Bachelorarbeit

Entwicklung einer Datenbanklesekomponente für Datenberechnungsnetzwerke im Kontext des Flow-Based Programming Frameworks

zur Erlangung des akademischen Grades

Bachelor of Science Informatik

vorgelegt dem
Fachbereich Mathematik, Naturwissenschaften und
Informatik der
Fachhochschule Gießen-Friedberg

Michael Eckel
17. August 2009

Referent: Herr Prof. Dr. Thomas Karl Letschert
Korreferent: Herr M. Sc. Sebastian Süß

Danksagung

An dieser Stelle bedanke ich mich bei Herrn Prof. Dr. Thomas Karl Letschert und bei Herrn M. Sc. Sebastian Süß, die mir die Teilnahme am Projekt PARASUITE ermöglicht und mich während meiner Bachelorarbeit und der vorangegangenen Projektphase betreut haben.

Weiterer Dank geht an das Cognidata-Team aus Marburg, Dipl.-Inf. Christian Seidemann und Dipl.-Inf. Helmut Neubauer, die mir bei Problemen stets geholfen haben.

Großer Dank gilt auch meinem Kommilitonen Volker Steiß, meinem Bruder Stefan Eckel und meinem Vater Horst Eckel fürs Korrekturlesen dieser Bachelorarbeit.

Bei Sven Steinseifer bedanke ich mich für das Bereitstellen seiner Masterarbeit. Philipp Hoffmann und Eugen Labun danke ich für das Bereitstellen ihrer Bachelorarbeiten.

Versicherung der Selbstständigkeit

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Bürgeln, 17. August 2009

Michael Eckel

Inhaltsverzeichnis

Kapitel 1

Einführung

In diesem Kapitel wird das Projekt vorgestellt, in dessen Kontext diese Bachelorarbeit entstand. Dazu wird zunächst auf den Sinn und Zweck des Projektes eingegangen, bevor die Funktionsweise genauer erklärt wird. Danach werden kurz die eingesetzten Technologien beschrieben. Anschließend wird der Grund des Entstehens dieser Bachelorarbeit erörtert und die Ziele dieser Arbeit werden festgelegt. Am Ende des Kapitel wird der Aufbau dieser Arbeit geschildert.

1.1 Das Projekt PARASUITE

PARASUITE[1] steht für *Product Analysis and Reporting Application SUITE* und ist ein durch Drittmittel finanziertes Forschungsprojekt der Cognidata GmbH[2], in das auch die Philipps-Universität Marburg[3] und die Fachhochschule Gießen-Friedberg[4] involviert sind. PARASUITE ist ein entscheidungsunterstützendes System (engl.: *Decision Support System*, kurz *DSS*) im Bereich Produktlebenszyklus. Kunden werden dabei in den folgenden Bereichen des Produktlebenszyklus unterstützt:

Beginning-of-Life: Produktion und Markteinführung eines Produktes

Middle-of-Life: Produktwartung, Instandhaltung

End-of-Life: Entsorgung des Produkts

[1] Siehe `http://www.parasuite.eu/`
[2] Siehe `http://www.cognidata.de/`
[3] Siehe `http://www.uni-marburg.de/`
[4] Siehe `http://www.fh-giessen-friedberg.de/`

1.1.1 Kunden

Wie unterschiedlich die Unterstützung durch PARASUITE beim Kunden aussehen kann, wird in den folgenden Abschnitten anhand der beiden Kunden *Bombardier Transportation*[5] (im Folgenden auch nur *Bombardier* genannt) und *ThyssenKrupp Aufzüge*[6] (im Folgenden auch nur *Thyssen* genannt) vorgestellt.

Der Kunde *Bombardier Transportation*

Der erste Kunde von PARASUITE war der Lokomotivenhersteller Bombardier Transportation, welcher durch PARASUITE im Bereich *Beginning-of-Life* des Produktlebenszyklus unterstützt wird. Bombardier ist daran interessiert, bessere und fehlerrobustere Produkte herzustellen. PARASUITE wird eingesetzt, um aus riesigen, über Jahre gesammelten Mengen an Daten, Erkenntnisse über fehlerhafte und häufig ausfallende Bauteile von Lokomotiven zu ziehen. Bei den auszuwertenden Daten handelt es sich überwiegend um Fehler- und Ausfallmeldungen. Bei jedem Ausfall eines Bauteils bzw. bei einem Fehler in einem Bauteil wird ein Schnappschuss (engl.: *Snapshot*) der Zustände der einzelnen Komponenten erstellt und gespeichert. Um dies zu ermöglichen, sind jede Menge Sensoren in den Lokomotiven verbaut, die den Zustand verschiedener Komponenten überwachen und im Fall eines Fehlers die aktuellen Zustandsdaten liefern.

Eine einzige Lokomotive produziert pro Jahr ca. 500 Millionen solcher Fehlerdaten-Schnappschüsse. Nicht alle davon sind gravierende Ausfälle von wichtigen Teilen; auch Fehler und Ausfälle von kleineren Komponenten, wie Displays oder Kontrollleuchten, gehören zu diesen Daten. Bombardier möchte aus diesen Fehlerdaten Erkenntnisse darüber gewinnen, welche Teile unter welchen Umständen ausfallen und welche Ursachen einem Ausfall zu Grunde liegen. Somit werden schlecht funktionierende Bauteile ausfindig gemacht und können durch bessere, geeignetere ersetzt werden. Bei einer solchen Masse an Daten und der damit verbundenen Komplexität einer Erkenntnisgewinnung bedarf es ausgeklügelten Analysetechniken, die nur mit einem leistungsfähigen Computersystem und einer ebenso leistungsfähigen Software, die diese Analysetechniken anwendet, zu bewältigen ist. Mit PARASUITE wird versucht, genau solch eine Software zu entwickeln.

[5] Siehe `http://www.bombardier.com/en/transportation`
[6] Siehe `http://www.thyssenkrupp-aufzuege.de/`

Der Kunde *ThyssenKrupp Aufzüge*

Ein weiterer Kunde, der PARASUITE einsetzt, ist der Aufzüge-Hersteller *ThyssenKrupp Aufzüge*, welchen PARASUITE im Bereich *Middle-of-Life* des Produktlebenszyklus unterstützt. Thyssen ist daran interessiert, Wartungsarbeiten an Aufzügen, die bei Kunden verbaut sind, effizient zu planen. Dabei geht es primär um Personalverfügbarkeit und um Einsparung von Kraftstoffkosten.

Wie Bombardier auch, hat Thyssen über Jahre hinweg Daten gesammelt, in denen jeder Schadensfall mit entsprechenden Details, wie z. B. den ausgefallenen Teilen, gespeichert ist. Bevor PARASUITE eingesetzt wurde, gab es zwei Ansätze, wie ein Schadensfall abgewickelt wurde. Der erste Ansatz sah so aus, dass bei einer eingegangenen Störungs- oder Fehlermeldung ein kleines Handwerker-Team vor Ort zum Kunden geschickt wurde, um den Schaden zu beheben. Wenn zufällig ein anderer Kunde auf dem Weg lag, der auch einen Schaden zu vermelden hatte, wurde dieser mit in die Reparaturfahrt aufgenommen.

Der zweite Ansatz war schon vorausschauender. Aus dem Datenbestand wurden die Kunden ermittelt, die näherungsweise auf dem Weg zum fehlermeldenden Kunden lagen. Für diese wurde dann ermittelt, ob oder wann zuletzt häufig ausfallende Teile ausgetauscht wurden. Aufgrund des Datenbestands konnte außerdem grob ermittelt werden, wann welches verbaute Teil im Schnitt ausfällt bzw. defekt wird. Wurde diese mittlere Zeitspanne bei den entsprechenden Kunden schon überschritten oder drohte in naher Zukunft abzulaufen, wurden die entsprechenden Teile bei den Kunden vorsorglich („auf Verdacht“) ausgewechselt. So konnte es aber vorkommen, dass ein Teil, welches noch völlig in Ordnung war, ausgetauscht wurde. Deshalb ist auch diese vorausschauende Lösung nicht gut geeignet.

Durch den Einsatz von PARASUITE zur vorausschauenden Produktwartung (engl.: *Predictive Maintenance*) soll der ganze Prozess stark optimiert werden. So sollen Wartungsarbeiten effizient geplant werden können. PARASUITE soll sozusagen „vorhersehen“, wann ein Teil auszufallen droht. Dieses „Vorhersehen“ beruht natürlich auf Berechnungen auf den vom Kunden über Jahre gesammelten Daten. Die Ergebnisse sind Wahrscheinlichkeiten und lassen damit auch keine 100-prozentig sichere Vorhersage zu, sind aber dennoch sehr aussagekräftig. Durch den Einsatz von PARASUITE soll es möglich werden, ein breites Spektrum potentieller Ausfälle bereits im Voraus zu ermitteln. Folglich können Teile ausgetauscht werden, bevor es zu einem Ausfall kommt.

Reparaturarbeiten werden dadurch größtenteils planbar und mehrere Kunden können auf einer einzigen Reparaturfahrt eingeplant werden.

Ein weiterer Vorteil, der durch den Einsatz von PARASUITE entstehen soll, ist, dass Handwerkern für eine Reparaturfahrt gezielt Ersatzteile mitgegeben werden können, nämlich genau jene, welche von PARASUITE mit hoher Ausfallwahrscheinlichkeit eingestuft werden. Dies hat zur Folge, dass durch das niedrigere Beladungsgewicht des eingesetzten Fahrzeugs – im Gegensatz zu einem voll beladenen Fahrzeug mit allen Ersatzteilen – zusätzlich Kraftstoff eingespart werden kann.

1.1.2 Arbeitsweise von PARASUITE

PARASUITE ist kein Lösungssystem und auch kein entscheidungstreffendes System, sondern ein entscheidungsunterstützendes System. Deshalb müssen die Erkenntnisse, die PARASUITE herauskristallisiert hat, erst von Experten begutachtet werden, welche dann eine Entscheidung treffen können. Es ist durchaus möglich, dass bei gewonnenen Erkenntnissen nicht beeinflussbare Zusammenhänge[7] bestehen und deshalb keine Verbesserungen dort zulassen. Aus diesem Grund sind diese Erkenntnisse als Entscheidungsgrundlage anzusehen und nur durch Sachverständige bzw. Experten zu beurteilen, die sich in der Anwendungsdomäne auskennen.

Damit PARASUITE erfolgreich arbeiten kann, benötigt es vom Kunden gesammelte Daten. Diese Daten sind die Grundlage für Berechnungen und Analyseverfahren. Bei Neukunden müssen die bestehenden Daten zuerst auf das PARASUITE-System migriert werden, da sie in anderen Organisationsstrukturen vorliegen, als von PARASUITE benötigt. Diese Stufe wird in PARASUITE „Datenimport“ genannt. In Abbildung 1.1 ist die Arbeitsweise von PARASUITE als schematische Darstellung zu sehen. Beim Datenimport kann es vorkommen, dass die zu importierenden Daten von keiner guten Qualität oder sogar ungeordnet, redundant und/oder inkonsistent sind. Dann muss zusätzlich eine Datenbereinigung (engl.: *Data Cleaning*) durchgeführt werden (in Abbildung 1.1 „Aufbereitung“ genannt). Sind die Daten bereinigt, können sie in die PARASUITE-Datenbank gespeichert werden und das PARASUITE-System kann mit diesen Daten arbeiten.

[7] Solche Zusammenhänge können technischer, physikalischer, wirtschaftlicher oder sonstiger Art sein.

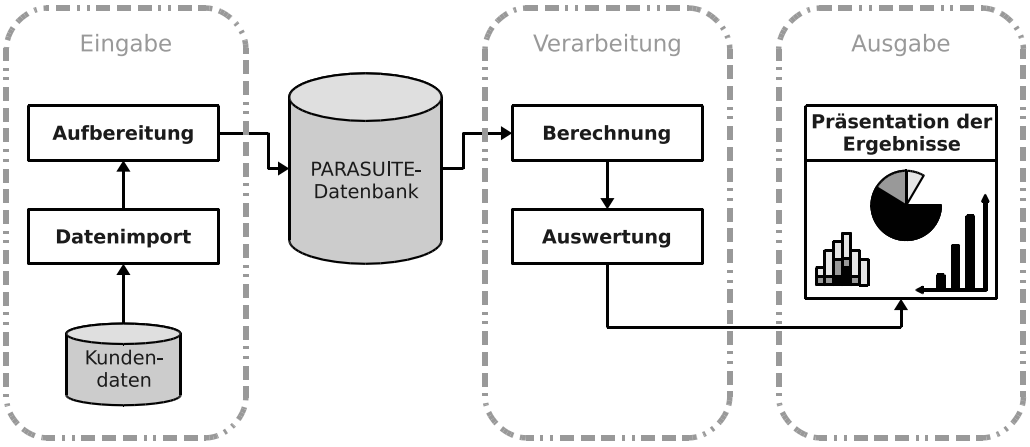

Abbildung 1.1: Schematische Darstellung der Arbeitsweise von PARASUITE

Für die Erkenntnisgewinnung aus den Daten müssen große Berechnungen auf den Daten durchgeführt werden, in PARASUITE auch Vorberechnungen genannt. Aus den Berechnungen resultiert eine wesentlich kleinere Datenmenge, welche sehr gut und schnell ausgewertet werden kann. Die Ergebnisse und damit die Erkenntnisse aus den Daten werden dem Benutzer anschließend grafisch ansprechend präsentiert.

1.1.3 Das PARASUITE-Framework

PARASUITE ist als Framework konzipiert und verwendet kundenspezifische Anpassungen. Jede Anpassung für einen Kunden verwendet als Basis das PARASUITE-Framework. Das Ganze ist nötig, da Daten der verschiedenen Kunden in der Regel semantisch sehr unterschiedlich sind und deshalb auch anders analysiert werden müssen. Zur Analyse ist es nötig, die Kundendaten zu verstehen, um so gezielt bestimmte Verfahren anzuwenden.

1.1.4 Eingesetzte Technologien

In PARASUITE wird *Java* als Programmiersprache eingesetzt. Als Plattform dient die *Java Enterprise Edition*. Die integrierte Entwicklungsumgebung der Wahl ist *Eclipse IDE for Java EE Developers*[8]. PARASUITE lässt sich

[8]Siehe `http://www.eclipse.org/downloads/moreinfo/jee.php`

grob in zwei Bestandteile zerlegen: PARASUITE-Frontend und PARASUITE-Backend. Eine grafische Benutzeroberfläche (GUI) basierend auf dem *Eclipse RCP*[9]-Framework stellt momentan das Frontend dar. Das PARASUITE-Backend läuft im Applikations-Server *JBoss*[10].

Neben diesen kommen viele weitere Hilfsprogramme zum Einsatz, welche den Software-Entwicklungsprozess unterstützen. Zu diesen zählen u. a. die Versionsverwaltung *Subversion*[11], der Style-Checker *CheckStyle*[12] und das Continuous-Integration-Tool *Hudson*[13].

1.2 Ziel dieser Bachelorarbeit

Bisher wurde in PARASUITE ein generisches Datenbankschema namens „PARASUITE1“ eingesetzt, welches in Abschnitt 2.2 näher beschrieben wird. Seit Ende des Jahres 2008 wird ein neues, generatives Datenbankschema namens „PARASUITE2“ eingesetzt, auf das in Abschnitt 2.3 genauer eingegangen wird. Der Umstieg auf das neue Datenbankschema brachte allerlei Veränderungen am bestehenden System mit sich. Unter anderem konnten die Daten nicht mehr in bestehender Weise aus der Datenbank gelesen werden. Dies betrifft auch die Berechnungsnetzwerke (in Kapitel 3 beschrieben), mit deren Hilfe PARASUITE seine Datenanalyseverfahren und *Data Mining*-Mechanismen realisiert. Ein solches Netzwerk besteht aus miteinander verbundenen Komponenten, welche meist Berechnungskomponenten sind. Ausnahmen sind hierbei Komponenten, die das Netz mit Daten speisen („Produzenten“) und solche, die die berechneten Daten wieder aus dem Netzwerk heraustransportieren („Konsumenten“). Für das alte Datenbankschema existierte für jede kundenspezifische Tabelle eine eigene Lesekomponente. Es kann hier auch von Tabellenlesekomponenten gesprochen werden. Dies schränkt die Generizität eklatant ein, da für jeden Kunden neue, angepasste Lesekomponenten entwickelt werden müssen.

An dieser Stelle setzt diese Bachelorarbeit an. Das Ziel ist die Konzeption und Implementierung einer Datenbanklesekomponente, die zur Einspeisung

[9] Siehe `http://wiki.eclipse.org/index.php/Rich_Client_Platform`
[10] Siehe `http://www.jboss.org/`
[11] Siehe `http://subversion.tigris.org/`
[12] Siehe `http://checkstyle.sourceforge.net/`
[13] Siehe `https://hudson.dev.java.net/`

der Daten aus der neuen Datenbank in die Berechnungsnetzwerke genutzt werden kann. An die Komponente sind folgende Anforderungen gestellt:

Konfigurierbarkeit: die Komponente soll konfigurierbar und somit anpassbar in ihrem Verhalten sein. Dabei soll Folgendes einstellbar sein:

- Tabelle, aus der gelesen werden soll
- Tabellenspalten, die gelesen werden sollen
- Datenfilter (Filterbedingungen)
- Sortierung (Mehrspaltensortierung)
- Lokalisierung der Daten
- Verhalten bei Duplikaten (doppelten Datensätzen)

Generizität: die Komponente soll in der Lage sein, aus jeder (kundenspezifischen) Datenbanktabelle zu lesen, was über die Konfiguration eingestellt werden kann.

Menschenlesbarkeit: die Konfiguration der Komponente soll textuell erfolgen und menschenlesbar sein[14].

1.3 Aufbau dieser Bachelorarbeit

Diese Bachelorarbeit ist in fünf Kapitel unterteilt.

In diesem ersten Kapitel wird das Projekt beschrieben, in dessen Kontext dieser Arbeit entstand. Danach wird kurz auf die eingesetzten Technologien eingegangen, bevor das Ziel dieser Bachelorarbeit festgelegt wird.

In Kapitel 2 wird auf die in PARASUITE zu speichernden Daten eingegangen und das alte und das neue Datenbankschema, welches PARASUITE zugrunde lag bzw. liegt, werden beschrieben. Danach werden die beiden Datenbankschemen miteinander verglichen und deren Vor- und Nachteile erarbeitet.

In Kapitel 3 werden dann die in „PARASUITE“ unabdinglichen Datenvorberechnungen beschrieben. Dazu wird auch das Programmierparadigma und Framework *Flow-Based Programming* vorgestellt, mit dessen Hilfe PARASUITE die Datenvorberechnungen realisiert.

[14]Diese Anforderung kam während der Entwicklungsphase der Komponente hinzu.

Mit Hilfe des *Flow-Based Programming* Frameworks wird anschließend in Kapitel 4 eine Datenbanklesekomponente für das neue Datenbankschema erstellt.

In Kapitel 5 wird ein Ausblick auf weitere Entwicklungen im PARASUITE-Projekt gegeben und die in dieser Bachelorarbeit vorgestellten Konzepte werden nochmals miteinander in Beziehung gebracht.

Kapitel 2

Zugrundeliegende Datenbank und deren Schnittstellen

Dieses Kapitel beschreibt das alte und das neue Datenbankschema, das PARASUITE zugrunde lag bzw. liegt. Dabei wird zuerst auf die Daten eingegangen, die im PARASUITE-System gespeichert werden sollen. Danach wird die Umsetzung der Speicherung im alten Datenbankschema beschrieben, bevor die Umsetzung im neuen Datenbankschema vorgestellt wird. Anschließend wird auf deren Vor- und Nachteile eingegangen und die Notwendigkeit für den Umstieg vom alten auf das neue Datenbankschema wird erläutert. Im letzten Abschnitt dieses Kapitels werden die softwaretechnischen Schnittstellen zur Datenbank, die von diversen Softwaremodulen und -komponenten in PARASUITE verwendet werden können, für das neue Datenbankschema vorgestellt. Die Schnittstellen für das alte Datenbankschema werden hier nicht weiter erläutert, da es zum Verfassungszeitpunkt dieser Arbeit nicht mehr benutzt wird. Als Datenbanksystem wird in PARASUITE *MySQL*[1] eingesetzt.

2.1 Zugrundeliegendes Datenmodell

Beim zugrundeliegenden Datenmodell stellt sich die Frage, welche Daten überhaupt gespeichert werden sollen. Da PARASUITE im Bereich Produktlebenszyklus tätig ist, müssen produktbezogene Daten gespeichert werden. Dazu gehören zum einen Produkte selbst und zum anderen Diagnosedaten, z. B. Fehler- und Ausfallmeldungen (im Folgenden einfach „Meldungen“ genannt). Somit gibt es zwei Klassen zu speichernder Objekte: Produkte und Meldungen. Klassen von Objekten werden nachfolgend auch Entitäten genannt. Abbildung 2.1 zeigt ein Diagramm, das die beiden Entitäten samt

[1] Siehe `http://www.mysql.com/`

ihren Eigenschaften und ihrer Beziehung zueinander darstellt. Die Beziehung kann wie folgt beschrieben werden:

> Zu einem Produkt kann es mehrere Meldungen geben.
> Eine Meldung gehört zu genau einem Produkt.

Beide Entitäten besitzen dabei diverse Eigenschaften. Ein Produkt hat z. B. einen Namen und eine Modellnummer. Eine Meldung hat z. B. eine Beschreibung und eine Uhrzeit, welche angibt, wann die Meldung aufgetreten ist.

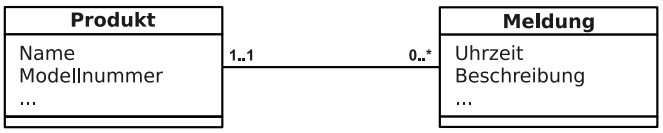

Abbildung 2.1: In PARASUITE zu speichernde Daten

Außerdem ist ist eine Lokalisierung von Daten und Metadaten nötig, da PARASUITE nicht nur von deutschsprachigen Anwendern genutzt wird, sondern auch von englischsprachigen. Die folgenden Abschnitte stellen die Umsetzung zur Speicherung dieser Daten im alten und im neuen Datenbankschema vor und beschreiben, wie die Lokalisierung von Daten in den beiden Datenbankschemen realisiert wird.

2.2 Das alte Datenbankschema

Das alte Datenbankschema „PARASUITE1“ wurde gegen Ende des Jahres 2008 abgeschafft und durch das neue Datenbankschema „PARASUITE2“ ersetzt. Das alte Datenbankschema ist ein generisches. Aufgrund der hohen Komplexität und der Menge der vorhandenen Tabellen wird hier nur das Grundprinzip dieses Schemas beschrieben. Eine detaillierte Beschreibung ist in [Neu07] zu finden.

2.2.1 Aufbau

Der grundlegende Aufbau des alten Datenbankschemas ist in Abbildung 2.2 zu sehen. Die Abbildung enthält acht Tabellen und deren Beziehungen untereinander. Die rot hervorgehobenen Beziehungen betreffen die Lokalisierung von Daten und werden im nächsten Abschnitt behandelt. Die oberen drei Tabellen

– *PRODUCT*, *PROPERTY_VALUE* und *PROPERTY_TYPE* – haben den Zweck, Produkte und ihre Eigenschaften zu speichern. Die beiden Tabellen in der Mitte – *TRANSLATION* und *LANGUAGE* – werden für die Lokalisierung von Daten verwendet. Meldungen und deren Eigenschaften werden in den unteren drei Tabellen – *NOTIFICATION*, *NOTIFICATION_VALUE* und *NOTIFICATION_TYPE* – gespeichert.

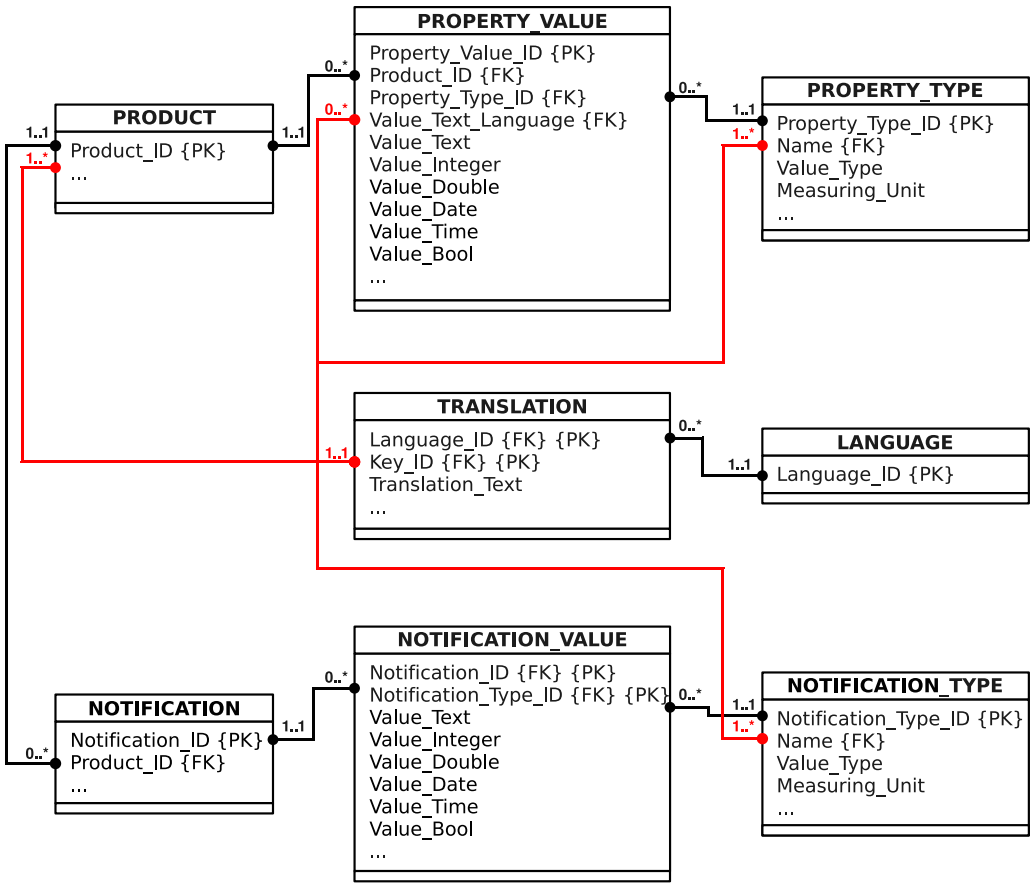

Abbildung 2.2: Aufbau des alten Datenbankschemas

Meldungen und Produkten liegt die gleiche Speicherstruktur zugrunde, bestehend aus je drei Tabellen (im Folgenden auch Dreierkombination genannt). Die Benennung der Tabellen für solch eine Dreierkombination ist im originalen Datenbankschema leider nicht konsistent und folglich auch nicht in Abbildung 2.2: Die meldungsspezifischen Tabellen tragen das Präfix *NOTIFICATION*, die produktspezifischen hingegen nicht nur das Präfix *PRODUCT*, sondern auch *PROPERTY*.

Im Folgenden dienen Produkte als Beispiel. Die Grundidee, die hinter der Dreierkombination von Tabellen steckt, ist die Zerlegung einer „normalen“ Datenbanktabelle in ihre entsprechenden Bestandteile: Tabelleninformationen, Spalteninformationen (inkl. Spaltentypen) und Werte. Die einzelnen Bestandteile werden jeweils als eigene Datenbanktabelle realisiert:

- Tabelleninformationen → *PRODUCT*
- Spalteninformationen → *PROPERTY_TYPE*
- Werte → *PROPERTY_VALUE*

Somit werden Informationen, die normalerweise implizit in einer „normalen“ Datenbanktabelle kodiert werden, explizit in eigens dafür vorgesehenen Tabellen kodiert. Informationen über eine Spalte werden zu einem Datensatz in der Tabelle *PROPERTY_TYPE*. Werte unter einer Spalte werden zu je einem Datensatz in der Tabelle *PROPERTY_VALUE*. Das macht die Speicherung der Daten flexibel und erlaubt es, neue Produkteigenschaften (≙ Spalten in einer „normalen“ Tabelle) hinzuzufügen, ohne die Notwendigkeit, neue Spalten zu irgendeiner Tabelle in der Datenbank hinzufügen zu müssen.

Die Tabelle *PRODUCT* enthält Produkte. Bei Bombardier sind dies Lokomotiven, bei Thyssen sind es Aufzüge. Ein Produkt wird dabei nur mit einem eindeutigen Bezeichner (*Product_ID*) gespeichert. Alle Produkteigenschaften liegen in der Tabelle *PROPERTY_TYPE*. Produkteigenschaften für Lokomotiven sind z. B. Fahrzeugnummer, Farbe und Fertigungsplattform, für Aufzüge z. B. Modelltyp, Maximallast und Volumen. Jede Produkteigenschaft besitzt dabei einen eindeutigen Bezeichner (*Property_Type_ID*), einen Namen (*Name*), eine Maßeinheit (*Measuring_Unit*) und eine Typangabe (*Value_Type*), die festlegt, welchen Datentyp eine Produkteigenschaft hat. Die Tabelle *PROPERTY_VALUE* enthält die konkret zu speichernden Werte. Jeder Wert besitzt einen eindeutigen Bezeichner (*Property_Value_ID*) und ist einer Produkteigenschaft (*Property_Type_ID*) und einem Produkt (*Product_ID*) zugeordnet. Der gespeicherte Wert steckt in einer der folgenden Spalten, je nachdem, welcher Typ in *PROPERTY_TYPE* in der Spalte *Value_Type* angegeben ist:

- *Value_Text_Language* (Typ `VARCHAR(32)`)
- *Value_Text* (Typ `VARCHAR(50)`)
- *Value_Integer* (Typ `BIGINT(20)`)
- *Value_Double* (Typ `DOUBLE`)
- *Value_Date* (Typ `DATE`)

- *Value_Time* (Typ TIME)
- *Value_Bool* (Typ TINYINT(1))

Davon ist pro Datensatz immer nur genau eine Spalte mit einem Wert belegt, alle anderen sind NULL. *Value_Text_Language* ist dabei lokalisierter Text. Auf Lokalisierung wird im nächsten Abschnitt genauer eingegangen. Die gerade beschriebene typisierte Speicherung von Werten wurde so gelöst, weil das Datenbanksystem – hier also *MySQL* – keine varianten Datentypen[2] unterstützt (siehe [Sunb]).

Somit können diverse Produkte mit ihren jeweiligen Eigenschaften angelegt werden, ohne dabei neue Spalten in irgendeiner Tabelle hinzufügen zu müssen. Auch Änderungen an Produkteigenschaften sind möglich, ohne eine Spalte umzubenennen.

Mit der Dreiertabellenkombination für Meldungen verhält es sich analog zu der für Produkte, mit zwei kleinen Ausnahmen:

1. In der Tabelle *NOTIFICATION* wird nicht nur ein eindeutiger Bezeichner (*Notification_ID*) für eine Meldung gespeichert, sondern auch die Beziehung zu dem Produkt (*Product_ID*), zu dem die Meldung gehört.
2. In der Tabelle *NOTIFICATION_VALUE* existiert keine Spalte, um lokalisierten Text zu speichern.

2.2.2 Lokalisierung

Lokalisierung wird vom eingesetzten Datenbanksystem *MySQL* von Haus aus nicht unterstützt, sodass an dieser Stelle ein Weg gefunden werden muss, dieser Anforderung gerecht zu werden. Das „PARASUITE1“-Datenbankschema geht dafür den Weg über zwei zusätzliche Tabellen: *TRANSLATION* und *LANGUAGE* (siehe auch Abbildung 2.2). Die Tabelle *LANGUAGE* enthält dabei alle verfügbaren Sprachen. Sie besitzt nur eine Spalte (*Language_ID*), welche gleichzeitig auch Primärschlüssel ist. Diese Spalte ist vom Datentyp CHAR(5) und ein Eintrag hat folgendes Format: <sprache>_<land>, wobei sprache und land je 2 Buchstaben haben. <sprache> darf dabei nur aus Kleinbuchstaben bestehen und ist ein *ISO Language Code* gemäß ISO 639-1[3]. <land> darf nur aus

[2] Variante Datentypen werden z. B. von Visual FoxPro (http://msdn.microsoft.com/de-de/library/cc483381(VS.71).aspx) oder VisualBasic (http://userpage.fu-berlin.de/~ram/pub/pub_jf47ht81Ht/vba_datentypen_de) unterstützt.

[3] Siehe [Int08] für weitere Informationen inkl. einer Liste aller *ISO Language Codes*.

Großbuchstaben bestehen und ist ein *ISO Country Code* gemäß ISO-3166-1[4]. Beispiele: `de_DE`, `en_GB`, `en_US`.

Die Tabelle *TRANSLATION* hat einen zusammengesetzten Primärschlüssel, der aus der Sprache (*Language_ID*) und einem MD5-Hash-Wert (*Key_ID*) besteht. Außerdem besitzt die Tabelle eine Spalte *Translation_Text*, welche die Übersetzung enthält. Um eine Übersetzung eines Wertes zu bekommen, muss vom zu übersetzenden Text ein MD5-Hash-Wert gebildet (*Key_ID*) und die gewünschte Sprache (*Language_ID*) angegeben werden. Mit dieser Information kann aus der Tabelle *TRANSLATION* dann der übersetzte Wert aus der Spalte *Translation_Text* gelesen werden. Das Speichern eines übersetzten Wertes funktioniert analog.

Das „PARASUITE1"-Datenbankschema speichert in all seinen Tabellen, in denen ein Textwert vorkommt, anstatt des originalen Wertes den MD5-Hash-Wert. Somit muss bei jedem Lesevorgang die Tabelle *TRANSLATION* „befragt" werden. Alle MD5-Hash-Wert-Spalten in anderen Tabellen als *TRANSLATION* haben eine Fremdschlüsselbeziehung zur Spalte *Key_ID* der Tabelle *TRANSLATION*. Diese Fremdschlüsselbeziehungen sind in Abbildung 2.2 der Übersicht halber mit roten Linien dargestellt.

Das hier vorgestellte alte Datenbankschema ist in der Lage, verschiedene Produkte mit jeweils unterschiedlichen Eigenschaften mit beliebig vielen Datensätzen zu speichern. Aber weder beim Kunden Bombardier noch beim Kunden Thyssen ist dies nötig, denn es existieren ausschließlich Produkte, die alle die gleichen Eigenschaften haben – bei Bombardier Lokomotiven und bei Thyssen Aufzüge.

2.3 Das neue Datenbankschema

Das neue Datenbankschema „PARASUITE2" ist ein generatives Datenbankschema und hat somit einen anderen Ansatz als das generische alte Datenbankschema. Zum Verfassungszeitpunkt dieser Arbeit ist das neue Datenbankschema die Basis für das PARASUITE-System und somit auch für die zu erstellende Datenbanklesekomponente (siehe Kapitel 4). Im Folgenden wird der Aufbau anhand des Kunden Bombardier näher beschrieben.

[4]Siehe [Int] für weitere Informationen inkl. einer Liste aller *ISO Country Codes*.

2.3.1 Aufbau

Im neuen Datenbankschema sind beliebige Strukturen erlaubt. Wie eine Struktur aussieht, ist komplett dem Ersteller überlassen. Es müssen dabei keine speziellen Restriktionen oder Konventionen beachtet werden. Wie in Abbildung 2.3 zu sehen, existieren in der Datenbank die kundenspezifischen Tabellen inklusive deren Beziehungen zueinander. Diese Tabellen können auf allgemein in Datenbanken gewohnte Weise angelegt und miteinander in Beziehung gebracht werden (Fremdschlüsselbeziehungen).

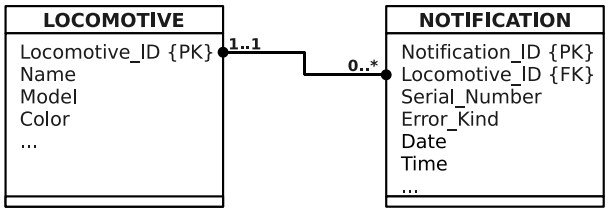

Abbildung 2.3: Neues Datenbankschema – Kundenspezifische Tabellen

In Abbildung 2.3 existieren zwei Tabellen – *LOCOMOTIVE* und *NOTIFICATION* – mit ihren jeweiligen Spalten. In diesen Tabellen sind die vom Kunden gesammelten Daten vorhanden, welche vom PARASUITE-System analysiert werden können. Es ist zu sehen, dass keinerlei zusätzliche Verwaltungsinformation in den Tabellen vorhanden ist, wie es beim alten Datenbankschema der Fall war (vergleiche 2.2).

Wie das alte Datenbankschema auch, wird das neue Datenbankschema der Anforderung nach Lokalisierung gerecht. Dabei werden lokalisierte Namen für alle kundenspezifischen Tabellen sowie für alle in den Tabellen vorhandenen Spalten benötigt. Außerdem besteht die Notwendigkeit nach lokalisierten Datenwerten in den Datensätzen der kundenspezifischen Tabellen. Das „PARASUITE2“-Datenbankschema löst all diese Probleme, wie in den folgenden Abschnitten genauer beschrieben wird.

2.3.2 Lokalisierte Tabellennamen

Für lokalisierte Tabellennamen existiert eine Tabelle *TABLE_LOC*, welche zu jeder im System vorhandenen kundenspezifischen Tabelle einen lokalisierten Namen in der jeweiligen Sprache speichert (siehe Abbildung 2.4). Für jede Lokalisierung existiert in der Tabelle *TABLE_LOC* eine eigene Spalte, die einen

Namen in folgendem Format hat: `<sprache>_<land>`, z. B. `de_DE` oder `en_GB` (vergleiche Abschnitt 2.2.2).

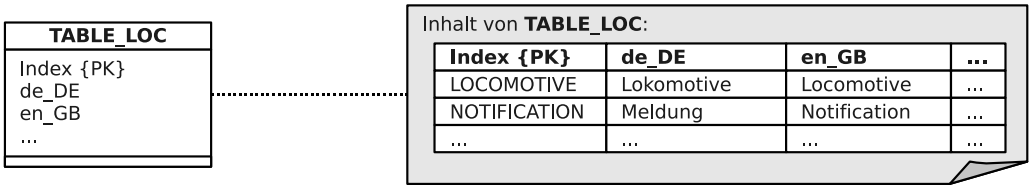

Abbildung 2.4: Neues Datenbankschema – Lokalisierte Tabellennamen

Unter der Tabellenspalte *Index* der Tabelle *TABLE_LOC* wird der Name der kundenspezifischen Tabelle angegeben. Die Spalte *Index* ist gleichzeitig der Primärschlüssel, da Tabellennamen innerhalb einer Datenbank eindeutig sein müssen. Ein Datensatz dieser Tabelle besteht somit – wie in Abbildung 2.4 zu sehen – aus dem Tabellennamen, wie er in der Datenbank vorkommt, und aus je einem Wert für jede Lokalisierung, hier im Beispiel für in Deutschland gesprochenes Deutsch (Spalte *de_DE*) und britisches Englisch (Spalte *en_GB*).

2.3.3 Lokalisierte Tabellenspalten

Um auch die Spaltennamen der einzelnen kundenspezifischen Tabellen mit lokalisierten Namen zu versehen, geht das „PARASUITE2“-Datenbankschema den Weg, eine eigene Tabelle für jede kundenspezifische Tabelle anzulegen, welche die Namen der Tabellenspalten in jeder Lokalisierung speichert (siehe Abbildung 2.5). Der Name solch einer „Spaltenlokalisierungstabelle“ folgt dem Format `<tabellenname>_LOCALIZED_HEADER`, wobei `<tabellenname>` für eine kundenspezifische Tabelle steht (z. B. *LOCOMOTIVE_LOCALIZED_HEADER*). Als Primärschlüssel hat diese Tabelle eine Spalte namens *ID*, die den Namen der kundenspezifischen Tabellenspalte angibt. Wie bei den lokalisierten Tabellennamen existiert auch hier für jede Lokalisierung eine eigene Spalte (im Beispiel *de_DE* und *en_GB*).

2.3.4 Lokalisierte Daten

Nachdem nun Tabellennamen und Tabellenspalten lokalisiert werden können, fehlt noch eine Möglichkeit, die gespeicherten Daten einer Tabelle, das heißt

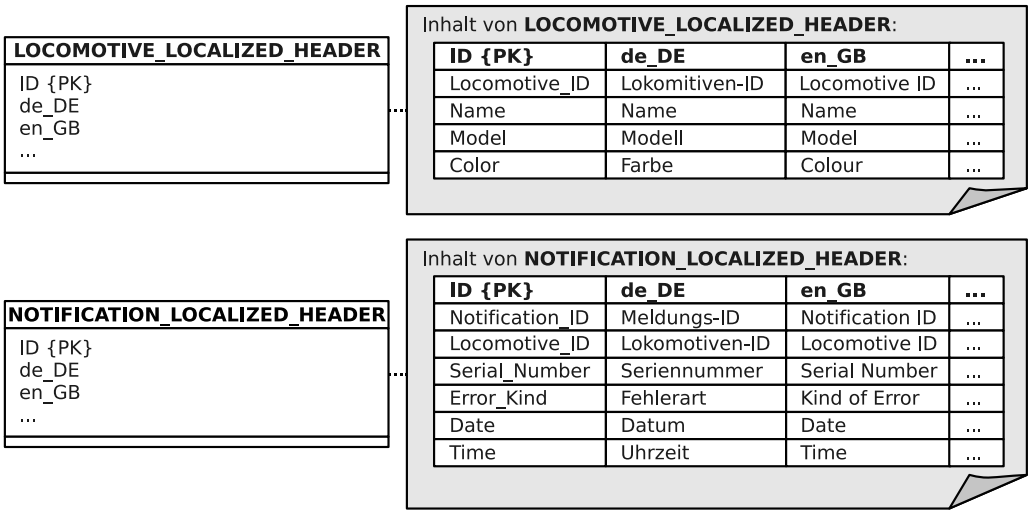

Abbildung 2.5: Neues Datenbankschema – Lokalisierte Tabellenspalten

die Daten für jede einzelne Spalte einer Tabelle, zu lokalisieren. Auch dafür bietet das „PARASUITE2“-Datenbankschema eine Möglichkeit. Somit ist es möglich, komplett lokalisierte Datensätze zu erhalten, indem alle Werte unter jeder Tabellenspalte lokalisiert werden. Allerdings ist es nicht immer sinnvoll, jegliche Art von Daten zu lokalisieren. Zum Beispiel ergibt es keinen Sinn, Zahlenwerte zu lokalisieren. Auch Kundendaten wie Name, Wohnort[5] und Adresse sind nicht sinnvoll zu lokalisieren – obwohl dies im „PARASUITE2“-Datenbankschema möglich wäre.

Realisiert ist das Ganze wie folgt: für jede zu lokalisierende Spalte einer kundenspezifischen Tabelle existiert eine eigene Tabelle. Das heißt, wenn eine Tabelle z. B. drei Spalten hat – *Name*, *Model* und *Color* – und alle sollen lokalisierte Werte haben, dann sind dazu drei zusätzliche Tabellen notwendig, nämlich für jede Spalte eine. Der Name einer solchen Tabelle hat folgendes Format: `<tabellenname>_LOCALIZED_COLUMN_<spaltenname>`, z. B. *LOCOMOTIVE_LOCALIZED_COLUMN_Color*, wie in Abbildung 2.6 zu sehen. Eine solche „Datenlokalisierungstabelle“ hat als Primärschlüssel die Spalte *ID*, welche den Wert der entsprechenden Spalte eines Datensatzes angibt. Dazu

[5] Bei Wohnorten – überwiegend bei größeren Städten – kann es durchaus sinnvoll sein, diese an eine Lokalisierung anzupassen. Zum Beispiel heißt die Stadt München auf Englisch bekanntlich *Munich*. Allerdings gibt es lokalisierte Namen nur für wenige Ortschaften (siehe [Wik09]).

kommen, wie bei lokalisierten Tabellennamen und -spalten auch, die Spalten für die jeweilige Lokalisierung (siehe Abbildung 2.6).

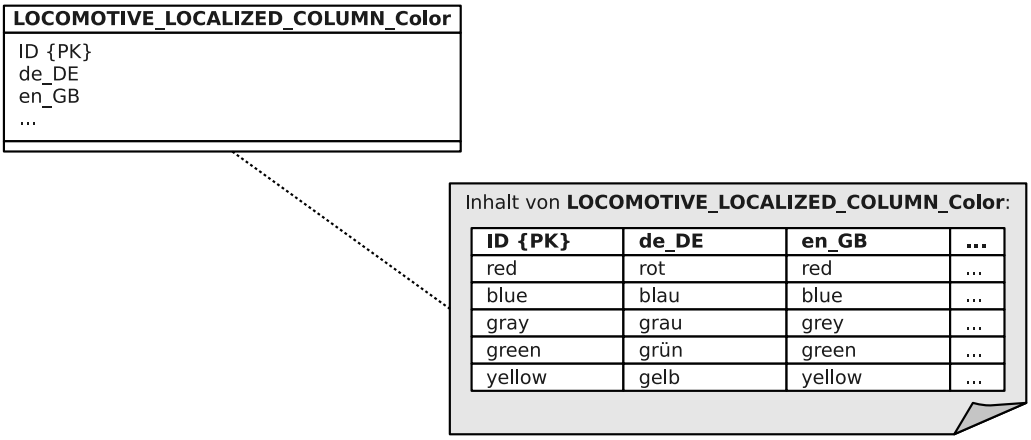

Abbildung 2.6: Neues Datenbankschema – Lokalisierte Daten

Mit den gerade vorgestellten Mechanismen zur Lokalisierung ist es möglich, alle kundenspezifischen Informationen zu lokalisieren. Außerdem können beliebig kundenspezifische Tabellen angelegt und miteinander in Beziehung gebracht werden. Damit erfüllt das neue Datenbankschema alle in Abschnitt 2.1 gestellten Anforderungen.

2.4 Vergleich der Datenbankschemen – Vor- und Nachteile

In diesem Abschnitt wird die Notwendigkeit für den Umstieg vom alten auf das neue Datenbankschema erläutert. Dabei wird auf Vor- und Nachteile der beiden Datenbankschemen eingegangen und der Bezug zu den in Abschnitt 2.1 festgelegten Anforderungen hergestellt.

2.4.1 Vergleich der Generizität

Das alte Datenbankschema ist, wie in Abschnitt 2.2 beschrieben, ein generisches Datenbankschema mit einer starren Struktur. Durch die Aufteilung

einer normalen Datenbanktabelle in drei einzelne Tabellen wandern Meta-Informationen, die normalerweise implizit im Datenbankschema kodiert sind, in explizit dafür vorgesehene Tabellen. Dadurch ergibt sich eine sehr flexible Art der Speicherung, ohne dass neue Tabellen oder Spalten hinzugefügt werden müssen. Somit steckt die Generizität implizit im Datenbankschema.

Beim neuen Datenbankschema ist dies nicht der Fall (siehe Abschnitt 2.3). Hier sind beliebige Strukturen erlaubt. Tabellen und Spalten können beliebig hinzugefügt, entfernt und verändert werden. Dadurch ändert sich bei solch einem Vorgang jedes Mal das Datenbankschema, da die Informationen über Tabellen und Spalten implizit in diesem kodiert sind. Damit ist das neue Datenbankschema nicht generisch. Um trotzdem generische Zugriffe auf die Informationen und Daten dieses Schemas zu bekommen, wurde der generische Teil in die Schnittstellen zur Datenbank verlagert. Diese Schnittstellen werden in Abschnitt 2.5 genauer beschrieben.

Somit sind generische Zugriffe auf beide Datenbankschemen möglich.

2.4.2 Vergleich der Lokalisierungsmöglichkeiten

Beide Datenbankschemen bieten die Möglichkeit zur Lokalisierung von Daten. Im alten Datenbankschema stecken alle übersetzten Werte in der Tabelle *TRANSLATION* (siehe Abschnitt 2.2.2). Jeder übersetzte Wert hat dabei den MD5-Hash-Wert des zu übersetzenden Textes als Primärschlüssel. Das heißt, zu jedem MD5-Hash-Wert existiert genau eine Übersetzung. Dieser MD5-Hash-Wert ist datenbankweit für alle zu übersetzenden Werte gleich. Es kann vorkommen, dass zu dem gleichen Wort in der einen Sprache verschiedene Übersetzungen in der anderen Sprache existieren. Ein Beispiel wäre das deutsche Wort *Birne*. Es kann ins Englische übersetzt werden mit *bulb* (Glühbirne) oder mit *pear* (Obst). Beide haben zwei völlig verschiedene Bedeutungen. Im alten Datenbankschema ist es nicht möglich, diese zwei Übersetzungen zu speichern. Somit muss einer Übersetzung der Vorzug gegeben werden. Ein weiterer, eher unwahrscheinlicher Fall wäre, wenn zwei unterschiedliche Worte oder Texte den gleichen MD5-Hash-Wert hätten. In diesem Fall könnte analog auch nur eine Übersetzung gespeichert werden.

Das neue Datenbankschema unterstützt die komplette Lokalisierung von kundenspezifischen Tabellen, deren Spalten und zugehörigen Werten (Datensätzen). (siehe Abschnitte 2.3.2, 2.3.3 und 2.3.4). Tabellennamen werden in der Tabelle *TABLE_LOC* gespeichert. Für die Spalten jeder kun-

denspezifischen Tabelle existiert je eine eigene Lokalisierungstabelle, z. B. *LOCOMOTIVE_LOCALIZED_HEADER*. Für die Lokalisierung von Werten einer Spalte existiert pro Spalte einer Tabelle eine eigene Tabelle, z. B. *LOCOMOTIVE_LOCALIZED_COLUMN_Color*. Jede Lokalisierungstabelle hat dabei pro Sprache eine Spalte, z. B. *de_DE* oder *en_GB*. Somit lassen sich alle Werte lokalisieren, was ein Vorteil gegenüber dem alten Datenbankschema ist.

2.4.3 Vergleich der Performanz und Handhabung

Die schlechtere Performanz des alten Datenbankschemas ist der Hauptgrund für den Umstieg vom alten auf das neue Datenbankschema. Um dies genauer zu erläutern, werden zunächst nur die zu speichernden Daten betrachtet (siehe Abschnitt 2.1). Dort gibt es Produkte und Meldungen zu diesen. Sollen nun alle Meldungen zu einem Produkt abgefragt werden, könnte das folgendermaßen formuliert werden:

> Zu **Produkt** X gib mir alle **Meldungen**.

An der Abfrage sind nur 2 Entitäten beteiligt, Produkte und Meldungen. Soll nun eine äquivalente Abfrage auf dem alten Datenbankschema ausgeführt werden, sieht die Abfrage wesentlich komplexer aus. Listing 2.1 zeigt dazu eine Abfrage in SQL.

Listing 2.1: Datenbankabfrage auf dem alten Datenbankschema

```
1  select
2    *
3  from
4    PRODUCT as p join
5    PROPERTY_VALUE as pv
6      on (p.PRODUCT_ID = pv.OBJECT_ID) join
7    PROPERTY_TYPE as pt
8      on (pv.PROPERTY_TYPE_ID = pt.PROPERTY_TYPE_ID) join
9    NOTIFICATION as n
10     on (p.PRODUCT_ID = n.PRODUCT_ID) join
11   NOTIFICATION_VALUE as nv
12     on (n.NOTIFICATION_ID = nv.NOTIFICATION_ID) join
13   NOTIFICATION_TYPE as nt
14     on (nv.NOTIFICATION_TYPE_ID = nt.
         NOTIFICATION_TYPE_ID)
15 where
```

```
16   p.PRODUCT_ID = '10202'
```

In der Abfrage tauchen 6 Tabellen auf, die alle mit Tabellen-Joins verbunden sind: die 3 Tabellen für Produkte und die 3 Tabellen für Meldungen.

Eine Abfrage auf dem neuen Datenbankschema hingegen sieht wesentlich einfacher aus (siehe Listing 2.2). Hier sind nur die Tabelle für Produkte (*LOCOMOTIVE*) und die Tabelle für Meldungen (*NOTIFICATION*) beteiligt, die auch mit einem Tabellen-Join verbunden sind.

Listing 2.2: Datenbankabfrage auf dem neuen Datenbankschema

```
1 select
2   *
3 from
4   LOCOMOTIVE as l join
5   NOTIFICATION as n on (l.Locomotive_ID = n.
        Locomotive_ID)
6 where
7   ...
```

Damit tauchen nur die Entitäten des zugrundeliegenden Datenmodells auf (siehe Abschnitt 2.1). Im Vergleich zum alten Datenbankschema ist eine Abfrage auf dem neuen Datenbankschema wesentlich performanter, was mit der Menge der vorhandenen Tabellen-Joins zusammenhängt.

Des Weiteren ist die Nähe des neuen Datenbankschemas zum Datenmodell zu erkennen. Für jede Entität im Datenmodell existiert im neuen Datenbankschema eine Tabelle. Dadurch sind Abfragen wesentlich leichter und intuitiver zu formulieren als im alten Datenbankschema, was die Handhabung erheblich vereinfacht.

2.4.4 Weitere Vor- und Nachteile

Im alten Datenbankschema existieren die Tabellen *PRODUCT_VALUE* und *NOTIFICATION_VALUE* (siehe 2.2). In diesen Tabellen ist für jeden möglichen Datentyp eine eigene Spalte vorhanden, von denen pro Datensatz immer nur eine mit einem Wert belegt ist. Dies rührt daher, dass MySQL keine varianten Datentypen unterstützt. Deshalb ist sehr viel ungenutzter Speicherplatz in diesen Tabellen vorhanden, was ein klarer Nachteil für das alte Datenbankschema gegenüber dem neuen ist.

Ein Nachteil des neuen Datenbankschemas ist allerdings, dass – bedingt durch die Lokalisierungstabellen – schnell sehr viele zusätzliche Tabellen entstehen können, was negativen Einfluss auf die Übersichtlichkeit hat.

2.5 Datenbankschnittstellen in PARASUITE

In jedem größeren System ist es empfehlenswert, ein zentrales Modul für den Datenbankzugriff zu haben. So muss nicht der fehleranfällige Weg gegangen werden, jedem Modul direkten Zugriff per SQL auf die Datenbank zu geben. Generell gibt es zwei gängige Ansätze, diesen zentralen Zugriff auf die Daten der Datenbank zu bewerkstelligen. Zum einen kann ein Persistenz-Framework wie Hibernate[6] verwendet werden, das objekt-relationales Mapping[7] betreibt und zum anderen gibt es den Ansatz des Data Access Objects[8] (kurz DAO). Durch Untersuchungen im Rahmen des PARASUITE-Projektes kommt es, bedingt dadurch, dass PARASUITE Massendaten verarbeitet, beim ersten Ansatz zu Performanzeinbrüchen, was auf das rechenintensive objekt-relationale Mapping zurückzuführen ist. Deshalb wird in PARASUITE der letztere Ansatz verwendet, bei dem diese Performanzeinbußen nicht bestehen.

Für den Zugriff auf die Daten der Datenbank existieren in PARASUITE zwei zentrale Schnittstellen – die „ParasuiteEntityDAOBean“ und der „Value List Handler“. Beide sind realisiert als Enterprise JavaBean[9] (EJB) und sowohl für lokalen als auch für entfernten Zugriff ausgelegt. Bevor die beiden Schnittstellen vorgestellt werden, werden die Klassen beschrieben, die als Parameter in den Methoden der Schnittstellen auftauchen. PARASUITE verwendet außerdem eine eigene Terminologie bei Tabellen und deren Spalten. Eine Tabelle heißt „PARASUITE Entity“ und eine Spalte heißt „PARASUITE Entity Type“. Die entsprechenden Java-Klassen für deren Repräsentation heißen folglich `ParasuiteEntity` und `ParasuiteEntityType`. Diese beiden Klassen und die Klasse `ParasuiteEntityData` werden deshalb zunächst genauer beschrieben, bevor auf die beiden Datenbankschnittstellen „ParasuiteEntityDAOBean“ und „Value List Handler“ genauer eingegangen wird.

[6] Siehe `http://www.hibernate.org/`

[7] Siehe `http://www.it-visions.de/glossar/alle/838/Objekt-Relationales%20Mapping.aspx`

[8] Siehe `http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html`

[9] Siehe `http://java.sun.com/products/ejb/`

2.5.1 Wichtige Java-Klassen für PARASUITE-Datenbankschnittstellen

Eine Instanz der Klasse `ParasuiteEntity` enthält Informationen über eine Tabelle in der Datenbank. Eine Instanz der Klasse `ParasuiteEntityType` beinhaltet Informationen über eine Spalte einer Tabelle. Eine Instanz der Klasse `ParasuiteEntityData` enthält Datensätze aus der Datenbank. Abbildung 2.7 zeigt die Klassen in UML-Notation.

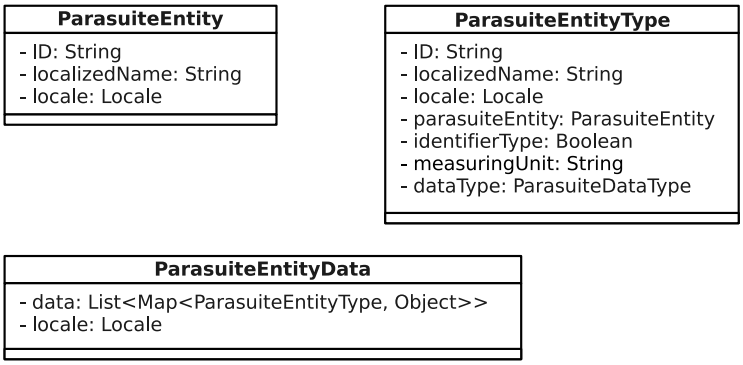

Abbildung 2.7: Neues Datenbankschema – Lokalisierte Tabellenspalten

ParasuiteEntity Die Klasse `ParasuiteEntity` deklariert 3 Instanzvariablen. `ID` ist dabei der Name der Tabelle in der Datenbank bzw. der Name des PARASUITE Entitys. `localizedName` ist der lokalisierte Name der Tabelle in der Datenbank in der in `locale` angegebenen Lokalisierung.

ParasuiteEntityType Die Klasse `ParasuiteEntityType` deklariert 7 Instanzvariablen. Analog zur Klasse `ParasuiteEntity` ist dabei `ID` der Name einer Tabellenspalte, die zur Tabelle gehört, die mit `parsuiteEntity` angegeben wird. `localizedName` ist der lokalisierte Name einer Tabellenspalte in der in `locale` angegebenen Lokalisierung. Mit der Instanzvariable `identifierType` vom Typ `Boolean` wird angegeben, ob es sich um eine Tabellenspalte handelt, die als Primärschlüssel definiert bzw. Teil eines Primärschlüssels ist, wenn die Tabelle einen zusammengesetzten Primärschlüssel enthält. `dataType` gibt an, welchen Typ die Tabellenspalte hat und ist vom `Enum`-Typ `ParasuiteDataType`, welcher u. a. folgende Enum-Konstanten definiert:

- UNKNOWN_TYPE
- VALUE_TEXT_LOCALIZED
- VALUE_TEXT
- VALUE_INTEGER
- VALUE_DOUBLE
- VALUE_BOOL
- VALUE_BOOL_TEXT_LOCALIZED
- VALUE_DATE
- VALUE_TIME
- VALUE_DATE_TIME

UNKNOWN_TYPE ist dabei ein Überbleibsel aus der frühen Entwicklungsphase des neuen Datenbankschemas, als noch unbekannte Typen erlaubt waren. Zum Verfassungszeitpunkt dieser Arbeit spielt dieser Typ keine Rolle mehr und ist hier nur der Vollständigkeit halber aufgeführt. Es existieren außerdem zwei *LOCALIZED*-Typen – VALUE_TEXT_LOCALIZED und VALUE_BOOL_TEXT_LOCALIZED. Diese beiden geben an, ob ein Text bzw. ein Wahrheitswert lokalisierbar sein soll. Ein Beispiel für lokalisierbaren Text (VALUE_TEXT_LOCALIZED) ist eine Farbe. Ein Beispiel für nicht lokalisierbaren Text (VALUE_TEXT) ist eine Modellbezeichnung eines Produkts. Für Wahrheitswerte verhält es sich analog: Ein Wahrheitswert, der lokalisiert werden kann (VALUE_BOOL_TEXT_LOCALIZED), ist z. B. der Zustand eines Schalters (AN/AUS) und einer, der nicht sinnvoll zu lokalisieren ist (VALUE_BOOL), z. B. ein Binärwert (0/1).

ParasuiteEntityData Die Klasse ParasuiteEntitydata deklariert 2 Instanzvariablen und enthält Datensätze aus der Datenbank. data ist dabei die Liste der Datensätze. Ein Eintrag der Liste repräsentiert einen Datensatz und besteht aus einer Map, die als Schlüssel ein ParasuiteEntityType-Objekt (Spalteninformationen) enthält und als Wert ein Objekt vom Typ Object. Dieses Objekt enthält den Wert und hat den Typ, der im ParasuiteEntityType-Objekt in der Variablen dataType angegeben ist. Die zweite Instanzvariable, die diese Klasse deklariert, ist locale, wie bei den beiden vorher vorgestellten Klassen auch.

2.5.2 Datenbankschnittstelle „ParasuiteEntityDAOBean“

Diese Schnittstelle existierte als erste und stellt im Prinzip alle nötigen Funktionen zur Verfügung. Aus Performanzgründen wurde noch die zweite Schnittstelle – der „Value List Handler“ – eingeführt, welcher in Abschnitt 2.5.3 beschrieben wird. Listing 2.3 zeigt die wichtigsten Ausschnitte aus

dem Quellcode der „Local“-Schnittstelle der „ParasuiteEntityDAOBean“. Die „ParasuiteEntityDAOBean“ ist eine *stateless* Session Bean.

Listing 2.3: Java-Code – Local-Schnittstelle `ParasuiteEntityDAO`

```
@Local
public interface ParasuiteEntityDAOLocal {

  List<ParasuiteEntity> getParasuiteEntities(Locale
      locale);

  List<ParasuiteEntityType> getParasuiteEntityTypes(
      ParasuiteEntity entity, Locale locale);

  ParasuiteEntity getParasuiteEntity(String entityName,
      Locale locale);

  ParasuiteEntityType getParasuiteEntityType(String
      entityName, String entityTypeName, Locale locale);

  ParasuiteEntityData getParasuiteEntityData(
      ParasuiteEntity entity, Locale locale, List<
      ParasuiteEntityType> desiredTypes,
      FilterExpressionGroup filterExpressionGroup, List<
      Map<ParasuiteEntityType, Boolean>> sortTypes,
      Boolean distinct, Integer limit);

  ...
}
```

Die Methode `getParasuiteEntities` liefert als Rückgabewert eine Liste aller in der Datenbank vorhandenen PARASUITE Entities (≙ Tabellen) in der als Parameter angegebenen Lokalisierung (`locale`). Die Methode `getParasuiteEntityTypes` liefert zu einer PARASUITE Entity alle vorhandenen PARASUITE Entity Types (≙ Tabellenspalten) in der gewünschten Lokalisierung. Mit der Methode `getParasuiteEntity` kann ein spezielles PARASUITE Entity-Objekt anhand seines Namens abgefragt werden. Wenn kein entsprechendes Entity mit dem angegebenen Namen gefunden wurde, wird `null` zurückgegeben. Mit der Methode `getParasuiteEntityType` kann ein spezielles PARASUITE Entity Type-Objekt anhand seines Namens abgefragt werden. Als Parameter wird der Name des PARASUITE Entitys und des PARASUITE Entity Types erwartet. Wird kein entsprechender PARASUITE

Entity Type mit dem angegebenen Namen gefunden, wird ebenfalls `null` zurückgegeben. Die Methode `getParasuiteEntityData` ist die komplexeste Methode dieses Interfaces. Sie liefert alle Datensätze einer Tabelle, die den Kriterien in den angegebenen Parametern entsprechen. Sie erwartet insgesamt 7 Parameter. Der Übersicht halber stellt Tabelle 2.1 die Parameter einschließlich Erklärung und dem SQL-Pendant dar.

Parameter	**Erklärung**	**SQL-Pendant**
entity	PARASUITE Entity, aus dem Daten abgefragt werden	`FROM`
locale	Lokalisierung	n/a[a]
desiredTypes	Auswahl der PARASUITE Entity Types	`SELECT`
filterExpressionGroup	Filter-Bedingungen	`WHERE`
sortTypes	Sortierung + Richtung	`ORDER BY`
distinct	keine Duplikate	`DISTINCT`
limit	maximale Anzahl der abzufragenden Datensätze	`LIMIT`

[a]Es ist keine vergleichbare Funktionalität in SQL vorhanden.

Tabelle 2.1: Parameter der Methode `getParasuiteEntityData`

Der Parameter `entity` legt fest, von welchem PARASUITE Entity gelesen werden soll, `locale` bestimmt, in welcher Sprache die Daten geholt werden. Mit `desiredTypes` wird festgelegt, welche Spalten gelesen werden. `filterExpressionGroup` legt Filter-Bedingungen fest und ist vom Typ `FilterExpressionGroup`, welcher in Abschnitt 4.3.1 genauer erklärt wird. Der Parameter `sortTypes` ist eine Liste von PARASUITE Entity Types, die die Sortierreihenfolge angibt und pro angegebenem PARASUITE Entity Type auch noch die Sortierrichtung (auf- oder absteigend) enthält. `distinct` bestimmt, ob zwei oder mehrere identische Zeilen nur einmal ausgegeben werden sollen oder nicht. Mit `limit` ist die Möglichkeit gegeben, die ausgegebenen Datensätze auf eine bestimmte Anzahl zu beschränken.

2.5.3 Datenbankschnittstelle „Value List Handler“

Der „Value List Handler“[10] ist dem gleichnamigen JEE-Pattern nachempfunden und realisiert das so genannte „Paging“-Prinzip. Die hier vorgestellte Implementierung ist keine exakte Nachahmung dieses Patterns, hat jedoch vieles mit diesem gemeinsam. Die im vorigen Abschnitt beschriebene „ParasuiteEntityDAOBean“ hat einen großen Nachteil beim Abfragen größerer Datenmengen. Diesen Nachteil bringt die Methode `getParasuiteEntityData` mit sich. Als Rückgabewert hat sie ein Objekt vom Typ `ParasuiteEntityData`, das alle Ergebnis-Datensätze der Abfrage enthält. Deshalb kann es unter Umständen sehr lange dauern, große Datenmengen abzufragen. Der Aufruf der Methode kehrt somit erst zurück, wenn alle Datensätze in das Rückgabeobjekt „gepackt“ sind.

Ein Anwendungsszenario wäre das Navigieren durch die Datensätze in der PARASUITE-GUI. Steht nur die „ParasuiteEntityDAOBean“ zur Verfügung, muss die GUI alle Datensätze auf einmal abfragen und dann jeweils die entsprechenden Einträge anzeigen. Im Extremfall kann es vorkommen, dass die GUI einen `OutOfMemoryError` aufgrund zu vieler zwischenzuspeichernder Daten produziert. Der Value List Handler wurde nötig, um genau dieses Problem zu lösen. So kann bei diesem angegeben werden, wie viele Datensätze abgefragt werden sollen und bei welchem Datensatz begonnen wird. Der Value List Handler ist als *stateful* Session Bean implementiert. Die Definition der Schnittstelle ist in Listing 2.4 zu sehen.

Listing 2.4: Java-Code – Remote-Schnittstelle des `ValueListhandler`

```
1 @Remote
2 public interface ValueListHandlerRemote {
3
4   void executeSearch(ParasuiteEntity entity, Locale
        locale, List<ParasuiteEntityType> desiredTypes,
        FilterExpressionGroup filterExpressionGroup, List<
        Map<ParasuiteEntityType, Boolean>> sortTypes,
        Boolean distinct);
5
6   int getSize();
7
8   ParasuiteEntityData getCurrentElement();
9
```

[10] Siehe http://java.sun.com/blueprints/corej2eepatterns/Patterns/ValueListHandler.html

```
10    ParasuiteEntityData getPreviousElements(int howMany);
11
12    ParasuiteEntityData getNextElements(int howMany);
13
14    void resetIndex();
15
16    int getIndex();
17
18    void setIndex(int index);
19
20    void destroy();
21 }
```

In der Dokumentation der Schnittstelle ist beschrieben, wie der Value List Handler zu benutzen ist. Zuerst muss die Methode `executeSearch` aufgerufen werden, welche bis auf den Parameter `limit` die gleiche Schnittstelle hat wie die Methode `getParasuiteEntityData` der „ParasuiteEntityDAOBean“ (siehe Abschnitt 2.5.2). Der `limit`-Parameter ist hier unnötig, da durch Aufruf der anderen Methoden der abgefragte Bereich angegeben werden kann. Nach dem Aufruf von `executeSearch` ist der value List Handler im Initialzustand, d. h. der „Navigationszeiger“, mit dem man durch die Daten navigiert, steht vor dem erstem Datensatz. Analog kann man bis hinter den letzten Datensatz navigieren. Die Methode `getSize` liefert die Anzahl vorhandener Datensätze für die getätigte Abfrage. `getCurrentElement` liefert das Element bzw. den Datensatz, auf das der Navigationszeiger aktuell zeigt, im Initialzustand jedoch `null`. Die Methode `getPreviousElements` liefert so viele Elemente, die sich direkt vor dem aktuellen Element befinden, wie in `howMany` angegeben, das aktuelle Element ausgeschlossen. `getNextElements` funktioniert analog und liefert die Elemente nach dem aktuellen Element. Die Methode `resetIndex` setzt den Navigationszeiger wieder vor das erste Element. `getIndex` liefert den aktuellen Index, an dem der Navigationszeiger steht, und mit `setIndex` kann der Navigationszeiger auf den im Parameter `index` angegebenen Index gesetzt werden. Die Methode `destroy` „vernichtet“ den Value List Handler sozusagen und gibt seine Ressourcen wieder frei. Komplett entladen werden kann er jedoch nur vom zuständigen Applikationsserver JBoss. Nach dem Aufruf von `destroy` ist der Value List Handler nicht mehr funktionstüchtig.

Kapitel 3

Datenvorberechnungen und Flow-Based Programming

Dieses Kapitel erklärt die Notwendigkeit von Datenvorberechnungen im PARASUITE-System und zeigt die Realisierung dieser mittels des Programmierparadigmas *Flow-Based Programming*. Dabei werden die Grundkonzepte und Bestandteile dieses Paradigmas erklärt und das Framework *JavaFBP*[1] wird vorgestellt, welches von John Paul Morrison[2] entwickelt und zusammen mit Sven Steinseifer weiterentwickelt wurde (vergleiche [Ste09]).

In der heutigen Welt nehmen Datenbestände immer mehr zu. Wissen – also Informationen, mit denen man etwas anfangen kann – über die Daten ist jedoch nicht automatisch vorhanden, sodass viele Daten ungenutzt in Datenbanken „herumliegen“. Mit Data Mining wird versucht, interessante Informationen oder Muster aus den Daten in großen Datenbanken zu gewinnen. Typische Einsatzgebiete für Data Mining sind z. B. Marktanalysen (Marktsegmentierung, Kundenverhalten), Risikoanalysen (Finanz- und Ressourcenplanung) und Betrugserkennung (Versicherungsbetrug, „Geldwäsche“) (siehe [Sei03, S. 3]). In PARASUITE werden auch Verfahren des Data Mining eingesetzt, um aus den großen Mengen an vorhandenen Kundendaten Informationen und Erkenntnisse zu gewinnen. Diese Verfahren sind sehr rechenintensiv und dauern in der Regel selbst auf modernen Mehrprozessorsystemen einige Stunden. Aus diesem Grund wurde der Berechnungsprozess in PARASUITE in zwei Schritte aufgeteilt. Zum einen gibt es Datenvorberechnungen, welche die rechenintensiven Data Mining-Verfahren durchführen und zum anderen gibt es Datenauswertungen, die die vorberechneten Daten als Grundlage nehmen, um diese dann für die grafische Präsentation auszuwerten und aufzubereiten. PARASUITE realisiert Datenvorberechnungen mit Hilfe von Flow-Based

[1] Siehe `http://jpaulmorrison.com/fbp/#JavaFBP`
[2] Siehe `http://jpaulmorrison.com/`

Programming und nutzt dazu das Flow-Based Programming-Framework JavaFBP.

3.1 Flow-Based Programming (FBP)

Wie oben bereits erwähnt, ist Flow-Based Programming (im Folgenden auch einfach FBP genannt) ein Programmierparadigma, welches von John Paul Morrison ins Leben gerufen wurde (siehe [Ste09, S. 3, Abschnitt „1.1 History“]). Anwendungen werden als Netzwerke von „black box“-Prozessen betrachtet, die Daten über vordefinierte uni-direktionale Verbindungen austauschen. Diese Komponenten können beliebig miteinander verbunden werden, ohne dass sie intern verändert werden müssen.

> In “Flow-Based Programming” (FBP), applications are defined as networks of “black box” processes, which exchange data across predefined one-way connections. These black box processes can be reconnected endlessly to form different applications without having to be changed internally. [Mor]

3.1.1 Ideologie und Idee hinter Flow-Based Programming

Dieser Unterabschnitt stellt die Ideologie und die Ideen, die hinter Flow-Based Programming stehen, dar. Als Quelle für diesen Unterabschnitt diente [Mor94, S. 6–13].

Morrison schreibt in seinem Buch [Mor94] über Flow-Based Programming (FBP) von einer revolutionären Technologie, die die ganze Zeit nur darauf gewartet hätte, endlich Anwendung zu finden. Das Problem, das die Verbreitung von FBP bisher verhindert hätte, sei der Paradigmenwechsel, welcher mit dieser Technologie einhergehe. Anwendungen würden einfach aus vorgefertigten Komponenten zusammengebaut werden, anstatt Codezeilen zu schreiben. Außerdem biete diese Technologie eine konsistente Sicht auf eine Software-Anwendung, angefangen beim Design auf höchster Ebene, bis hinunter zur Implementierung. Beim Entwickeln von Software läge der Fokus somit auf den Daten und deren Transformationen, anstatt darauf, sich mit prozeduralem Code zu beschäftigen. Die Softwareentwicklungsmethode „Rapid Prototyping“ sei gleich in diesem Prinzip inbegriffen, was zu zuverlässigeren und besser wartbaren Systemen führe. Des Weiteren sei die

Technologie bestens geeignet, um verteilte Systeme aufzubauen und scheine sich in die gleiche Richtung wie objektorientierte Programmierung zu entwickeln. Flow-Based Programming sei somit eine wahrliche Revolution im Softwareentwicklungsprozess, nicht zuletzt deshalb, weil Benutzer und Programmierer die gleiche Sprache sprechen würden. Flow-Based Programming wird, im Gegensatz zu prozedularen Sprachen, als Koordinationssprache angesehen. Herkömmliche Programmiersprachen würden dem Computersystem sagen, „was“ auszuführen sei bzw. welche Operationen durchzuführen seien, wohingegen Koordinationssprachen wie FBP dem Computersystem sagen würden, „wie“ verschiedene Module koordiniert werden sollen. Darüberhinaus schreibt Morrison von „konfigurierbarer Modularität“, womit die Fähigkeit zur Wiederverwendung von unabhängigen Komponenten verdeutlicht werden soll, welche einfach in anderer Weise miteinander kombiniert werden können. Diese Eigenschaft kennzeichne jedes gelungene und erfolgreiche System.

3.1.2 Analogien und weitere Konzepte von FBP

Ein vergleichbares Prinzip zu FBP ist das z. B. in UNIX-basierten Systemen verwendete „Pipelining“. Die Ausgabe eines Programms wird zur Eingabe des nächsten Programms in der Pipeline. Das Ausführen des Befehls `ls | grep '*FBP*' | lpr` auf der Kommandozeile z. B. bewirkt, dass durch das Programm `ls` alle Namen der Dateien und Verzeichnisse im aktuellen Verzeichnis auf die Standard-Ausgabe (`stdout`) geschrieben werden, welche durch das Pipe-Symbol „`|`“ zur Standard-Eingabe (`stdin`) des nächsten Programms werden, in diesem Falle `grep`. Das Programm `grep` seinerseits filtert alle Dateien und Verzeichnisse heraus, die „FBP“ in ihrem Namen tragen, und das Programm `lpr` druckt diese Datei- und Verzeichnisnamen auf den Standard-Drucker. Mit Flow-Based Programming lässt sich dieses Prinzip auch realisieren.

> ... UNIX supports the concept of “pipelining”, where the output of one process becomes the input of another, and so on repeatedly. This is definitely a form of configurable modularity, and I found a lot of their experience using this technique relates closely to things we discovered using FBP. [Mor94, S. 293–294, Abschnitt „UNIX and its descendants“]

Um eine Vorstellung davon zu bekommen, wie ein Netzwerk in Flow-Based Programming funktioniert, ist in Abbildung 3.1 ein kleines Beispiel-Netzwerk zu sehen.

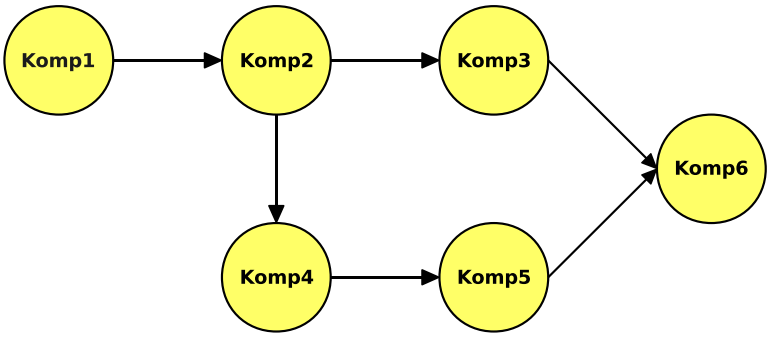

Abbildung 3.1: Simples Beispiel-Netzwerk in Flow-Based Programming

Im Beispiel ist *Komp1* (steht für „Komponente 1“) der Produzent und beliefert das Netz mit Daten. Woher *Komp1* die Daten nimmt, ist an dieser Stelle irrelevant. Über eine Verbindung fließen die Daten dann zu *Komp2*, welche in diesem Fall Daten über zwei Verbindungen heraussendet. Was *Komp2* konkret macht, ist an dieser Stelle ebenfalls uninteressant. Wichtig ist nur, dass eine Komponente mehrere ausgehende Verbindungen besitzen kann und es Sache der Komponente ist, welche Daten wo herausgesendet werden. Deshalb sollte eine Komponente immer gut dokumentiert werden. Im Beispiel-Netzwerk ist zu sehen, dass die Daten der einen Verbindung über *Komp3* zu *Komp6* fließen und die der anderen Verbindung über *Komp4* und *Komp5* zu *Komp6*. Was hiermit demonstriert werden soll, ist, dass eine Komponente auch mehrere eingehende Verbindungen haben kann (siehe [Ste09, S. 4]).

Abbildung 3.1 ist relativ simpel gehalten, aber zur Demonstration des Prinzips ausreichend. Es sind zwei wesentliche Bestandteile von Flow-Based Programming zu sehen: Komponenten und Verbindungen. In Wirklichkeit besitzen Komponenten Ports, über welche Daten empfangen und gesendet werden können. Verbindungen können nur an diese Ports „angeschlossen“ werden, nicht direkt an die Komponenten. Die Daten, die über Verbindungen transportiert werden, heißen in FBP Informationspakete (engl.: *Information Packets*). Des Weiteren gibt es zusammengesetzte Komponenten (im Folgenden „Subnetze“ genannt), die Teilnetzwerke beinhalten können. Nach außen sehen diese aus wie normale Komponenten und können auch in dieser Weise benutzt werden. Somit wird nicht nur eine abstrakte Sicht auf komplexe Bestandteile des Gesamtnetzwerks erreicht, sondern jedes Subnetz kann beliebig oft an verschiedenen Stellen im Netzwerk wiederverwendet werden.

> The next concept I want to describe is the ability to group components into packages which can be used as if they were components in their own right. This kind of component is called a “composite component”. It is built up out of lower-level components, but on the outside it looks just like any other component. [Mor94, S. 38]

> It is possible to use hierarchical decomposition to build data-flow networks. That means, one can build a component out of several components. Such a component is called a “subnet” in FBP terminology. Ports of the subnet will be assigned to ports of the inner components. Such a subnet can be used like a normal component in a network. The user of the subnet does not need to know he is using a subnet rather than a normal component. [Ste09, S. 5, Abschnitt „1.2.2 Hierarchical Decomposition“]

John Paul Morrison sagt in seinem Buch [Mor94] außerdem, dass Entwickler versuchen sollen, bestehende Komponenten zu verwenden, anstatt neue zu entwickeln, außer es gibt einen guten Grund für eine Neuentwicklung.

> New black boxes can be written as needed, but a developer tries to use what is available first, before creating new components. In FBP, the emphasis shifts from building everything new to connecting preexisting pieces and only building new when building a new component is cost-justified. [Mor94, S. 12–13]

In den folgenden Abschnitten wird das „JavaFBP“-Framework vorgestellt und die Bestandteile von FBP werden genauer beschrieben.

3.2 Das Framework „JavaFBP“ und dessen Bestandteile

John Paul Morrison stellt für sein FBP-Paradigma fertige Frameworks für verschiedene Programmiersprachen zur Verfügung, u. a. für Java und C#.NET. Für Java hat dieses Framework den Namen „JavaFBP“, für C# analog „C#FBP“.[3] Die folgenden Abschnitte gehen deshalb auch stellenweise auf die Realisierung der Konzepte in Java ein.

[3] Beide Frameworks sind bei SourceForge unter der URL `http://sourceforge.net/projects/flow-based-pgmg/` zum Download verfügbar.

3.2.1 Komponenten und Ports

In JavaFBP ist jede Komponente als eigener Thread realisiert. Dadurch existiert ein hoher Grad an Nebenläufigkeit und die Ausführung von FBP-Netzwerken ist prädestiniert für parallele Verarbeitung auf Mehrprozessorsystemen.

> For implementing the processes, JavaFBP uses the concurrency facilities of the `Thread` class provided by Java. [Ste09, S. 18]

Für eine Komponente ist es sinnvoll, eine möglichst kleine, spezielle Aufgabe zu erfüllen. So kann sie sinnvoll mit anderen Komponenten kombiniert werden, um eine größere Funktionalität zu erreichen und die Flexibilität zu steigern. Dieses Prinzip ist aus der UNIX- und Linux-Welt bekannt, wo viele kleine Kommandozeilenprogramme zur Verfügung stehen, die je eine spezielle Aufgabe erfüllen, aber in Kombination sehr komplexe Probleme lösen können. (siehe [PW08]).

Ports

Wie oben bereits angesprochen, besitzen Komponenten Ports, über die sie Daten senden und empfangen. Jede Komponente hat eine fest definierte Anzahl an Ports, die der Komponentenentwickler bestimmen kann. Diese Ports können unterschiedlicher Art sein (vergleiche [Ste09, S. 3]). Vom JavaFBP-Framework werden folgende vier Arten zur Verfügung gestellt:

- Eingangsport (engl.: *Input Port*)
- Ausgangsport (engl.: *Output Port*)
- Array-Eingangsport (engl.: *Array Input Port*)
- Array-Ausgangsport (engl.: *Array Output Port*)

Über einen Eingangsport kann eine Komponente Daten empfangen und über einen Ausgangsport Daten senden. Dabei kann jeweils nur eine einzige ein- bzw. ausgehende Verbindung an den Port angeschlossen werden. An einen Array-Port ist es möglich, mehrere ein- bzw. ausgehende Verbindungen anzuschließen. Von PARASUITE werden noch zwei weitere Port-Arten benötigt, und zwar ein Konfigurationseingangsport (engl.: *Config Input Port*), über den die Komponente konfiguriert wird, und ein Konfigurationsausgangsport (engl.: *Config Output Port*), über den die Komponente ihre Konfiguration senden kann. Technisch sind diese beiden zusätzlichen Port-Arten als *Input Port*

bzw. *Output Port* realisiert, werden aber in einer FBP-Komponente als Konfigurationsports behandelt (mehr dazu in Kapitel 4). Abbildung 3.2 zeigt eine FBP-Komponente mit allen möglichen Port-Arten inklusive der von PARASUITE eingeführten Konfigurationsports.

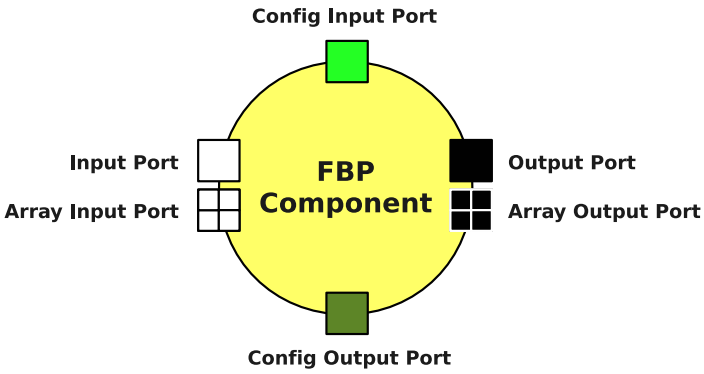

Abbildung 3.2: FBP-Komponente mit Ports

Port-Annotationen

Ports werden im Java-Code mit Hilfe von Annotationen an der Komponentenklasse definiert. Listing 3.1 zeigt ein Beispiel einer Komponentenklasse, die mit allen möglichen Port-Arten annotiert ist. Die Definitionen `@InPort` und `@OutPort` müssen dabei den Namen des Ports als Parameter übergeben bekommen. Dieser Name muss eindeutig sein. Um eine Komponentenklasse mit mehreren Ports gleicher Art auszustatten, muss die Annotation `@InPorts` bzw. `@OutPorts` verwendet werden, welche die einzelnen Port-Definitionen enthält. Listing 3.2 zeigt ein Beispiel.

Listing 3.1: Port-Annotationen an einer FBP-Komponentenklasse (1)

```
1 @InPort("IN")
2 @InPort(value = "ARRAYIN", arrayPort = true)
3 @OutPort("OUT")
4 @OutPort(value = "ARRAYOUT", arrayPort = true)
5 public class FBPComponent1 extends Component {
6   ...
7 }
```

Listing 3.2: Port-Annotationen an einer FBP-Komponentenklasse (2)

```
@InPorts({
  @InPort("IN1"),
  @InPort("IN2")
  })
@OutPorts({
  @OutPort("OUT1"),
  @OutPort(value = "OUT2", array = true)
})
public class FBPComponent2 extends Component {
  ...
}
```

Implementierung einer FBP-Komponente

Eine FBP-Komponente muss von der Klasse `Component` aus dem JavaFBP-Framework abgeleitet werden. Dabei müssen die Methoden `execute` und `openPorts` überschrieben werden, damit die Komponente sinnvoll arbeiten kann. Listing 3.3 zeigt das Grundgerüst einer FBP-Komponente (vergleiche [Ste09, S. 17]).

Listing 3.3: Grundgerüst einer FBP-Komponente

```
@InPort("IN")
@OutPort("OUT")
public class SampleFBPComponent extends Component {
  private InputPort inputPort = null;
  private OutputPort outputPort = null;

  /* Hauptverarbeitungsschleife */
  @Override
  protected void execute() {
    ... // die Ports koennen hier verwendet werden
  }

  /* Oeffnet die Ports */
  @Override
  protected void openPorts() {
    this.inputPort = super.openInput("IN");
    this.outputPort = super.openOutput("OUT");
  }
}
```

Die Methode `openPorts` öffnet die definierten Ports. Die Methode wird vom JavaFBP-Framework nach dem Erzeugen einer Komponenteninstanz als erste aufgerufen. Bei den Methoden `super.openInput` und `super.openOutput` muss der Name des Ports als Parameter angegeben werden, welcher mit dem in der Annotation angegebenen identisch sein muss. An dieser Stelle ist eine Schwäche des JavaFBP-Frameworks ganz klar zu erkennen, und zwar ist die Namensangabe der Ports redundant. An dieser Stelle sei auf Abschnitt 5.2 verwiesen, in dem auf Nachteile und mögliche Verbesserungen genauer eingegangen wird.

Bei der Methode `execute` handelt es sich um die Hauptverarbeitungsmethode der Komponente. Typischerweise enthält diese eine Schleife, die am Eingangsport ankommende Pakete liest und verarbeitet, bis der Port vom Framework geschlossen wird. Das Framework schließt den Port typischerweise, wenn diejenige Komponente, die Daten an den Eingangsport sendet, terminiert oder den Port schließt. Wie oben bereits erwähnt, gibt es nicht nur Verarbeitungskomponenten, sondern auch solche, die das Netz mit Daten speisen. Diese Art von Komponenten haben normalerweise keinen Eingangsport, da sie Produzenten bzw. die Datenquelle sind. Deshalb initiieren diese Komponenten in den meisten Fällen das sukzessive Schließen der Ports und damit das Beenden der Komponenten, bis das ganze Netzwerk terminiert. Analog haben Komponenten, die als Datensenke dienen, keinen Ausgangsport (siehe auch [Ste09, S. 4]).

3.2.2 Informationspakete

Informationspakete (engl.: *Information Packets*), kurz IPs, werden die Datenpakete genannt, die über Verbindungen von Komponente zu Komponente wandern. Dabei sind die Daten, die ein solches Paket enthalten kann, vom Typ `Object` und somit beliebig (siehe [Ste09, S. 6, Abschnitt „1.3.2 Asynchronous Message Passing“]).

Listing 3.4 zeigt, wie ein Informationspaket `p` mit einer Instanz von `Object` über einen Ausgangsport gesendet und der Ausgangsport danach geschlossen wird.

Listing 3.4: Senden eines Informationspakets in einer FBP-Komponente

```
1 @OutPort("OUT")
2 public class FBPComponent1 extends Component {
3   private OutputPort outputPort = null ;
```

```
4
5   @Override
6   void execute() {
7     Object o = new Object();
8     Packet p = this.create(o);
9     outputPort.send(p);  // Paket senden
10    outputPort.close();  // Port schliessen
11  }
12
13  @Override
14  protected void openPorts() {
15    this.outputPort = super.openOutput("OUT");
16  }
17 }
```

3.2.3 Verbindungen

FBP-Verbindungen verknüpfen eine Komponente oder ein Subnetz mit einer anderen Komponente oder einem anderen Subnetz. FBP-Verbindungen sind gemeinsam genutzte Datenstrukturen. Sie sind realisiert als blockierende FIFO-Puffer. Versucht ein Konsument (*Input Port*) aus einem leeren Puffer zu lesen, blockiert dieser, bis ein Informationspaket eintrifft oder der Port geschlossen wird. Analog blockiert ein Produzent (*Output Port*), wenn ein solcher Puffer voll ist, bis wieder Platz im Puffer ist (siehe [Ste09, S. 6–7]). Logischerweise können nur Ausgangsports zu Eingangsports verbunden werden, nicht umgekehrt.

3.2.4 Subnetze

Subnetze bestehen, wie oben angesprochen, aus mehreren miteinander verbundenen Komponenten. Nach außen hin sieht ein Subnetz aus wie eine normale Komponente und hat auch eigene Ports. Den Subnetz-Ports können Ports von Komponenten, die sich im Subnetz befinden, zugewiesen werden. Hierbei dürfen Subnetz-Eingangsports nur mit Komponenten-Eingangsports und Subnetz-Ausgangsports nur mit Komponenten-Ausgangsports verbunden werden. Diese Zuweisung ist also keine FBP-Verbindung mit FIFO-Puffer wie im vorigen Abschnitt beschrieben. Subnetze lassen sich beliebig ineinander verschachteln.

> That means, one can build a component out of several components. Such a component is called a "subnet" in FBP terminology. Ports of the subnet will be assigned to ports of the inner components. Such a subnet can be used like a normal component in a network. The user of the subnet does not need to know he is using a subnet rather than a normal component. [Ste09, S. 6]

3.3 Java-basierte FBP-Netzwerkerstellung

FBP-Netzwerke mittels Java-Code zu entwerfen war früher die einzige Möglichkeit, FBP-Netzwerke zu erstellen. Inzwischen existiert eine weitere Möglichkeit mittels XML, welche in Abschnitt 3.4 beschrieben wird. Um ein FBP-Netzwerk mittels Java-Code zu erstellen, muss eine eigene Klasse angelegt werden, die von der abstrakten Klasse `Network` des JavaFBP-Frameworks erbt (vergleiche [Ste09, S. 4–5, Abschnitt „2.1.1 The FBP Language of JavaFBP"]). Diese Klasse kann natürlich auch anonym sein. Die Methode `define` muss dabei implementiert werden. Es existieren drei Methoden, mit Hilfe derer ein Netzwerk beschrieben werden kann:

1. `component(String name, Class type)`
2. `connect(String sender, String receiver)`
3. `initialize(Object content, String receiver)`

Erstere erstellt eine Komponente und erwartet als Parameter den Namen, den die Komponente im FBP-Netzwerk bekommen soll, und den Klassentyp der Komponente. Beispiel:

```
component("reader", ListReader.class)
```

Zweitere stellt eine Verbindung zwischen zwei Komponenten her und erwartet die beiden `String`-Parameter `sender` und `receiver`, welche in folgendem Format angegeben werden: `<komponentenname>.<portname>`. Dabei müssen die Komponenten wirklich einen Port mit diesem Namen besitzen und zuvor mit `component(...)` angelegt worden sein. Beispiel:

```
connect("reader.OUT", "writer.IN")
```

`<komponentenname>` bezieht sich auf den Parameter `name` der Methode `component`. Mit Hilfe der Methode `initialize` kann ein Parameter vom Typ `Object` an eine Komponente übergeben werden. Beispiel, um ein Array mit drei `String`s zu übergeben:

```
initialize({ "v1", "v2", "v3" }, "reader.CONFIG")
```

Ein Beispiel für ein komplettes FBP-Netzwerk zeigt Listing 3.5. Im Listing wird davon ausgegangen, dass die zwei FBP-Komponentenklassen – *PacketProducer* und *PacketConsumer* – existieren und folgende Eigenschaften haben: Erstere hat einen Konfigurationsport *CONFIG* und einen Ausgangsport *OUT*, letztere hat nur einen Eingangsport namens *IN*. *PacketProducer* liest ein Array mit Daten am Konfigurationsport und sendet die einzelnen Elemente als separate Informationspakete über den Ausgangsport. *PacketConsumer* schreibt die gelesenen Daten der einzelnen Informationspakete auf die Standardausgabe.

Listing 3.5: Java-basiertes Erstellen eines FBP-Netzwerks

```
1  public class FBPNetworkJava {
2    public static void main(String[] args) {
3      Network net = new Network() {
4        public Component producer;
5        public Component consumer;
6
7        @Override
8        protected void define() throws Exception {
9          /*
10          * FBP-Komponente "PacketProducer" mit
11          * Ausgangsport "OUT" und
12          * Konfigurationsport "CONFIG"
13          */
14         producer = component("producer", PacketProducer.
               class);
15
16         /*
17          * FBP-Komponente "PacketConsumer" mit
18          * Eingangsport "IN"
19          */
20         consumer = component("consumer", PacketConsumer.
               class);
21
22         /* Verbindungen erstellen */
23         connect("producer.OUT", "consumer.IN");
24
25         /* "PacketProducer" initialisieren */
26         initialize({ "PK1", "PK2", "PK3" }, "producer.
               CONFIG");
```

```
27          }
28        };
29        net.go();
30      }
31
32  }
```

In der `main`-Methode wird ein `Network`-Objekt mit Hilfe einer anonymen Klasse erstellt. In der anonymen Klasse werden 2 Variablen für FBP-Komponenten angelegt – `producer` und `consumer`. In der implementierten Methode `define` wird den angelegten Komponentenvariablen dann je ein `Component`-Objekt zugewiesen mit entsprechendem Namen und Typ. Danach wird eine Verbindung zwischen den beiden Komponenten erstellt, und zwar vom Ausgangsport von `producer` zum Eingangsport von `consumer`. Am Ende der Methode `define` wird die `producer`-Komponente noch mit einem String-Array initialisiert. Listing 3.6 zeigt die Ausgabe des Netzwerks bzw. der `consumer`-Komponente.

Listing 3.6: Bildschirmausgabe der `consumer`-FBP-Komponente

```
1 PK1
2 PK2
3 PK3
```

3.4 XML-basierte FBP-Netzwerkerstellung

Die XML-basierte Erstellung von FBP-Netzwerken wurde von Sven Steinseifer eingeführt und bietet zwei Vorteile (siehe [Ste09, S. 6, Abschnitt „1.2.3 Data Grouping“]). Zum einen ist sie programmiersprachenunabhängig und zum anderen kann die Netzwerkbeschreibung menschenlesbar serialisiert werden. Das ist wichtig, denn Netzwerke werden auf dem Client erstellt und zur Berechnung an den Server gesendet. Treten bei der Ausführung des Netzwerks irgendwelche Fehler auf, kann die Netzwerkbeschreibung leichter auf Fehler hin überprüft werden (Debugging) als es mit binär serialisierten Objekten möglich wäre.

Bei der Einführung von XML-Netzwerkbeschreibungen wurde der Mangel an einer Struktur für Komponenteninitialisierungsdaten gleich mit beseitigt. Seitdem ist die interne Struktur für Initialisierungsdaten vom Typ `HashMap<String, Object>` anstatt nur vom Typ `Object`. Somit ist es möglich, Initialisierungsparameter mit Name und Wert zu erstellen. Für Werte

existieren in der XML-Beschreibung diverse Typen, die vom XML-Parser beachtet werden. Diese Typen sind folgende:

- `INT`
- `FLOAT`
- `STRING`
- `BOOL`
- `LONG`
- `DOUBLE`
- `DATE`

Der XML-Parser wandelt die Werte anhand dieser Typangabe in entsprechende Java-Typen um. Außerdem wurde der Typ `LIST` eingeführt, damit Listenstrukturen erstellt werden können. Vom XML-Parser wird eine solche Liste in ein Objekt vom Typ `java.util.List` umgewandelt. Eine Listenstruktur kann beliebig tief geschachtelt werden, was bedeutet, dass ein Eintrag einer Liste wieder eine Liste sein kann. Somit ist es möglich, einfache hierarchische Strukturen zu realisieren. Wie normale Initialisierungsparameter auch, hat eine Liste einen Typen für ihre Einträge.

In Listing 4.1 ist eine XML-Netzwerkbeschreibung zu sehen, welche funktionell analog zu der in Listing 3.5 beschriebenen Java-basierten ist, mit einer Ausnahme: die Initialisierung der `producer`-Komponente geschieht nicht mittels Array, sondern mit Hilfe einer Liste. Der Initialisierungsparameter in Listing 3.5 heißt `VALUES` und hat als Wert eine Liste vom Typ `STRING`. Des Weiteren ist anzumerken, dass die Information über Ports einer Komponente im XML-Tag `<connection>` zu finden ist, anstatt im `<component>`-Tag. Ansonsten sollte die XML-Beschreibung selbsterklärend sein.

Listing 3.7: XML-Beschreibung eines FBP-Netzwerks

```
<network>
  <structure>
    <component type='PacketProducer' name='producer' />
    <component type='PacketConsumer' name='consumer' />
    <connection sender='producer' sourcePort='OUT'
          receiver='consumer' destinationPort='IN' />
    <init component='producer' port='CONFIG'>
      <parameter name='VALUES'>
        <list type='STRING'>
          <item>PK1</item>
          <item>PK2</item>
          <item>PK3</item>
        </list>
      </parameter>
    </init>
  </structure>
```

```
</network>
```

Wie eine solche XML-Beschreibung im PARASUITE-System lauffähig gemacht werden kann, zeigt Listing 3.8.

Listing 3.8: XML-basiertes Erstellen eines FBP-Netzwerks

```
public class FBPNetworkXML {
  public static void main(String[] args) {
    String networkDescription = ... ;

    /* Verbindung zur Middleware herstellen */
    ParasuiteMiddleware.setPROVIDER_URL("localhost");
    ParasuiteMiddleware.connect();

    /* Netzwerk abarbeiten */
    DataflowResult result = (DataflowResult)
        ParasuiteMiddleware
        .execMAction(MActions.DATAFLOW,
            networkDescription);

    /* Verbindung zur Middleware trennen */
    ParasuiteMiddleware.disconnect();
  }

}
```

In der `main`-Methode wird zuerst eine String-Variable angelegt, die die XML-Netzwerkbeschreibung enthält. Diese entspricht der in Listing 4.1 gezeigten und wurde hier der Übersichtlichkeit halber weggelassen. Um das so beschriebene FBP-Netzwerk nun im PARASUITE-Backend auszuführen, muss zuerst eine Verbindung zur PARASUITE-Middleware aufgebaut werden. Danach wird die Middleware-Aktion zum Ausführen eines FBP-Netzwerks aufgerufen: `execMAction(MActions.DATAFLOW, networkDescription)`. `MAction.DATAFLOW` gibt dabei an, dass es sich bei der auszuführenden Aktion um eine Ausführung eines FBP-Netzwerks handelt. Nach Abschluss der Aktion wird die Verbindung zur Middleware getrennt.

Kapitel 4

Konzeption und Implementierung einer Datenbanklesekomponente

Dieses Kapitel beschreibt die Erstellung einer Datenbanklesekomponente für das neue Datenbankschema „PARASUITE2“ und stellt somit den Hauptteil dieser Arbeit dar. Die in Abschnitt 1.2 gestellten Anforderungen an Konfigurierbarkeit, Generizität und Menschenlesbarkeit fließen dabei maßgeblich in die Entwicklung mit ein. Dieses Kapitel geht zuerst auf den Aufbau und die Arbeitsweise der Datenbanklesekomponente ein, um dann das Hauptaugenmerk auf die Konfiguration der Komponente zu richten und dort insbesondere auf die Filterkonfiguration.

4.1 Aufbau und Arbeitsweise der Datenbanklesekomponente

Die Datenbanklesekomponente benötigt nur zwei Ports, einen Konfigurationsport und einen Ausgangsport. Der Konfigurationsport namens *CONFIG* enthält dabei die komplette Konfiguration der Komponente. Über den Ausgangsport namens *OUT* sendet die Komponente die aus der Datenbank gelesenen Daten. Dabei wird jeder Datensatz als eigenes Informationpaket gesendet, um der nebenläufigen Verarbeitung gerecht zu werden. Würden die gesamten Daten in einem Informationspaket gesendet werden, würde die Verarbeitung annähernd sequentiell anstatt nebenläufig ablaufen (siehe [Ste09, S. 6]). Abbildung 4.1 enthält eine schematische Darstellung vom Aufbau der Datenbanklesekomponente. Da Komponenten im PARASUITE-System englische Namen haben, heißt die Java-Klasse der Datenbanklesekomponente `DatabaseReader2`. Das Anhängsel „2“ rührt daher, dass für das alte Datenbankschema „PARASUITE1“ schon eine Komponente mit diesem Namen existierte.

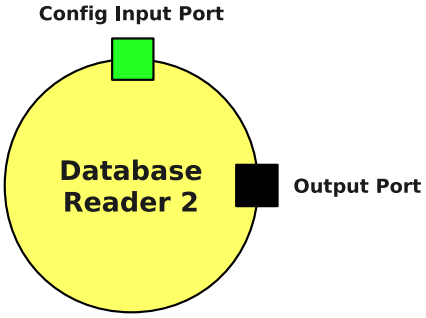

Abbildung 4.1: Schemadarstellung der Datenbanklesekomponente

Die Arbeitsweise der Datenbanklesekomponente lässt sich der Reihe nach in folgenden Schritten beschreiben:

1) Lesen der Konfiguration vom Konfigurationsport (*CONFIG*)
2) Verarbeiten der Konfiguration
3) Schließen des Konfigurationsports
4) Hauptverarbeitungsschleife
   Ende-Bedingung: alle Datensätze sind gelesen
   a) Lesen eines Datensatzes aus der Datenbank
   b) Datensatz transformieren und in ein Informationspaket stecken
   c) Informationspaket über den Ausgangsport *OUT* senden
5) Schließen des Ausgangsports

4.2 Konfiguration der Datenbanklesekomponente

Die Konfiguration spielt bei der Datenbanklesekomponente eine zentrale Rolle. Mit ihr kann das Verhalten der Komponente entscheidend verändert werden. Die Datebanklesekomponente erwartet 6 Konfigurationsparameter, welche in Tabelle 4.1 aufgelistet sind. Die erste Spalte gibt dabei den Namen des Konfigurationsparameters an. Die zweite Spalte gibt die Syntax des Konfigurationsparameters in Form eines in Java gebräuchlichen regulären

Ausdrucks (REGEX[1]) an (siehe [Suna]). Die dritte Spalte zeigt ein Beispiel. Die Konfigurationsparameter `COLUMNS` und `ORDER` sind jeweils Listen. Der zugehörige REGEX und das Beispiel stehen deshalb für einen Listeneintrag.

Name	Syntax (REGEX)	Beispiel
LOCALE	`^[a-z]{2}_[A-Z]{2}$`	`de_DE`
TABLE	`^[a-zA-Z_0-9]+$`	`LOCOMOTIVE`
COLUMNS	`^[a-zA-Z_0-9]+$`	`Time`
FILTER	n/a[a]	–
ORDER	`^[a-zA-Z_0-9]+\((ASC|DESC)\)$`	`Time(ASC)`
DISTINCT	`^(true|false)$`	`true`

[a]Ein Filterausdruck ist ein komplexerer Ausdruck, der nicht mit einem regulären Ausdruck beschrieben werden kann. Filtern ist u. a. deshalb ein eigener Abschnitt (4.3) gewidmet.

Tabelle 4.1: Konfigurationsparameter der Datenbanklesekomponente

Diese Konfigurationsparameter sind von der Bedeutung äquivalent mit denen in Tabelle 2.1 (Seite 26). Jedoch sind die Namen unterschiedlich. Folgende Entsprechungen gelten:

- *LOCALE* → locale
- *TABLE* → entity
- *COLUMNS* → desiredTypes
- *FILTER* → filterExpressionGroup
- *ORDER* → sortTypes
- *DISTINCT* → distinct

In Listing 4.1 ist ein Beispiel einer Konfiguration in XML zu sehen. Die Beschreibung ist keine komplette FBP-Netzwerkbeschreibung und enthält nur den zur Konfiguration relevanten Teil. Außerdem enthält die abgebildete Konfiguration keinen Filterausdruck; Filterausdrücke werden in Abschnitt 4.3 behandelt.

Listing 4.1: Beispiel einer XML-Konfiguration der Datenbanklesekomponente

```
1 ...
2   <init component='dbreader2' port='CONFIG'>
3     <parameter name='LOCALE' type='STRING'>
4       de_DE
5     </parameter>
```

[1]Von REGular EXpression. Siehe `https://wwwbs.informatik.htw-dresden.de/fbs/regex/fr1regex.html`

```
6     <parameter name='TABLE' type='STRING'>
7       LOCOMOTIVE
8     </parameter>
9     <parameter name='COLUMNS'>
10      <list type='STRING'>
11        <item>Locomotive_ID</item>
12        <item>Time</item>
13      </list>
14    </parameter>
15    <parameter name='FILTER' type='STRING'>
16      <!-- ... Filter-String ... -->
17    </parameter>
18    <parameter name='ORDER'>
19      <list type='STRING'>
20        <item>Time(ASC)</item>
21        <item>Locomotive_ID(DESC)</item>
22      </list>
23    </parameter>
24    <parameter name='DISTINCT' type='BOOL'>
25      true
26    </parameter>
27  </init>
28 ...
```

Somit ist es möglich, aus verschiedenen Tabellen unterschiedliche Spalten zu lesen, die Daten zu filtern und zu sortieren. Außerdem werden die Daten in einer bestimmten Lokalisierung abgefragt und das Beachten bzw. Ignorieren von doppelten Datensätzen kann eingestellt werden. Mit einer solchen Konfigurationsmöglichkeit ist die Datenbanklesekomponente gemäß Anforderung besonders generisch gehalten.

4.3 Filterkonfiguration der Datenbanklesekomponente

Die Datenbanklesekomponente verwendet intern die in 2.5.2 und 2.5.3 beschriebenen Datenbankschnittstellen „ParasuiteEntityDAOBean“ und „Value List Handler“. Um Daten aus der Datenbank zu lesen, verwendet die Datenbanklesekomponente die Methode `executeSearch` des Value List Handlers. Der komplexeste Parameter dieser Methode ist `filterExpressionGroup` vom

Typ `FilterExpressionGroup`, im Folgenden einfach „Filtergruppe“ genannt. Dieser Parameter definiert Filterbedingungen, die beim Abfragen der Daten aus der Datenbank auf die Daten angewendet werden. FBP-Komponenten und deren Konfiguration sind dabei Bestandteil einer FBP-Netzwerkbeschreibung, welche in XML erstellt wird (vergleiche 3.4). Als Bestandteil der Komponentenkonfiguration, muss eine Filtergruppe demnach auch textuell serialisierbar sein. Außerdem besteht die Anforderung nach Menschenlesbarkeit (siehe 1.2).

In diesem Unterkapitel werden verschiedene Ansätze und Lösungen zum Problem der Serialisierung und entsprechenden Deserialisierung von Filtergruppen vorgestellt. Dabei wird zunächst genauer auf Filtergruppen und deren Struktur eingegangen, bevor die verschiedenen Serialisierungs- und Deserialisierungsmethoden beschrieben werden.

4.3.1 Was sind Filtergruppen?

Eine Filtergruppe ist Teil einer hierarchischen Gesamtstruktur, die Filterausdrücke repräsentiert. Eine zentrale Eigenschaft einer Filtergruppe ist, dass sie leicht in eine SQL-`WHERE`-Bedingung transformiert werden kann. Diese Eigenschaft ist wichtig, da der Value List Handler intern per SQL auf die Datenbank zugreift. Abbildung 4.2 zeigt ein UML-Diagramm der hierarchischen Gesamtstruktur, zu der auch eine Filtergruppe gehört.

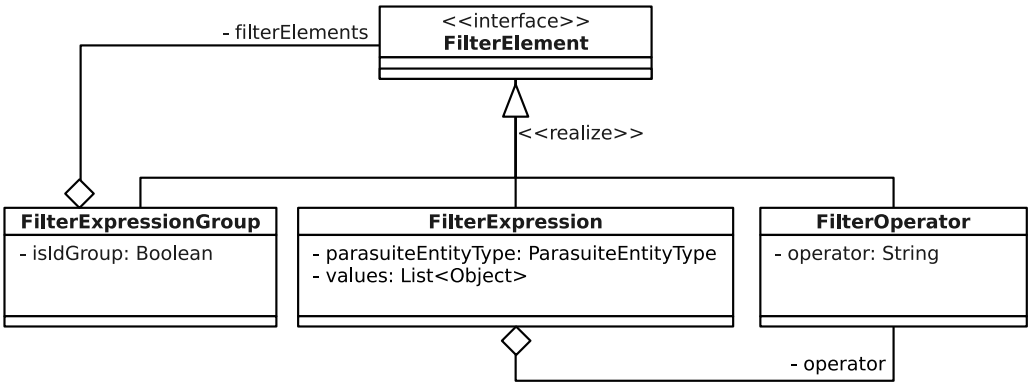

Abbildung 4.2: UML-Klassendiagramm – Hierarchische Struktur für Filterausdrücke

Im UML-Diagramm sind die drei Klassen `FilterExpressionGroup`, `FilterExpression` und `FilterOperator` zu sehen, die alle das gleiche In-

terface `FilterElement` implementieren. Im folgenden werden die Klassen beschrieben.

FilterOperator Die Klasse `FilterOperator` deklariert eine Variable `operator`, welche einen der folgenden Operatoren als `String` enthält:

- `=`
- `!=`
- `<`
- `<=`
- `>`
- `>=`
- `BETWEEN`
- `NOT BETWEEN`
- `IN`
- `NOT IN`
- `IS NULL`
- `AND`
- `OR`

Der Konstruktor stellt dabei sicher, dass nur die gerade genannten Operatoren angegeben werden können. Des Weiteren ist anzumerken, dass von dieser Klasse logische und Vergleichsoperatoren gleichermaßen definiert sind. `AND` und `OR` sind dabei die logischen Operatoren, alle anderen sind Vergleichsoperatoren. Schöner wäre es, diese Operatoren kategorisch in zwei Klassen aufzuteilen (z. B. `FilterOperatorLogical` und `FilterOperatorComparation`).

FilterExpression Die Klasse `FilterExpression` repräsentiert einen einzelnen Filterausdruck (im Folgenden auch Einzelfilterausdruck genannt) nach dem Muster `<operand> <operator> <wert(e)>`. Dabei entspricht `<operand>` einer Tabellenspalte (Variable `parasuiteEntityType`). `<operator>` ist vom gerade vorgestellten Typ `FilterOperator` (Variable `operator`) und muss ein Vergleichsoperator sein. `<wert(e)>` enthält – je nach Operator – keinen, einen oder mehrere Werte als Liste vom Typ `Object` und kann somit beliebige Typen aufnehmen. Der exakte Typ variiert je nachdem, welcher Typ in `parasuiteEntityType` angegeben ist (siehe 2.5.1). Da dies unterschiedliche Typen sein können, ist die Variable `values` vom Typ `List<Object>` und kann somit beliebige Typen aufnehmen. Ein Beispiel für einen einzelnen Filterausdruck kann textuell ausgedrückt etwa so aussehen:

$$\underbrace{\texttt{LOCOMOTIVE.Date}}_{\text{Operand}} \quad \underbrace{\texttt{BETWEEN}}_{\text{Operator}} \quad \underbrace{\texttt{(2009-01-01, 2009-08-31)}}_{\text{Werte}}$$

FilterExpressionGroup Die Klasse `FilterExpressionGroup` repräsentiert eine Gruppe von Filterausdrücken (im Folgenden einfach Filtergruppe genannt). Ein Filterausdruck kann dabei vom Typ `FilterExpression` sein oder wieder vom Typ `FilterExpressionGroup`. Die einzelnen Filterausdrücke sind dabei durch Operatoren (`FilterOperator`) miteinander verknüpft. Dabei sind nur die logischen Operatoren `AND` und `OR` von `FilterOperator` erlaubt. Außerdem besitzt die Klasse `FilterExpressionGroup` ein Feld `isIdGroup` vom Typ `Boolean`. Dieses Feld gibt an, ob es sich um eine Filtergruppe handelt, die als Filterbedingung eindeutige Bezeichner („IDs“) einer Tabelle prüft. Ein Beispiel in SQL könnte so aussehen:

```
select * from X where id in (10002, 10005, 10008, ...)
```

Für eine Datenbankabfrage ist das `isIdGroup`-Feld nicht von Bedeutung, dafür aber für die PARASUITE-GUI. Dort können Datensätze aus einer Liste ausgewählt und ein Filter kann mittels eines Filter-Dialogs auf diese angewendet werden (im nächsten Abschnitt werden Filter-Dialoge näher beschrieben). Die Menge ausgewählter Datensätze kann sehr groß sein und damit auch die Anzahl der IDs in der Filterbedingung. Da die GUI eine kurze textuelle Repräsentation der Filterbedingung anzeigt, ist es wichtig zu wissen, wann es sich um eine Filterbedingung handelt, die eindeutige Bezeichner prüft, und wann nicht. Mit dieser Information kann im Falle einer langen Liste mit IDs eine verkürzte textuelle Repräsentation dieser Liste dargestellt werden (siehe nächster Abschnitt).

Filter-Dialog der PARASUITE-GUI

In der PARASUITE-GUI existiert ein Filter-Dialog, in dem Filtergruppen interaktiv zusammengestellt werden können, wie in Abbildung 4.3 zu sehen. Die dahinter liegende Struktur ist die gerade vorgestellte (Abschnitt 4.3.1).

Auf diese Weise kann ein Filterausdruck bzw. eine Filtergruppe für einen Benutzer von PARASUITE bequem erstellt werden. Zum Verfassungszeitpunkt dieser Bachelorarbeit befindet sich eine grafische Oberfläche zum Erstellen von FBP-Netzwerken bereits in Entwicklung. Dabei entstehen auch Konfigurationsdialoge für FBP-Komponenten, die es ermöglichen, Komponenten interaktiv zu konfigurieren. Der Filter-Dialog enthält im unteren Teil des Fensters eine kurze textuelle Repräsentation des zusammengebauten Filters. Hier ist auch zu sehen, dass eine Filterbedingung, die nur eindeutige Bezeichner prüft dargestellt wird als `(..ID CONDITIONS..)`.

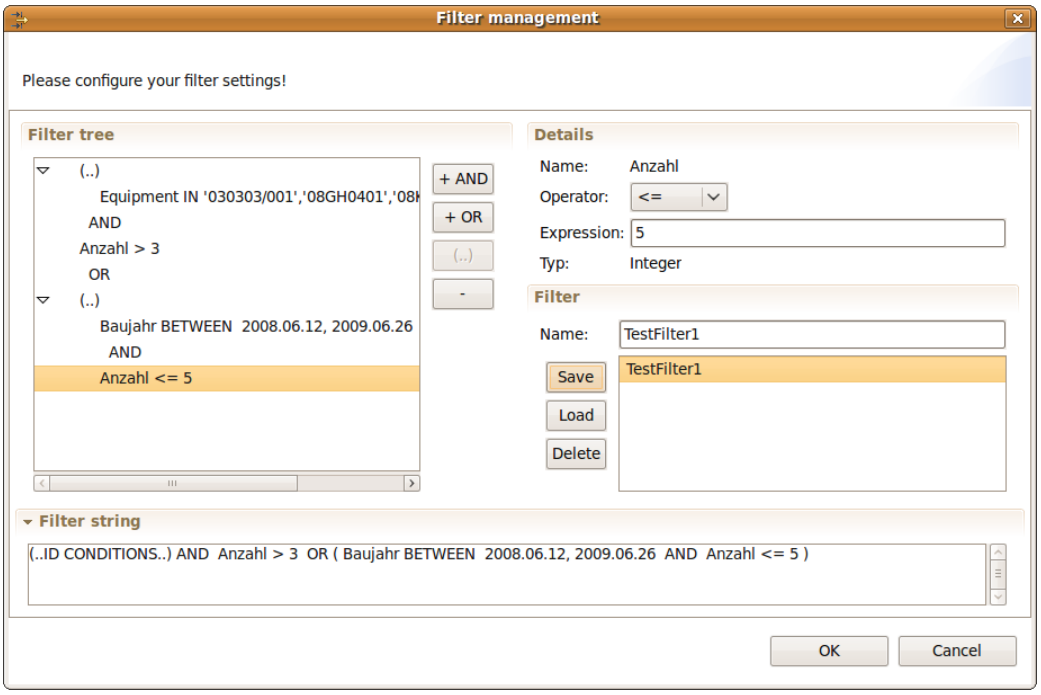

Abbildung 4.3: Screenshot – Filter-Dialog in der PARASUITE-GUI

4.3.2 Serialisierung einer Filtergruppe mittels BASE64

Zu Beginn der Entwicklung bestand die Anforderung nach menschenlesbarer, textueller Serialisierung einer Filtergruppe noch nicht, wodurch die Möglichkeit einer Kodierung mittels BASE64 vorhanden war. Mittels BASE64 kann ein beliebiges Binärobjekt in eine textuelle Repräsentation umgewandelt werden (vergleiche [url]). Somit ist es möglich, eine binär serialisierte Filtergruppe in BASE64 umzuwandeln und innerhalb einer FBP-Netzwerkbeschreibung übers Netz zum Server zu senden. Auf dem Server kann die Filtergruppe dann von BASE64 zurück in binär gewandelt und deserialisiert werden. Im praktischen Einsatz im PARASUITE-System trat allerdings nach einiger Zeit ein Versionskonflikt auf, der damit zusammenhing, dass ein alter Client eine veraltete Version einer Filtergruppe zum Server sendete. Somit konnte die Filtergruppe nicht korrekt deserialisiert und das FBP-Netzwerk nicht ausgeführt werden. Aus diesem Grund wurde eine neue Anforderung an den Datenbankleseknoten gestellt, und zwar die menschenlesbare, versionsunabhängige Serialisierung. Somit scheidet die Serialisierung einer Filtergruppe mittels BASE64 aus.

4.3.3 Serialisierung einer Filtergruppe mittels SQL

Zum Umwandeln einer Filtergruppe in eine SQL-`WHERE`-Bedingung existiert in PARASUITE eine eigene Utility-Klasse namens `FilterSQLPrinter`. Eine von dieser Klasse erzeugte SQL-Bedingung am Beispiel des Kunden Bombardier zeigt Listing 4.2.

Listing 4.2: SQL-Filterausdruck

```
1 LOCOMOTIVE.Locomotive_ID IN ('021100101-1')
2 AND
3 (
4   NOTIFICATION.Time NOT BETWEEN
5       '20:10:53' AND '20:37:12'
6 )
```

Mit SQL steht damit bereits eine textuelle Serialisierung, die menschenlesbar ist, zur Verfügung. Das Problem, das sich hier ergibt, ist allerdings die Rückwandlung von SQL in eine Filtergruppe (Deserialisierung). Für diese Aufgabe ist ein SQL-Parser nötig. In den folgenden zwei Unterabschnitten werden zwei Ansätze beschrieben, die versuchen, dieses Problem zu lösen. Der erste Ansatz ist die Implementierung mittels eines nichtdeterministischen endlichen Automaten. Der zweite Ansatz versucht dies mittels selbstgebautem Scanner und Parser.

Deserialisierung mittels nichtdeterministischem endlichen Automaten

Der erste Ansatz der Deserialisierung einer Filtergruppe ist das Entwerfen eines nichtdeterministischen endlichen Automaten (NEA), der die SQL-`WHERE`-Bedingung in einen Token-Strom umwandelt. Dieser Token-Strom wird dann einem Parser zur Verfügung gestellt, welcher die weitere Verarbeitung vornimmt. Abbildung 4.4 zeigt einen in JFLAP[2] entworfenen NEA, der Zustände und Übergänge zwischen den Zuständen mit dem Zweck darstellt, bestimmte Tokens zu erkennen. Da die Erstellung eines solchen Automaten ein sehr zeitaufwändiger und fehleranfälliger Prozess ist und Änderungen oft größeren Aufwand erfordern, wurde schnell von diesem Ansatz abgelassen und ein

[2] JFLAP steht für „Java Formal Languages and Automata Package“ und ist ein Tool, um u. a. endliche Automaten grafisch zu entwerfen. Für nähere Informationen und den vollen Funktionsumfang siehe `http://www.jflap.org/`.

effizienterer Ansatz verfolgt, welcher Gebrauch von regulären Ausdrücken in Java macht und im nächsten Abschnitt beschrieben wird.

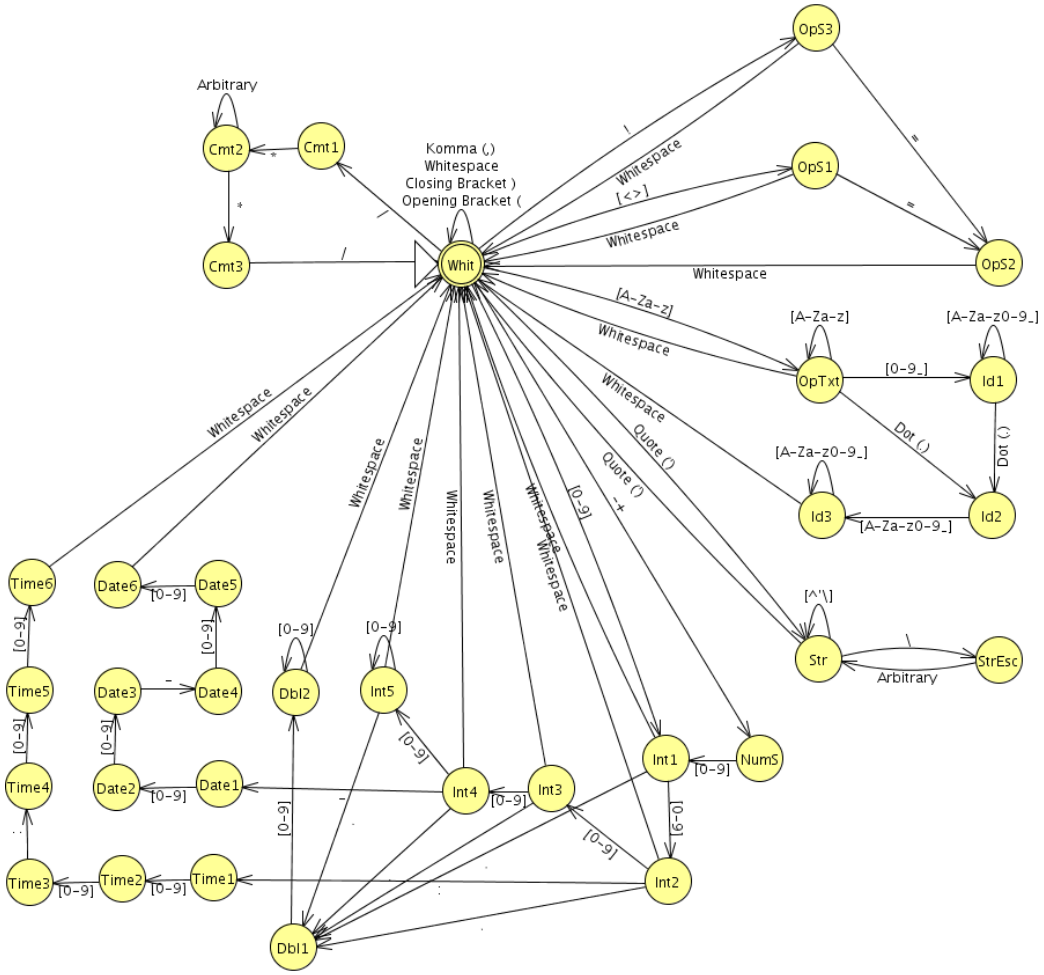

Abbildung 4.4: Screenshot – Tokenizer (NEA)

Deserialisierung mittels selbstgebautem Scanner und Parser

Der zweite Ansatz, die SQL-`WHERE`-Bedingung in eine Filtergruppe zurückzuwandeln, ist die Erstellung eines Scanners mit Hilfe von regulären Ausdrücken in Java und einem Parser, der den vom Scanner erzeugten Token-Strom verarbeitet. Der Scanner ist dabei generisch entworfen, sodass beim Hinzufügen, Ändern und Entfernen von Tokens, die der Scanner in der Lage ist zu erkennen, keine Code-Änderungen am Scanner nötig sind. Tokens sind

hierbei als entkoppelte `Enum`-Klasse realisiert. Der Parser konsumiert die vom Scanner erzeugten Tokens und wendet die entsprechenden Produktionen an. Das Ergebnis des Parsers ist ein abstrakter Syntaxbaum. Bei Filtergruppen ohne lokalisierte Werte ist das Parsen kein Problem. Bei lokalisierten Werten kann es im SQL-Code allerdings Unterabfragen auf eine Lokalisierungstabelle (siehe Abschnitt 2.3) geben. Ein Beispiel einer solchen Unterabfrage ist in Listing 4.3 zu sehen. Zum besseren Verständnis ist in diesem Beispiel nicht nur die SQL-`WHERE`-Bedingung angegeben, sondern die komplette SQL-Abfrage.

Listing 4.3: SQL-Unterabfrage in einer Filtergruppe

```
1  /*
2   * Alle Lokomotiven, die rote Farbe haben
3   */
4  select
5    *
6  from
7      LOCOMOTIVE
8  where
9    Color =
10   (
11     select ID
12     from LOCOMOTIVE_LOCALIZED_COLUMN_Color
13     where de_DE = 'rot'
14   )
```

Der SQL-Parser scheitert an dieser Stelle, da die entscheidende Information – hier der Wert `red` aus der Spalte *`Color`* der Tabelle *`LOCOMOTIVE`* – nicht direkt im SQL-Code steht. Diese Information wird indirekt durch eine Unterabfrage anhand des lokalisierten deutschen Wertes `rot` aus der Datenbank beschafft. Für den Parser wäre es nur möglich, an diese Information zu kommen, indem er selbst eine direkte SQL-Abfrage auf der Datenbank absetzt. Dies ist natürlich zu vermeiden, da das PARASUITE-System geeignete Datenbankschnittstellen (siehe Abschnitt 2.5), die von der wirklichen Implementierung abstrahieren, zur Verfügung stellt und von allen Komponenten in PARASUITE, die mit Daten der Datenbank arbeiten, benutzt werden sollen.

Ein weiteres scheinbares Problem ist das Fehlen von zusätzlichen Informationen über Filtergruppen (siehe Abschnitt 4.3.1) und alle beteiligten Klassen, wie `ParasuiteEntity` und `ParasuiteEntityType` (siehe Abschnitt 2.5.1). Die einzige Information, die der SQL-Code über eine Tabelle bzw. `ParasuiteEntity` liefert, ist die Information über den Tabellen-

namen. Dieser muss aber in der Datenbank, und somit im PARASUITE-System auch, eindeutig sein. Infolgedessen kann mittels der Methode `getParasuiteEntity` der `ParasuiteEntityDAOBean` (siehe Abschnitt 2.5) ein gültiges `ParasuiteEntity`-Objekt anhand des Tabellennamens erfragt werden. Für `ParasuiteEntityType`-Objekte gilt dies analog. Der Parser benötigt somit zum Deserialisieren einer Filtergruppe Informationen aus der Datenbank, welche aber leicht mit Hilfe der Datenbankschnittstellen beschafft werden können. Für Filtergruppen (Klasse `FilterExpressionGroup`) fehlt im SQL-Code allerdings die Information `isIdGroup`. Diese kann theoretisch vom `FilterSQLPrinter` mit Hilfe eines SQL-Kommentars in den SQL-Code eingefügt werden, was z. B. so aussehen könnte: `/* isIdGroup=true */`. Somit sind diese augenscheinlichen Probleme ohne großen Aufwand zu lösen, allerdings unter der Einschränkung, dass die Datenbank benötigt wird.

Durch Unterabfragen im SQL-Code und somit durch das Fehlen von wichtigen Informationen im SQL-Code (vergleiche Listing 4.3), ist es nicht in annehmbarer Art und Weise möglich, diese fehlenden Information zu beschaffen. Dadurch ist das Parsen des SQL-Codes keine gute Lösung und kann nicht zur Deserialisierung einer Filtergruppe genutzt werden.

4.3.4 Serialisierung einer Filtergruppe mittels der „PARASUITE Filter Expression Language“

Abgeleitet aus dem Ansatz SQL zu parsen, und damit eine Sprache zu parsen, ist die Erstellung einer eigenen Sprache für Filtergruppen. Diese Sprache wurde im Rahmen dieser Arbeit entwickelt und heißt „PARASUITE Filter Expression Language“, kurz PFEL. Der Sinn und Zweck dieser Sprache ist es, alle nötigen und wichtigen Informationen einer Filtergruppe (siehe Abschnitt 4.3.1) textuell und menschenlesbar zu repräsentieren. Äquivalent zum Parsen von SQL (siehe Abschnitt 4.3.3) wird auch hier die Datenbank benötigt, um aus der textuellen Repräsentation von Tabellen und Spalten gültige `ParasuiteEntity`- und `ParasuiteEntityType`-Objekte zu konstruieren (siehe Abschnitt 2.5.1). Im Gegensatz zum Ansatz SQL zu parsen, ist es aber möglich, die komplette Filtergruppe mit all ihren Informationen wiederherzustellen.

Im Folgenden wird die Sprache PFEL genauer beschrieben und die Serialisierung und Deserialisierung einer Filtergruppe mittels PFEL vorgestellt.

Definition der Sprache PFEL

PFEL ist definiert über eine lexikalische Struktur durch Angabe von Tokens und einer kontextfreien Grammatik. Die definierten Tokens sind in Listing 4.4 zu sehen. Dabei ist auf der linken Seite der Name des Tokens in spitzen Klammern angegeben und rechts davon – getrennt mit einem Doppelpunkt – der reguläre Ausdruck (vergleiche [Suna]), auf den das Token passt.

Listing 4.4: Tokens für PFEL

```
1  <WHITESPACE>  :  [ \n\t\f\r]
2  <BRACKET>  :  \(|\)|\{|\}
3  <COMMA>  :  ,
4  <IDENTIFIER>  :  IDENTIFIER\[LOCALE=[a-z]{2}_[A-Z
       ]{2}\]\(([A-Za-z_][A-Za-z0-9_]*\.[A-Za-z_][A-Za-z0-9
       _]*)\)
5  <OPERATOR>  :  OPERATOR\((<=|>=|=|!=|<|>|(NOT_)?IN|(NOT_
       )?BETWEEN|IS_NULL)\)
6  <OPERATOR_LOGICAL>  :  OPERATOR_LOGICAL\((AND|OR)\)
7  <UNKNOWN_TYPE>  :  UNKNOWN_TYPE\((([^\)]*)|NULL)\)
8  <VALUE_TEXT_LOCALIZED>  :  VALUE_TEXT_LOCALIZED
       \((('(\\.|[^\\'])*')|NULL)\)
9  <VALUE_TEXT>  :  VALUE_TEXT\((('(\\.|[^\\'])*')|NULL)\)
10 <VALUE_INTEGER>  :  VALUE_INTEGER
       \((([+-]?(0|([1-9][0-9]*)))|NULL)\)
11 <VALUE_DOUBLE>  :  VALUE_DOUBLE\((([+-]?(0|([1-9][0-9]*)
       )\.([0-9]+))|NULL)\)
12 <VALUE_BOOL>  :  VALUE_BOOL\((true|false|NULL)\)
13 <VALUE_BOOL_TEXT_LOCALIZED>  :
       VALUE_BOOL_TEXT_LOCALIZED\((('(\\.|[^\\'])*')|NULL)
       \)
14 <VALUE_DATE>  :  VALUE_DATE
       \((('[0-9]{4}-[0-9]{2}-[0-9]{2}')|NULL)\)
15 <VALUE_TIME>  :  VALUE_TIME
       \((('[0-9]{2}:[0-9]{2}:[0-9]{2}')|NULL)\)
16 <VALUE_DATE_TIME>  :  VALUE_DATE_TIME
       \((('[0-9]{4}-[0-9]{2}-[0-9]{2}_
       [0-9]{2}:[0-9]{2}:[0-9]{2}')|NULL)\)
17 <VALUELIST_BEGIN>  :  VALUE_LIST\[TYPE=(UNKNOWN_TYPE|
       VALUE_TEXT_LOCALIZED|VALUE_TEXT|VALUE_INTEGER|
       VALUE_DOUBLE|VALUE_BOOL|VALUE_BOOL_TEXT_LOCALIZED|
       VALUE_DATE|VALUE_TIME|VALUE_DATE_TIME)\]\(
18 <TAG>  :  \[[A-Z0-9_]+\]
```

Die kontextfreie Grammatik von PFEL ist in Listing 4.5 zu sehen. Das Listing zeigt die in PFEL verwendeten Produktionen in Backus-Naur-Form[3] (BNF). Literale sind dabei in einfachen Hochkommas angegeben (Beispiel: `'literal'`).

Listing 4.5: Kontextfreie Grammatik von PFEL

```
1 Start  ::=  FilterExpressionGroup
2 FilterExpressionGroup  ::=  '(' FilterElements ')'  |
      '(' <TAG> FilterElements ')'
3 FilterElements  ::=  FilterElement    |
      FilterElement <OPERATOR_LOGICAL> FilterElements
4 FilterElement  ::=  FilterExpression |
      FilterExpressionGroup
5 FilterExpression  ::=  '{' <IDENTIFIER> <OPERATOR>
      ValueList '}'
6 ValueList  ::=  <VALUELIST_BEGIN> Values ')'
      | <VALUELIST_BEGIN> ')'
7 Values  ::=  Value | Value ',' Values
8 Value  ::=  <UNKNOWN_TYPE> | <VALUE_TEXT_LOCALIZED>
      | <VALUE_TEXT> | <VALUE_INTEGER> | <VALUE_DOUBLE>
      | <VALUE_BOOL> | <VALUE_BOOL_TEXT_LOCALIZED>
      | <VALUE_DATE> | <VALUE_TIME> | <VALUE_DATE_TIME>
```

Printer für PFEL

Um eine Filtergruppe zu serialisieren, existiert die Utility-Klasse `FilterPFELPrinter`. Listing 4.6 zeigt ein Beispiel eines PFEL-Filterausdrucks, der vom `FilterPFELPrinter` aus einer Filtergruppe erzeugt wurde. Die Filtergruppe, die dem `FilterPFELPrinter` für das Beispiel zugrunde lag, ist die gleiche wie in Listing 4.2. Der Beginn einer Filtergruppe wird dabei eingeleitet mit einer öffnenden Klammer („`(`“) und endet mit einer schließenden („`)`“). Direkt hinter der öffnenden Klammer kann das optionale Tag `[IDCONDITIONS]` stehen, welches kennzeichnet, dass es sich um eine Filtergruppe handelt, die IDs überprüft (vergleiche Abschnitt 4.3.1). Ein einzelner Filterausdruck wird mit einer öffnenden, geschwungenen Klammer („`{`“) begonnen und endet mit einer schließenden („`}`“). Zwischen diesen geschwungenen Klammern steht ein Identifizierer, gefolgt von einem Operator und einer Liste mit Werten. Ein Identifizierer repräsentiert eine

[3] Siehe `http://dictionary.reference.com/browse/Backus-Naur%20Form`

Tabellenspalte mit entsprechender Angabe der Lokalisierung. Der Operator in einem einzelnen Filterausdruck darf nur ein Vergleichsoperator sein (siehe Abschnitt 4.3.1). Eine Liste mit Werten enthält eine Typangabe und darf nur Werte dieses Typs beinhalten. Eine Filtergruppe kann aus mehreren einzelnen Filterausdrücken und Filtergruppen verknüpft mit logischen Operatoren bestehen.

Listing 4.6: PFEL-Filterausdruck

```
( [IDCONDITIONS]
  {
    IDENTIFIER[LOCALE=de_DE](LOCOMOTIVE.Locomotive_ID)
    OPERATOR(IN)
    VALUE_LIST[TYPE=VALUE_TEXT](
      VALUE_TEXT('021100101-1')
    )
  }
  OPERATOR_LOGICAL(AND)
  (
    {
      IDENTIFIER[LOCALE=de_DE](NOTIFICATION.Time)
      OPERATOR(NOT_BETWEEN)
      VALUE_LIST[TYPE=VALUE_TIME](
        VALUE_TIME(20:10:53),
        VALUE_TIME(20:37:12)
      )
    }
  )
)
```

Parser für PFEL

Die Java-Klasse des PFEL-Parsers heißt `FilterPFELParser`. Diese verwendet intern den `FilterPFELScanner`, welcher einen Token-Strom entsprechend den in Listing 4.4 aufgeführten Tokens liefert. Der Parser konsumiert die vom Scanner gelieferten Tokens und entscheidet jeweils, welche Produktion (vergleiche Listing 4.5) anzuwenden ist. Listing 4.7 zeigt die Java-Klasse des PFEL-Parsers samt Methoden. Die Implementierung der Methoden ist zur Vereinfachung weggelassen.

Listing 4.7: Java-Klasse `FilterPFELParser`

```
1  import java.io.File;
2
3  public final class FilterPFELParser {
4    /* Konstruktoren */
5    public FilterPFELParser(final String text) {}
6    public FilterPFELParser(final File file) {}
7
8    /* Filtergruppe parsen */
9    public FilterExpressionGroup parse() {}
10
11   /* Hilfsmethoden fuer Tokens */
12   private String parseTokenTag() {}
13   private FilterOperator parseTokenOperator() {}
14   private FilterOperator parseTokenOperatorLogical() {}
15   private TokenType parseTokenValueListBegin() {}
16
17   /* Hilfsmethoden fuer Produktionen */
18   private FilterExpressionGroup
         parseFilterExpressionGroup() {}
19   private List<FilterElement> parseFilterElements() {}
20   private FilterElement parseFilterElement() {}
21   private FilterExpression parseFilterExpression() {}
22   private Map<Character, Object> parseValueList() {}
23   private List<Map<Character, Object>> parseValues() {}
24   private Map<Character, Object> parseValue() {}
25 }
```

Der PFEL-Parser hat zwei Konstruktoren: einen, der einen `String` und einen, der eine Datei entgegennimmt. Der `String` oder die Datei müssen dabei einen gültigen PFEL-Filterausdruck enthalten. Mit der Methode `parse` kann der übergebene PFEL-Ausdruck dann zu einer gültigen Filtergruppe geparst werden. Die Methode `parse` ist – abgesehen von den Konstruktoren – die einzige `public`-Methode dieser Klasse. Alle anderen Methoden sind `private` und dienen als Hilfsmethoden. In Listing 4.7 sind die Methoden gruppiert dargestellt und mit Kommentaren versehen. Es gibt die Gruppe der Hilfsmethoden für Tokens, die je ein bestimmtes Token parsen. Manche Tokens werden auch direkt in den anderen Hilfsmethoden geparst, weshalb nicht für jedes Token eine Hilfsmethode existiert. Des Weiteren ist eine Gruppe mit Hilfsmethoden für Produktionen vorhanden. Dabei existiert für jede Produktion (siehe Listing 4.5) eine Hilfsmethode.

Wie eine Filtergruppe mit Hilfe des PFEL-Parsers geparst werden kann, zeigt Listing 4.8. Dabei wird zuerst ein PFEL-Ausdruck definiert, der in der Variable `pfelString` gespeichert ist. Der wirkliche PFEL-Ausdruck ist hier zur Vereinfachung weggelassen und könnte z. B. den in Listing 4.6 gezeigten Ausdruck enthalten. Danach wird eine Instanz vom `FilterPFELParser` erzeugt und in der Variable `parser` gespeichert. Dabei wird der gerade definierte PFEL-Ausdruck an den Konstruktor übergeben. Anschließend wird der PFEL-Parser mit der Methode `parse` angewiesen, den PFEL-Ausdruck zu parsen und eine dem PFEL-Ausdruck entsprechende Filtergruppe zurückzugeben. Die Filtergruppe wird in der Variablen `feg` gespeichert und kann somit verwendet werden.

Listing 4.8: Parsen eines PFEL-Filterausdrucks

```
1  public class PFELParserTest {
2    public static void main(String[] args) {
3      /* Zu parsender PFEL-Ausdruck */
4      String pfelString = ... ;
5
6      /* PFEL-Parser erzeugen */
7      FilterPFELParser parser =
8          new FilterPFELParser(pfelString);
9
10     /* PFEL-String parsen */
11     FilterExpressionGroup feg = parser.parse();
12
13     ...
14   }
15 }
```

Eine Alternative zum Selbstbau des Scanners und Parsers für PFEL wäre es, einen Parsergenerator zu verwenden – für Java z. B. JavaCC[4] oder AntLR[5]. Aber bei kleineren Sprachen wie PFEL ist ein selbstgebautes Scanner-Parser-Gespann ohne größeren Aufwand zu erstellen und besser wartbar als eines, das mit einem Parsergenerator erzeugt wurde.

Die Sprache PFEL hat einen Nachteil, der sich z. B. beim Testen von Filtergruppen und dem PFEL-Parser bemerkbar macht. Da der PFEL-Parser die Datenbank benötigt, um eine gültige Filtergruppe zu bauen, ist es nur möglich, unter der Annahme zu testen, dass in der Datenbank bestimmte

[4]Siehe `http://javacc.dev.java.net/`
[5]Siehe `http://www.antlr.org/`

Tabellen, Spalten oder Datensätze vorhanden sind. Angenommen, es existiert ein PFEL-Ausdruck, der eine Spalte namens *Selling_Date* der Tabelle *LOCOMOTIVE* enthält:

```
... IDENTIFIER[LOCALE=de_DE](LOCOMOTIVE.Selling_Date) ...
```

Wird die Spalte *Selling_Date* nun aus irgendeinem Grund aus der Tabelle *LOCOMOTIVE* gelöscht, bekommt der Parser bei der Anfrage an die Datenbankschnittstelle `ParasuiteEntityDAOBean` (Methode `getParasuiteEntityType`, siehe Abschnitt 2.5.2) kein gültiges `ParasuiteEntityType`-Objekt zurück, sondern `null`, was zu Fehlern im Parse-Vorgang führt. PFEL ist damit eng gekoppelt an den Inhalt der Datenbank. JUnit-Tests, die den PFEL-Parser testen, müssen zwangsläufig PFEL-Ausdrücke an den PFEL-Parser reichen. Ändert sich der Datenbankinhalt, können die erstellten JUnit-Tests fehlschlagen. Da das Testen von Software aber eine hohe Priorität hat (vergleiche [VM00]), müssen alle Informationen, die eine Filtergruppe enthält, mit in die textuelle Serialisierung einfließen. Nur so kann ohne Datenbankzugriff das Ursprungsobjekt wieder hergestellt werden. Der nächste Abschnitt beschreibt deshalb die Serialisierung einer Filtergruppe mittels XML.

4.3.5 Serialisierung einer Filtergruppe mittels XML

In diesem Abschnitt wird die Serialisierung und Deserialisierung einer Filtergruppe (siehe Abschnitt 4.3.1) in XML beschrieben. XML wurde hier gewählt, weil es sehr gut zum Darstellen hierarchischer Strukturen geeignet ist.

Printer für XML

Für die Serialisierung einer Filtergruppe in XML ist der `FilterXMLPrinter` zuständig. Ein Beispiel einer XML-Repräsentation einer Filtergruppe ist in Listing 4.9 zu sehen. Dem Beispiel liegt dabei die gleiche Filtergruppe zugrunde wie den Beispielen des PFEL-Parsers (siehe Abschnitt 4.3.4) und des SQL-Parsers (siehe Abschnitt 4.3.3) auch.

Listing 4.9: Ein XML-Filterausdruck

```
1 <FilterExpressionGroup>
2   <IsIdGroup>true</IsIdGroup>
3   <FilterExpression>
```

```
4       <Locale>de_DE</Locale>
5       <ParasuiteEntityType>
6         <Id>Locomotive_ID</Id>
7         <LocalizedName>Lokomotiven-ID</LocalizedName>
8         <Locale>de_DE</Locale>
9         <ParasuiteEntity>
10          <Id>LOCOMOTIVE</Id>
11          <LocalizedName>Lokomotive</LocalizedName>
12          <Locale>de_DE</Locale>
13        </ParasuiteEntity>
14        <IdentifierType>true</IdentifierType>
15        <DataType>VALUE_TEXT</DataType>
16        <MeasuringUnit isNull="true"></MeasuringUnit>
17      </ParasuiteEntityType>
18      <FilterOperator type="compare">IN</FilterOperator>
19      <ValueList type="VALUE_TEXT">
20        <Value>021100101-1</Value>
21      </ValueList>
22    </FilterExpression>
23    <FilterOperator type="logical">AND</FilterOperator>
24    <FilterExpressionGroup>
25      <IsIdGroup>false</IsIdGroup>
26      <FilterExpression>
27        <Locale>de_DE</Locale>
28        <ParasuiteEntityType>
29          <Id>Time</Id>
30          <LocalizedName>Uhrzeit</LocalizedName>
31          <Locale>de_DE</Locale>
32          <ParasuiteEntity>
33            <Id>NOTIFICATION</Id>
34            <LocalizedName>Meldung</LocalizedName>
35            <Locale>de_DE</Locale>
36          </ParasuiteEntity>
37          <IdentifierType>false</IdentifierType>
38          <DataType>VALUE_TIME</DataType>
39          <MeasuringUnit isNull="true"></MeasuringUnit>
40        </ParasuiteEntityType>
41        <FilterOperator type="compare">NOT_BETWEEN</
              FilterOperator>
42        <ValueList type="VALUE_TIME">
43          <Value>20:10:53</Value>
44          <Value>20:37:12</Value>
```

```
45        </ValueList>
46      </FilterExpression>
47    </FilterExpressionGroup>
48 </FilterExpressionGroup>
```

Die XML-Repräsentation der einzelnen Bestandteile einer Filtergruppe wird im Folgenden beschrieben.

ParasuiteEntity Eine Instanz der Klasse `ParasuiteEntity` (siehe Abschnitt 2.5.1) wird in einen XML-Filterausdruck, wie in Listing 4.10 zu sehen, serialisiert. Dabei erhält jedes Instanzfeld der Klasse einen eigenen XML-Tag.

Listing 4.10: XML-Filterausdruck – `ParasuiteEntity`

```
1 <ParasuiteEntity>
2   <Id>LOCOMOTIVE</Id>
3   <LocalizedName>Lokomotive</LocalizedName>
4   <Locale>de_DE</Locale>
5 </ParasuiteEntity>
```

ParasuiteEntityType Eine Instanz der Klasse `ParasuiteEntityType` (siehe Abschnitt 2.5.1) wird analog zu einem `ParasuiteEntity` in einen XML-Filterausdruck serialisiert (siehe Listing 4.11). Die Serialisierung des `ParasuiteEntity`-Objekts ist hier der Übersichtlichkeit halber ausgelassen.

Listing 4.11: XML-Filterausdruck – `ParasuiteEntityType`

```
1 <ParasuiteEntityType>
2   <Id>Locomotive_ID</Id>
3   <LocalizedName>Lokomotiven-ID</LocalizedName>
4   <Locale>de_DE</Locale>
5   <ParasuiteEntity>...</ParasuiteEntity>
6   <IdentifierType>true</IdentifierType>
7   <DataType>VALUE_TEXT</DataType>
8   <MeasuringUnit isNull="true"></MeasuringUnit>
9 </ParasuiteEntityType>
```

FilterOperator Eine Instanz der Klasse `FilterOperator` (siehe Abschnitt 4.3.1) wird in einen XML-Filterausdruck, wie in Listing 4.12 zu sehen, serialisiert. Dabei wird unterschieden zwischen logischen und Vergleichsoperatoren.

Listing 4.12: XML-Filterausdruck – `FilterOperator`

```
<!-- Vergleichs-Filteroperator -->
<FilterOperator type="compare">IN</FilterOperator>

<!-- Logischer Filteroperator -->
<FilterOperator type="logical">AND</FilterOperator>
```

FilterExpression Eine Instanz der Klasse `FilterExpression` (siehe Abschnitt 4.3.1) wird in einen XML-Filterausdruck, wie in Listing 4.13 zu sehen, serialisiert. Dabei erhält jedes Instanzfeld der Klasse `FilterExpression` einen eigenen XML-Tag. Die Serialisierung des `ParasuiteEntityType`-Objekts ist hier der Übersichtlichkeit halber ausgelassen.

Listing 4.13: XML-Filterausdruck – `FilterExpression`

```
<FilterExpression>
  <Locale>de_DE</Locale>
  <ParasuiteEntityType>...</ParasuiteEntityType>
  <FilterOperator type="compare">IN</FilterOperator>
  <ValueList type=">VALUE_TEXT">
    <Value>021100101-1</Value>
  </ValueList>
</FilterExpression>
```

FilterExpressionGroup Eine Instanz der Klasse `FilterExpressionGroup` (siehe Abschnitt 4.3.1) wird in einen XML-Filterausdruck, wie in Listing 4.15 zu sehen, serialisiert. Das Instanzfeld `isIdGroup` erhält hier einen eigenen XML-Tag. Danach folgt entweder ein einzelner Filterausdruck (`FilterExpression`) oder eine Filtergruppe (`FilterExpressionGroup`). Daraufhin kann beliebig oft ein logischer `FilterOperator` gekoppelt mit wieder entweder einem einzelnen Filterausdruck oder einer Filtergruppe (siehe dazu auch Abschnitt 4.3.1) folgen. Der Aufbau in BNF-Form einer `FilterExpressionGroup` in XML ist in Listing 4.14 zu sehen. Literale sind dabei in einfachen Hochkommas angegeben. Ein Beispiel für einen XML-Filterausdruck ist in Listing 4.15 zu sehen.

Listing 4.14: XML-Filterausdruck – Aufbau einer `FilterExpressionGroup`

```
<!-- Aufbau einer Filtergruppe in BNF-Darstellung -->
'<FilterExpressionGroup>'
IsIdGroup
(FilterExpression|FilterExpressionGroup)
(
  FilterOperator
  (FilterExpression|FilterExpressionGroup)
)*
'</FilterExpressionGroup>'
```

Listing 4.15: XML-Filterausdruck – `FilterExpressionGroup`

```
<FilterExpressionGroup>
  <IsIdGroup>true</IsIdGroup>
    <FilterExpression>...</FilterExpression>
  <FilterOperator type="logical">AND</FilterOperator>
    <FilterExpressionGroup>...</FilterExpressionGroup>
</FilterExpressionGroup>
```

Parser für XML

Das Parsen von XML-Strukturen wird vom Java-Framework gut unterstützt. Dabei stehen verschiedene XML-Parser zur Verfügung, z. B. ein SAX-Parser und ein DOM-Parser.[6] Der `FilterXMLParser`, welcher den XML-Filterausdruck parst, ist mit Hilfe des DOM-Parsers aus dem Java-Framework realisiert. Die Klasse `FilterXMLParser` ist in Listing 4.16 zu sehen. Dabei ist die Implementierung der Methoden zur Vereinfachung weggelassen.

Listing 4.16: Beispiel einer XML-Ausgabe für Filtergruppen

```
public final class FilterXMLParser {
  /* Konstruktor (private) */
  private FilterXMLParser() {}

  /* Filtergruppe parsen */
  public static FilterElement parse(final String
      xmlFilterElement) {}

  /* Hilfsmethoden */
```

[6] Siehe `http://www.torsten-horn.de/techdocs/java-xml.htm`

```
9   private static FilterElement parseFilterElement(final
        Node node) {}
10  private static FilterExpressionGroup
        parseFilterExpressionGroup(final Node node) {}
11  private static FilterExpression parseFilterExpression(
        final Node node) {}
12  private static FilterOperator parseFilterOperator(
        final Node node) {}
13  private static ParasuiteEntity parseParasuiteEntity(
        final Node node) {}
14  private static ParasuiteEntityType
        parseParasuiteEntityType(final Node node) {}
15 }
```

Die Klasse `FilterXMLParser` ist eine zustandslose Utility-Klasse. Aus diesem Grund ist der Konstruktor auf `private` gesetzt. Analog zum `FilterPFELParser` (siehe Abschnitt 4.3.4) existiert auch im `FilterXMLParser` nur eine `public`-Methode namens `parse`, die einen Filterausdruck in XML-Form als Parameter erwartet. Als Ergebnis liefert sie eine gültige Filtergruppe. Die `parse`-Methode verwendet intern die in Listing 4.16 aufgeführten Hilfsmethoden, um die einzelnen Elemente im XML-Filterausdruck zu parsen.

Somit kann eine Filtergruppe mit all ihren Informationen serialisiert und wieder deserialisiert werden. Beim Deserialisierungsprozess bzw. beim Parsen ist keine Verbindung zur Datenbank nötig, um eine gültige Filtergruppe zu konstruieren. Alle Informationen stecken im XML-Filterausdruck. Damit ist die Serialisierung einer Filtergruppe mittels XML auch geeignet für Testzwecke mittels JUnit und die in diesem Abschnitt vorgestellte Methode der Serialisierung erfüllt alle gestellten Anforderungen. Ein Filtergruppenobjekt lässt sich textuell und menschenlesbar serialisieren und enthält alle Informationen. Die in dieser Arbeit zu entwickelnde Datenbanklesekomponente wird damit allen gestellten Anforderungen gerecht.

Ein Nachteil der Serialisierung mittels XML sei an dieser Stelle allerdings erwähnt: Der serialisierte XML-Ausdruck kann schon bei einer geringen Menge an Filterbedingungen sehr groß werden und viel Overhead beinhalten. Verglichen damit enthält ein PFEL-Filterausdruck weitaus weniger Overhead, wodurch die Bedeutung des Filters schneller erfasst werden kann.

Kapitel 5

Ausblick und Zusammenfassung

In diesem Kapitel werden die in dieser Bachelorarbeit vorgestellten Konzepte zusammengefasst und nochmals miteinander in Verbindung gebracht. Außerdem wird ein Ausblick auf weitere Entwicklungen gegeben, die aus dieser Arbeit hervorgehen könnten.

5.1 Zusammenfassung

Mit PARASUITE wird ein entscheidungsunterstützendes System im Bereich Produktlebenszyklus entwickelt, das bei Kunden wie *Bombardier Transportation* und *ThyssenKrupp Aufzüge* eingesetzt wird. Berechnungen auf Daten der Kunden liefern dabei Entscheidungsgrundlagen für die vorausschauende Produktwartung. Zur Speicherung der Kundendaten setzte PARASUITE bisher ein generisches Datenbankschema ein, welches aber durch ein neues, generatives ersetzt wurde, das nicht kompatibel zum alten ist.

Um die Daten der Kunden zu berechnen, werden in PARASUITE spezielle Berechnungsnetzwerke eingesetzt. Dazu muss es Komponenten geben, die solche Berechnungsnetzwerke mit Daten aus der Datenbank speisen. Bisher existierten für diesen Zweck sehr spezielle Komponenten, die auf dem alten Datenbankschema arbeiteten. Diese Komponenten sind mit dem neuen Datenbankschema nicht kompatibel und können auch nicht portiert werden.

Daraus hervorgehend wird eine neue Komponente benötigt, die in der Lage ist, auf dem neuen Datenbankschema basierende Daten zu lesen und damit die Berechnungsnetzwerke zu speisen. Die Konzeption und Implementierung einer generischen, konfigurierbaren Komponente, die diese Aufgabe erfüllt, wird in dieser Bachelorarbeit beschrieben.

Die Bachelorarbeit ist dabei so gegliedert, dass zuerst das PARASUITE-Projekt genauer vorgestellt wird und die Einsatzgebiete beim Kunden aufgezeigt werden. Außerdem werden die Ziele dieser Bachelorarbeit festgelegt, welches die Konzeption und Implementierung einer Datenbanklesekomponente darstellt, die die Anforderungen an menschenlesbare Konfigurierbarkeit und Generizität erfüllen soll. Danach werden das alte und das neue Datenbankschema beschrieben, wobei detailliert auf deren Speicherstrukturen eingegangen wird. Daraufhin werden die beiden Datenbankschemen miteinander verglichen und Vor- und Nachteile beider aufgeführt. Des Weiteren werden die Schnittstellen zur Datenbank für das neue Datenbankschema vorgestellt, die von der zu entwickelnden Datenbanklesekomponente benutzt werden. Nachfolgend werden das Programmierparadigma Flow-Based Programming (FBP) und das Framework JavaFBP vorgestellt, mit dessen Hilfe PARASUITE seine Datenberechnungen realisiert. Basierend darauf, wird die Entwicklung der Datenbanklesekomponente geschildert, wobei besonders auf die Konfiguration der Komponente eingegangen wird. Die Konfiguration muss, gemäß Anforderung, menschenlesbar serialisierbar sein. Die komplexesten zu serialisierenden Strukturen in der Konfiguration sind dabei Filterbedingungen für die auszulesenden Daten. Diesbezüglich werden verschiedene Methoden zur Serialisierung dieser Filterbedingungen vorgestellt und eine anforderungsgerechte Lösung wird erarbeitet.

5.2 Ausblick

Basierend auf den in dieser Arbeit vorgestellten Konzepten, wird in diesem Abschnitt darauf eingegangen, wohin die Entwicklung der Datenbanklesekomponente noch führen und welche weiteren Entwicklungen sich ergeben könnten.

5.2.1 Generische Lesekomponente

Die in dieser Arbeit vorgestellte Datenbanklesekomponente ist in Bezug auf Datenbanktabellen, die sie auslesen kann, generisch. Sie ist in der Lage, FBP-Netzwerke mit Daten aus der Datenbank zu speisen und kann dabei sehr flexibel konfiguriert werden. In Zukunft könnte auch das Lesen aus Dateien unterschiedlicher Formate interessant werden, was vor allem beim

Datenimport eine Rolle spielt. Geeignete Dateiformate wären z. B. XML-Dateien und CSV-Dateien. Anstatt eine Dateilesekomponente zu entwickeln, könnte an dieser Stelle direkt eine generische Lesekomponente entwickelt werden, die mittels Konfiguration in der Lage ist, aus Datenbanken und aus Dateien zu lesen.

5.2.2 Generische Schreibkomponente

Analog zur generischen Lesekomponente wäre eine generische Schreibkomponente denkbar, die in einem FBP-Netzwerk als Konsument agiert und Daten aus dem Netz heraustransportiert. Die Schreibkomponente könnte dabei in ähnlicher Weise konfigurierbar sein wie die Lesekomponente und, je nach Konfiguration, in eine Datenbanktabelle oder eine Datei schreiben.

5.2.3 Anpassung des FBP-Frameworks

Im JavaFBP-Framework können FBP-Komponentenklassen mit Port-Annotationen versehen werden, um die Komponenten mit entsprechenden Ports auszustatten (siehe Abschnitt 3.2.1). Wie in Abschnitt 3.2.1 erwähnt wird, ist hier eine Schwachstelle des JavaFBP-Frameworks zu erkennen. Port-Annotationen müssen dabei an die Klasse geheftet werden und der Name des Ports muss als Parameter angegeben werden. Listing 5.1 zeigt die relevanten Teile der Java-Klasse einer FBP-Komponente.

Listing 5.1: Port-Annotationen an einer FBP-Komponentenklasse

```
1  @InPort("IN")
2  @OutPort("OUT")
3  public class SampleFBPComponent extends Component {
4    private InputPort inputPort = null;
5    private OutputPort outputPort = null;
6
7    ...
8
9    @Override
10   protected void openPorts() {
11     this.inputPort = super.openInput("IN");
12     this.outputPort = super.openOutput("OUT");
13   }
14 }
```

Im Listing sind zwei Annotationen an der Klasse zu sehen, die einen Eingangsport (*IN*) und einen Ausgangsport (*OUT*) für die FBP-Komponente definieren. Für jeden definierten Port müssen entsprechende Variablen in der Java-Klasse vorhanden sein, in diesem Fall `inputPort` und `outputPort`. In der Methode `openPorts` müssen die Ports dann „geöffnet“ werden, wobei der gleiche Name angegeben werden muss wie in der entsprechenden Port-Annotation an der Klasse. Diese Information ist hier ganz klar redundant. An dieser Stelle könnte angesetzt werden, um das JavaFBP-Framework entsprechend abzuändern, sodass die Namensinformation des entsprechenden Ports nur einmal vorhanden ist. Somit könnte eine Java-Klasse für eine FBP-Komponente aussehen wie in Listing 5.2. Die Port-Annotationen könnten dabei direkt an den Variablen für Ports stehen und durch das FBP-Framework automatisch geöffnet werden. Dadurch wird das redundante Vorkommen von Portnamen vermieden und eine FBP-Komponente wird übersichtlicher.

Listing 5.2: Alternative Port-Annotationen

```
1 public class SampleFBPComponent extends Component {
2   @InPort("IN")
3   private InputPort inputPort = null;
4
5   @OutPort("OUT")
6   private OutputPort outputPort = null;
7
8   ...
9 }
```

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Literaturverzeichnis

[Hof08] Hoffmann, Philipp: *Konzeption einer generischen Datenimport-Schnittstelle*, Fachhochschule Gießen-Friedberg, Bachelorarbeit, August 2008

[Int] International Organization for Standardization: *English country names and code elements*. `http://www.iso.org/iso/country_codes/iso_3166_code_lists/english_country_names_and_code_elements.htm`. – [Online; Stand 28. Juni 2009]

[Int08] International Organization for Standardization: *Codes for the Representation of Names of Languages*. `http://www.loc.gov/standards/iso639-2/php/English_list.php`. Version: November 2008. – [Online; Stand 28. Juni 2009]

[Let08] Letschert, Prof. Dr. T.: *Programmiersprachen – Konzepte und Realisationen*. Fachhochschule Gießen-Friedberg, 2008

[Mor] Morrison, John P.: *Flow-Based Programming*. `http://jpaulmorrison.com/fbp/`. – [Online; Stand 3. August 2009]

[Mor94] Morrison, John P.: *Flow-Based Programming*. van Nostrand Reinhold, 1994

[Neu07] Neubauer, Helmut: *Entwicklung eines Entscheidungsunterstützungssystems für Problemstellungen im Bereich Produktlebenszyklusmanagement – Datenbankdesign und Datenimport*, Philipps-Universität Marburg, Diplomarbeit, 2007

[PW08] *Kapitel* 2.1.1 Kleine, spezialisierte Programme. In: Plötner, Johannes ; Wendzel, Steffen: *Linux*. Galileo Computing (2). – ISBN 978–3–8362–1090–4

[Sei03] Seidl, Prof. Dr. T.: *Praktikum "Data-Mining-Algorithmen"*. `http://www.dme.rwth-aachen.de/dateien/pruefungseinheiten/7/Prakt.pdf`. Version: 2003. – [Online; Stand 1. August 2009]

[Ste09] Steinseifer, Sven: *Evaluation and Extension of an Implementation of Flow-Based Programming*, Fachhochschule Gießen-Friedberg, Masterarbeit, Juli 2009. – [eingereicht]

[Suna] Sun Microsystems Inc.: *The Java Tutorials – Regular Expressions*. `http://java.sun.com/docs/books/tutorial/essential/regex/`. – [Online; Stand 28. Juli 2009]

[Sunb] Sun Microsystems Inc.: *MySQL 5.1 Referenzhandbuch – 11 Datentypen*. `http://dev.mysql.com/doc/refman/5.1/de/data-types.html`. – [Online; Stand 12. August 2009]

[url] *Base64 Encoder und Decoder*. `http://www.homepage-performance.de/base64.html`. – [Online; Stand 13. August 2009]

[VM00] Viega, John ; McManus, John: *The Importance of Software Testing*. `http://www.cutter.com/research/2000/crb000111.html`. Version: Januar 2000. – [Online; Stand 26. Juli 2009]

[Wic07] Wickesser, Craig: *Has JPA Killed the DAO?* `http://www.infoq.com/news/2007/09/jpa-dao`. Version: September 2007. – [Online; Stand 20. Juli 2009]

[Wik09] Wiktionary – Das freie Wörterbuch: *Liste alternativer Städtenamen*. `http://de.wiktionary.org/wiki/Verzeichnis:Liste_alternativer_St%C3%A4dtenamen`. Version: Juni 2009. – [Online; Stand 26. Juni 2009]