Plans beschrieben und erläutert, wie diese modelliert und in ein ausführbares Format transformiert werden. Mashup Plans sind ein wesentlicher Bestandteil dieser Arbeit. Die folgenden Abschnitte basieren auf [HRWM15] und [HM16].

2.7.1 Extended Data Mashup Ansatz

[HM16] präsentiert in seiner Arbeit den Extended Data Mashup Ansatz. Dieser bietet eine höhere Flexiblität für die Modellierung und Ausführung von Data Mashups als der herkömmliche Data Mashup-Ansatz und ist in drei Stufen eingeteilt:

- Modellierungsstufe
- Transformationsstufe
- Ausführungsstufe

Ein Domänenexperte definiert die Datenquellen und die Datenoperationen mithilfe eines domänenspezifischen Modells, dem **Mashup Plan**. Danach werden Transformationspatterns ausgewählt, um eine Implementierung finden zu können, die die Anforderungen erfüllt (z.B. robust oder zeitkritisch). Anschließend wird der Mashup Plan, abhängig vom gewählten Transformationspattern, in ein ausführbares Model transformiert. Dieser wird danach auf einer geeigneten Engine ausgeführt. Das Ergebnis wird in einer Datenbank gespeichert und ist für Visualisierung, Datenanalyse etc. zugänglich.

2.7.2 Mashup Plan Modellierung

Bei der Modellierung eines Mashup Plans müssen diverse Beschränkungen beachtet werden:

Jeder Mashup Plan beinhaltet genau einen Startknoten, mindestens einen DSD und DPD und einen Endknoten. Die technischen Details und Eigenschaften von DPDs und DSDs, die zur Modellierung von Mashup Plans verwendet werden, werden von IT-Experten erstellt und in den entsprechenden Repositories abgelegt. Dadurch wird es Domänen-experten ermöglicht, bei der Erstellung von Mashup Plans auf diese Repositories zuzugreifen, ohne die technischen Details bestimmen zu müssen.

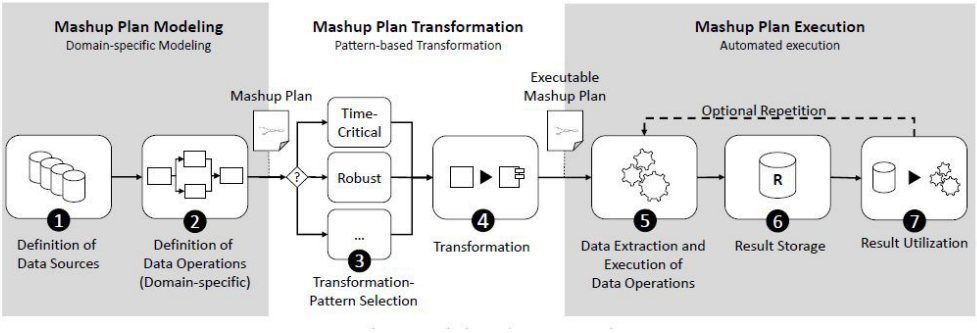


Abbildung 2.12: Extended Mashup Ansatz [HRWM15]

Mashup Plan

Mashup Plans sind verwandt mit dem Ansatz der Pipes and Filter und werden von Domänenexperten erstellt, welche wenig technisches Wissen und Programmierkenntnisse besitzen. Ein Mashup Plan ist ein gerichteter, zusammenhängender und nicht ausführbarer Flussgraph bestehend aus Knoten und Kanten. Die Kanten beschreiben den Daten- und Kontrolfluss zwischen den Knoten, welche unterteilt werden können in **DSD**s (Data Source Descriptions) und **DPD**s (Data Processing Descriptions. DSDs basieren auf Business Objects, den sogenannten Artefakten und ermöglichen Nutzern, ohne Kenntnisse über technische Details, Datenquellen zu modellieren.

Ein Mashup Plan enthält mindestens eine DSD und eine DPD. Eine DSD bildet den Startknoten eines Mashup Plans, wobei jeder Mashup Plan einen Endpunkt hat. Die verwendeten DPDs und DSDs werden in entsprechen Verzeichnissen gespeichert, auf die der Nutzer bei der Modellierung zugreifen kann.

Abbildung 2.13 stellt einen Mashup Plan grafisch dar.

DPD und DSD Modellierung

Jede DSD enthält Informationen über die Lokation der Datenquelle und ihre Zugriffsinformation (z.B. database port, URL etc.). DSDs bestimmen die zu integrierenden Daten in für Domänenexperten lesbaren Formaten. Dazu können Artefakte verwendet werden, welche die Daten repräsentieren. Diese Artefakte entsprechen geschäftsrelevanten Objekten und abstrahieren von den spezifischen Daten. Dies können beispielsweise Produktionsmaschinen sein, Informationssysteme usw. Diese Artefakte verwalten relevante Informationen über diese Geschäftsobjekte und ihre Lebenszyklen in abstrake Form. Diese Artefakte erleichtern Domänenexperten die Modellierung von Datenquellen.

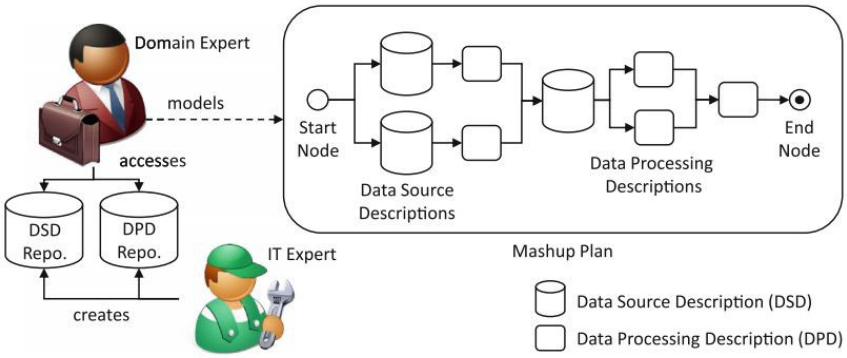


Abbildung 2.13: Mashup Plan [HRWM15]

Parametrisierte und abstrakte DPDs beschreiben Datenoperationen eines Mashups ohne dabei die technischen Details zu verwenden. DPDs können in *Data Selection* und *Data Combination* DPDs unterteilt werden. Ein DPD beschreibt im Gegensatz zu einem DSD, wie Daten verarbeitet bzw. modifiziert werden. Demnach entspricht ein DPD einer Operation, d.h. einem Code, der Daten verarbeitet. Die Implementierung eines DPD ist dabei kontextabhängig und es können mehrere Implementierungen für eine einzelne DPD existieren, abhängig von Datentypen, Datenstrukturen etc. Für die Umwandlung von DPDs und DSDs zu den entsprechenden Implementierungen kann der Ansatz von [RSM14] verwendet werden.

2.7.3 Patternbasierte Transformation

Der nicht ausführbare Mashup Plan muss in ein ausführbares Format überführt werden. Das Ergebnis der Transformation ist ein ausführbarer Mashup Plan. Dieser ist ein gerichteter Graph, der ausschließlich aus executable data processing nodes (eDPN) und Datenkontrolfluss-Kanten besteht. Ein eDPN enspricht einer Implementierung, einem Stück Code, welches aufgerufen wird. Informationen über den Datenzugriff und die Eingabeparameter eines eDPNs befinden sich im eDPN-Verzeichnis.

Die patternbasierte Transformation besteht aus 5 Ausführungsschrittegliedern:

- Modellierung des Mashup Plans
- Auswahl der Transformationspattern
- patternbasierte Transformation des Mashup Plans in ausführbares Format

- cloud-basierte Data Mashup Ausführung, abhängig von Benutzeranforderungen
- Speicherung/ Visualisierung des abgeleiteten Resultats

Ausführungsschritt 1 Die Erstellung des Mashup Plans wurde in 2.7.2 ausführlich erläutert und wird darum hier nicht näher beschrieben.

Auswahl des Transformation Patterns

Transformations Patterns enthalten zusätzliche Information über das Szenario, in dem ein Mashup Plan ausgeführt wird. Für jedes Szenario gibt es ein Transformations Pattern. Das Transformation Pattern Time-Critical Mashup beschreibt beispielsweise Anwendungsfälle, in denen die Zeit ein wichtiger Faktor. Die dazu ausgesuchte Implementierng sollte effizient sein. Der Transformation Pattern Robust Mashup hingegen verlangt eine robuste Ausführung, d.h. daten-persistent, mit hoher Verfügbarkeit und Fehlerbehandlung.

Für Patterns wird ein erweiterbares Patternkatalog eingesetzt, welches Informationen über Patterns zur Verfügung stellt. So gibt es Einsicht über alle existierenden Pattern, wie diese miteinander kombiniert werden können und welche Defizite und Einschränkungen sie haben. Jeder Eintrag im Katalog beschreibt eine einzelnes Transformation Pattern und besteht aus mehreren Bestandteilen:

- Beschreibung der Problemstellung, das mithilfe des Patterns gelöst wird
- vom Pattern bereitgestellte Lösung des beschriebenen Problems
- Fallbeispiel, wie der Pattern verwendet werden kann
- Evaluierung
- Information, ob und wie das Pattern mit anderen Transformationspatterns kombiniert werden kann

Der Benutzer muss bei der Auswahl des Patterns zumeist zusätzliche Parameter festlegen, die für das Auffinden der entsprechenden Implementierung erforderlich sind. Demnach muss der Benutzer, die maximale Laufzeit der Ausführung bestimmen, wenn er beispielsweise den zeitkritischen Mashup Pattern ausgesucht hat. Wählt der Nutzer im Gegensatz dazu das Robust Mashup Pattern, muss festgelegt werden, ob Fehlerbehandlung notwendig ist oder Logging unterstützt wird.

Die ausgewählten Patterns beeinflussen, wie das Mashups ausgeführt wird. Das Robust Mashup Pattern z.B. erfordert eine robuste Ausführung des Mashups. Folglich wird eine Workflow Engine als Execution Engine verwendet.

Patternbasierte Transformation

Im dritten Ausführungsschritt wird der nicht ausführbare Mashup Plan anhand der zuvor gewählten Patterns in ein ausführbares Format gebracht. Sowohl Abbildung des Mashup Plans auf ein ausführbares Modell als auch Auswahl der geeigneten Ausführungs-Engine werden mithilfe eines regelbasierten Transformationsansatzes, ähnlich dem Ansatz von [RSM14] ausgesucht. Patterns werden strukturiert und zu einer Implementierung verbunden. Dazu wird ein sogennanter Pattern Graph benutzt. Ein Pattern Graph ist ein baumbasierter, direkter Graph, bestehend aus Knoten und Kanten. Die Knoten stellen entweder ein Pattern oder eine Implementierung dar, Kanten dagegen beschreiben Spezialisierungen. Es gibt zwei Kantentypen:

- consists of- Kanten
- implemented by-Kanten

Die consists of-Kante verbindet Patterns miteinander und deutet darauf hin, dass ein Pattern aus mehreren Sub-Patterns besteht. Demnach kann die bestehende Problemlösung nur gelöst werden, wenn alle Sub-Patterns ausgeführt werden. Die Implemented by-Kante verknüpft Implementierungsknoten miteinander. Ist ein Pattern mit mehreren Implementierungen verbunden, bedeutet dies, dass es von einer dieser Implementierungen realisiert werden kann. Ist dies der Fall, wird die realisierende Implementierung entweder manuell ausgesucht oder automatisch. Im Wesentlichen wird ein anfangs generisches Pattern im Wurzelknoten des Baumgraphen schrittweise konkretisiert und aufgeteilt in Subpatterns bis schließlich in der letzten Stufe diese durch Implementierungsfragmente ersetzt werden.

Patterns können demzufolge über mehrere Abstraktionsstufen hierarchisch strukturiert werden, wobei jedes Pattern durch mehrere Implementierungen realisiert werden kann. Der Wurzelknoten im Patterngraphen entspricht hierbei dem Pattern mit dem höchsten Abstraktionsgrad. Es handelt sich demnach um das Pattern, welches im Patternkatalog beschrieben wird.

Für jeden Eintrag im Patterkatalog exisitiert ein Pattern Graph. Die Parametrisierung des Patterns bestimmt, welcher Pfad im Patterngraphen eingeschlagen wird, um die Implementierungen in den Blattknoten zu erreichen. Diese Entscheidung wird anhand von Regeln festgelegt, die beim Durchlaufen des Patterngraphen ausgewertet werden. Diese Regeln vergleichen die Parameter des Pattern mit vordefinierten Eigenschaften der Implementierungen, um somit die geeignetesten Implementierungen bestimmen zu können. Obgleich keine Implementierung gefunden werden kann, die alle Anforderungen erfüllt, wird letztlich eine Implementierung ausgewählt. Der Benutzer entscheidet daraufhin, ob diese verwendet wird oder nicht. Sobald eine geeignete Implementierung gefunden wurde, wird der Mashup Plan in eine ausführbare Darstellung überführt. Hirmer et al. benutzen modularisierte Implementierungsfragmente, die zusammengescriptet werden und somit das ausführbare Modell erstellen. Wird beispielsweise die Ausführung mittels einer Workfow Engine durchgeführt, kann der ausführbare Workflow automatisch generiert werden, indem Knoten der Business Process Execution Language (BPEL) aufgerufen werden , um die Operationen des definierten Mashup Plans ausführen zu können. Die Programmierlogik von DSDs und DPDs wird in Codefragmenten gespeichert, z.B. als Java Web Services, die vom Workflow ausgeführt werden. Die Transformation läuft bei Ausführung auf einer Node-RED Engine ähnlich ab. Hier werden vordefinierte JavaScript Codefragmente miteinander verbunden.

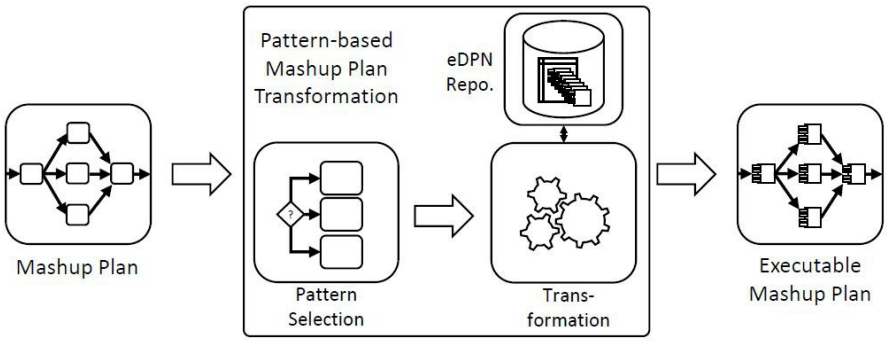


Abbildung 2.14: Komponenten der Mashup Plan Transformation [HRWM15]

TOSCA-basierter Einsatz und Ausführung

Um Data Mashups ausführen zu können, sind mehrere Softwarekomponenten erforderlich, die den Datenfluss bewerkstelligen, die Programmierlogik für DSDs und DPDs bereitstellen und das Ergebnis in einer Datenbank beziehungsweise Data Warehouse visualisieren oder abspeichern. Hirmer et al. setzen sich als Ziel, diese Komponenten lediglich bei Initiierung eines Data Mashups On-Demand bereitzustellen. Diese Komponenten werden einmalig bei der ersten Ausführung des Mashup Flows zur Verfügung gestellt. Wird der Mashup nicht mehr gebraucht, können die bereitgestellen Komponenten wieder abgerüstet werden, um Kosten zu sparen. Hirmer et al. nutzen aus diesem Grund Cloud-Computing-Technologien, den OASIS Standard TOSCA und die Ergebnisse ihrer vorangegangenen Arbeit. Die Komponenten, die bereitgestellt werden sollen, werden beim Durchlaufen des Pattern Graphen ermittelt. Ausgehend von dieser Information kann automatisch eine TOSCA Topologie erstellt werden, die alle erforderlichen Komponenten enthält und Informationen beinhaltet, wie diese miteinander verbunden sind. Dies kann durch das Konzept der Node Templates und Relationship Templates, die im TOSCA Standard beschrieben sind, erreicht werden.

2.7.4 FlexMash

FLexMash ist ein Data Mashup Tool, welches an der Universität Stttgart entwickelt wurde. Es ermöglicht die grafische und domänenspezifische Modellierung von Mashup Plans und ihre Ausführung. Dabei kann der Benutzer (Domänenexperte)seine Anforderungen eingeben, welche die Art der Ausführung bestimmen. Dadurch wird einerseits eine höhere Flexibilität bei der Ausführung gewährleistet

und andererseits der Benutzerkreis vergrößert, da Domänenexperten, ohne technisches Wissen, in der Lage sind mit FlexMash Data Mashups zu erstellen

FlexMash wird, wie die meisten Mashup Tools, online bereitgestellt und kann über einen Webbrowser verwendet werden. Da FlexMash in einer Cloud Computing Infrastruktur als Dienst angeboten wird, ermöglicht es einen leichten Zugriff, einfachen Einsatz und bessere Skalierbarkeit. Die Architektur lässt sich in vier Hauptbestandteile zerteilen.

Der **Mashup Plan Modeler** bietet dem Benutzer die Möglichkeit, festzulegen, wie Daten schrittweise verarbeitet werden sollen. Desweiteren können hier Patterns betrachtet und ausgewählt werden. Die patternbasierte **Model Transformation**-Komponente (PbMT) enthält den **Pattern-Implentation Selector**, der,wie zuvor beschrieben, automatisch eine geeignete Implementierung für die parametrisierten Patterns auswählt. Dieser Prozess wird mittels eines Patterngraphen und einem regelbasierten Ansatz realisiert. Aus Gründen der Übersichtlichkeit wird die Abbildung des Patterns auf eine Implementierung möglichst einfach gehalten. Folglich existieren für jedes Pattern genau eine Implementierung. Die PbMT enthält zudem die Logik für die Abbildung des Mashup Plans in ein ausführbares Format, sowie die Logik für die Ausführung des Modells auf der geeigneten Engine.

Die Komponente **Utils** stellt hierzu Methoden bereit, die diese Funktionalität unterstützen. Die Ausführungsengines, die die umgewandelten Modelle ausführen sollen, sind nicht Bestandteil von FlexMash, sondern cloud-basierte externe Dienste. Die letzte Komponente in der Architektur ist für die Visualisierung des Outputs aus der Ausführung zuständig.

3 Grundkonzept einer Fragment-Repository

In diesem Kapitel wird das Konzept für ein Fragment-Repository, welches für die einzelnen Komponenten eines Mashup Plans automatisch Code-Fragmente sucht und zur Verfügung stellt, beschrieben. Die gefundenen Code-Fragmente aus diesem Verzeichnis ersetzen anschließend die entsprechenden DSDs und DPDs im Mashup Plan.

Dabei wird in 3.1 die Funktion des Repository-Dienstes in abstrakter Form beschrieben. Darüber hinaus wird erläutert, welche Eingabewerte an das Verzeichnis übergeben werden und welche Ausgabe daraus resultiert.

In 3.2 die Architektur des Repository-Dienstes erklärt. Des Weiteren wird erläutert, wie die drei Ebenen der Architektur funktionieren und auf welche Weise diese miteinander agieren, um die Grundfunktionen des Repository-Dienstes zu bewerkstelligen. Zudem wird beschrieben, wie ein Benutzer des Repository-Dienstes Anfragen schicken kann und wie die Ergebnisse ausgegeben werden.

In 3.3 werden die unterschiedlichen Technologien, die bei diesem Konzept zum Einsatz kommen, näher betrachtet. Dabei wird erwähnt, welche Vorteile die Verwendung dieser Technologien mit sich bringen und welche wesentlichen Merkmale sie aufweisen. Das in dieser Arbeit beschriebene Konzept kombiniert verschiedene Datenbanken für die Realisierung des Repository-Dienstes.

Zu den Kernpunkten dieser Arbeit gehört das Auffinden einer geeigneten Implementierung für ein verwendetes Pattern. Diese Prozedur wird auch Mapping gennannt. Da für ein Pattern durchaus mehr als nur eine Implementierung vorhanden sein kann, muss dieses Mapping-Problem durch ein Konzept gelöst werden. Dazu wird das Regelbasierte Mapping verwendet. In 3.4 wird beschrieben, wie eine geeignete Implementierung regelbasiert durch Parametrisierung der Patterns gefunden wird. Anschließend wird ein regelbasierter Ansatz vorgestellt, auf dem Regelbasierte Mapping dieser Arbeit basiert.

Zuletzt wird in 3.4.2 an einem Beispiel die Aufteilung eines verschachtelten Patterns innerhalb der Patternhierarchie Schritt für Schritt beschrieben. Hierbei entspricht ein verschachteltes Pattern einem Knoten des Mashup Plans, welches selbst Knoten enthält, die selbst Pattern sind. Dieser muss stufenweise in seine einzelne Bestandteile zerlegt werden.

3.1 Funktion des Fragment-Repositories

In diesem Abschnitt wird zusammengefasst, welche Funktion der Repository-Dienst haben soll. Abbildung 3.1 stellt die Funktion des Repository-Dienstes grafisch dar. Wie bereits erwähnt wurde, ist Ziel des Repository-Dienstes konkrete Implementierungen für die Knoten eines Mashup Plans

automatisch zur Verfügung zu stellen. Diese sogenannte Mapping von Patterns soll dabei regelbasiert umgesetzt werden.

Der Repository-Dienst erhält als Eingabe einen Mashup Plan. Dieser wird, angefangen beim Startknoten bis zum Endknoten traversiert. Dabei wird für jeden Knoten, welches nicht ausführbar ist, im Repository-Dienst nach der geeigneten Implementierung gesucht. Da für jedes Pattern, d.h. jeden Knoten des Mashup Plans, mehr als eine Implementierung in der Datenbank des Verzeichnisses vorhanden sein kann (Mapping-Problem) wird ein regelbasiertes Mapping verwendet. Diese wird in 3.4 genauer beschrieben. Jeder Knoten des Mashup Plans übergibt als Eingabe ein JSON-Objekt an das Verzeichnis. Mithilfe der darin enthaltenen Informationen kann, die Lookup-Funktion des Verzeichnisses nach der geeigneten konkreten Implementierung suchen und diese zur Verfügung stellen. Sind alle nicht ausführbaren Knoten durch Code-Fragmente aus dem Verzeichnis ersetzt worden und der Endknoten erreicht, wird als Ausgabe ein ausführbarer Mashup Plan übergeben. Dies kann beispielsweise, abhängig von den Nutzeranforderungen des Benutzers, ein BPEL-Workflow sein.

Neben der Verwaltung der Code-Fragmente, bietet der Repository-Dienst unterschiedliche Funktionen an. So kann ein Benutzer neue Code-Fragmente in die Datenbank einpflegen, bestehende Code-Fragmente ändern und updaten, löschen oder nach bestimmten Implementierungen mithilfe verschiedener Kriterein suchen.

3.2 Architektur eines Fragment-Repositories

In diesem Abschnitt wird die grundlegende Architektur des in dieser Arbeit entworfenen Konzepts einer Repository für Code-Fragmente beschrieben. In Abbildung 3.2 ist diese Architektur des Entwurfs abgebildet. Es handelt sich dabei um ein System, das aus insgesamt drei Ebenen besteht. Zwischen den drei Ebenen findet ein Informations- und Datenfluss sowohl von der obersten Ebene bis runter in die unterste Ebene als auch umgekehrt von der untersten Ebene hinauf zur obersten Ebene statt. Im Folgenden werden die einzelnen Ebenen näher beschrieben.

- *Präsentations-Schicht:* Auf der obersten Ebene, welche die Präsentationsschicht bildet, wird der Datenaustausch initiiert. Sie stellt die Schnittstelle zwischen den Benutzern bzw. Anwendungen und dem Repository-Dienst dar, über die mit dem Repository-Dienst interagiert werden kann.

  Unterschieden wird hier momentan zwischen zwei verschiedenen Formen der Interaktion. Bei der ersten Interaktionsform, verwendet der Benutzer einen Web-Browser oder Client-Dienst um Anfragen an die Repository zu senden. Zurückgelieferte Resultate werden wiederum im Browser bzw. Client angezeigt. Da es sich bei dem Repository-Dienst um einen REST-Dienst handelt, erfolgen die Anfragen in Form von HTTP-Anfragen über das HTTP-Protokoll, welches die Methoden Put, Post, Get und Delete unterstützt.

  Im zweiten Szenario wird der Repository-Dienst programmatisch aufgerufen, d.h. der Programmcode bestimmter Repository-Funktionen wird durch Methodenaufrufe in die Anwendung eingebunden. Konkret handelt es sich hier um die an der Universität Stuttgart entwickelte Anwendung Flex-Mash, welche als Grundlage für diese Arbeit gedient hat. Flex-Mash ruft eine

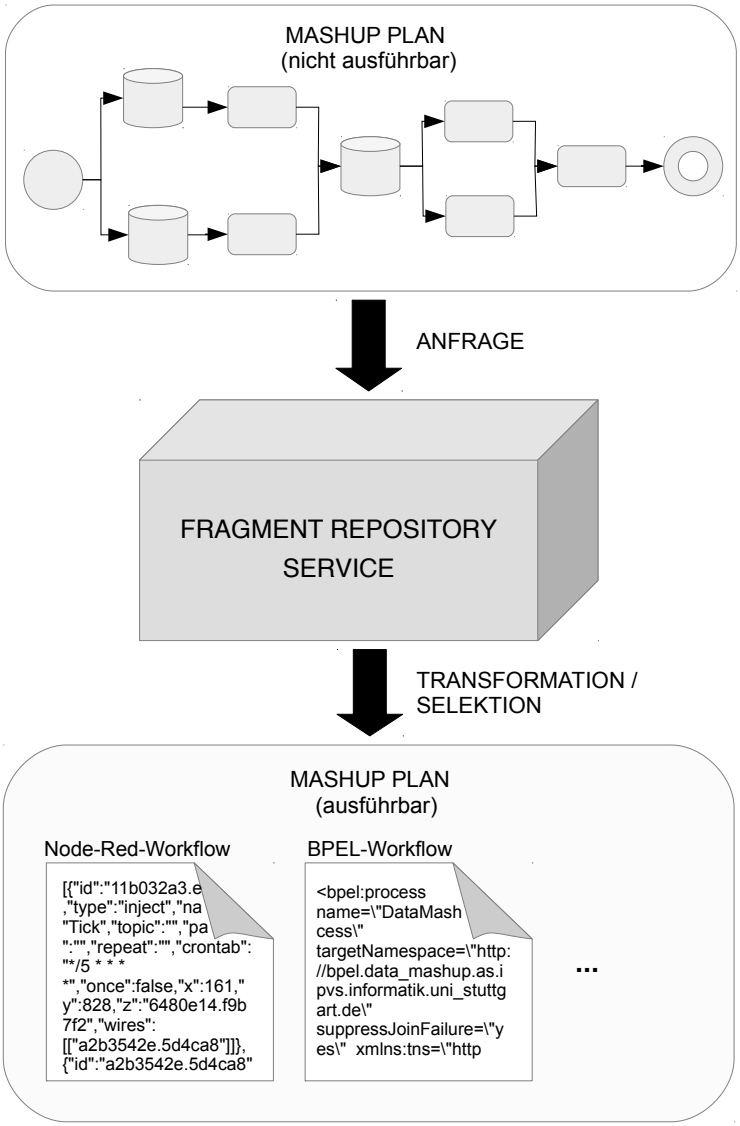


Abbildung 3.1: Fragment-Repository

Transform-Methode des Repository-Dienstes auf und übergibt ihm hierfür einen JSON-Flow als Eingabeparameter. Nach der Ausführung der Methode liefert der Repository-Dienst einen transformierten JSON-Flow als Ausgabe zurück, welcher dann in der Flex-Mash-Anwendung weiterverarbeitet wird.

- *Datenzugriffs-Ebene:* Die zweite Ebene der Architektur bildet die Datenzugriffs-Schicht. Sie stellt eine intermediäre Schicht zwischen dem Benutzer bzw. Anwendungen und der Datenschicht dar. Alle Anfragen der ersten Ebene werden hier entgegengenommen und an die Datenschicht weitergeleitet. Umgekehrt werden Daten von der Datenebene bezogen und an die Präsentationsschicht weitergeleitet.

  Die gesamte „business logic“ findet auf dieser Ebene statt. Ein Rest-Controller nimmt alle Rest-Anfragen und ihre Eingabeparameter an den Repository-Dienst entgegen und ruft interne Methoden auf, welche die Eingabedaten verarbeiten und auf der Datenebene entsprechende Datenmanipulations-Operationen bzw. Datenanfragen einleiten. Des Weiteren wird auf der Datenzugriffs-Ebene eine Pattern-Transformer-Funktion bereitgestellt, welche von der Fex-Mash-Anwendung aufgerufen wird. Diese übergibt der Transformer-Funktion einen Mash-Flow, welcher ein oder mehrere Patterns als Knoten enthält und somit unausführbar ist. Der Pattern-Transformer wandelt den Ausgangs-Workflow Schritt für Schritt, gegebenenfalls rekursiv, um, indem es regelbasiert Transformationsschritte ausführt, welche Patternknoten mit Code-Fragmenten und oder weiteren Pattern-Knoten ersetzt. Als Endresultat liefert der Pattern-Transformer einen ausführbaren JSON-Workflow zurück, der keine Pattern-Knoten mehr enthält.

  Diese zusätzliche Ebene verhindert den direkten Zugriff auf die Datenebene und die gesamte Logik, d.h. die Implementierungdetails der Funktionen des Repositiories, welche auf den relevanten Daten der Datenebene arbeiten, werden vor dem Benutzer verborgen.

- *Daten-Ebene:* Die dritte Ebene bildet die Daten-Ebene bzw. das Datenmodell. Hier werden alle Daten bereitgestellt, welche durch den Repository-Dienst verwaltet werden. Die Daten werden in drei verschiedenen Datenquellen vorgehalten. Der Großteil der Daten, der durch die Fragmente repräsentiert wird, ist in einer NoSQL-Datenbank abgelegt. Da NoSQL-Datenbanken für den Umgang mit großen Datenmengen ausgelegt sind (Big Data), gewährleistet der Einsatz einer solchen Datenbank hier gute Zugriffszeiten und eine hohe Verfügbarkeit. Die zu den Fragmenten gehörenden Metadaten werden in einer relationalen Datenbank abgelegt. In einer dritten Datenbank bzw. NoSQL-Tabelle werden die Web-Services gespeichert, welche von den Fragmenten aufgerufen werden.

Durch die Aufteilung in verschiedene Ebene wird der Grundsatz eines „separation of concerns“ realisiert. D.h. Aufgabenbereiche werden voneinander getrennt. Dadurch ist die Architektur übersichtlicher strukturiert und die Gesamtkomplexität des Systems wird niedrig gehalten. Dies erlaubt eine einfache Erweiterbarkeit des Systems (z.B. Erweiterung der Benutzerschnittstelle durch eine Benutzeroberfläche, Hinzufügen neuer Logik in Datenzugriffsebene etc.). Des Weiteren wird eine Trennung von fachlicher und technischer Ebene erzielt, indem Anfragen and die Repository als fachliche Abstraktionen von ihren konkreten, technischen Umsetzungen getrennt werden.

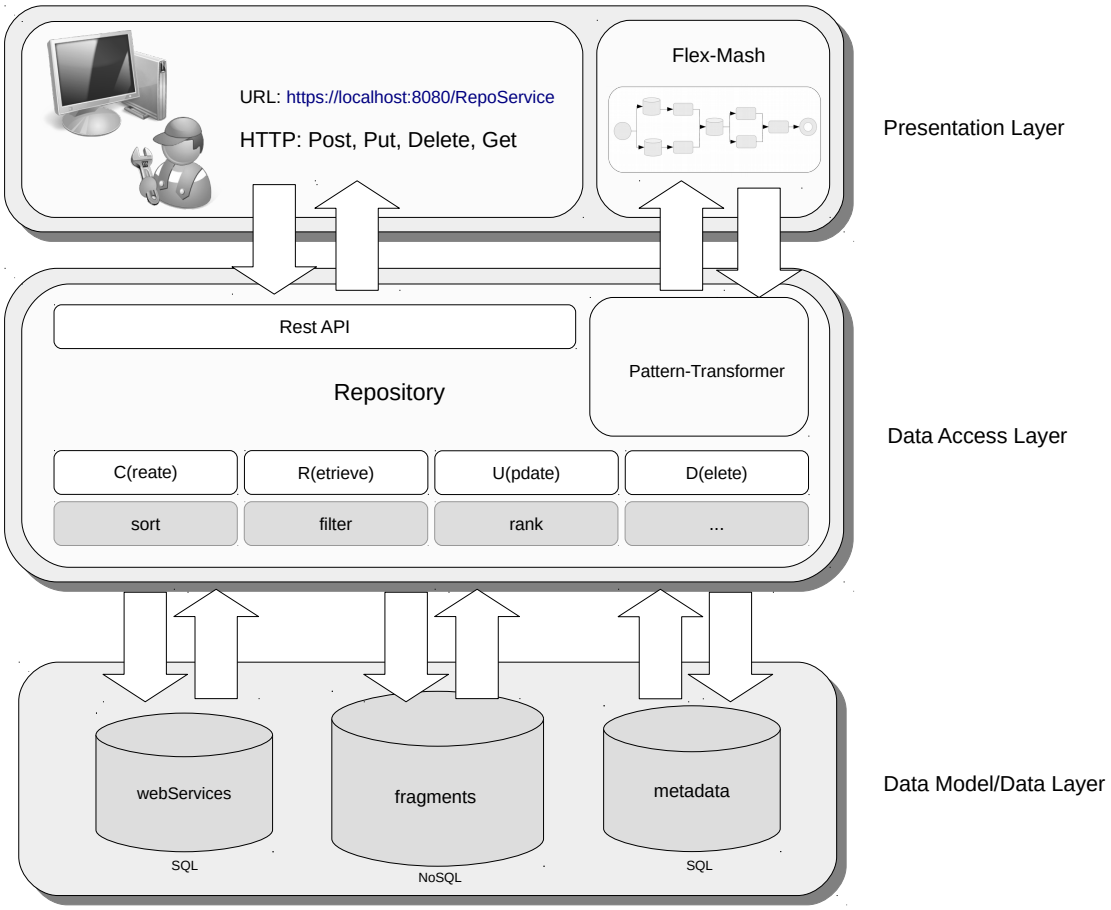


Abbildung 3.2: Architektur der Fragment-Repository

Die Interaktion mit der Repository erfolgt auf dem gegenwärtigen Implementierungsstand über die Verwendung von Rest-Anfragen in Form von HTTP-Befehlen an die Deployment-URL des Repository-Dienstes. Da momentan keine Benutzeroberfläche implementiert ist, muss ein REST-Client verwendet oder die Eingabe über die Adresszeile eines Web-Browser erfolgen. Die erfordert jedoch technisches Know-How und Domänenwissen, insbesondere bei der Formulierung der Eingabeparameter und beim Auslesen der Ausgabeparameter im JSON-Format.

3.3 Verwendete Technologien

In diesem Abschnitt werden die zur Realisierung des Konzepts für das Code-Fragmente-Verzeichnis verwendete Technologien beschrieben. Dazu gehören die NoSQL-Datenbank MongoDB und das relationale Datenbanksystem MySQL. Des Weiteren wird das verwendete Open-Source Framework

Spring beschrieben und ihre Vorteile genannt. Dazu wird das die Architektur von Spring mit den wichtigsten Komponenten aufgelistet. Ferner werden die wesentliche Merkmale dieses Open-Source Rahmenwerks vorgestellt und anschließend die Vorteile genannt, die eine Verwendung von Spring mit sich bringt. In 3.3.2 wird die für dieses Konzept verwendete NoSQL-Datenbank MongoDB beschrieben. Dabei wird erläutert, in welcher Form Daten in MongoDB gespeichert werden und welches die charakteristischen Merkmale dieser Datenbank sind. Ferner werden neben den Vorteilen auch die Unterschiede zu SQL-Datenbanken erwähnt. MongoDB wird in dieser Arbeit für die Speicherung der Code-Fragmente und der damit verbundenen Web Services verwendet.

3.3.1 Spring Framework

Für die Implementierung dieser Abschlussarbeit wurde unter Anderem das Spring Rahmenwerk verwendet. In diesem Abschnitt wird daher das Spring Framework genauer beschrieben.

Spring ist ein java-basiertes Open-Source Rahmenwerk und dient dazu die Entwicklung von Java-EE-Appikationen zu vereinfachen. Hauptbestandteile sind die **Dependency Injection** (DI) und das aspektorientierte Programmieren (aspect-oriented programming, AOP). Das Spring Rahmenwerk zeichnet sich dadurch aus, dass einfache Java-Objekte, sogenannte **Plain Old Java Objects** (POJO), als Java-Beans verwaltet werden. Neben Spring MVC gehören der **Inversion of Control**-Container und das **Aspect oriented Programming** zu den wesentlichen Hauptfunktionen dieses Rahmenwerks.

Das Spring Framework erfreut sich einer wachsender Beliebtheit in der Java Community, da sie eine Alternative, ein Ersatz, als auch als Erweiterung des *JavaBeans*-Modells betrachet werden kann. JavaEE ist weitverbreitet, hat allerdings Einschränkungen in Bezug auf Reusability von Code. Verwendet man das Spring Framework gemeinsam mit JavaEE erleichtert es die Entwicklung von Applikationen. Die wesentlichen Funktionen des Spring Frameworks sind [Wal12]:

- Verknüpfungen durch Dependency Injection über den Inversion-of-Control Container
- auf Plain Old Java Objects basierendes Programmiermodell
- Querschnittsthemen werden durch Aspect orientend Programming (AOP) bereitgestellt

Spring gründet seinen Angriff auf die Komplexität von Java auf vier Kernstrategien:

- Leichtgewichtige und minimal invasive Entwicklung mit *POJOs* (Plain Old Java Objects)
- Lockere Kopplung durch Injizieren von Abhängigkeiten und Interface-Orientierung
- Deklarative Programmierung durch Aspekte und übliche Konventionen
- Reduzierung von Boilerplate-Code durch Aspekte und Vorlagen

Spring besitzt eine mehrschichtige Architektur. Wird eine E-Commerce-Applikation entwickelt, erfolgt eine Trennung der Schichten. Der Benutzer kann selbst entscheiden, welchen Komponenten er verwenden möchte. Spring übernimmt Infrastruktur und Benutzer kann sich auf die Anwendung konzentrieren .Demnach ergeben sich mit der Benutzung von Spring folgende Vorteile. Ein Anwendungs-entwickler kann eine Java-Method bei einer Datenbanktransaktion ausführen, ohne sich dabei mit der Transaktions-API beschäftigen zu müssen. Desweiteren kann beispielsweise eine

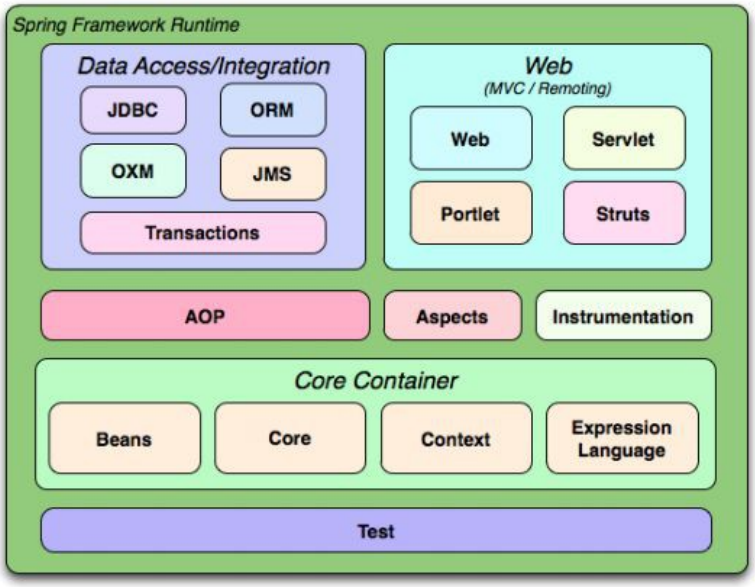


Abbildung 3.3: Die Architektur des Spring Framworks [JHD+04]

lokale Java-Method eine Remote Procedure ausführen, wobei die Auseinandersetzung des Benutzers mit einer Remote API entfällt [BSKM12].

Architektur von Spring

Die Architektur von Spring lässt sich in 7 Kernbestandteile zusammenfassen:

- Core Container
- Spring Context
- Spring DAO
- Spring ORM
- Spring Web Module
- Spring MVC framework

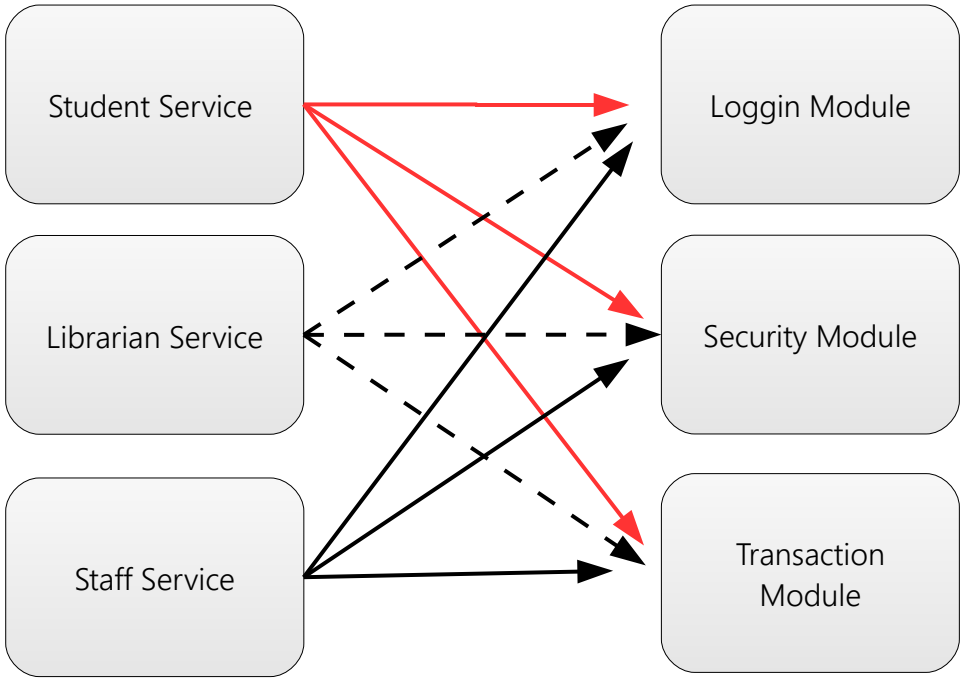


Abbildung 3.4: Normales System ohne AOP

Aspect oriented Programming, AOP, ermöglicht die Trennung verschiedener Themengebiete in einem System. Aspekte können mithilfe von Spring XML-Files miteinander verknüpft werden. Anhand des folgenden Beispieles wird dies dies veranschaulicht. In einem Bücherei-System benötigen diverse Diensttypen, wie z.B. ein Studentenservice etc., Funktionalitäten, die von Modulen für das Logging, Security und Transaktionen bereitgestellt werden. Das ursprüngliche Modell würde wie folgt aussehen:

Dasselbe Szenario würde mithilfe der AOP-Funktionalität von Spring folgendes Resultat ergeben. Jede der drei Funktionen werden allen Diensten zur Verfügung gestellt.

Inversion of Control ruft nicht die Applikation das Framework auf, sondern das Rahmenwerk die Komponenten, die von der Applikation bestimmt werden. Die Abhängigkeiten werden dynamisch zur Laufzeit injiziert.

Spring MVC Modell

Das Spring Rahmenwerk stellt sein eigenes MVC Modell zur Verfügung. Die Hauptkomponenten sind :

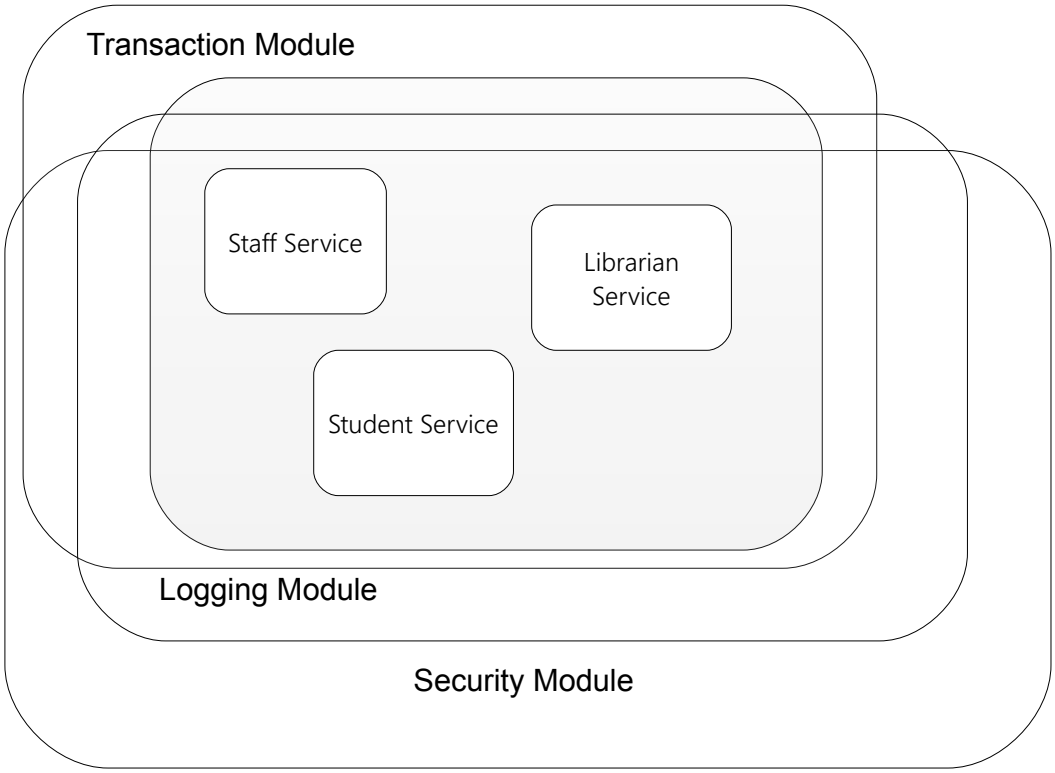


Abbildung 3.5: Ansatz mit AOP

- **DispatcherServlet**: empfängt den Request, welches durch ein web.xml file an ihn übergeben wird.
- **Controller**: Es bearbeitet die Anfrage und wird vom Benutzer erstellt. Controller sind Objekte, die auf Benutzeraktionen reagieren.
- **View**: visualisiert dem Endnutzern das Endresultat
- **ModelAndView**: assoziiert Anfrage mit der View; wird von Controller erstellt und gibt bei Ausführung Daten und Namen der View an
- **ViewResolver**: löst View auf, basierend auf Ausgabe des ModelandView; wählt das Ausgabemedium
- **HandlerMapping**: Mithilfe dieses Komponents assoziiert DispatcherServlet ankommende Anfragen mit individuellen Controllern.

Spring und XML

XML (Extensible Markup Language) wird bei eine hohen Anzahl von Frameworks für die Behandlung von Konfigurationsinformationen verwendet. Die in einer XML-Datei abgespeicherten Informationen können modifiziert und die Änderungen können über die Applikation sichtbar gemacht werden. XML-Dateien vereinfachen den Entwicklungsprozess und damit verkürzt sich auch die Entwicklungszeit. Es gibt drei Typen von XML-Dateien:

- web.xml file
- applicationContext.xml file
- DispatcherServlet.xml file

Vorteile von Spring

Der Einsatz von Spring bietet vielzählige Vorteile. So kann das Rahmenwerk effektiv genutzt werden in Kombination mit anderen Frameworks,wie *Hibernate* und *Struts*. Des Weiteren bietet Spring vereinfachten Zugriff auf die Datenbank, durch die Verwendung vom Hibernate Framework und die Vermeidung der Behandlung von Fehlermechanismen. Anwendungen, die mithilfe von Spring erstellt wurden, sind lediglich von wenigen APIs abhängig. Aufgrund seiner mehrschichtigen Architektur, kann der Benutzer selbst entscheiden, welche Komponenten verwendet werden sollen. Ferner verkürzt die Inversion of Control-Eigenschaft von Spring die Zeit für das Testen des Codes und das Spring Web MVC Rahmenwerk ist robust, flexibel und eignet sich gut für sich schnell entwickelnde Webapplikationen.

3.3.2 MongoDB

Die Codefragmente werden in einer NoSQL-Datenbank abgespeichert. Diese enthält zwei Tabellen (Collections). In einer Collection werden Codefragmente abgelegt. Dies können beispielsweise BPEL-Codefragmente sein. Diese wiederum rufen mit invoke-Knoten Web Services auf, welche die angeforderte Aufgabe ausführen. Diese Web Services werden in einer weiteren Collection abgespeichert. Alle Datensätze in den unterschiedlichen Datenbanksysteme werden über eine eindeutige ID-Nummer miteinander verbunden. Somit kann klar zugeordnet werden, welches Codefragment zu welchen Metadaten gehört. Es ist zu beachten, dass ein Knoten des Mashup Plans durch mehrere Implementierungen realisiert werden kann. Dies hat zur Folge, dass es für ein Knoten in der entsprechenden Datenbank mehrere Codefragmente vorliegen können, welche dieselbe Aufgabenstellung lösen, jedoch technisch unterschiedlich realisiert sind.

Mongo DB ist eine leistungsstarke und effiziente Datenbank, welche zur Klasse der dokumentorientierten NoSQL-Datenbanksystemen gehört . Die Firma 10g entwickelte Mongo DB mithilfe der Programmiersprache C++ und veröffentlichte diese im Jahr 2009. Mongo ist die Abkürzung für „Humongous“, welches enorm bedeutet und lässt darauf schließen, was eine der wichtigsten Eigenschaften der MongoDB Datenbank ist: die performante Verarbeitung von großen Datenmengen. Der Aufbau

und die Datenstruktur eignen sich insbesondere für die Verwendung in Webapplikationen und dem Internet [Mon12].

MongoDB bietet Features wie *consistency fault tolerance*, *persistance*, *aggregation*, *ad hoc queries*, *indexing*und *auto sharding*. Daten werden in MongoDB in Dokumenten abgespeichert. Diese setzen sich aus einem sortierten Satz von Eigenschaften zusammen, die aus einem Namen und einem Wert bestehen. Werte können einfache Felder, Datentypen oder andere Dokumente sein. Dokumente sind schemalos, d.h. sie besitzen keinen festen Aufbau, sondern werden dynamisch mit Daten befüllt, die benötigt werden. MongoDB eignet sich insbesondere für Anwendungen wie Content Management Systems, Archivierung, Real Time Analytics etc.

Bei der Erstellung von MongoDB standen vier Kernziele im Mittelpunkt:

- Mächtigkeit
- Flexibilität
- Geschwindigkeit/Skalierbarkeit
- einfache Benutzbarkeit

Mächtigkeit wird erreicht durch Bereitstellung einer mächtigen Abfragesprache, Index-Strukturen, und vielen weiteren Funktionen. Hierbei wird jedoch nicht auf Funktionalität verzichtet, die Benutzer von SQL gewohnt sind. Die Flexibilität ergibt sich aus einem schemalosen Datenmodell, welches mit der Datenbankanwendung wächst und sich verändern kann. Ferner ist der Zugang zum Datenbanksystem für Programmierer möglichst einfach gehalten. Eine große Geschwindigkeit und Skalierbarkeit wird durch die Verwendung von Key-Value-Stores erlangt. Zudem bietet MongoDB schnelle Antwortzeiten und eine weitgehend einfache Erweiterbarkeit der Speicherkapazität des Systems im Online-Betrieb. Zusätzliche Performance-Steigerung wird dadurch erreicht, dass die Kommunikation zwischen Server und Client über einem leichtgewichtigem TCP/IP-Protokoll läuft und somit weniger Overhead erzeugt als HTTP/REST. MongoDB verzichtet auf einige Features von relationalen Datenbanken, wie z.B. Join-Operationen und Transaktionskontrolle und erzielt damit höhere Geschwindigkeiten.

Dokumente in MongoDB

Dokumente in MongoDB werden im BSON-Format, binäres JSON abgespeichert, welches sich durch seine Effizienz und Platzersparnis auszeichnet. Ferner kann BSON aufgrund seiner auf C basierenden Repräsentation von Typen performant codiert und decodiert werden. Dokumente werden in sogenannten *Collections* zusammengefasst, wobei jede Collection eine benannte Menge von Dokumenten ist. Jeder Collection ist in einer Datenbank abgespeichert und jedes MongoDB-System enthält eine Menge von Datenbanken. Analog zu relationalen Datenbanken stellen Dokumente das Gegenstück zu Spalten dar, während Collections den Tabellen einer relationalen Datenbank entsprechen. Collections können Dokumente mit unterschiedlichem Aufbau beinhalten. Des Weiteren bekommt jedes Dokument automatisch einen Primärschlüssel zugeordnet, welches den eigenen Bedürfnissen angepasst werden kann. [NPP13].

Vorteile von MongoDB

MongoDB wird mittlerweile bevorzugt gegenüber relationalen Datenbanken bei Projekten, die große Datenmengen verarbeiten müssen. Einige wichtige Eigenschaften sind [HHLD11]:

- MongoDB unterstützt die Verwendung von komplexen Datentypen. Das Datenformat BSON erlaubt die Speicherung von komplexen Datentypen
- leistungsstarke Abfrage-Sprache erlaubt nahezu alle Funktionen wie beispielsweise Abfragen in Single-Tables relationaler Datenbanken und unterstützt Indexing
- High-Speed-Zugriff auf große Datenmengen: ab einem Datenvolumen von 50 GB ist die Zugriffsgeschwindigkeit von MongoDB zehn mal schneller als MySQL.

Unterschiede MongoDB und SQL

Es gibt viele Diskussionen darüber, ob NoSQL-Datenbanken anstelle von relationalen Datenbanken eingesetzt werden sollten. Ist die Datenbank nicht-strukturiert und sehr groß, empfiehlt es sich eine NoSQL-Datenbank zu verwenden. Diese Frage lässt sich in Bezug auf eine durchschnittlich große Datenbank mit strukturierten Daten nicht so einfach beantworten. Daten in einem relationalen Datenmodell werden in einem Datenbank-Schema dargestellt, wobei die Daten in den Zeilen und Spalten einer Tabelle abgelegt werden. Es ist zu beachten, dass jede Zeile dieselbe Anzahl an Spalten mit dem demselben Typ besitzt.

Tabellen in relationalen Datenbanken liegen normalisiert vor, was dazu führt, dass mehrere Tabellen erzeugt werden. Eine Anfrage auf diese Tabellen erfordert das Abrufen und Kombinieren von Informationen aus diesen unterschiedlichen Tabellen. Dazu werden Join-Operationen verwendet. Je größer das Schema und die Anzahl der Tabellen ist, umso länger Zeit nimmt die Abfrage auf die relationale Datenbank in Anspruch, um bestimmte Informationen zu beziehen. NoSQL vereinfacht die Bearbeitung von nicht-strukturierten Daten. Daten können semi-strukturiert vorliegen, so dass ähnliche Datenobjekte mit unterschiedlichen Eigenschaften gruppiert werden koennen. Unstrukturierte Daten können unterschiedlichen Typs sein und kein Format besitzen. Diese Daten können durch kein Schema-Typ dargestellt werden.

Die typischen Eigenschaften von SQL-Datenbanken, wie z.B. die ACID-Eigenschaften, erzeugen einen Overhead, welcher in NoSQL-Datenbanken teilweise vollständig eliminiert wird, um die Leistung zu vergrößern. Der Großteil an NoSQL-Datenbanken legen ihre Daten in Key-Value-Paaren ab. Dabei kann der Value ein Wort, eine Zahl oder eine komplexere Struktur sein. Die Erzeugung von Abfragen (Queries) ist jedoch relativ schwierig, da es keine Standard-Abfrage-Sprache gibt und die Operationen begrenzt eingesetzt werden können. Ferner gibt es keine Join-Operation. Im Wesentlichen ist die Verarbeitung bzw. Bearbeitung von Daten einfacher, erschwinglicher und flexibler.

MongoDB erlaubt mit *Autosharding Datenbankserver* automatisch auf verschiedene physikalische Maschinen aufzuteilen. Diese horizontale Skalierung verteilt die Arbeits- und Datenlast. Ferner werden Daten auf mehrere Server verteilt, wobei mithilfe von Replikation die Verfügbarkeit des Services erhöht wird. Die horizontale Skalierung ist bei SQL-Datenbanken nicht möglich, da Operationen wie Join sehr viel Zeit beanspruchen würden.

3.3.3 MySQL

Für die Speicherung der Metadaten, welche alle notwendigen Informationen über die Codefragmente enthalten, wird eine SQL Datenbank verwendet. Die Metadaten beinhalten eine einheitliche ID-Nummer und weitere Informationen, für die Beschreibung der Funktionalität und Eigenschaften der Codefragmente. Die Eigenschaften einer relationalen Datenbank wurden in 2.1.2 genauer beschrieben.

3.4 Regelbasiertes Mapping

Das Regelbasierte Mapping von Patterns basiert auf dem Ansatz von [RSM14]. Der Repository-Dienst erhält als Eingabe einen Mashup Plan. Dieser wird von Starknoten bis Endknoten traversiert. Dabei wird für jeden Knoten ein geeignetes Code-Fragment aus dem Verzeichnis gesucht und übergeben. Jeder Knoten entspricht einem Pattern. Für dieses Pattern werden vom Benutzer Parameter mitgegeben. Diese Parameter werden in Form eines JSON-Objekts an das Verzeichnis weitergeleitet. Dieser enthält folgende Komponenten in Form von Key-Value-Paaren:

- Namen des Knotens : Database
- Typ : MySQL/Pattern
- Pattern-Typ : robust
- Operations-Typ : extract
- Property-Feld : Eingabe-, Ausgabeparameter, Filter Criteria....

Der Name beschreibt die Funktion des Knotens, der Typ gibt an ob es sich hierbei um ein Pattern handelt,der weitere Knoten enhalten kann, die wiederum auch Patterns sein können, oder ein Knoten ist, der keine weiteren Patterns enthält.

Des Weiteren gibt der Operations-Typ die Funktion des Knoten an und das Property-Feld ist ein JSON-Objekt, welches alle Eingabe- und Ausgabeparameter und alle weiteren Parameter des Knotens enthält. Sollte der Knoten ein Pattern sein, der weitere Knoten enthält, werden diese Knoten mit ihren Parametern im Property-Feld gespeichert. Der Pattern-Typ gibt an, welches Transformations Pattern vom Benutzer ausgesucht wurde.

Anhand dieser Informationen kann das Repository-Dienst nun nach den geeigneten Implementierungen suchen und bei erfolgreicher Suche diese übergeben. Dabei werden die mitgegeben Informationen mit den Metadaten der Code-Fragmente im Verzeichnis verglichen.

Handelt es sich bei dem Knoten um ein Pattern, d.h. typ hat den Wert *Pattern*, muss dieser in der Patternhierarchie so lange in seine Bestandteile zerlegt werden, bis in der untersten Ebene der Hierarchie keine Patterns mehr vorzufinden sind.

Dabei ist zu beachten, dass alle Parameter vollständig in der Anzahl und fehlerfrei übergeben werden müssen, damit das Verzeichnis das passende Code-Fragment finden kann. Ist dies nicht der Fall und es wurden zu wenige Parameter übergeben oder Parameter in einem falschen Format eingegeben,

ist eine erfolgreiche Suche von Implementierungen nicht möglich und es wird eine Fehlermeldung ausgegeben. Werden beispielsweise zwei Datenbanken-Namen als Parameter erwartet und es wird lediglich eine angegeben, hat dies zur Folge, dass die Suche nicht durchgeführt werden kann. Wurden alle erforderlichen Parameter hingegen angegeben und ein Pattern in der Patterhierarchie bis zur letzten Blattebene gelangt ist und somit auf die kleinsten Bestandteile aufgeteilt wurde, wird der Pattern-Typ-Eintrag überprüft. Dieser ist ausschlaggebend, dafür welche mögliche Implementierung ausgewählt wird. Besitzt der Eintrag Pattern im JSON-Objekt den Wert Robust, wird dementsprechend das Code-Fragment selektiert, welches in seinem JSON-Object für den Schüssel Pattern denselben Wert Robust hat. Dieses Code-Fragment enthält für den Schlüssel Fragment einen Wert in Form eines BPEL-Strings.

Zusammengefasst ist das Auffinden des geeigneten Code-Fragments abhängig von den eingegeben Parametern und dem Wert des Eintrags Pattern im übergegeben JSON-Object eines Knotens. Somit können die Anforderungen des Benutzers bei der Suche nach konkreten Implementierungen berücksichtigt werden.

3.4.1 Regelbasierte Transformation

In diesem Abschnitt wird die regelbasierte Transformation näher beschrieben. Dabei wird Bezug auf den Ansatz von [RSM14] genommen. In Abbildung 3.6 wird das Verarbeitungsmodell der Patterntransformation grafisch dargestellt.

Der regelbasierte Pattern Transformer wandelt nicht ausführbare Workflows, welche mehrere Pattern enthalten, in ausführbare Workflows um. Dies wird anhand einer erweiterbaren Menge von Transformationsregeln und der Pattern Transformation Engine durchgeführt. Wie aus der Abbildung zu sehen ist, erhält der Pattern Transformer einen nicht ausführbaren Workflow. Dieser wird Knoten für Knoten durchlaufen und bei Auffinden von einem Pattern nach einer geeigneten Transformationsregel gesucht. Eine Transformationsregel setzt sich aus einem Bedingungs-, einem Fragment- und einem Aktionsteil zusammen. Eine Transformationsregel tritt lediglich dann in Kraft, wenn die Bedingungen im Bedingungsteil erfüllt werden. Werden alle Anforderungen erfüllt, bestimmt die Fragmente-Komponente aus einer Workflow Fragment Library ein Template für ein Workflow-Fragment, welches verwendet werden soll.

Anschließend fügt die Aktions-Komponente dem zu verwendenen Fragment alle notwendigen Implementierungsdetails hinzu. Ferner werden hierbei Parameter mit höherem Abstraktionsgrad auf Parameter niedrigeren Abstraktionsgrads abgebildet. Das Pattern wird somit zu einem Workflow-Fragment überführt, Dieser kann eventuell weitere Patterns enthalten und ist folglich weiterhin nicht ausführbar. Dementsprechend wird der Transformationsvorgang solange rekursiv durchgeführt, bis keine Pattern im Eingabe-Workflow vorhanden sind und der daraus resultierende und aus Workflow-Fragmenten zusammengesetzte neue Workflow ausführbar ist. Patterns können einem Abstraktionslevel zugeordnet werden. So lässt sich eine Patternhierarchie erzeugen, welches einer besseren Strukturierung und Klassifizierung dient und aus mehreren Ebenen aufgebaut ist.

Für jede Ebene dieser Patternhierarchie existiert eine Rule Sequence. Diese ordnet jedem Pattern einer Ebene Transformationsregeln zu und bestimmt die Reihenfolge, in welcher Transformationsregeln

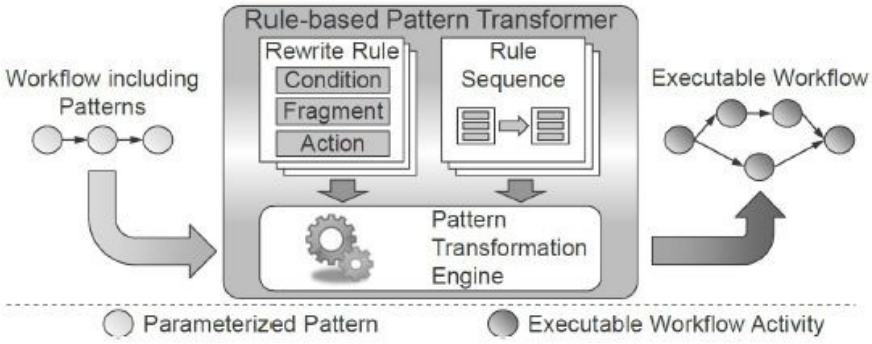


Abbildung 3.6: Pattern Transformer [LR00]

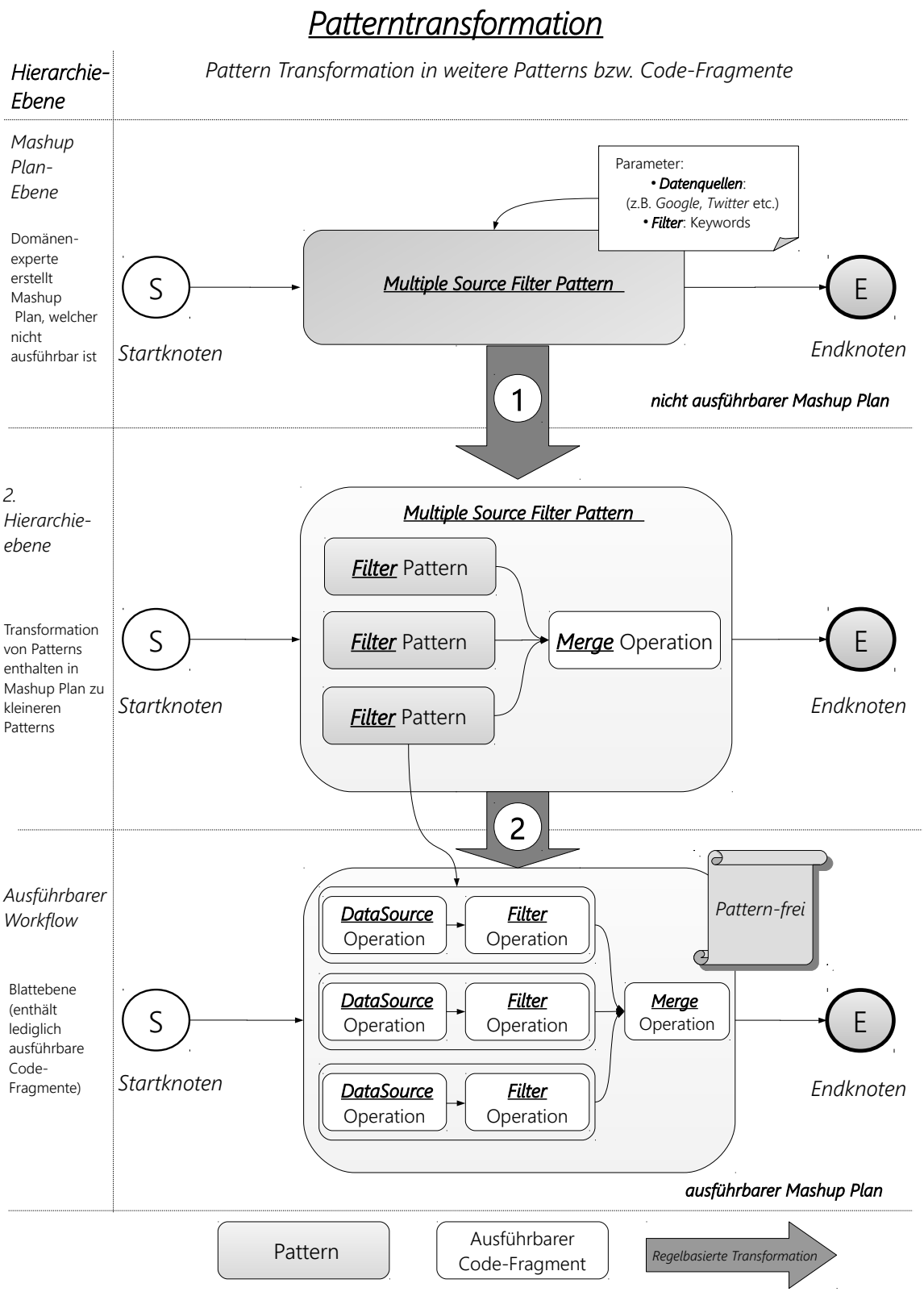
ausgeführt werden. Die erste Regel der Rule Sequence, deren Bedingungen erfüllt sind, wird ausgeführt, wobei die restlichen Regeln im Gegensatz dazu nicht eingesetzt werden.

3.4.2 Patternhierarchie im Beispiel

In diesem Abschnitt wird anhand eines Beispieles ein Mashup Plan, welches nicht ausführbar ist, in ein ausführbares Format gebracht und dabei die Patternhierarchie näher erläutert.

In der Abbildung ist ein Mashup Plan gegeben. Dieser besteht aus einem Startknoten, einem Endknoten und einem weiteren Knoten, mit dem Namen *Multiple Source Filter Pattern*. Es ist zu beachten, dass ein Mashup Plan weitaus komplexer aufgebaut sein kann, im Beispiel jedoch möglichst simple strukturiert ist, damit es für den Leser besser verständlich ist.

Der vorliegende Mashup Plan ist nicht ausführbar, da das Pattern Multiple Source Filter Pattern selbst nicht ausführbar ist. Der Domänenexperte, welcher den Mashup Plan erstellt, muss alle benötigten Parameter mitgeben. Diese sind für die spätere Ausführung des Mashup Plans erforderlich. So erfordert das Multiple Source Filter Pattern die Angabe von mehreren Data Source-Parametern, wie beispielsweise Twitter, Google etc. Des Weiteren müssen für eine Filterung der definierten Datenquellen Suchbegriffe mitgegeben werden. Die Transformation von Patterns basiert auf Regeln.


Patterntransformation
Hierarchie-Ebene
Pattern Transformation in weitere Patterns bzw. Code-Fragmente
Mashup Plan-Ebene
Domänen-experte erstellt Mashup Plan, welcher nicht ausführbar ist
Parameter:
• Datenquellen:
(z.B. Google, Twitter etc.)
• Filter: Keywords
S
Multiple Source Filter Pattern
E
Startknoten
Endknoten
1
nicht ausführbarer Mashup Plan
2. Hierarchie-ebene
Transformation von Patterns enthalten in Mashup Plan zu kleineren Patterns
Multiple Source Filter Pattern
Filter Pattern
Filter Pattern
Filter Pattern
Merge Operation
2
Ausführbarer Workflow
Blattebene (enthält lediglich ausführbare Code-Fragmente)
DataSource Operation
Filter Operation
DataSource Operation
Filter Operation
DataSource Operation
Filter Operation
Merge Operation
Pattern-frei
ausführbarer Mashup Plan
Pattern
Ausführbarer Code-Fragment
Regelbasierte Transformation

Diese Regeln müssen erfüllt werden, damit ein Pattern in weitere Komponenten zerlegt werden kann. Werden Parameter jedoch fehlerhaft eingegeben oder wird die Anzahl der erforderlichen Parameter nicht eingehalten, kann die regelbasierte Transformation nicht ausgeführt werden. Im Beispielfall ist es erforderlich, dass der Benutzer mehrere Datenquellen angeben muss, aus denen bestimmte Daten extrahiert werden sollen. Dazu werden Suchbegriffe, sogenannte Keywords, angegeben, mit deren Hilfe die erwünschten Daten aus den zuvor bestimmten Datenquellen gefiltert werden können. Sind alle Parameter angegeben und somit die Konditions-Klausel der regelbasierten Transformation erfüllt, kann das Pattern, welches nicht ausführbar ist in seine Bestandteile aufgeteilt werden.

Das hier vorhandene Multiple Source Filter Pattern setzt sich aus kleineren Komponenten zusammen, welche alle zur Gesamtfunktion des Pattern beitragen. Alle dieser Komponenten müssen ausführbar sein, damit das Pattern selbst auch ausgeführt werden kann. Diese gilt auch für den Mashup Plan, der aus Komponenten zusammengesetzt ist, welche ausführbar sowie unausführbar sein können. Solange nicht alle Bestandteile in ein ausführbares Format überführt werden, ist eine Ausführung des Mashup Plans nicht möglich. Das Multiple Source Filter Pattern besteht aus mehreren Filter Patterns und einer Merge Operation. Die Merge Operation ist ausführbar und ruft einen beispielsweise einen Web Service auf, der diese Funktionalität anbietet und durchführt. Die Filter Patterns sind dagegen nicht ausführbar und müssen weiter in ihre wesentlichen Bestandteile zerlegt werden. Das Filter Pattern selbst setzt sich aus den Komponenten Data Source Operation und Filter Operation zusammen. Im Schritt zwei aus der Abbildung 3.7 werden diese Patterns unterteilt in jeweils eine Filter und ein Data Source Operation. Mit diesem Transformationschritt ist auch die unterste Hierarchieebene, die Blattebene in der Patternhierarchie erreicht. Auf dieser Ebene ist der Mashup Plan vollständig transformiert und in ein ausführbares Format überführt worden. Dementsprechend setzt sich der Mashup Plan aus Bestandteilen zusammen, die keine Patterns sind und somit alle Knoten im Mashup Plan, beginnend beim Startknoten S bis zum Endknoten E, einzeln ausführbar sind. Dieser nach dem Ansatz des regelbasierten Mapping von Patterns umgewandelte Mashup Plan kann anschließend von einer beliebigen Engine ausgeführt werden, welche zuvor vom Benutzer durch die Auswahl des Transformationspattern bestimmt wurde.

Ein Mashup Plan kann aus zahlreichen Patterns bestehen, welche in ein ausführbares Format gebracht werden müssen. Jedes einzelne Pattern muss innerhalb der Patterhierarchie in seine kleinsten Bestandteile aufgelöst werden, für welche wiederum aus dem Code-Fragmente-Verzeichnis die entsprechenden Implementierungen gesucht und eingesetzt werden. Dabei können Patterns selbst Patterns enthalten, d.h. Patterns können verschachtelt vorkommen. Umso größer der Verschachtelungsgrad in einem Pattern ist, desto länger ist der Pfad im Pattern Graphen von der Mashup-Ebene bis zur untersten Ebene, der Blatt-Ebene, um dieses Pattern in seine kleinsten ausführbaren Komponenten zu überführen.

4 Patternbeispiele

In diesem Abschnitt werden nun einige Patternbeispiele vorgestellt. Diese können als Grundbausteine in Mashup Plans verwendet werden oder auch als Baustein eines anderen Patterns fungieren. Im Folgenden werden das *Source-to-Source*Pattern, *Filter*Pattern, *Split*Pattern, *Merge*Pattern näher beschrieben. Des Weiteren wird das *Data Iteration*Pattern in paralleler als auch in sequentieller Form erläutert.

4.1 Source-to-Source Pattern

Das **Source-to-Source Pattern**(S2SP) ist ein relativ simples Pattern, welches in Abbildung 4.1 grafisch dargestellt wird. Bei diesem Pattern werden aus einer Datenquelle (Datenquelle 1) Datensätze mithilfe eines Filters extrahiert und anschließend durch eine Data Operation transformiert. Der transformierte Datensatz wird anschließend in einer anderen Datenbank abgespeichert. Der Benutzer muss hierbei Parameter für die Filter Operation und die Data Operation eingeben, sowie die Datenquelle nennen, aus der die Datensätze extrahiert werden sollen. Werden diese Parameter nicht eingegeben, kann das Pattern nicht ordnungsgemäß ausgeführt werden. Dieses Pattern gehört zu den Basispattern, welche sehr häufig in größeren Pattern zum Einsatz kommen. Ferner kann eine Data Operation unterschiedlichen ETL-Operationen entsprechen. Diese sind in diesem Pattern unäre Operationen, welche nur eine Eingabemenge erwarten.

4.2 Filter Pattern

Ein weiteres simples Basispattern ist das **Filter Pattern** (DFP). Wie in Abbildung 4.2 zu sehen ist, setzt sich dieses Pattern aus einer Datenquelle (Datenquelle1) und einer Filter Operation zusammen. Die Funktion dieses Patterns besteht darin, Datensätze aus einer gegebenen Datenbank mithilfe einer Filter Operation zu extrahieren. Der Benutzer dieses Patterns muss lediglich die Datenquelle bestimmen, aus der Daten entnommen werden sollen und nach welchen Kriterien diese gefiltert werden sollen.

4.3 Data Split Pattern

Das **Data Split Pattern** (DSP) dient dazu, eine Datenmenge aus einer Datenquelle (Datenquelle1) zu extrahieren und diese in n kleinere Teilmengen aufzuteilen. Abbildung 4.3 stellt diese Pattern grafisch dar. Der Benutzer gibt hier als Parameter eine Datenquelle und einen Filterkriterium an. Hierbei kann

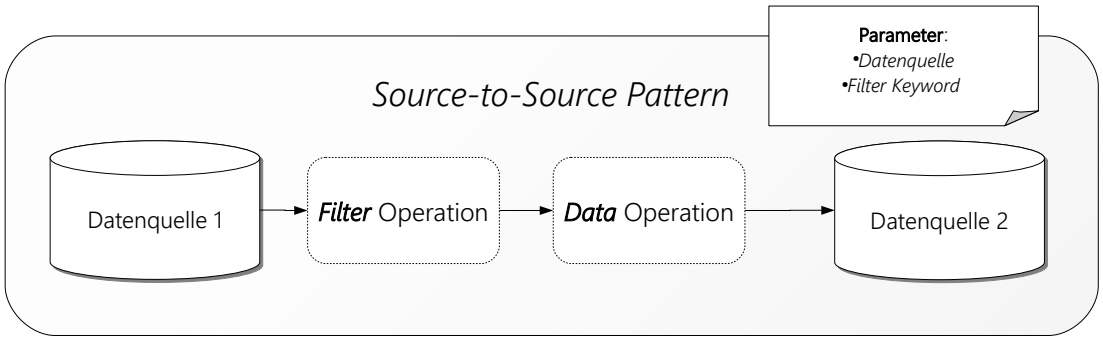


Abbildung 4.1: Source-to-Source Pattern

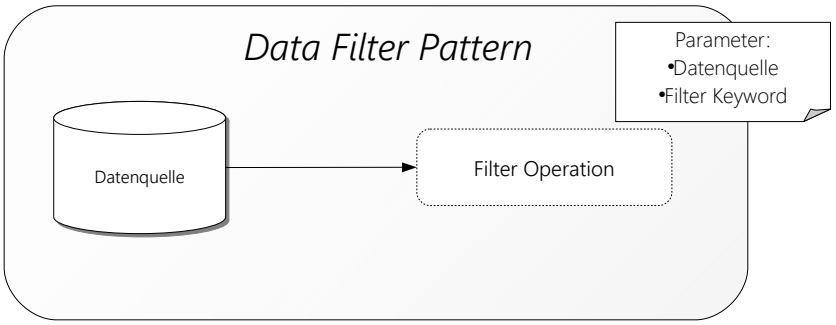


Abbildung 4.2: Data Filter Pattern

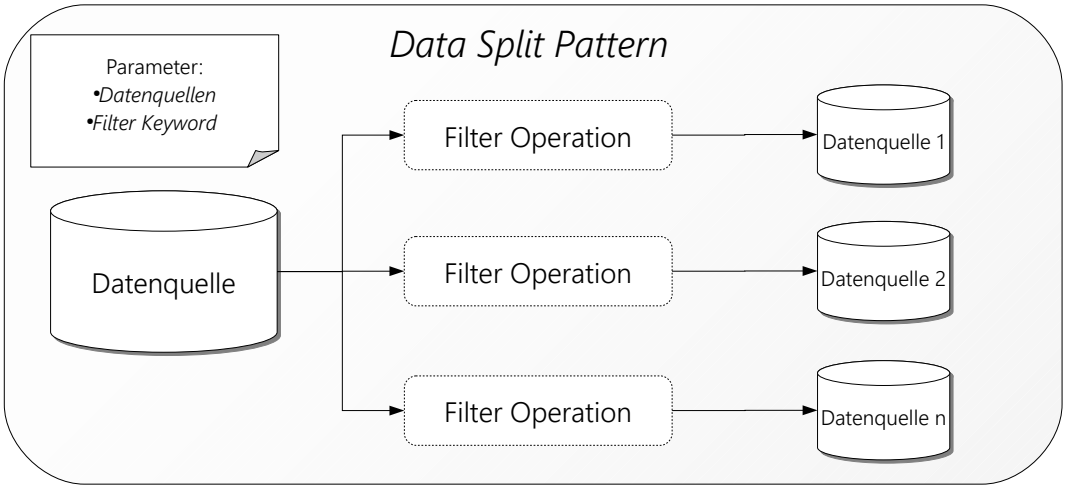


Abbildung 4.3: Data Split Pattern

der Benutzer beispielsweise einen Wert für die Filteroperation eingeben, die einen sogenannten Range bildet. Handelt es sich bei der Datenbank beispielsweise um eine Datenbank, die bestimmte Produkte mit Preisen enthält, kann durch ein Range-Wert = 100 diese Datenbank aufgeteilt werden in kleinere Datenbanken. Diese enthalten jeweils alle Produkte teuer als 100, günstiger als 100 und alle Produkte, deren Wert 100 beträgt.

4.4 Data Merge Pattern

Das **Data Merge Pattern**(DMP) bildet das Gegenstück zum zuvor vorgestellten Data Split Pattern. Abbildung 4.4 stellt diese grafisch dar. Bei diesem Pattern werden n Datenmengen zu einer Datenmenge zusammengefasst. Der Benutzer muss dabei alle Datenmengen als Parameter mitgeben und zusätzlich die Filter-Operation festlegen, mit welcher die Datensätze extrahiert werden können.

4.5 Data Iteration Pattern

Das **Data Iteration Pattern**(DIP) ist ein weiteres Pattern, welches jedoch im Gegensatz zu den bereits vorgestellten Patterns etwas komplexer aufgebaut ist. Abbildung 4.5 zeigt eine grafische Darstellung dieses Patterns. Hier werden mehrere Prozesse parallel ausgeführt. Daher handelt es sich hierbei um das Parallele Data Iteration Pattern. Das Gegenstück dazu ist das Sequentielle Data Iteration Pattern.

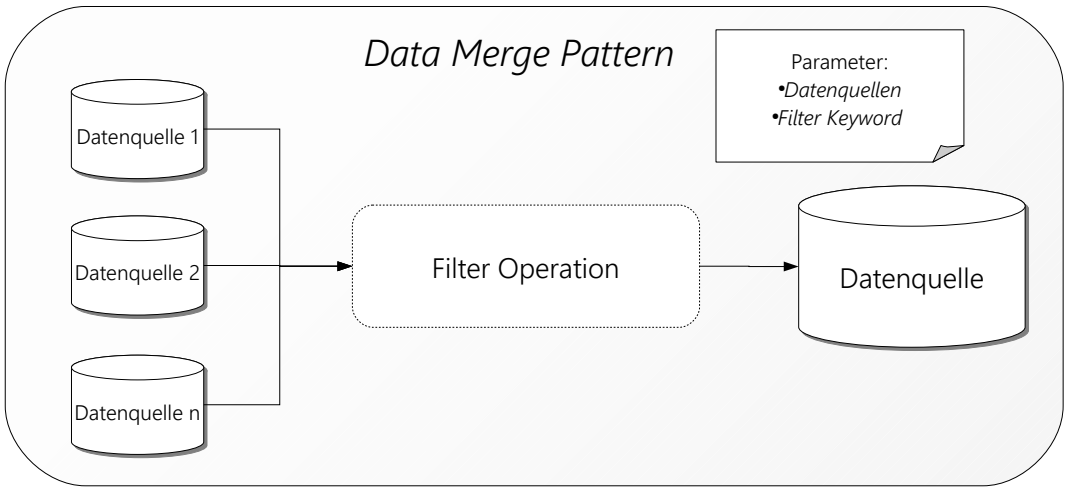


Abbildung 4.4: Data Merge Pattern

Das Parallele Data Iteration Pattern bekommt als Eingabe eine Datenmenge aus einer Datenbank, welche mithilfe einer Filter-Operation in 1...n Teilmengen aufgeteilt wird. Anschließend werden auf diese Teilmengen Operationen ausgeführt. Ziel ist es eine definierte Operation auf n Teilmengen parallel auszuführen. Aus diesen Operationen resultieren die Teilmengen 1*....n*, welche schließlich diese zu einer neuen Datenmenge in einer Datenbank zusammenfasst.

4.6 Sequentielles Data Iteration Pattern

Das **Sequentielle Data Iteration Pattern** (SDIP), welches in Abbildung 4.6 grafisch dargestellt ist,erwartet, wie auch beim Parallelen Data Iteration Pattern, eine Datemenge S als Eingabe. Hierbei wird diese jedoch nicht in kleinere Teilmengen aufgeteilt und es werden keine n Teilprozesse parallel ausgeführt. Die n Teilprozesse werden stattdessen nacheinander ausgeführt. Bei jedem Iterationsschritt wird mithilfe einer Filter-Operation eine Teilmenge T bestimmt, auf welches anschließend eine Operation ausgeführt wird. Diese Operation überführt die Teilmenge T in T*. Diese wird im nächsten Verarbeitungschritt in die Datenmenge S in der Zieldatenbank integriert. Die daraus resultierende Datenmenge S* dient schließlich als Eingabe für den nächsten Iterationsschritt. Die Anzahl der Iterationsschritte bestimmt der Benutzer bei der Eingabe der Parameter.

Die vorgestellten Patterns sind lediglich Beispiele. Es können weitaus komplexere Patterns für ein Mashup Plan erstellt werden. Dabei können Pattern verschachtelt vorkommen, d.h. Pattern können

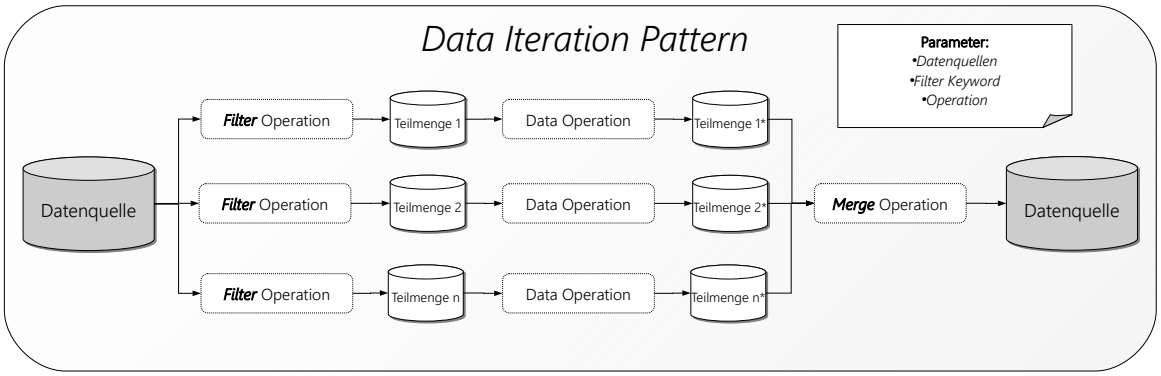


Abbildung 4.5: Data Iteration Pattern

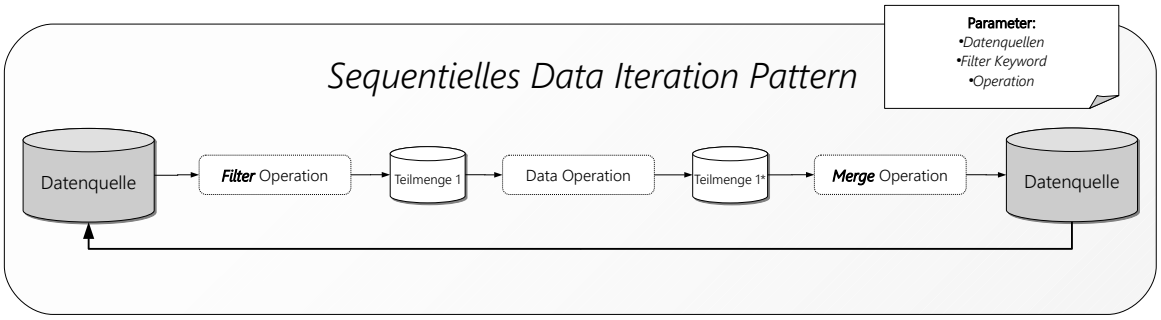


Abbildung 4.6: Sequentielles Data Iteration Pattern

selbst Patterns enthalten. Diese lassen sich dann anhand des regelbasiertes Mappings von Patterns innerhalb der Patternhierarchie auflösen bis in der untersten Ebene der Hierarchie lediglich ausführbare Knoten vorhanden sind. Besteht ein Mashup Plan aus zahlreichen komplexen Patterns, ist der Zeitaufwand für die Transformation dieses nicht ausführbaren Mashup Plans in ein ausführbares Format deutlich höher.

5 Implementierung

Dieses Kapitel beschäftigt sich mit der konkreten Implementierung des in den vorangegangenen Kapiteln beschriebenen Konzepts einer Fragment-Repository. Abschnitt 5.1 bietet eine kurze Auflistung aller Technologien, die zum Einsatz kamen um das Konzept umzusetzen. Anschließend wird in den darauf folgenden Kapiteln jeweils für jede Ebene der Architektur beschrieben, wie diese konkret implementiert wurde. In Abschnitt 5.2 werden zunächst die in der Datenebene verwendeten *Technologien* erklärt und das *Datenmodell* beschrieben, welche zur Umsetzung von Fragmenten und Patterns verwendet wurde.

Anschließend wird in 5.3 erklärt, welche Technologien die *business logic* des Repositories umsetzen und wie die grundsätzlichen *Methoden*, die vom Repository bereitstellt werden, realisiert wurden.

In Abschnitt 5.4 wird separat anschließend die Funktionsweise der Methode transformFragment genauer erklärt, welche die Transformation eines nicht ausführbaren Workflows, welches Patterns enthält, in einen ausführbaren Workflow ausführt. Da fuer die Präsentationsschicht keine Benutzeroberfläche erstellt wurde und lediglich mit REST-Clients und Browser gearbeitet wurde, wird die Präsentationsschicht hier nicht weiter ausgeführt.

5.1 Verwendete Technologien

Für die Umsetzung des Konzepts wurden folgende Technologien eingesetzt.

- **JSON:** ist ein kompaktes, leicht-gewichtiges Datenformat für den Datenaustausch zwischen Anwendungen in einfacher, lesbarer Textform. Die unausführbaren Ausgangs-Workflows des Flex-Mash-Tools werden in JSON übersetzt.
- **Jackson:** Java-basierte Bibliothek zum Serialisieren von Java in JSON und umgekehrt
- **REST/HTTP :** REST-Paradigma zur Erstellung von Web-Services unter Bereitstellung einer einfachen API, basierend auf HTTP-Befehlen *Post*, *Get*, *Put*, *Delete*. Wurde als Paradigma für die Entwicklung des Repository-Dienstes verwendet
- **Spring Framework:** Rahmenwerk zur Entwicklung von mehr-stufigen Applikationen unter Verwendung von einfachen Java-Klassen und Features wie Dependence Injektion und Annotationen. Spring Tool und Framework wurde als Editor verwendet und für die Erzeugung der mehrstufigen Architektur der Repository, insbesondere der Repository-Funktionaliäten
- **MongoDB:** NoSQL-Datenbank im Bereich Big Data, welches ein JSON-ähnliches Format BSON verwendet. Die Fragmente und der Hauptteil der dazugehörenden Informationen werden in MongoDB Collections gespeichert.

- **MySQL:** Die zu den Fragmenten gehörenden Metadaten werden in eine relationalen Datenbank MySQL gespeichert.
- **PostMan:** Front-End-Werkzeug zur Adressierung und zum Testen von REST-Web-Services.

5.2 Datenebene

Auf der Datenebene werden alle relevanten Informationen und Daten für das Repository bereitgestellt. Dieser Abschnitt beschreibt kurz die *Organisation* der Daten in verschiedenen Datenbanken und Tabellen sowie das verwendete *Datenmodell*.

Bei den auf dieser Ebene verwalteten Daten handelt es sich um die Fragment-Daten, ihre Metadaten und weitere Informationen wie z.B. Informationen bezüglich der Web-Services, welche innerhalb der Fragmente aufgerufen werden. Alle diese Daten, außer dem eigentlichen Fragment-Code werden durch das Flex-Mash-Tool bereitgestellt und werden bei der Registrierung von neuen Fragmenten in die Repository berücksichtigt. Während die Fragment-Daten und die Namen der von ihnen aufgerufenen Web Services in zwei verschiedenen Collections „fragments“ und „fragmentWebServices“ in einer MongoDB-Datenbank gespeichert werden, werden alle zu den Fragmenten gehörenden Metadaten in einer Tabelle „fragmentMetadata“ innerhalb einer MySQL-Datenbank gespeichert. Diese Art der Datenaufteilung kann damit begründet werden, dass der Großteil der Daten in diesem Kontext von den Fragmenten ausgemacht werden und NoSQL-Datenbanken im Bereich Big Data Vorteile gegenüber den relationalen Datenbanken bieten.

So können komplexere Strukturen bis hin zu vollständigen Objekte, welche beispielsweise verschachtelt, weiteren Kinds-Objekte enthalten, einfacher in NoSQL-Datenbanken wie MongoDB gehandhabt werden als dies in relationalen Datenbanken der Fall ist. Der Grund hierfür ist, dass NoSQL-Datenbanken wie MongoDB kein Datenschema erfordern und somit flexibler sich den ändernden Ansprüchen unterschiedlicher Datenformate gerecht werden. Im Rahmen des Flex-Mash-Projektes werden Daten unterschiedlicher Quellen und unterschiedlicher Formate innerhalb eines Mashup-Flows integriert.

Für komplexer aufgebaute Objekte wie dem Fragment-Objekt(vergleiche Abschnitt 5.3.1 mit mehreren möglichen Verschachtelungen, müssten in einer relationalen Datenbank mit normalisierten Tabellen, multiple Tabellen erstellt und diese über Primärschlüssel-Fremdschlüssel-Paare miteinander verknüpft werden. Bei einer Abfrage der Daten müssten umgekehrt mehrere JOIN-Operationen ausgeführt werden, um alle Informationen bzgl. eines Fragments zu erhalten. Dies ist zeitaufwendig und kann im Big Data- Kontext wie Twitter und Google zu langen Wartezeiten führen. Werden die relevanten Daten verteilt in einem verteilten Datenbanksystemen gehalten, kann dies zusätzlich die Wartezeit der Anfragen erhöhen. Aus genannten Gründen wurde entschieden, den Großteil der Daten in einer NoSQL-Datenbank zu speichern. Somit enthält die Collection „fragmentsälle Fragmente. Das Json-Objekt, das einem Fragment entspricht, besitzt eine eindeutige ID, welche bei der Registrierung des Fragments durch MongoDB automatisch erzeugt und vergeben wird. Das eigentliche Fragment, d.h. der in einer bestimmten Workflow-Sprache formulierte Code-Schnipsel, welcher ausführbar ist, wird in Form eines Strings vorgehalten. Dabei kann es sich beispielsweise um den Programmcode für eine Aktivität in einem BPEL- oder Node-Red-Workflow handeln. Dieser Code-String ist als Wert

einem Schlüsselwert „fragment“ zugeordnet. Gemeinsam mit der internen ID eines Fragments bildet dieser Schlüsselwert-Wert-Paar einen Eintrag in der Collection „fragments“. Jedes Fragment besitzt Metadaten, welche diesen genauer beschreiben. Diese sind für den Repository-Dienst essentiell, da sie bei Anfragen, die an die Repository gestellt werden als Suchkriterien verwendet werden. Innerhalb der Repository wird die intern vergebene ID zur eindeutigen Identifizierung der Fragmente verwendet. Da diese ID's jedoch den Benutzern des Repository-Dienstes nicht bekannt sind, müssen Fragmente über ihre Metadaten gesucht und identifiziert werden.

Zu den Metadaten gehören:

- Name: der Name des Fragments ist informationstragend, so dass es bereits die Funktion oder den Einsatzbereich des Fragments andeutet (z.b. database)
- Typ: diese Metainformation gibt an, ob es sich bei dem Fragment um ein Pattern handelt oder nicht. Im Falle eines Patterns wird der Wert „pattern“ verwendet. Ansonsten wird die Art der von dem Fragment ausgeführten Operation angegeben (z.B. „MySQL“)
- Operation: diese Metainformation benennt die ausgeführte Operation (z.B. „extract“)
- Property: bei dieser Metainformation handelt es sich um ein eigenes JSON-Array, welches verschachtelt im JSON-Object für Metadata enthalten ist. Es enthält Werte für die Schlüsselwerte Eingabe- und Ausgabeparameter, welche bei Pattern-Fragmenten eine Rolle spielen, sowie „filter criteria“, welche z.B. bei der Extraktion und Filterung von Daten aus Datenquellen angegeben werden müssen. Innerhalb von Workflows werden oft externe Web-Services aufgerufen. Diesen werden Eingabe-Parameter übergeben und die vom Web-Service zurückgelieferte Ausgabe dann in Ausgabeparametern abgespeichert und im weiteren Verlauf des Workflows weiterverarbeitet. Bei der Transformation von Mashup-Flows in BPEL-Workflows werden im Flex-Mash-Tool z.B. BPEL-Fragmente erstellt, welche die Web-Services SQLFilter, SQLExtractor, TwitterFilter oder TwitterExtractor aufrufen. Die Informationen über diese Web-Services sind für die Ausführung des Fragments von Bedeutung und müssen ebenso in der Repository abgespeichert werden. In einer Collection „fragmentWebServices“ wird die Zuordnung von Fragmenten und Web-Services festgehalten, wobei ein Eintrag der Collection jeweils einen Schlüsselwert-Wert-Paar für die Id des Fragments und eines für den Namen des im Fragment aufgerufenen WebServices enthält. Die Gesamtheit der Daten bezüglich eines Fragments, welche verteilt über zwei verschiedene Datenbanken in zwei Collections und einer Tabelle abgelegt ist, wird über die intern eindeutige ID des Fragments „zusammengehalten“, d.h. die Verknüpfung und korrekte Zuordnung der zusammengehörenden Daten erfolgt über die Fragment-ID .

5.3 Datenzugriffsebene

Nachdem in Abschnitt 5.2 das Datenmodell und die Organisation der Daten in der Datenzugriffsebene beschrieben wurden, werden in diesem Abschnitt die implementierten Funktionen und die für diese Funktionen verwendeten Datenformate im Detail beschrieben. Die Umsetzung dieser Ebene stellte den Hauptteil bei der Implementierung des Repository-Dienstes dar. In 5.3.1 wird das Datenformat Fragment erklärt, welches bei der Registrierung eines Fragments verwendet wird. Auf die einzelnen Funktionen des Repositories, welche die „business logic“ realisieren, wird in 5.3.3 eingegangen.

Listing 5.1 Beispiel für JSON-Objekt für die Registrierung eines Fragments

```
{
       fragment:    <bpel:invoke name=\"InvokeNYTExtractor\"
          partnerLink=\"SQLExtractorParnterLink\"
      operation=\"extract\" portType=\"ns:SQLExtractor
      \" inputVariable=\"SQLExtractorParnterLinkRequest\" outputVariable=
      \"SQLExtractorParnterLinkResponse\"></bpel:invoke>";
       metadata:   [ {
              name   :  MySQL ,
              type:   MySQLv1.0
              op:    extract
             property:  [ {
                   inputVariable:  SQLExtractorParnterLinkRequest,
                  outputVariable:SQLExtractorParnterLinkResponse,
                   filter  criteria:  someFilter
            } ]
      } ]
}
}
```

Schliesslich wird die Umsetzung der Transformation von Workflows durch die Funktion transformFlow in Abschnitt 5.4.3 näher erklärt.

5.3.1 Die Klasse Fragment

Das wichtigste Element bei der Erzeugung von ausführbaren Workflows aus Mashup-Flows bildet das Fragment. Es stellt den grundlegenden Baustein dar, aus welchen die ausführbaren Workflows zusammengesetzt werden. Neben dem eigentlichen Fragment bzw. Fragment-Code werden innerhalb der Repository die zum Fragment gehörenden Metadaten und Informationen über die vom Fragment aufgerufenen Web-Services abgespeichert (vergleiche Abschnitt Abschnitt 5.2). Intern entspricht die Gesamtheit aller dieser Daten, welche über eine eindeutige Fragment-ID verknüpft sind, einem Fragment. Dementsprechend muss der Benutzer bei der Registrierung eines Fragments alle notwendigen Daten angeben. Hierfür wird der in Abschnitt Abschnitt 5.3.3 näher beschriebenen Methode „registerFragment“ als Eingabeparameter ein einzelnes JSON-Objekt übergeben. Der Aufbau dieses JSON-Objekts wird in Listing 5.1 anhand eines Beispiels dargestellt.

Um dieses JSON-Objekt intern weiterzuverarbeiten wurde eine Java-Klasse Fragment erstellt, welche den Aufbau des JSON-Objekts widerspiegelt. Den Schlüsselwerten des JSON-Objekts entsprechend, wurden Felder mit getter- und Setter-methoden für die Klasse erstellt. Diese Klasse ist in Listing 5.2 dargestellt. Der Übersichtlichkeit halber werden die importierte Java-Klassen nicht angezeigt. Das Feld „fragment“ ist ein String, der als Wert den Programm-Code für das Fragment enthält. Das Feld „metadata“ ist vom Typ JSONObject und enthält die Metadaten für das Fragment. Dieses JSON-Object muss dem in Listing 5.1 gezeigten Wert des Schlüsselwertes entsprechend aufgebaut sein. Bei der Registrierung des Fragments wird programmatisch auf Existenz bestimmen Schlüsselwert-Wert-Paare hin geprueft. Bei fehlenden Werten wird eine Fehlermeldung ausgegeben. Prinzipiell hätte das Feld metadata ebenso als eine Inner-Class in Java implementiert werden können, da das Metadaten-Objekt eng an die Fragment-Klasse gekoppelt ist. Beim letzten Feld „creation date“ handelt es sich um einen

Listing 5.2 Java-Klasse zur Repräsentation eines Fragments

```
package repoFragments;
@Document
public class Fragment {
      @NotNull
      private String fragment;
      @NotNull
      private JSONObject metadata = new JSONObject();
      @NotNull
      private Date creationDate;

      public String getFragment() {
            return fragment;
      }

      public void setFragment(String fragment) {
            this.fragment = fragment;
      }

      public JSONObject getMetadata() {
            return metadata;
      }

      public void setMetadata(JSONObject metadata) {
            this.metadata = metadata;
      }

      public Date getCreationDate() {
            return creationDate;
      }
}
}
```

Zeitstempel der bei der erstmaligen Registrierung des Fragments mit dem Registrierungszeitpunkt als Wert belegt wird. Dieser Zeitstempel wird später beispielsweise bei der Sortierung von Fragmenten verwendet. Alle Felder besitzen die Annotation „@NotNull“, da sie unbedingt mit Werten belegt sein müssen damit eine erfolgreiche Registrierung erfolgen kann. Die gesamte Klasse Fragment besitzt die Annotation „document“, welches eine Spring-Annotation ist, die die Java-Klasse als Eintrag in einer Collection innerhalb einer MongoDB-Datenbank spezifiziert.

5.3.2 Der Repository-Dienst

Die business logic in der Datenebene wird von der Repository bereitgestellt. Um eine Repository programmtechnisch zu realisieren, wurde das Spring Framework verwendet (vergleiche Abschnitt Abschnitt 5.2). Durch Verwendung von Spring können mehrstufige Applikationen unter Verwendung einfacher Java-Klassen („POJOS-Plain Old Java Objects) und Features wie Annotationen und Dependency Injection erstellt werden. Durch Annotationen wird die Verwendung von XML-Konfigurations-Dateien, welche z.B. die Beziehungen zwischen den verschiedenen Java-Klassen festlegen, ueberfluessig. Zudem liefert Spring eine MongoDB-Repository-Schnittstelle, welche Standard-Funktionen, wie

Listing 5.3 Die Klasse FragmentRepository

```
package myFragments;

@RepositoryRestResource(collectionResourceRel = "fragments", path = "fragments")
public interface FragmentRepository extends MongoRepository<Fragment, Integer> {
      List<Fragment> findByFragmentId(@Param("fragmentId") int fragment_id);

      public void deleteByFragmentId(@Param("fragmentId") int fragment_id);

      @Query(value="{ }", fields="{'creationDate': false})
      List<Fragment> findAllFragments(Sort sort);
}
```

z.B. Anfragen zum Finden aller Elemente (findAll()) automatisch bereitstellt. Weitere Funktionen können einfach hinzugefügt werden, indem die Anfrage mit Hilfe von Annotationen formuliert werden kann. Das Grundgerüst der Repository-Klasse ist in Listing 5.3 gezeigt.

Es handelt sich dabei um eine Schnittstellendefinition, welche die vom Spring-Framework bereitgestellte Repository-Schnittstelle MongoRepository beerbt. In diesem Beispiel wurde eine zusätzliche Funktion findallFragments zu den Standard-Funktionen deleteByFragmentId und findByFragmentId hinzugefuegt. FindAllFragments liefert alle Fragmente zurueck, sortiert diese jedoch. Zudem wird mit Hilfe der Annotation „Query“ eine Projektion definiert, so dass das Schlüsselwert-Wert- Paare creationdate nicht im Anfrageergebnis angezeigt wird. Über die Annotation @RepositoryRestResource wird der Name der Collection in MongoDB spezifiziert und über welchen Pfad dieser Methode des Repositories aufgerufen werden kann. Da über diese Schnittstelle jedoch nur Signaturen von Methoden definiert werden können und somit die Möglichkeiten bei der Implementierung von Methoden eingeschränkt sind, wurde eine weitere Klasse FragmentRestController definiert, welche den Hauptteil der Funktionalitäten des Repositories bereitstellt. Diese Klasse agiert wie ein klassischer Controller in einer Model-View-Controller-Architektur (MVC), d.h. alle Anfragen an die Repository werden von dieser Klasse entgegengenommen und weiterverarbeitet. Es werden Operationen auf der zu Grunde liegenden Datenebene ausgeführt und Anfrageergebnisse an den Benutzer zurueckgeliefert. In Listing 5.4 wird ein Beispiel einer Methode updateMetadata des FragmentRestControllers gezeigt, mit der die Metadaten eines Fragments aktualisiert werden können.

Mit Hilfe der Annotation *@RequestMapping* wird der URL-Pfad festgelegt, über den die Methode des REST-Services aufgerufen werden kann. Die geschweiften Klammern im URL-Pfad kennzeichnen die Parameter, die dann mit Hilfe der Annotation *@PathVariable* in der Signatur der Methode adressiert werden können. Die Annotation *@ResponseBody* legt schliesslich fest, dass das Ergebnis des Methodenaufrufs dem Benutzer zurückgeliefert wird. Die restlichen Methoden der Klasse werden im folgenden Abschnitt beschrieben.

5.3.3 Die Funktionen des Repository-Dienstes

Im folgenden Abschnitt werden die Methoden, die vom REST-Controller des Repository-Service bereitgestellt werden, näher beschrieben. Diese bilden die Schnittstelle, über die der Repository-Dienst angesprochen werden kann. Für jede Methode wird kurz ihre Funktion erklärt und angegeben,

Listing 5.4 Eine Methode der Klasse FragmentRepository

```
@RequestMapping(value="/updateMetadata/{fragmentIdParam}/{metadataParam}",
                                              method=RequestMethod.POST)
@ResponseBody
public void updateMetadata(@PathVariable("fragmentIdParam") int fragmentIdParam,
 @PathVariable("metadataParam") String metadataParam) throws SQLException{

      List<Fragment> resultFragments = new ArrayList<Fragment>();
      String sql = "UPDATE fragmentMetaDataTable SET metadata='" + metadataParam + "' WHERE
          fragmentId='" + fragmentIdParam + "';";
      resultFragments = repo.findByFragmentId(fragmentIdParam);
      if (resultFragments.isEmpty()) {
            System.out.println("There is not fragment with ID :" + fragmentIdParam);
      }
      else {
            executeDMDBOperation(sql);
            System.out.println("Metadata for fragment with fragmenId " + fragmentIdParam + " has
                been    updated" );
      }
}
```

welche Eingabe- und welche Ausgabeparameter sie besitzt und welches Format bzw. Struktur diese Parameter haben muessen.

Die Gesamtheit der hier beschriebenen Methoden dienen der Verwaltung der Fragment-Repository und werden über eine REST-Schnittstelle bereitgestellt. Es liegt jedoch momentan keine Benutzeroberfläche vor, d.h. der Aufruf der Methoden erfolgt über einen REST-Client (wie z.b. Postman) oder einen Web-Browser. Darüber hinaus erfordert die Verwendung technisches Wissen wie z.B. über Datei-Formate wie JSON. Dadurch ist diese Schnittstelle weniger für den Gebrauch durch Benutzer ohne IT-Know-How gedacht, sondern eher für Benutzer, die über Domänen- und technisches Wissen verfügen und den Repository-Service bereitstellen bzw. in ihre Anwendung einbinden wollen.

Die Menge der aufgeführten Methoden stellt den aktuellen Stand des Funktionsumfangs der Repository dar und kann in zukünftigen Arbeiten (auch hinsichtlich der Eingabe- und Ausgabeparameter der einzelnen Methoden) erweitert bzw. angepasst werden.

ıtemize

findByMetaData: Mit Hilfe dieser Funktion kann der Benutzer Fragmente innerhalb des Repositories anhand ihrer Metadaten suchen und finden. Bei der Suche muss als Eingabeparameter ein JSON-Objekt mit Paaren von Schlüsselwerten und ihren Werten übergeben werden. Die Suche nach dem passenden Fragment in der Repository erfolgt dann in Form eines Abgleichs. Schlüsselwerte, die im Eingabeparameter angegeben wurden, werden bei der Suchanfrage nur dann berücksichtigt, falls Fragmente in der Repository existieren, welche diesen Schlüsselwert in der Metadaten-Tabelle als Spalte aufweisen. Ist dies der Fall, wird im zweiten Schritt geprueft, ob der Wert in der Tabelle mit dem im Eingabeparameter angegebenen Wert übereinstimmt.

Stimmen die Werte bei keinem Fragment überein, wird eine Benachrichtigung ausgegeben, dass kein passendes Fragment mit geforderten Eigenschaften in der Repository gefunden wurde. Andernfalls

werden alle Fragmente zurueckgeliefert, welche für gegebene Schlüsselwerte, die passenden Werte aufweisen. Anzumerken ist, dass Fragmente ebenfalls in der Antwortmenge ausgegeben werden, falls sie für gewisse geforderte Schlüsselwerte keinen Wert in der Tabelle aufweisen, d.h. nur für existierende Schlüsselwerte müssen die Werte uebereinstimmen. Der Eingabeparameter dieser Methode muss ein JSON-Obkekt mit Schlüssel-Wert-Wert-Paaren sein, ansonsten aber keinem vorgegebenem Format folgen.

- **EingabeParameter:** JSONObject mit Key-Value-Paaren
- **Ausgabe:** vollständiges Fragment mit ID, Fragment-Code und Metadata

updateMetaData:

Mit Hilfe dieser Funktion kann der Nutzer die Metadaten für Fragmente innerhalb des Repositories aktualisieren. Hierbei muss als Eingabeparameter nicht ein vollständiges JSON-Objekt übergeben werden, d.h. es müssen nicht für alle möglichen Spalten in der Metadaten-Tabelle entsprechende neue Key-Value-Paare angegeben werden, sondern nur jene, deren Werte aktualisiert werden sollen. Falls ein angegebener Schlüsselwert nicht existiert, wird eine Fehlermeldung ausgegeben.

Um das Fragment eindeutig zu identifizeren, dessen Metainformationen aktualisiert werden sollen, muss zudem eine Fragment-ID angegeben werden. Dies bedingt, dass dem Benutzer im Vornherein die Fragment-ID bekannt ist. Diese kann durch die Methoden showFragments bzw. listFragments in Erfahrung gebracht werden.

Anzumerken ist für diese Methode, dass Metadaten normalerweise relativ selten verändert werden, da sie eine beschreibende Funktion besitzen und eng an das beschriebene Objekt gebunden sind. Da diese Methode bis auf Dateiformatskontrollen keine weiteren Bedingungen prueft, sollten Änderungen nur von Benutzern mit Domänenwissen vorgenommen werden.

- **EingabeParameter:** FragmentId, neue Metadaten als JSON-Objekt mit zu ändernden Schlüsselwert-Wert-Paaren:
- **Ausgabe:** Bestätigung über erfolgreiche Operation oder Fehlermeldung

updateFragment: Ähnlich der Methode updateMetadata können auch hier Fragment-Daten aktualisiert werden. Im Gegensatz dazu kann hier jedoch ein vollständiges Fragment aktualisiert werden. Als Eingabeparameter erwartet die Methode ein JSON-Objekt, welches alles Schlüsselwert-Werte-Paare enthält, die aktualisiert werden sollen. Ist als Schlüsselwert die Fragment-ID nicht angegeben, wird eine Fehlermeldung ausgegeben. Falls darüber hinaus ein angegebener Schlüsselwert nicht gefunden werden kann, wird ebenso eine Fehlermeldung ausgegeben.

Das JSON-Objekt, das als Parameter übergeben wird, darf nur Schlüsselwerte enthalten, die auch in der Klasse Fragment(vgl. Abschnitt 5.3.1)zu finden sind. Um das Fragment eindeutig zu identifizeren, dessen Metainformationen aktualisiert werden sollen, muss insbesondere eine Fragment-ID angegeben werden. Dies bedingt, dass dem Benutzer im Vornherein die Fragment-ID bekannt ist. Diese kann durch die Methoden showFragments bzw. listFragments in Erfahrung gebracht werden.

Die Methode kann Änderungen in allen Datenquellen vornehmen. D.h. je nachdem welcher Schlüsselwert angegeben wurde, wird entsprechend in der Metadaten-Tabelle bzw. in der Collection Fragments bzw. der Collection fragmentWebServices Daten manipuliert.

- **EingabeParameter:** FragmentId, neue Daten als JSON-Objekt mit zu ändernden Schlüsselwert-Wert-Paaren
- **Ausgabe:** Bestätigung über erfolgreiche Operation oder Fehlermeldung oder Fehlermeldung

registerFragment:

Diese Methode stellt eine der wichtigsten Methoden der Repository dar. Sie ermöglicht dem Benutzer die Erzeugung bzw. das Abspeichern von neuen Fragmenten in der Repository. Die Eingabe des Benutzers muss dabei der Java-Klasse eines Fragments entsprechend aufgebaut sein (siehe Abschnitt 5.3.1), d.h. es muss ein JSON-Objekt übergeben werden, in dem insbesondere jene Schlüssel-Werte, deren Wertebereich im Datenmodell als Not-Null festgelegt ist, mit Werten belegt sein muessen.

Bei der Registrierung eines Fragments ist die gleichzeitige Registrierung von Metadaten und Eigenschaften(Properties), welche das Fragment näher beschreiben, obligatorisch. Dies ist notwendig, um später bei Suchanfragen Fragmente eindeutig identifizieren und das Suchergebnis anhand von Suchkriterien einschränken zu können. Da die Suche in Form eines Metadaten-Abgleichs abläuft, müssen die Schlüsselwerte für metadata und properties im übergebenen JSON-Objekt dementsprechend zwingend Werte enthalten. Insbesondere der Wert für den Schlüssel „pattern“ ist wichtig, um das Fragment richtig zu kategorisieren, so dass bei einer Transformation die Fragmente nach ihrer Eigenschaft „robust“ oder „time-critical“ selektiert werden können.

Anzumerken ist, dass jedes Fragment, das erfolgreich in der Repository abgelegt werden kann, eine eindeutige ID besitzt. Diese wird jedoch nicht als Eingabeparameter vom Benutzer festgelegt, sondern automatisch vom System zugeordnet. Unter dieser intern eindeutigen Fragment-ID wird das Fragment in einer Tabelle Fragments in einer NoSQL-Datenbank abgespeichert. Gleichzeitig wird der Aufruf einer weiteren Methode getriggert, welche in einer relationalen Datenbank die zu dem Fragment gehörenden Metadaten unter derselben ID abspeichert. Des Weiteren werden in einem zweiten Methodenaufruf die Namen der Web Services, welche im gespeicherten Code-Fragment aufgerufen werden, in einer Tabelle in der NoSQL-Datenbank festgehalten.

- **EingabeParameter:** JSON-Object, welches der Klasse Fragment entspricht
- **Ausgabe:** Bestätigung über erfolgreiche Operation oder Fehlermeldung oder Fehlermeldung

deleteFragment: Diese Methode stellt das Gegenstück zur oben beschriebenen Methode registerFragement dar. Mit Hilfe dieser Methode kann ein Fragment aus der Repository gelöscht werden. Hierzu muss lediglich die ID des zu löschenden Fragments als Parameter übergeben werden. Da die Fragment-ID bei der erstmaligen Registrierung des Fragments in die Repository intern vergeben wird, muss diese seitens des Benutzers zunächst in Erfahrung gebracht werden, falls diese nicht vorher bekannt ist. Hierfür bieten sich die Methode showFragments oder listFragments an, welche weiter unten beschrieben werden.

Falls die eingegebene ID nicht existiert, wird eine Fehlermeldung ausgegeben. Die Ausführung dieser Methode entfernt das Fragment mit der entsprechenden ID aus der Fragment-Tabelle bzw. -Collection

in der NoSQL-Datenbank. Zusätzlich müssen alle mit diesem Fragment verbundenen Daten aus den weiteren Datenbanken bzw. Tabellen entfernt werden, da sie keine weitere Verwendung mehr finden. Hierzu werden intern entsprechende weitere Methoden getriggert, welche die zum Fragment gehörenden Metadaten aus der Metadaten-Tabelle sowie den Eintrag aus der Fragment-Web-Service-Tabelle entfernen, welche den im Fragment aufgerufenen Web-Service gespeichert hat.

- **EingabeParameter:** Fragment-ID
- **Ausgabe:** Bestätigung über erfolgreiche Operation oder Fehlermeldung oder Fehlermeldung

Bei der Registrierung von höherrangigen Pattern-Fragmenten in der Repository müssen im Gegensatz zur Registrierung einfacher Fragmente bzw. Patterns folgende Aspekte berücksichtigt werden: Höherrangige Patterns repräsentieren mindestens einen Knoten, in der Regel jedoch mehrere Knoten im Workflow. Dies bedeutet, dass diese Fragmente später durch den Transformationsprozess im Patterntransformer in ein oder mehrere Fragmente bzw. JSON-Knoten umgewandelt werden. Als Eingabeparameter muss hier ein JSON-Objekt übergeben werden, welches verschachtelt alle relevanten Knoten enthält. Dabei muss jeder enthaltene Knoten wiederum der Klasse Fragment entsprechend aufgebaut sein.

showFragments: Diese Methode erlaubt es, alle Fragmente in der Collection Fragments aufzulisten. Dabei werden die Fragmente momentan vollständig als JSON-Objekte aufgeführt, d.h. die Fragment-ID, der Schlüsselwerte Fragment mit dem eigentlichen Fragment-Code als Wert, der Schlüsselwert metadata mit dem JSON-Objekt, welches die Metadaten enthält.

In zukünftigen Arbeiten könnte die Methode so abgeändert werden, dass für ein Fragment jeweils komplett alle im Repository vorliegenden Daten angezeigt werden, d.h. alle seine Metadaten und die Namen von ihm aufgerufenen WebServices. Des Weiteren könnten durch Angabe von Parametern gewisse Werte aus dem Resultat ausgeschlossen werden, ähnlich einer Projektion bei SQL-Datenbanken, so dass beispielsweise nur die Fragment-ID aller Fragmente in der Ergebnismenge aufgelistet wird. Die Auflistung folgt keinerlei Reihenfolge oder Sortierung.

- **EingabeParameter:** keine
- **Ausgabe:** Auflistung aller Fragment-JSON-Objekte aus der Collection fragments.

listFragments: Mit Hilfe dieser Methode können wie bei der Methode showFragments alle Fragmente aus der Collection Fragments aufgelistet werden. Im Gegensatz dazu erfolgt hier jedoch eine Auflistung aller Fragmente sortiert nach ihrem Zeitstempel, welcher belegt, wann ein Fragment in der Repository registriert wurde. Als Eingabeparameter erwartet die Methode einen Wert, der die Sortierreihenfolge angibt. Zulässige Werte sind hier „Ascending“ bzw „Descending“ für eine aufsteigende bzw. absteigende Reihenfolge.

In zukünftigen Arbeiten könnte die Methode so abgeändert werden, dass für ein Fragment jeweils komplett alle im Repository vorliegenden Daten angezeigt werden, d.h. alle seine Metadaten und die Namen von ihm aufgerufenen WebServices. Des Weiteren könnten durch Angabe von Parametern gewisse Werte aus dem Resultat ausgeschlossen werden, ähnlich einer Projektion bei SQL-Datenbanken, so dass beispielsweise nur die Fragment-ID aller Fragmente in der Ergebnismenge aufgelistet wird. Zudem könnte der Wert, nach dem sortiert werden soll als Parameter angegeben werden. Dieser

Wert muss zwangsläufig einem Spaltenwert einer Tabelle bzw. einem Schlüsselwert einer Collection entsprechen.

- **EingabeParameter:** Wert für aufsteigende bzw. absteigende Sortierung
- **Ausgabe:** Auflistung aller Fragment-JSON-Objekte aus der Collection Fragments absteigend sortiert nach creation time.

5.4 Transformation von Mashup-Flows

Die *Transformation* von Mashup-Flows wandelt einen unausführbaren, abstrakten JSON-Flow in ausführbaren Code um. Eine solche Transformation wurde bereits in der vorangegangenen Arbeit im Rahmen des Flex-Mash-Projekts implementiert. Um den Bezug zu diesem bereits existierenden Transformationsprozess herzustellen und somit den Anknüpfungspunkt dieser Arbeit zu verdeutlichen, folgt in Abschnitt 5.4.1 eine kurze Bestandsaufnahme bzgl. der Transformation von Mashup-Flows. Anschließend wird in Abschnitt 5.4.3 die für diese Arbeit erstellte Methode zur Transformation erklärt.

5.4.1 Bestandsaufnahme

Der folgende Abschnitt bezieht sich auf den letzten Stand der Implementierung des Tools FlexMash zum Zeitpunkt der Anfertigung dieser Arbeit. **FlexMash** bietet dem Benutzer die Möglichkeit einen Data Mashup Flow zu transformieren. Bei dem Mashup Flow handelt es sich um einen Workflow, der über die Benutzeroberfläche des Tools modelliert wurde. Dieser ist nicht ausführbar, da er lediglich aus abstrakten, Plattform-unabhängigen Knoten zusammengesetzt ist, denen kein konkreter Code-Fragment in einer spezifischen Programmiersprache zugeordnet ist. Über die Benutzeroberfläche kann der Benutzer die Transformation des Workflows anstoßen.

Hierzu wird eine **HTTP**-Anfrage an das Flex-Mash-Tool gesendet. Diese enthält als Parameter das bestimmende Kriterium für die Transformation, welche die Haupteigenschaft des zu erzeugenden Workflows festlegt. Je nachdem ob als Parameter *„robust“* oder *„time-critical“* gewählt wurde, wird eine entsprechende Workflow-Sprache ausgewählt, in der die ausführbare Version des Ausgangsworkflows umgesetzt bzw. in welche sie transformiert werden soll. Somit bestimmt dieser Parameter die Transformationsmethode, welche verwendet werden muss, um einen Workflow zu erhalten, der dem vom Benutzer geforderten Kriterium entspricht. Hierzu wird der modellierte Workflow zunächst in einen JSON-Flow umgewandelt und liegt als JSON-Array vor, in welchem jedes Feld einen JSON-Knoten enthält (siehe Abbildung **??**). Jeder JSON-Knoten wiederum enthält ein Array targets, in welchem die Namen aller seine Nachfolger-Knoten, mit denen er verbunden ist, abgespeichert sind.

Bei der Transformation wird jeder JSON-Knoten in einer for-Schleife durchlaufen und für jeden Knoten jeweils ein Objekt der Klasse *FlowNode* erstellt. Diese Klasse ist eine vordefinierte Java-Klasse, welche alle notwendigen Felder besitzt um intern einen Workflow-Knoten zu repräsentieren (siehe Listing 5.5). Dies beinhaltet beispielsweise die Namen aller Nachbarknoten in Form eines Arrays „target“. Schließlich liegt der Ausgangs-Workflow als interne Repräsentation in Form von lose gekoppelten

Listing 5.5 Die Klasse FlowNode

```
package de.unistuttgart.ipvs.as.flexmash.utils.transformation_utils;

import java.util.ArrayList;

/**
 * Data model of a flow node
 */
public class FlowNode {
      public String name;
      public String type;
      public String source;
      public ArrayList<String> target;
      public String criteria;
      public String adress;
      public String user;
      public String password;
      public String table;
      public String hashtag;
      public String database_name;
      public String filter_criteria;
      public String category;

      public FlowNode() {
            target = new ArrayList<>();
      }
}
```

Knoten des Formats *FlowNode* vor. Beginnend mit dem Startknoten wird nun eine Queue erstellt, in der für den aktuell betrachteten Knoten die Transformationsmethode aufgerufen wird. Diese ordnet jedem Knoten anhand seines Typs einen String zu, welcher den Programmcode bzw. Code-Fragment für den Knoten darstellt. Diese Zuordnung von Code zu Knoten erfolgt momentan *hart-codiert*, d.h. alle möglichen Code-Strings sind im Programm-Code von Flex-Mash integriert. Nach der Bearbeitung des aktuellen Knotens, wird dieser aus der Queue entfernt und seine Nachfolge-Knoten aus seinem Array target in die Queue eingereiht. Dies wird wiederholt bis keine Knoten mehr enthalten sind. Als Endresultat der Transformation wird ein String zurückgegeben, der durch Konkatenation der Code-Strings der einzelnen Knoten erzeugt wird.

5.4.2 Konzept der Transformation

Der in dieser Arbeit entwickelte Repository-Dienst wird zwei Aspekte der Transformation von relevanten JSON- Workflows abdecken. Zum Einen werden in der Repository alle in Frage kommenden Code-Fragmente gespeichert und Suchfunktionen bereitgestellt, so dass bei der Transformation das geeignete Fragment gesucht und gefunden werden kann. Dadurch wird vermieden, dass Code-Fragmente hart-kodiert im eigentlichen Programm-Code integriert werden müssen. Zum Anderen wird der Repository-Dienst einen *Pattern-Transformer* bereitstellen, der die Transformation von nicht-ausführbaren Workflows mit Patterns in ausführbare, pattern-freie Workflows bewerkstelligt. Die

vom Repository-Dienst bereitgestellte Methode transformFlow knüpft an genau der Stelle an, an der der Ausgangs-Workflow durchlaufen wurde und in Form eines Arrays von JSON-Knoten vorliegt.

Die Methode **transformFlow** wird vom Repository bereitgestellt und ermöglicht die Transformation eines Work-Flows. Die Transformation bezeichnet hier die Umwandlung eines Workflows, welcher anfänglich nicht ausführbar ist, in einen ausfürbaren Workflow. Die Ausführbarkeit des Workflows wiederum bedeutet in diesem Kontext, dass der Workflow in einer konkreten Workflow-Sprache vorliegt, welche ausgeführt werden kann, wie z.B. BPEL oder NodeRed. In Bezug auf die Ausführbarkeit und den Transformationsprozess, bietet es sich an, die Hierarchie von Workflow-Patterns nochmals zu betrachten (vergleiche **??**). Der Ausgangs-Workflow ist nicht ausführbar, da er aus abstrakten Bestandteilen zusammengesetzt ist welchen keine konkreten Implementierungcodes zugeordnet sind. Dabei handelt es sich entweder um einfache Workflow-Knoten oder Patterns, welche zusätzlich über mindestens einen Workflow-Knoten abstrahieren.

Während einfache Workflow-Knoten Elemente der ersten Hierarchie-Ebene von abstrakten Workflow-Bestandteilen oberhalb der Blattebene darstellen, gehören Patterns den höheren Hierarchie-Ebenen an. Man kann einfache Workflow-Knoten ebenso als „einfache Patterns“ und eigentliche Patterns als „höher-rangige Patterns“ betrachten. Die eigentlichen Patterns müssen durch Transformationen Schritt für Schritt zunächst auf die Elemente der einfachen Workflow-Elemente abgebildet werden. Bei den Elementen der Blattebene, d.h. auf der untersten Hierarchie-Ebene unterhalb der abstrakten Workflow-Elemente, handelt es sich nicht mehr um abstrakte Bestandteile, sondern konkrete Implementierungen, welche ausgeführt werden können. Ziel der Transformations-Methode ist es, alle abstrakten Workflow-Bestandteile, sowohl einfache Knoten, als auch Patterns, von oben in Richtung nach unten in der Pattern-Hierarchie umzuwandeln, so dass nur noch ausführbare Elemente der Blatt-Ebene vorliegen.

Die Methode transformFlow soll nun genau dort ansetzen, wo im FlexMash-Tool aus dem in der Benutzeroberfläche modellierten Workflow ein JSON-Flow *„mashupFlowAsJSON“* erstellt wurde. Statt nun den mit Hilfe einer for-Schleife über das Array(aus dem JSON-Flow wurde in einem vorigen Schritt ein Array mit allen Knoten extrahiert) zu iterieren und fur jeden JSON-Knoten eine Methode aufzurufen, welche das JSON-Objekt in ein Code-Fragment in der geeigneten Workflow-Sprache umwandelt (hart-codiert), wird der gesamte JSON-Flow als Eingabe-Parameter einer Methode transformFlow übergeben. Als Ausgabe gibt diese schliesslich einen vollständig ausführbaren Workflow zurück.

5.4.3 Ablauf der Methode transformFlow

In Abschnitt 3.4.1 wurde bereits der Ablauf der Pattern-Transformation näher erklärt. Der folgende Abschnitt soll sich mehr auf den technischen Aspekt beziehen. Da zum Zeitpunkt der Anfertigung dieser Teil der Implementierung noch nicht vollständig bereitstand, werden die folgenden Sachverhalte konzeptionell und unter Vorbehalt erklärt. Die Methode transformFlow erhält einen JSON-Flow als Eingabe und extrahiert aus diesem das JSON-Array nodes. Die folgenden Ausführungen basieren auf der Grundlage, dass JSON-Knoten, insbesondere JSON-Knoten des Typs Pattern die in Listing 5.6 dargestellte Struktur haben. Anzumerken ist, dass ein Pattern-Knoten alle Knoten, auf die es bei einer Transformation abgebildet wird, verschachtelt in einem JSON-Array nodes innerhalb des JSON-Arrays properties enthält.

Listing 5.6 Beispiel für JSON-Knoten des Typs Pattern

```
{
      "name" : "dataFilterPattern",
      "type": "Pattern"
      "operation": "filter"
      "table": ""
      "source": ""
      "property": [ {
            "inputVariables": ["NYT"]
            "outputVariables": [ ],
            "filter criteria": "someFilter"
            "nodes" [
                  {
                        "name" : "dataSource_NYT2625",
                        "type": "Pattern"
                        "operation": "extract"
                        "table": ""
                        "source": ""
                        "property": [ {"inputVariables": [ {"input":"sourceName"}]
                              "outputVariables": [ ],
                              "filter criteria": ""
                              "targets": [{ "target": "filter3478"}]
                              "nodes" []
                        } ]
                  }
                  {
                        "name" : "filter3478",
                        "type": "Pattern"
                        "operation": "filter"
                        "table": ""
                        "source": ""
                        "property": [ {"inputVariables": ["sourceName" ]
                        "outputVariables": [ ],
                        "filter criteria": "someFilter"
                        "nodes" []
                  } ]
            } ]
      } ]
} }
```

Die Methode läuft in zwei Phasen ab. Diese sind in Abbildung 5.1 dargestellt.

- **Bereinigungs-Phase:** Ziel dieser Phase ist es einen JSON-Array von Patterns zu „bereinigen“, d.h. aus einem JSON-Array, welcher Knoten des Typs „pattern“ enthält, ein JSON-Array zu erzeugen, welches keine Patterns mehr enthält und nur noch aus Knoten besteht, welche nicht vom Typ „patternßind. Hierfür wird zunächst über das Array nodes iteriert und jeweils für jeden im aktuellen Array-Feld enthaltenen JSON-Knoten der Typ überprüft. Solange der Typ des Knotens nicht „pattern“ ist, wird mit dem nächsten Knoten fortgefahren. Andernfalls, liegt ein Pattern-Knoten vor und dieser muss entsprechend weiterverarbeitet werden. Hierfür wird die Methode mapPattern aufgerufen, welche das Pattern eventuell rekursiv auf ein Fragment

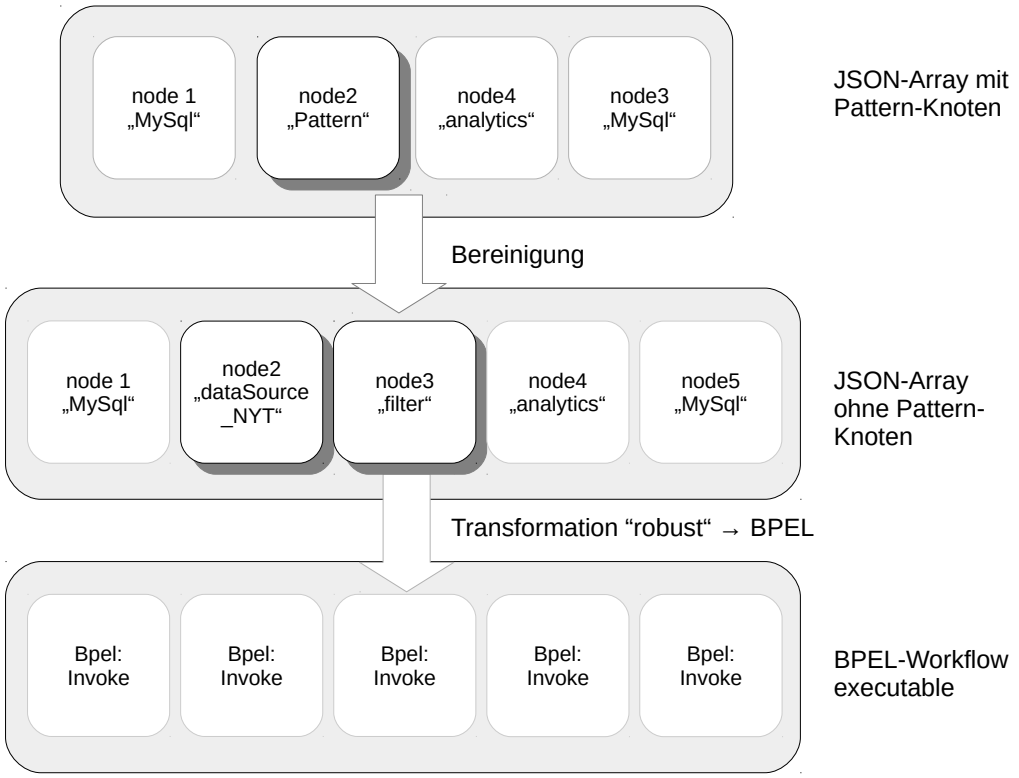


Abbildung 5.1: Die zwei Phasen der Transformation

der nächst-niedrigeren Hierarchiestufe abbildet, bis im bearbeiteten Fragment keine Knoten des Typs Fragment mehr vorhanden sind. Die Methode wird in Abschnitt**??** näher erläutert.

- **Transformations-Phase:** Die Transformations-Phase läuft dann entsprechend der in Flex-Mash existierenden Implementierung ab. Je nachdem, ob der Schlüssel Pattern mit dem Wert „robust“ oder „time-critical“ belegt ist, wird eine Konvertierungs-Funktion aufgerufen, welche den Workflow entweder in BPEL oder Node-Red transformiert. Diese Methoden wurden im Vergleich zu den ursprünglichen Methoden im Flex-Mash-Tool um eine Methode erweitert, welche bei der Auswahl des geeigneten Fragments nicht mehr hart-codiert einen Code-String einfügt, sondern in der Repository nach einem geeigneten Fragment sucht.

5.4.4 Die Methode mapPattern

Die Methode **mapPattern** erhält ein JSON-Object als Eingabe, welche für den Schlüsselwert type den Wert „pattern“ besitzt. Da es sich bei diesem JSON-Knoten um ein Pattern handelt, muss diese in ein Fragment der nächst-niedrigeren Hierarchie-Ebene umgewandelt werden, welche ausführbar ist oder weiter Patterns enthält. Ist das Letztere der Fall, wird die Methode rekursiv erneut für diese Patterns aufgerufen, bis nur noch Fragmente übrig bleiben, welche nicht vom Typ Pattern sind. Jeder

mapPattern-Aufruf liefert als Ausgabe einen JSON-Array mit ein oder mehreren enthaltenen JSON-Knoten. Die in jedem mapPattern-Aufruf JSON-Arrays werden anschliessend zu einem JSON-Array zusammengefasst und dieses Array bildet schliesslich die Gesamtausgabe.

In jedem mapPattern-Aufruf wird aus dem JSON-Object für das Pattern das im Wert des Schlüsselwertes „properties“ enthaltene, verschachtelte JSON-Array „nodesëxtrahiert. Dieses Array enthält alle JSON-Knoten, auf die das Pattern abgebildet werden muss. Jeder dieser JSON-Knote entspricht in ihrem Aufbau dem Aufbau der Flex-Mash-üblichen JSON-Knoten. D.h. die üblichen Schlüsselwerte „name“, „type“, „pattern“, „properties“ usw. sind enthalten und zudem verschachtelt ein Array „targets“, welches die Namen der Nachfolgeknoten enthält.

Die Abbildung erfolgt regelbasiert. D.h.bevor die mapPattern-Methode die Abbildung vornimmt, prüft sie ob alle für das Pattern notwendige Parameter im Array properties enthalten sind und dem geforderten Format entsprechen. Erst wenn dies der Fall ist, wird die Transformation durchgeführt. Bei der Transformation werden die Eingabeparameter eventuell an die entsprechenden Fragmente „weitergereicht“. Dies bedeutet, dass Schlusselwerte der Fragmente mit geeigneten Werten programmatisch belegt werden.

5.4.5 Die neue Methode convert

Diese Methode **convert** ist angelehnt an die Methode convert der Klassen *MashupPlanToBPELConverter* und *MashupPlanToNodeREDFlowConverter* im Flex-Mash-Tool. Anstatt jedoch den Fragmenten hart-codierte Code-Strings zuzuordnen, ruft die neue Methode eine weitere Methode **fragmentLookup** auf. Diese Methode gleicht bei der Suche nach dem geeigneten Fragment im Repository die im Array *„properties*"des aktuellen JSON-Knotens enthaltenen Ein- und Ausgabe-Parameter mit den im Array Metadata enthaltenen Werten der Fragmente im Repository ab. Hier muss sowohl die Anzahl der Parameter übereinstimmen, als auch das Format ihrer Werte. Zudem muss die Metainformation „pattern“ mit dem Wert des Schlüsselwertes „pattern“ übereinstimmen um eine geeignete Konvertierungsmethode für BPEL bzw. Node-Red aufzurufen. Hier kommen momentan entweder *„robust“* oder *„time-critical“* als mögliche Werte in Frage.

6 Related Work

In diesem Abschnitt werden Ansätze präsentiert und beschrieben, die eine vergleichbare Funktionalität anbieten. Dabei wird insbesondere der Verzeichnis-Dienst Fragmento genaür beschrieben.

Fragmento

Fragmento ist ein Open-Source-Verzeichnis-Dienst, dessen Fokus auf dem Konzept von Prozess-Fragmenten im Bereich von Compliance Management in prozess-basierten Anwendungen liegt. Ziel ist die Verwaltung von Prozessen und Prozess-Fragmenten im Bereich der Compliance. Dabei werden Compliance Controls als wiederverwendbare (reusable) Prozess-Fragmente betrachtet. Ein Prozessfragment beschreibt [SKLS11] wie folgt :

We understand a process fragment as a connected, possibly incomplete process graph which may also contain additional artifacts like the fragment context. A process fragment is not necessarily directly executable as some parts may be explicitly stated as opaqü in order to mark points of variability.

Eine weitere detaillierte Beschreibung des Begriffs Prozess-Fragment ist in [SLM$^+$10] gegeben:

textitA process fragment is defined as a connected graph with significantly relaxed completeness and consistency criteria compared to an executable process graph

Dementsprechend besteht ein Prozess-Fragment aus Aktivitäten, Platzhaltern, den sogennanten *Regions*- und *Kontroll*-Kanten, welche Abhängigkeiten zwischen diesen beschreiben. Ferner kann ein Prozess-Fragment einen Kontext (z.B. Variablen) definieren und einen Prozess-Start- und einen Prozess-End-Knoten. Darüber hinaus kann es über mehrere eingehende und ausgehende Kanten verfügen und besteht aus mindestens einer Aktivität. Zudem sollte es möglich sein es zu einem ausführbaren Prozess-Graph überführen zu können, da ein Prozess-Fragment nicht unbedingt direkt ausführbar ist und teilweise undefiniert sein kann.

Fragmento ist demnach ein Verzeichnis für Prozess-Artefakte, welche die Speicherung und das Auffinden, sowie die Verwaltung von Versionen von allen Artefakten, die mit Prozessen verbunden sind, ermöglicht. Es unterstützt die CRUD-Operationen: Create, Read, Update und Delete.

Das Verzeichnis vergibt eindeutige ID-Nummern für jedes Artefakt, dass darin abgelegt wird. Mithilfe dieser eindeutigen Nummern können Relationen zwischen den Artefakten erzeugt werden. Ferner ermöglicht Fragmento die Verwaltung dieser Relationen und die Annotation von Prozess-Fragmenten für bestimmte Prozesse, um damit Bedingungen beschreiben zu können. Da Artefakte in das XML-Format serialisiert werden können, erlaubt Fragmento zudem die Speicherung von XML-Artefakten. Wie auch in dem Ansatz für ein Fragmente-Repository, welches in dieser Diplomarbeit beschrieben wird, verwendet Fragmento Metadaten, um die Artefakte beschreiben zu können.

Ein Artefakt in Fragmento setzt sich aus folgenden Komponenten zusammen:

- eine eindeutige ID-Nummer
- Metadaten (Name, Beschreibung, Keywords etc.)
- XML-Dokument
- Typ (Fragment, WSDL etc.)
- Relation zu anderen Artefakten

Die erfolgreiche Suche nach bestimmten Artifakten in Fragmento kann unterschiedlich durchgeführt werden. So kann kann beispielweise nach übereinstimmungen (Matches) in der Beschreibung oder im Inhalt oder mithilfe der Angabe eines Intervalls nach der dem Erstellungsdatum gesucht werden. Ferner ist es m?glich nach einem bestimmten Artefakt-Typ auszurichten oder die Suche auf jene Artefakte zu reduzieren, welche in Relation zu einem bestimmten Artefakt stehen.

Eine weitere wichtige Funktionalität, die Fragmento anbietet, ist die Verwaltung der im Verzeichnis abgelegten Artefakte, sowie der Relationen zwischen den Artefakten. Neü Artefakte können erstellt und im Verzeichnis gespeichert werden. Dabei können mehrere Versionen eines Artefakts vorhanden sein, nach welchen anhand der Versions-Historie gesucht werden kann. Dementsprechend ist es m?glich die neüste Version eines Artefakts oder auch eine beliebige Version zu benutzen. Ferner können Relationen verwaltet werden, indem diese gelöscht, upgedatet oder neu erstellt werden. Relationen beschreiben, welche Artefakte züinander geh?ren und ermöglichen es Annotationen eines Artefakts an andere Artefakte, beispielsweise die Annotation eines Prozess-Fragments an ein Prozess.

Fragmento wird als ein Web Service angeboten, wobei die zur Verfügung gestellten Operationen über ein SOAP/HTTP Binding bereitgestellt werden. Die Operationen, welche zum Auffinden von Artefakten verwendet werden, werden als REST-Funktionen (z.B. HTTP/GET) realisiert. Dadurch wird die Integration von anderen Tools erleichtert.

Erstellung eines Artefakts

Wird ein neues Artefakt erstellt, erzeugt das Verzeichnis ein neues textitversioniertes Objekt. Diese Objekt enthält alle Versionen des Artefakts und stellt eine Versions Historie bereit. Damit kann man auf die erste Version eines Artefakts, die sogenannte textitroot version sowie auf die aktuelle Version, die textitbase version zugreifen. Das Version Descriptor Object ist ein weiteres Objekt, welches der internen Repräsentation der Version eines Artefakts entspricht. Dieser enthält neben dem Erstellungsdatum Metadaten und einen Verweis auf das XML-Dokument des Artefakts. Es ist möglich Relationen zwischen diesen Version Descriptor Objekten zu erzeugen.

Erweiterbarkeit

Fragmento kann durch zahlreiche Funktionalitäten erweitert werden, welche hilfreich sind in Bezug auf die Verwaltung von Prozessen, Prozess-Fragmenten und damit verbundenen Artefakten. Dazu gehören Validators, Custom Query Functions und View Transformations.

Design und Implementierung

Fragmento wurde mit der Programmiersprache Java geschrieben. Das Backend von Fragmento basiert auf einem Technologie-Stack. Die Repository-Anwendung läuft auf einem Tomcat Server

und Hibernate dient als Data Abstraction Layer. Ferner dient das Spring Framework für das Object Lifecycle Management sowie PostgreSQL als Datenbank für die Speicherung der Daten. Für die Erstellung der Web Service Schnittstelle kommen Axis 2 Libraries zum Einsatz. Der Web Client wurde mithilfe von Java Server Pages (JSP) erstellt, Tag Libraries für die View und Servlets für die Bearbeitung von Client-Anfragen verwendet.

Neben Fragmento gibt viele weiter kommerzielle und nicht-kommerzielle Verzeichnisdienste, die den Ansatz des Software Reuse verfolgen. Im Folgenden werden einige kommerzielle Beispiele genannt und beschrieben.

+1Reuse Repository

Das +1Reuse System wurde von +1 Software Engineering Co. in Kalifornien entwickelt und läuft auf der Sun Workstation Plattform. Es unterstützt Reuse Repositories, die vom Benutzer erstellt und verwaltet werden und das sogenannte Selectives Reuse. Selective Reuse verbessert in hohem Ma?e die Fähigkeit des Benutzers Quellcode und Dokumentationen von früheren Projekten mit beliebiger Granularituet wiederzuverwenden. Folglich stellt jedes vorangegangene Projekt eine Reuse Library dar, deren Submodelle wiederverwendet werden können. Es können Design, Dokumentation, Qüllcode, Header Files, Test Cases etc. wiederverwendet werden. +1Reuse erlaubt 3 Formen der Wiederverwendung:

- User-Defined Reuse Library
- Filtered Reuse Library und
- Selective Reuse

Da durch den Reuse von existierendem Code und Dokumentationen die Produktivität des Programmierers erhöht werden kann, ermöglicht *+1Reuse* jeglichen Quellcode, Dokumentationen, Header- und Testdateien wiederzuverwenden, indem der Ansatz von Submodellen unterstützt wird. Nachdem ein Submodell ausgesucht wurde, werden das Submodell und die damit verbundenen Dateien in das neu Projekt kopiert und ermöglichen dadurch die Lösung von aufkommenden Problemfällen.

Software Asset Library Management System

SALMS ist ein System zur Klassifizierung, Beschreibung und Auffindung von wiederverwendbaren Assets[Mor98]. Der Reuse von Software Assets in allen Phasen des Software Engineering Life Cycles führt neben der Steigerung der Produktivität auch zu Qualitätsverbesserungen. SALMS bietet eine zentrale Repository an, die Mechanismen zur Klassifizierung und Speicherung von Software Assets als auch Techniken für das Auffinden von wiederverwendbaren Assets bereitstellt. Mithilfe von SALMS schließt sich die Lücke zwischen der Entwicklung von wiederverwendbaren Komponenten und der Erstellung von einer Software durch den Einsatz von Reusable Software. Des Weiteren werden Features für die Anforderungsverwaltungs-Aktivität und für die Erstellung und Verwaltung einer technischen Bibiliothek des Unternehmens bereitgestellt. SALMS kann über PC oder UNIX Workstations im Unternehmensnetzwerk verteilt werden und ist somit für jeden Entwickler zugänglich. Das User Interface basiert auf der WEB Technologie. Ein Asset ist hierbei eine Sammlung von Artefakten, die während des Life-Cycles erstellt werden. Demnach verkörpern Anforderungen, Architekturmodelle, Design Spezifikationen, Source Code oder Test Skripte ein Asset.

Automated Software Reuse Repository

ASRR ist ein Verzeichnis, welches aus zwei Hauptkomponenten aufgebaut ist: dem Administrations-Tool und der Reuse Repository. Das Administrations-Tool übernimmt administrative Funktionen, wie das Hinzufügen, Löschen oder Modifizieren von Benutzern und ihren Attributen. Attribute beschreiben den Sicherheitsgrad, Gruppen- und Sicherheitspermissionen für das Hinzufügen, Bearbeiten und Löschen von Modulen. Die Reuse Repository ermöglicht dem Benutzer Module in das Verzeichnis hochzuladen und sie in einer Repository abzulegen, die nach bestimmten Komponenten abgesucht werden kann. Zusätzliche Funktionen verwalten die Login-Prozedur des ASRR (*Program Control*), begrenzen die Funktionsmöglichkeiten des Benutzers Module zu bearbeiten, löschen, hinzufügen, hochladen, betrachten oder runterzuladen (Program Control) oder erhöhen die Sicherheit, indem inaktive Benutzer nach einer bestimmten Zeitperiode automatisch ausgeloggt werden. Zudem gewährt ASRR dem Benutzer einfachen Zugriff auf das Verzeichnis, für die Suche nach Reuse Komponenten. Die Suche ist flexibel, da der Benutzer nach Wortketten (Strings) suchen und dabei die Wortbegriffe *not*, *or oder* and verwenden kann. Spezifische Informationen über das Reuse Modul beinhalten Details über die Plattform, die genutzt wird, Erleichterungen für die Wiederverwendung und zusätzliche Informationen werden dem Benutzer zur Verfügung gestellt.

AIRS

AIRS ist eine AI-basierende Library-System für Software Reuse. Es wurde von E.J. Ostertag, J.A. Hender, C.Braun und R. Prieto-Diaz entwickelt und ermöglicht Benutzern eine Software.Verzeichnis nach Komponenten zu durchsuchen, die vom Benutzer bestimmten Anforderungen erfüllen müssen [GR91]. Eine Komponente wird hierbei als ein (Feature, Term)- Paar beschrieben. Das Feature stellt ein Klassifizierungs-Kriterium dar und wird definiert anhand von damit verbundenen Begriffen (Terms)[EMD94]. Komponenten können widerum zu sogenannten Packages zusammengefasst werden. Diese sind logische Einheiten, die aus eine Menge von ähnlichen Komponenten bestehen. Packages werden ebenfalls, wie auch die Komponenten selbst, anhand von Features beschrieben. Sie enthalten jedoch, im Gegensatz zur Beschreibung einer Komponente, eine Menge von Mitglieds-Komponenten. Ausgehend von einer Zielbeschreibung werden die Reuse Komponenten und Packages, deren Beschreibung einen hohen Ähnlichkeitsgrad zur Zielbeschreibung aufweisen, als Kandidaten aus dem Verzeichnis ausgewählt [JC94].

Der Ähnlichkeitsgrad wird bestimmt mittels einer Kennzahl, die sogannte Distance, welches den erwartenden Aufwand zur Erreichung des angestrebten Zieles, bei einer gegebenen Komponente als Kandidat, darstellt. Der Distance-Wert wird berechnet durch bestimmte Funktionen, den Comparators. Beispiele hierfür sind die SubSumption-, Closeness- und Package Comparators-Funktionen. Die AIRS Klassifizierungsansatz basiert auf der Formalisierung von Konzepten und ähnelt dem Ansatz der Faceted Classification [PD91]. Die Implementierung vom Prototyps des AIRS Systems wurde auf zwei unterschiedliche Software Libraries angewandt, um die Funktionalität des Systems darzustellen. Es wurden eine Menge von Ada Packages für die Datenstrukturmanipulation und eine Menge von Komponenten der Programmiersprache *C* für dein Einsatz in Command, Control und Informationssystemen verwendet.

Vergleich

Neben kommerziellen *Reusable Component Repositories* kommen auch sogenannte*Government Repositories* zum Einsatz. Kommerzielle Ansätze werden in Case Environments integriert. Einige größere

Repositories benutzen web-basierte Techologien für die Bereitstellung von Diensten. Sie nutzen flache Dateien, welche in HyperTextMarkup Language (HTML) geschrieben sind. *Electronic Library Services* and *Applications* (ELSA) erweitern den Ansatz, indem sie Multimedia Oriented Repository Environment (MORE) nutzen.

7 Zusammenfassung und Ausblick

7.1 Zusammenfassung

In Anbetracht der heutzutage großen Datenmengen, die in Unternehmen oder im wissenschaftlichen Bereich enstehen können, kommen immer häufiger Mashup-Applikationen wie Intel Mashmaker, Yahoo Pipes oder IBM Mashup Hub zum Einsatz, welche die Erstellung, Bearbeitung und Visualisierung von Daten aus diesen heterogenen Datenmengen erleichtern. Da diese Datenmengen stetig ansteigen und oftmals verteilt sind (Big Data), ist es erforderlich (semi-) strukturierte und unstrukturierte Daten weitgehend automatisiert und generisch zusammenzuführen und zu analysieren. Der Prozess der ad-hoc Zusammenfuehrung multipler Datenquellen ist auch unter den Namen Data Mashup, usiness Mashup oder nterprise Mashup bekannt. Es existieren zahlreiche Tools, welche die Erstellung solcher Mashups ermögichen. Diese weisen jedoch Nachteile in Bezug auf das technische Wissen des Nutzerkreises und der Flexiblität auf. So erfordern derartige Tools hohe technische Anforderungen und können dadurch nicht von jedem beliebigem Benutzer verwendet werden. Lediglich solche Nutzer, die das nötige technische Wissen mitbringen, sind in der Lage diese Tools zu nutzen. Ein weiterer Nachteil ist die eingeschränkte Flexibilität, die bestehende Lösungsansätze aufweisen, da diese nur eine Form der Ausführung unterstützen. Folglich können auf unterschiedliche Anforderungen von Benutzerkreisen nicht direkt eingegangen werden und dadurch Ergebnisse erzielt werden, welche nicht dem erwünschten Resultat entsprechen. Während für eine bestimmte Benutzergruppe beispielsweise die robuste Ausführung von Datenverarbeitungsprozessen im Vordergrund steht, setzt im Gegensatz dazu ein anderer Benutzerkreis die zeitlich effiziente Ausführung voraus.

An der Universität Stuttgart wurde das Data Mashup Tool FlexMash entwickelt, mit dem Ziel die Nachteile und Einschränkungen von Data Mashup-Ansätzen zu beseitigen. FlexMash ermöglicht Domänenexperten die Modellierung von Data Mashups, wobei die Ausführung an die unterschiedlichen Anforderungen der Benutzer angepasst wird. Dadurch wird einerseits eine höhere Flexibilität bei der Ausführung gewährleistet und andererseits die Benutzerkreis vergrößert, da Domänenexperten, ohne technisches Wissen, in der Lage sind mit FlexMash Data Mashups zu erstellen. Dazu muss der Benutzer lediglich ein sogenannten Mashup Plan erstellen. Dieser Mashup Plan entspricht einem abstraktem Modell eines Data Mashups und setzt sich aus Datenquellen, sogenannten Data Source Descriptions (DSDs) und Datenoperationen (DPDs) zusammen. Der Domänenexperte modelliert solch ein Mashup Plan, indem er DSDs und DPDs miteinander verknüpft. Der erstellte Mashup Plan ist nicht ausführbar und möglichst abstrakt gehalten. Der Mashup Plan kann anschließend, abhängig von den unterschiedlichen Nutzeranforderungen in unterschiedliche ausführbare Formate überführt werden. Dabei ist zu beachten, dass die Bereitstellung von konkreten Implementierungen in FlexMash momentan lediglich hardcodiert ist. Diese sollte automatisiert und das Mapping-Problem gelöst werden.

In dieser Arbeit wurde ein Konzept für ein Verzeichnisdienst erstellt, welches die automatisierte Bereitstellung von Implementierungen der einzelnen Komponenten des erstellten Mashup Plans (DSDs und DPDs) ermöglicht. Dazu wurde zunächst ein Konzept für das Verzeichnisdienst erstellt, die beschreibt wie der Dienst aufgebaut ist und welche Funktionen zur Verfügung gestellt werden.Des Weiteren wurde ein Konzept entwickelt, die das Mapping-Problem, basierend auf der regelbasierten Transformation, löst.

In 3.2 wurde der Aufbau des Verzeichnisdienstes mit den einzelnen Komponenten und ihren Relationen zueinander beschrieben. Dabei wurden die drei Ebenen des Verzeichnisdienstes näher betrachtet, ihre Funktion in der Gesamtstruktur und die Interaktion dieser Ebenen zueinander erläutert. Zudem wurde geklärt, wie ein Benutzer mit dem Verzeichnis-Dienst kommunizieren, d.h. Anfragen schicken kann und in welcher Form die Ergebnisse ausgegeben werden.

In 3.3 wurden anschließend die unterschiedlichen Technologien vorgestellt, die verwendet werden, um das Verzeichnis-Dienst zu realisieren. Dabei wurden auf die charakteristischen Merkmale dieser Technologien eingegangen. Des Weiteren wurde die Architektur der NoSQL-Datenbank, welche die konkreten Implementierungen für die DPDs und DSDs in einem Mashup Plan enthalten und bei Anfrage zur Verfügung stellen, näher betrachtet.

Im nachfolgenden Kapitel 4 wurden Beispiele für Patterns vorgestellt. Dabei wurden Basispattern beschrieben, welche nicht komplex aufgebaut sind und als Komponenten größerer und komplexerer Patterns zum Einsatz kommen können. Neben dem Aufbau dieser Patterns wurde auch erwähnt, wie diese Funktionieren, aus welchen Komponenten sie bestehen. Ferner wurde erläutert, welche Parameter eingegeben werden müssen.

Kapitel 5 beschäftigt sich mit der technischen Umsetzung des zuvor erstellten Konzepts. Zunächst wurde in 5.2 beschrieben, wie die Daten in der Datenebene verwaltet werden. Dabei wurde wurde das Datenmodell und die Organisation der Daten in der Datenzugriffsebene beschrieben. Anschließend wurden die implementierten Funktionen und die für diese Funktionen verwendeten Datenformate im Detail beschrieben. Zudem wurde erwähnt, wie der Repository-Dienst technisch mithilfe des Spring Rahmenwerks realisiert. Im darauf folgenden Abschnitt wurden die von dem Repository-Dienst angebotenen Funktionen vorgestellt. Im nächsten Abschnitt wurde die momentane Transformation von Mashup-Flows des FlexMash-Projekts beschrieben. Dabei wird ein JSON-Flow in ein ausführbares Format transformiert. Nach einer Bestandsaufnahme, welche die Umsetzung der Transformation im FlexMash-Projekt beschreibt, wurde die Transformation für das Konzept des Repository-Dienstes für diese Arbeit vorgestellt. Im Kapitel 5 wurde anschließend die technische Realisierung der Architektur des Dienstes beschrieben. Dabei wurde beschrieben, auf welchen Technologien der Repository-Dienst basiert und das Datenformat für die Code-Fragmente detailliert beschrieben. Ferner wurde die Hauptfunktion des Dienstes, die Transformation eines unausführbaren Mashup Plan in ein ausführbares Format besprochen. Im letzten Kapitel 6 wurden schließlich ähnliche Konzepte von Code-Fragmente-Diensten vorgestellt und näher betrachtet.

7.2 Ausblick

Für diese Arbeit wurde ein Konzept für ein Code-Fragmente-Verzeichnis erstellt. Dabei wurden Grundfunktionen implementiert, welche das Verzeichnis einem Benutzer zur Verfügung stellen soll. In zukünftigen Arbeiten könnten diesbezüglich weitere komplexere Funktionen erstellt werden.

Ferner wurde für die Speicherung der Code-Fragmente eine NoSQL-Datenbank MongoDB verwendet. Da die Zahl der darin enthaltenen Code-Fragmente im Einsatz relativ hoch sein kann, ist es erforderlich diese Code-Fragmente zu kategorisieren. Diese Kategorisierung bzw. Klassifizierung von Fragmenten würde die Suche nach geeigneten Komponenten erheblich verkürzen. Statt alle vorhandenen Code-Fragmente abzusuchen, könnte man bei Angabe einer bestimmten Kategorie des Code-Fragments direkt in dieser die Suche starten. Diese Klassifizierung könnte in zukünftigen Arbeiten thematisiert werden.

Des Weiteren wäre eine grafische Benutzeroberfläche, ähnlich der GUI von Fragmento denkbar, da momentan auf die Dienste des Repository über einen REST-Client(Postman) oder über die Adress-Zeile des Web-Browsers zugegriffen wird.

Zudem kann es vorkommen, dass es für einen Knoten im Mashup Plan mehrere ausführbare Codefragmente gibt, welche alle gegeben Anforderungen erfüllen. In diesem Fall könnten zukünftige Arbeiten versuchen eine Metrik zu erstellen, anhand welcher es möglich ist, alle in Frage kommenden Code-Fragmente miteinander zu vergleichen. Diese Metrik, welches sich aus unterschiedlichen Kriterien zusammensetzt, würde anschließend alle Code-Fragmente, die sich als Implementierung für einen Knoten im Mashup Plan eignen, in Form einer Ranking-Liste ausgeben. Somit könnte das Code-Fragment ausgewählt werden, welches das höchste Ranking aufweist und damit die meisten Anforderungen erfüllt.

Der Verzeichnisdienst sollte außerdem in der Lage sein, stets eine Lösung bereitzustellen. Werden alle erforderlichen Parameter beispielsweise nicht vollständig eingegeben bzw. fehlerhaft eingegeben, wird dennoch das Code-Fragment ausgesucht, dass die höchste Priorisierung hat. Dadurch ist gewährleistet, dass das Verzeichnis sogar dann Lösungen zur Verfügung stellt, wenn die Kriterien nicht alle erfüllt sind. Diese Thematik könnte in weiteren Arbeiten erörtert werden.

Die Versionverwaltung von bestehenden Code-Fragmenten ist ein weiteres Feld, welches sich für zukünftige Arbeiten eignet. Ähnlich der Versionsverwaltung von *Fragmento*, sollte das Verzeichnis eine Versions-Historie aufweisen. Diese enthält, angefangen bei der ersten Version, alle bisherigen Versionen einschließlich der aktuellen Version. Sobald eine Version aktualisiert wird, wird dieser automatisch die um die Zahl eins erhöhte Versionnummer zugewiesen. Dadurch wird gewährleistet, dass Benutzer auch in der Lage sind, auf die Codefragmente zuzugreifen, welche nicht aktuell sind, jedoch ihren Anforderungen genügen.

Eine weitere zukünftige Arbeit könnte die Implementierung ein Authorisierungsprozesses sein, welcher die Benutzer in ihrer Interaktion mit dem Verzeichnis kontrolliert. Dadurch würde beispielsweise festgelegt werden, welcher Benutzer die Erlaubnis erhält, neue Code-Fragmente in das Verzeichnis zu registrieren, bereits bestehende Code-Fragmente zu aktualisieren oder zu löschen.

Literaturverzeichnis

[ABFG04] D. Austin, A. Barbir, C. Ferris, S. Garg. Web services architecture requirements. *W3C Working Group Notes*, S. 22, 2004. (Zitiert auf Seite 30)

[Ale77] C. Alexander. *A pattern language: towns, buildings, construction.* Oxford University Press, 1977. (Zitiert auf Seite 16)

[AZ05] P. Avgeriou, U. Zdun. Architectural patterns revisited–a pattern. 2005. (Zitiert auf den Seiten 5, 33 und 34)

[Bea09] A. Beaulieu. *Einführung in SQL*. O'Reilly Germany, 2009. (Zitiert auf Seite 14)

[Bet01] U. Bettag. Web-services. *Informatik-Spektrum*, 24(5):302–304, 2001. (Zitiert auf Seite 30)

[BSKM12] A. Bawiskar, P. Sawant, V. Kankate, B. Meshram. Spring Framework: A Companion to JavaEE. *International Journal of Computational Engineering and Management IJCEM*, 1(15):41–49, 2012. (Zitiert auf Seite 51)

[DLHPB09] G. Di Lorenzo, H. Hacid, H.-y. Paik, B. Benatallah. Data integration in mashups. *ACM Sigmod Record*, 38(1):59–66, 2009. (Zitiert auf Seite 36)

[DVXB+09] P. De Vrieze, L. Xu, A. Bouguettaya, J. Yang, J. Chen. Process-oriented enterprise mashups. In *Grid and Pervasive Computing Conference, 2009. GPC'09. Workshops at the*, S. 64–71. IEEE, 2009. (Zitiert auf Seite 35)

[EMD94] D. Eichmann, T. McGregor, D. Danley. Integrating structured databases into the web: The MORE system. *Computer Networks and ISDN Systems*, 27(2):281–288, 1994. (Zitiert auf Seite 88)

[FBB+14] M. Falkenthal, J. Barzen, U. Breitenbücher, C. Fehling, F. Leymann. From pattern languages to solution implementations. In *Proceedings of the Sixth International Conferences on Pervasive Patterns and Applications (PATTERNS 2014), Venice, Italy*. 2014. (Zitiert auf Seite 16)

[FJZ+12] M. Falkenthal, D. Jugel, A. Zimmermann, R. Reiners, W. Reimann, M. Pretz. Maturity Assessments of Service-oriented Enterprise Architectures with Iterative Pattern Refinement. In *GI-Jahrestagung*, S. 1095–1101. Citeseer, 2012. (Zitiert auf Seite 16)

[GR91] A. Guillermo, P.-D. Ruben. Domain analysis concepts and research directions, 1991. (Zitiert auf Seite 88)

[HHLD11] J. Han, E. Haihong, G. Le, J. Du. Survey on NoSQL database. In *Pervasive computing and applications (ICPCA), 2011 6th international conference on*, S. 363–366. IEEE, 2011. (Zitiert auf Seite 56)

[HM16] P. Hirmer, B. Mitschang. FlexMash–Flexible Data Mashups Based on Pattern-Based Model Transformation. In *Rapid Mashup Development Tools*, S. 12–30. Springer, 2016. (Zitiert auf den Seiten 5, 17, 18 und 38)

[HR83] T. Haerder, A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys (CSUR)*, 15(4):287–317, 1983. (Zitiert auf Seite 15)

[HRWM15] P. Hirmer, P. Reimann, M. Wieland, B. Mitschang. Extended Techniques for Flexible Modeling and Execution of Data Mashups. In *DATA*, S. 111–122. 2015. (Zitiert auf den Seiten 5, 36, 38, 39, 40 und 43)

[HSSJS08] V. Hoyer, K. Stanoesvka-Slabeva, T. Janner, C. Schroth. Enterprise mashups: Design principles towards the long tail of user needs. In *Services Computing, 2008. SCC'08. IEEE International Conference on*, Band 2, S. 601–602. IEEE, 2008. (Zitiert auf Seite 35)

[JC94] J.-J. Jeng, B. H. Cheng. A formal approach to reusing more general components. In *Knowledge-Based Software Engineering Conference, 1994. Proceedings., Ninth*, S. 90–97. IEEE, 1994. (Zitiert auf Seite 88)

[JEA+07] D. Jordan, J. Evdemon, A. Alves, A. Arkin, S. Askary, C. Barreto, B. Bloch, F. Curbera, M. Ford, Y. Goland, et al. Web services business process execution language version 2.0. *OASIS standard*, 11(120):5, 2007. (Zitiert auf Seite 24)

[JHD+04] R. Johnson, J. Hoeller, K. Donald, Sampaleanu, et al. The Spring Framework–Reference Documentation. *Interface*, 21, 2004. (Zitiert auf den Seiten 5 und 51)

[Lan03] T. Langner. Web Services mit Java. *Markt+ Technik*, 2003. (Zitiert auf Seite 30)

[LR00] F. Leymann, D. Roller. Production workflow: concepts and techniques. 2000. (Zitiert auf den Seiten 5, 20, 21 und 59)

[Mel10] I. Melzer. *Service-orientierte Architekturen mit Web Services: Konzepte-Standards-Praxis*. Springer-Verlag, 2010. (Zitiert auf Seite 27)

[MLM+06] C. M. MacKenzie, K. Laskey, F. McCabe, P. F. Brown, R. Metz, B. A. Hamilton. Reference model for service oriented architecture 1.0. *OASIS standard*, 12, 2006. (Zitiert auf Seite 27)

[Mon12] I. MongoDB. MongoDB, 2012. (Zitiert auf Seite 55)

[Mor98] E. Morandin. SALMS v5. 1: A System for Classifying, Describing, and Querying about Reusable Software Assets. In *The Proceedings of 5th International Conference on Software Reuse (ICSR'98)*. 1998. (Zitiert auf Seite 87)

[Mül06] J. Müller. *Workflow-based integration: grundlagen, technologien, management*. Springer-Verlag, 2006. (Zitiert auf den Seiten 5, 20, 22 und 23)

[NPP13] A. Nayak, A. Poriya, D. Poojary. Type of NOSQL databases and its comparison with relational databases. *International Journal of Applied Information Systems*, 5(4):16–19, 2013. (Zitiert auf Seite 55)

[PD91] R. Prieto-Diaz. Implementing faceted classification for software reuse. *Communications of the ACM*, 34(5):88–97, 1991. (Zitiert auf Seite 88)

[Rei12] R. Reiners. A Pattern Evolution Process-From Ideas to Patterns. In *Informatiktage*, S. 115–118. 2012. (Zitiert auf Seite 16)

[RG00] R. Ramakrishnan, J. Gehrke. *Database management systems*. McGraw-Hill, 2000. (Zitiert auf den Seiten 11 und 15)

[RHJN04] L. Rapanotti, J. G. Hall, M. Jackson, B. Nuseibeh. Architecture-driven problem decomposition. In *Requirements Engineering Conference, 2004. Proceedings. 12th IEEE International*, S. 80–89. IEEE, 2004. (Zitiert auf den Seiten 5 und 33)

[RSM11] P. Reimann, H. Schwarz, B. Mitschang. Design, implementation, and evaluation of a tight integration of database and workflow engines. *Journal of Information and Data Management*, 2(3):353, 2011. (Zitiert auf den Seiten 5, 24 und 25)

[RSM14] P. Reimann, H. Schwarz, B. Mitschang. A pattern approach to conquer the data complexity in simulation workflow design. In *OTM Confederated International ConferencesÖn the Move to Meaningful Internet Systems"*, S. 21–38. Springer, 2014. (Zitiert auf den Seiten 40, 42, 57 und 58)

[SKLS11] D. Schumm, D. Karastoyanova, F. Leymann, S. Strauch. Fragmento: advanced process fragment library. In *Information Systems Development*, S. 659–670. Springer, 2011. (Zitiert auf Seite 85)

[SLM+10] D. Schumm, F. Leymann, Z. Ma, T. Scheibler, S. Strauch. Integrating compliance into business processes. *Multikonferenz Wirtschaftsinformatik 2010*, S. 421, 2010. (Zitiert auf Seite 85)

[SN96] R. W. Schulte, Y. V. Natis. Service oriented architectures, part 1. *Gartner, SSA Research Note SPA-401-068*, 1996. (Zitiert auf Seite 27)

[Wag11] F. Wagner. Nutzung einer integrierten Datenbank zur effizienten Ausführung von Workflows. In *BTW Workshops*, S. 145–149. 2011. (Zitiert auf Seite 24)

[Wal12] C. Walls. *Spring im Einsatz*. Carl Hanser Verlag GmbH Co KG, 2012. (Zitiert auf Seite 50)

Alle URLs wurden zuletzt am 15. 01. 2015 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift