Mirko Haustein

Entwurf und prototypische Implementierung eines Mandatsmanagements für SEPA Direct Debits

MASTERARBEIT

HOCHSCHULE MITTWEIDA (FH)
UNIVERSITY OF APPLIED SCIENCES

Institut für Technologie- und Wissenstransfer Mittweida

Mittweida, 2009

Mirko Haustein

Entwurf und prototypische Implementierung eines Mandatsmanagements für SEPA Direct Debits

eingereicht als

MASTERARBEIT

an der

HOCHSCHULE MITTWEIDA (FH)
UNIVERSITY OF APPLIED SCIENCES

Institut für Technologie- und Wissenstransfer Mittweida

Mittweida, 2009

Erstprüfer: Prof. Dr.-Ing. Frank Zimmer
Zweitprüfer: Dr.-Ing. Frank Schubert

Vorgelegte Arbeit wurde verteidigt am:

Bibliographische Beschreibung

Haustein, Mirko:

Entwurf und prototypische Implementierung eines Mandatsmanagements für SEPA Direct Debits - 2009. - 78 S. Mittweida, Hochschule Mittweida, Institut für Technologie- und Wissenstransfer, Masterarbeit, 2009

Referat:

Ziel dieser Masterarbeit ist der Entwurf und die prototypische Implementierung eines Mandatsmanagements für SEPA Direct Debits. Zu Beginn wird eine Einführung in SEPA gegeben und die Besonderheiten erläutert. Danach wird konkret auf das SEPA-Lastschriftverfahren eingegangen und der neue Zahlungsablauf analysiert. Die Ergebnisse der Analyse werden als Anforderungen und Aufgaben für ein Mandatsmanagement zusammengestellt. Anschließend wird für eine der identifizierten Anforderungen eine Lösung erarbeitet und anhand eines Prototypen implementiert. Mit einem abschließenden Test wird die Funktionsweise des Prototypen sichergestellt und auf Korrektheit überprüft.

Mirko Haustein

Inhaltsverzeichnis

Abkürzungsverzeichnis

AAV	Abbuchungsauftragsverfahren
API	Application Programming Interface
B2B	Business to Business
BIC	Bank Identifier Code
Blob	Binary Large Object
CAMT	Cash Management
CDF	Cancelled Debit File
CSM	Clearing and Settlement Mechanism
DBMS	Datenbankmanagementsystem
DHTML	Dynamic HTML
DMS	Dokumenten-Management-System
DNF	Debit Notification File
DOM	Document Object Model
DTAUS	Datenträgeraustauschverfahren
DTAZV	Datenträgeraustausch Auslandszahlungsverkehr
DTD	Document Type Definition
DTO	Data Transfer Object
DVF	Debit Validation File
EBA	Euro Banking Association
EBICS	Electronic Banking Internet Communication Standard
ECBS	European Committee for Banking Standards
EEV	Einzugsermächtigungsverfahren
EPC	European Payments Council
EZB	Europäische Zentralbank

GrpHdr	GroupHeader
GUI	Graphical User Interface
IBAN	International Bank Account Number
IDF	Input Debit File
ISO	International Organisation for Standardization
JAXB	Java API for XML Binding
JDBC	Java Database Connectivity
LSA	Lastschrift-Abkommen
PACS	Payment Clearing on Settlement
PAIN	Payment Initiation
PmtInf	PaymentInformation
PSD	Payment Services Directive
RSF	Results of Settlement
SAX	Simple API for XML
SCF	SEPA Cards Framework
SCT	SEPA Credit Transfer
SDD	SEPA Direct Debit
SDF	Settled Debit File
SEPA	Single Euro Payments Area
SQL	Structured Query Language
StAX	Streaming API for XML
STP	Straight Through Processing
UCI	Unique Creditor Identifier
UNIFI	UNIversal Financial Industry message scheme
W3C	World Wide Web Consortium
XML	Extensible Markup Language
ZKA	Zentraler Kreditausschuss

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

1 Einleitung

1.1 Problemstellung und Abgrenzung

Unterschiedliche technische Standards und Zahlungsverfahren behindern bis zum jetzigen Zeitpunkt bargeldlose Zahlungen innerhalb Europas. Um dieses Problem zu beseitigen, ist ein europaweit einheitlicher Zahlungsraum mit dem Namen Single Euro Payments Area (SEPA) definiert worden. Seit Januar 2008 können bargeldlose Zahlungen innerhalb des SEPA-Raumes durch die Einführung von SEPA Credit Transfers getätigt werden. Zum Jahresende 2009 soll ein weiteres Zahlungsinstrument folgen. Die SEPA-Lastschrift (SEPA Direct Debit) orientiert sich dabei an dem bereits vorhandenen Lastschriftverfahren und basiert auf dem Prinzip: »Ich fordere Geld der anderen Seite und schreibe dieses mir gut.« [Bun08, Seite 21]

Für eine Lastschrift muss der Kontoinhaber des zu belastenden Kontos dem Einzieher der Forderung eine Genehmigung erteilen. Hierfür ist eine Autorisierung durch den Zahlungspflichtigen zwingend notwendig. Für diese Genehmigung existieren derzeit verschiedene Varianten. Während beim deutschen Lastschriftverfahren dem Gläubiger eine einmalige (aber widerrufbare) Einzugsermächtigung erteilt wird, muss beispielsweise in Italien jede einzelne Lastschriftbelastung durch den Kunden autorisiert werden.

So haben sich in ähnlicher Weise spezifische Funktionalitäten der Zahlungsinstrumente über Jahrzehnte hinweg in allen europäischen Märkten herausgebildet. Die nationalen Zahlungssysteme zeichnen sich durch ein hohes Maß an Effizienz und Sicherheit aus, beschränken jedoch die Interoperabilität zwischen den Systemen auf ein Mindestmaß. Um die Vorzüge des europäischen Binnenmarkts nutzen zu können, müssen die Grenzen im Zahlungsverkehr abgebaut werden.

Dieses versucht man durch die Einführung der SEPA-Zahlungsinstrumente zu erreichen. Die Genehmigung der SEPA-Lastschrift durch den Zahlungspflichtigen erfolgt durch eine Einverständniserklärung. Diese wird zwischen dem Zahlungspflichtigen und dem Zahlungsempfänger schriftlich als ein Mandat vereinbart. Um eine Lastschrift auslösen zu können, ist die Angabe des Mandats zwingend notwendig. Die Erteilung und der Nachweis von Mandaten ist ein neuer Prozess und benötigt deshalb die Unterstützung durch IT-Systeme. Ein solches System stellt das Mandatsmanagement dar und soll in dieser Arbeit konzipiert und entwickelt werden.

1.2 Zielsetzung und Aufbau der Arbeit

Zu Beginn dieser Masterarbeit wird eine Einführung in SEPA im Kapitel 2 gegeben und die Besonderheiten erläutert.

Im Kapitel 3 wird konkret auf das neue SEPA-Lastschriftverfahren eingegangen und mit dem bisherigen deutschen Lastschriftverfahren verglichen. Weiterhin werden der Prozess zur Erteilung der Mandate und deren Nachweis beim Versand der neuen SEPA-Lastschrift analysiert.

Die Ergebnisse dieser Analyse werden im Kapitel 4 als Anforderungen und Aufgaben für ein Mandatsmanagement zusammengestellt.

Anschließend wird für eine der identifizierten Anforderungen eine Lösung im Kapitel 5 erarbeitet und anhand eines Prototypen im Kapitel 6 implementiert. Dabei werden auftretende Probleme bei der Implementierung genannt und deren Lösungsmöglichkeiten erläutert sowie auf Optimierungsmöglichkeiten eingegangen.

Im Kapitel 7 wird die Funktionsweise des Prototypen sichergestellt und auf Korrektheit überprüft.

Mit einem abschließenden Resümee im Kapitel 8 und einem Ausblick auf weitere Verbesserungen schließt diese Arbeit.

2 SEPA - Single Euro Payments Area

Die Grundlagen für einen einheitlichen europäischen Wirtschaftsraum wurden bereits 1992 mit der Einführung des Euro geschaffen. Durch unterschiedliche technische Standards der Mitgliedsländer sowie unterschiedliche Zahlungsverfahren wurde ein europaweit einheitlicher Zahlungsraum veranlasst [SN08].

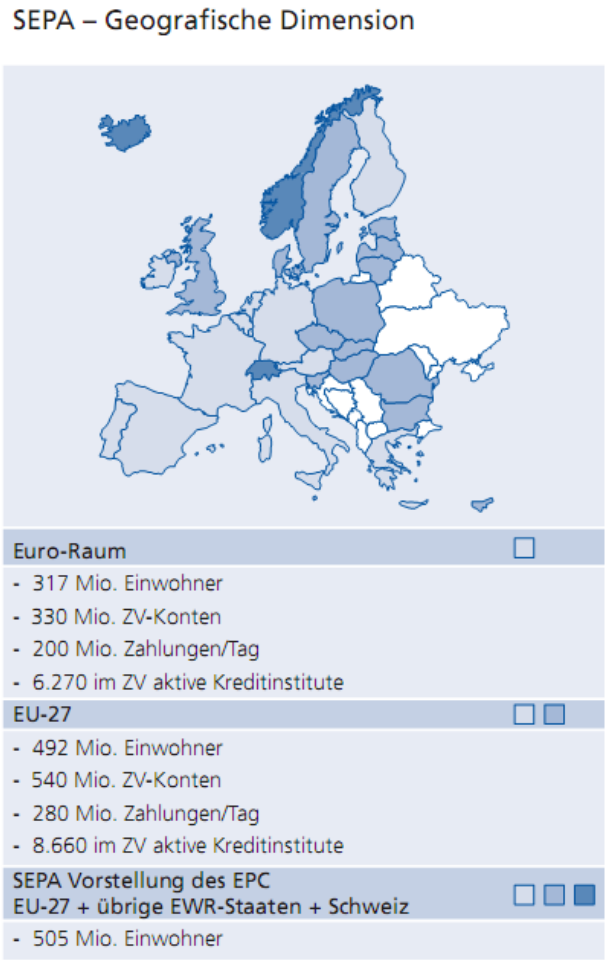


Abbildung 2.1: Darstellung der beteiligten SEPA-Länder [Deu08a, Seite 2]

Dieser Zahlungsverkehrsraum ist dabei mit SEPA abgekürzt, welcher für Single Euro Payments Area steht und die 27 Mitgliedsstaaten der Europäischen Union beinhaltet. Die Notwendigkeit dieser Umstellung lässt sich durch folgende Aufzählung besser verdeutlichen:

- Die Märkte im unbaren Zahlungsverkehr sind stark fragmentiert.
- Jedes Land verfügt über eigene technische Standards (beispielsweise in Bezug auf die Kontonummern-Systematik oder das Format für den Zahlungsaustausch).
- Die einzelnen Zahlungsinstrumente unterscheiden sich von Land zu Land (italienisches gegenüber deutschem Lastschriftverfahren).
- Der Zahlungsverkehr wird fast ausschließlich über nationale Dienstleister und Clearinghäuser abgewickelt.
- Es fehlen einheitliche Rechtsvorschriften in den einzelnen Mitgliedsstaaten.

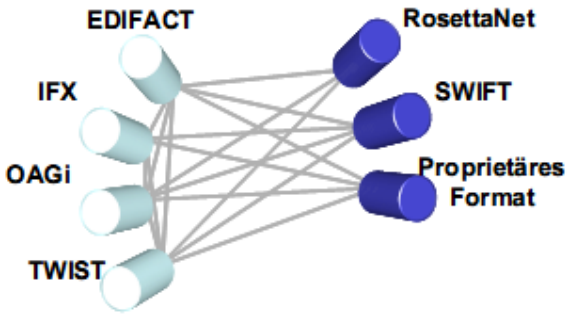


Abbildung 2.2: Fehlende Interoperabilität zwischen parallelen Standards [Bun06, Seite 3]

Vor allem Auslandszahlungen sind meist um ein Vielfaches teurer als Inlandszahlungen. Das liegt an einer höheren Fehlerquote, da oftmals dem Auftraggeber die Zahlungssysteme des Empfängerlandes nicht genau bekannt sind und daher falsche Angaben gemacht werden. Ebenso stellt diese Unklarheit ein Hindernis für die Schaffung eines Europäischen Binnenmarktes dar und wird von der EU-Kommision schon seit vielen Jahren kritisiert. Mit SEPA sollen die traditionellen Strukturen aufgebrochen werden. Dabei soll es künftig Zahlungsinstrumente geben, die jeder Kunde in Europa einsetzen kann. Bankkunden können künftig ihren gesamten Zahlungsverkehr einheitlich in Euro über ein beliebiges Kreditinstitut in Europa abwickeln. Dabei wird die Abschottung der bisherigen nationalen Märkte zu Gunsten eines europaweiten Zahlungsmarktes aufgehoben und somit ein europaweiter Wettbewerb geschaffen. Da SEPA nicht nur den grenzüberschreitenden europäischen Zahlungsverkehr betrifft, sondern zu einer vollständigen Integration führen soll, berührt der Umbau der europäischen Zahlungsverkehrslandschaft auch nationale Strukturen.

2.1 Definition des Begriffes SEPA

2.1.1 SEPA - Die Vision

Laut [Deu08a] lässt sich die Vision an folgendem Zitat verdeutlichen: »Durch SEPA soll in Europa das bargeldlose Zahlen über Ländergrenzen hinweg genauso einfach, sicher und effizient werden, wie es heute innerhalb der Nationalstaaten ist. Bargeldlose Euro-Zahlungen sind dann von einem einzigen Konto aus mittels einheitlicher Zahlungsinstrumente (SEPA-Überweisung, SEPA-Lastschrift und SEPA-Kartenzahlungen) möglich.« Um dieses Ziel zu erreichen ist im Juni 2002 der European Payments Council (EPC) als wichtigstes Gremium der europäischen Kreditwirtschaft zur Schaffung von SEPA gegründet wurden. Der EPC setzt sich vor allem aus den europäischen und nationalen Bankenverbänden zusammen. Das deutsche Kreditgewerbe begleitet diese Anstrengungen durch eine nationale Spiegelung der EPC-Gremienstruktur im Zentralen Kreditausschuss und ist aus Spitzenverbänden des deutschen Kreditgewerbes zusammengesetzt. Vor allem die Europäische Zentralbank (EZB) und die nationalen Zentralbanken der Euro-Länder fördern den SEPA-Gedanken, um die Vorteile der Währungsunion besser ausnutzen zu können.

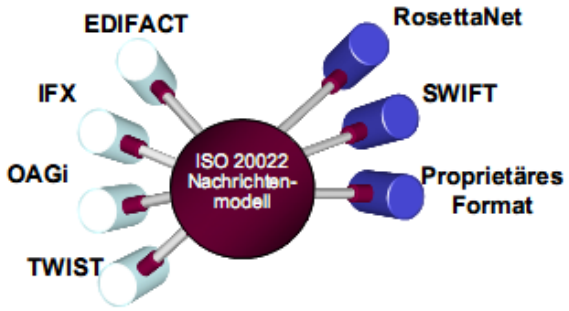


Abbildung 2.3: Gemeinsamer Modellierungsansatz nach SEPA [Bun06, Seite 3]

2.1.1.1 Vorteile durch Nutzung von SEPA

Grundsätzlich wird durch die Einführung und Nutzung von SEPA ein wesentlich größerer Wettbewerb zwischen den jeweiligen Banken gefördert. Dies spiegelt sich beispielsweise bei der Auswahl eines leistungsstarken Kreditinstitutes in ganz Europa durch den Kunden wieder. Mit dem Einsatz von europaweit einheitlichen Verfahren soll der Zahlungsverkehr schneller und sicherer werden, so dass die Kunden später nach einer maximalen Abwicklungszeit von drei Werktagen über den Überweisungsbetrag verfügen können. Geplant ist, dass sich diese Wartezeit im Jahre 2012 auf einen Bankarbeitstag verkürzt. Einen wesentlichen Faktor in der Verarbeitung stellt dabei die vollautomatische Abwicklung, das Straight Through Processing (STP) dar. Dies bedeutet, dass Bankprozesse ohne manuellen Eingriff abgewickelt werden können. Später soll der Zahlungsverkehr durch innovative elektronische Produkte weiter modernisiert werden.

2.1.2 Die SEPA-Zahlungsinstrumente

Der Startschuss für den produktiven Einsatz von SEPA ist im Januar 2008 erfolgt. Seit diesem Zeitpunkt können Bankkunden ihre Überweisungen bereits im SEPA-Format tätigen. Weiterhin soll im Jahre 2010 eine kritische Masse an Transaktionen über diese neuen Zahlungsinstrumente abgewickelt werden. Somit hätte die Integration einen irreversiblen Stand erreicht. Die SEPA-Zahlungsinstrumente lassen sich wie folgt unterscheiden.

- SEPA-Überweisung / SEPA Credit Transfer (SCT)
- SEPA-Lastschrift / SEPA Direct Debit (SDD)
- SEPA-Kartenzahlung / SEPA Cards Framework (SCF)

2.1.2.1 SEPA-Überweisung

Mit dem Ziel dass eine Überweisung nur maximal drei Bankarbeitstage dauert, bis dem Empfänger der Betrag gutgeschrieben wird, ist die bereits heute eingesetzte EU-Standardüberweisung ausgebaut worden. Dabei muss die International Bank Account Number (IBAN) und der Bank Identifier Code (BIC) für die Kontoidentifizierung angegeben werden. Bereits seit dem 28.01.2008 können europaweite Zahlungen unter Nutzung der SEPA-Überweisung getätigt werden.

2.1.2.2 SEPA-Lastschrift

Bisher konnten nur mit hohem Aufwand Rechnungen aus dem Ausland beglichen werden. Mit der zukünftigen europäischen Lastschrift soll dieses vereinfacht und beschleunigt werden. Die Grundlage dafür bieten einheitliche Standards, die vom EPC herausgegeben werden. Der produktive Einsatz von SEPA-Lastschriften ist per Definition für den 01.11.2009 vorgesehen. Bis dahin muss die EU-Richtlinie für SEPA-Lastschriften in nationales Recht umgesetzt werden. Eine ausführliche Beschreibung über SDD ist im Abschnitt 3.2 zu finden.

2.1.2.3 SEPA-Kartenzahlung

Das EPC definiert mit einem Rahmenwerk für den Kartenzahlungsverkehr generelle Anforderungen an Banken und Kartensystemen. Dabei werden drei Möglichkeiten festgelegt. Erstens, die Ablösung nationaler durch internationale Kartenprogramme, zweitens die Kooperation derselben und drittens die Ausdehnung des Wirkungsbereiches nationaler Kartensysteme durch Expansion. Als Ziel soll erreicht werden, dass der Karteninhaber seine Debitkarte in der gesamten SEPA-Region genau so einfach wie in seinem Heimatland einsetzen kann. Für diesen Zweck müssen bis Ende 2010 alle Kartenlesegeräte technologisch SEPA-konform umgerüstet sein.

2.1.3 Die SEPA-Datenformate

Vor dem Hintergrund des angestrebten STP-Prozessing hat das EPC eine europaweit einheitliche Lösung erarbeitet. Dabei basiert das SEPA-Datenformat auf XML-Basis, speziell auf UNIversal Financial Industry message scheme (UNIFI), und wird ebenfalls auch als ISO 20022 bezeichnet. Das Datenformat strebt eine weltweite Konvergenz von existierenden und neuen Nachrichtenstandards aus verschiedenen Bereichen des Finanzwesens an. Ausgehend von den ISO-Nachrichten für Überweisungen und Lastschriften, sind die SEPA-Nachrichtenstandards abgeleitet und als Untermenge definiert. Für eine effiziente Nutzung innerhalb der EU sind dabei Einschränkungen an ISO 20022 vorgenommen worden, welche durch das EPC im Dezember 2006 verabschiedet wurden. Dies wird deutlich am Beispiel der Implementation Guidelines für die jeweiligen Datenformate. Während die Umsetzung für den Interbankenzahlungsverkehr (PACS-Nachrichten) verbindlich festlegt wurde, stellen die Spezifikationen für die Kunde-Bank-Schnittstelle (PAIN-Nachrichten) nur empfehlenden Charakter dar. Deshalb wurden für die PAIN-Nachrichtentypen auch keine XML-Schemata durch das EPC erstellt.

2.1.4 Die verschiedenen SEPA-Nachrichtenarten

Prinzipiell lassen sich die Zahlungsverkehrsnachrichten nach ISO 20022 in drei Gruppen unterteilen. Erstens die Payment Initiation (PAIN)-Nachrichten, welche Nachrichten in der Kunde-Bank-Beziehung umfassen. Die zweite Gruppe stellen die Payment Clearing on Settlement (PACS)-Nachrichten dar. Diese werden im Interbankenbereich, also zwischen Bank-Bank, respektive zwischen Bank und Dienstleister ausgetauscht. Diese Zahlungen erfolgen nicht direkt vom Kunden zur Bank, sondern müssen den Umweg über ein Clearing-Haus (Bank) nehmen. Deshalb bezeichnet man dies als Clearing on Settlement. Für den Fehlerfall, Abbruch oder Statusänderungen von Zahlungen, gibt es ebenfalls zugehörige Nachrichten, so genannte R-Transaktionen. Diese können beispielsweise einen Payment Status Report,Payment Cancellation Request, Customer Payment Reversal oder weitere umfassen. Dabei sind diese Nachrichten sowohl für PACS, als auch für PAIN spezifiziert. Die dritte Gruppe, Cash Management (CAMT), sind Nachrichten, die in der Bank-Kunde-Beziehung eingesetzt werden. Diese werden in dieser Arbeit jedoch nicht genutzt und sollen somit nicht weiter berücksichtigt werden. Der Aufbau der Nachrichtenbezeichnung ist nach ISO 20022 einheitlich definiert und in Tabelle 2.1 dargestellt.

	pain.008.001.01
pain	Legt das Geschäftsfeld fest
008	Definiert den Nachrichtennamen nach UNIFI
001	Stellt das genutzte Message-Level dar
01	Gibt die Message-Version an

Tabelle 2.1: Aufbau der Nachrichtenbezeichnung

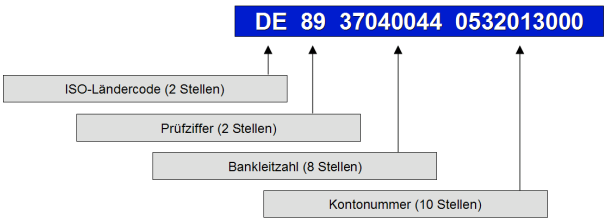


Abbildung 2.4: Aufbau der IBAN für Deutschland [Deu08b]

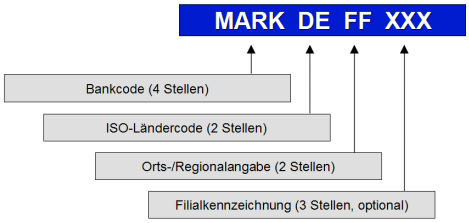


Abbildung 2.5: Aufbau des Bank Identification Codes [Deu08b]

2.1.4.1 IBAN und BIC

Die eindeutige Identifikation des Kontoinhabers erfolgt bei SEPA mit Hilfe der IBAN und dem BIC und löst damit die bisherige Kontonummer sowie die Bankleitzahl ab. Standardisiert ist die IBAN durch das European Committee for Banking Standards (ECBS) und die International Organisation for Standardization (ISO), welche das Format vorgeben. Dieser Code ist in Deutschland mit einer Länge von 22 Stellen genau festgelegt. Grundsätzlich muss die Kennung getrennt in Viererblöcken geschrieben werden und darf keine Sonderzeichen oder Kleinbuchstaben enthalten. International kann die IBAN inklusive dem zweistelligen Ländercode maximal 34 Stellen lang sein. Hinsichtlich der Vergabe der IBAN gibt es in den einzelnen Ländern verschiedene Ansätze. Während in Deutschland die Deutsche Bundesbank für die Vergabe zuständig ist, verfolgen andere Länder einen vom Markt getriebenen Ansatz. Dort bestimmen die Bankinstitute die Vergabemodalitäten und die zeitliche Ausrichtung selbst. Die IBAN beginnt mit einem zweistelligen Länderkennzeichen (gemäß ISO 3166-1), beispielsweise DE für Deutschland, und einer zweistelligen Prüfziffer (gemäß ISO 7064), die nach einem festgelegten Algorithmus berechnet werden kann. Danach folgen der acht Stellen lange Bankcode (Bankleitzahl) und die in Deutschland maximal zehnstellige Kontonummer. Hat die Kontonummer weniger als zehn Stellen, wird sie von vorn mit Nullen aufgefüllt.

Theoretisch sind in dieser IBAN alle notwendigen Informationen zur Identifizierung des Kontoinhabers enthalten. Da weltweit jedoch nicht jedes Land mit IBAN agiert, wird zusätzlich noch der BIC zur eindeutigen Identifikation mit angegeben. Mit Hilfe des BIC, der oftmals auch als SWIFT-Code bezeichnet wird, können Banken weltweit eindeutig identifiziert werden. Dabei besitzt er entweder acht oder elf Stellen und enthält, neben dem Land und dem Bankennamen, außerdem Informationen über Bankfilialen. Somit erlaubt der BIC das Routing von internationalen Zahlungen.

3 Das Lastschriftverfahren

3.1 Bisheriges Lastschriftverfahren in Deutschland

3.1.1 Grundstruktur des Lastschriftverfahrens

Das Lastschriftverfahren wird als banktechnischer Sammelbegriff für die Verbindung einer Vielzahl von Rechtsbeziehungen und Abwicklungsvorgängen verwandt und ist eine Form der bargeldlosen Zahlung. Dabei erteilt der Zahlungsempfänger seiner Hausbank den Auftrag vom Konto des Zahlungspflichtigen einen bestimmten Betrag abzubuchen. Im Gegensatz zu einer Überweisung wird der Zahlungsvorgang bei einer Lastschrift nicht vom Zahlungspflichtigen, sondern vom Zahlungsempfänger ausgelöst [Göß04, Seite 159 ff.].

Hierbei sind neben dem Zahlungspflichtigen und dem Zahlungsempfänger zusätzlich die Bank des Zahlungspflichtigen (Zahlstelle oder auch Schuldnerbank genannt) sowie die Bank des Zahlungsempfängers (1. Inkassostelle) beteiligt. Aufgrund der zwischen beiden Parteien getroffenen Lastschriftvereinbarung lässt der Zahlungsempfänger durch seine Bank (Inkassobank) die Forderung von der Bank des Zahlungspflichtigen einziehen und bekommt den Betrag zunächst unter Vorbehalt des Eingangs gutgeschrieben. Daraufhin belastet die Bank des Zahlungspflichtigen nach Erhalt der Weisung von der Bank des Zahlungsempfängers (Gläubigerbank) das Konto des Schuldners mit dem gezahlten Betrag und reicht das so Erlangte an die Gläubigerbank weiter. Dies geht aus [Dyn08] hervor.

Haben Gläubiger und Schuldner ihr Konto bei der gleichen Bank, spricht man von einer Hauslastschrift und der Zwischenschritt über andere Banken entfällt. Die Rechtsgrundlage für Lastschriften bildet das Lastschrift-Abkommen (LSA) zwischen den Spitzenverbänden der deutschen Kreditwirtschaft [Göß04, Seite 178].

Das Lastschriftverfahren wird in zwei verschiedenen Formen durchgeführt. Die Einreichung einer Lastschrift kann mittels dafür vorgesehener Vordrucke, per Datenträgeraustausch oder Online per Datenübertragung erfolgen [Bar08].

3.1.2 Einzugsermächtigungsverfahren

Das Einzugsermächtigungsverfahren (EEV) ist die bekannteste und verbreitetste Methode der Lastschriftverfahren. Hierbei gibt der zahlungspflichtige Kontoinhaber dem Zahlungsempfänger die schriftliche Erlaubnis von seinem Konto Beträge einzuziehen [Göß04, Seite 160]. In der

gängigen Praxis wird dieser Nachweis nicht verlangt, aber die beteiligten Banken können diesen bei Bedarf anfordern. Der Zahlstelle (Schuldnerbank) liegt diese Ermächtigung im Allgemeinen nicht vor. Aus diesem Grund muss der Zahlungspflichtige mit seiner Bank eine entsprechende Inkassovereinbarung abschließen. (siehe Abschnitt 3.1.4 Lastschriftinkasso).

Bei dem Einzugsermächtigungsverfahren erfolgt die Gutschrift des einzuziehenden Betrages unter Vorbehalt, somit ist eine Verfügung erst dann möglich, wenn der Bank der Geldbetrag von der Bank des Zahlungspflichtigen gutgeschrieben worden ist.

Wird das Konto des Zahlungspflichtigen unberechtigt belastet, hat er das Recht, ohne Angabe von Gründen, bei seiner Bank innerhalb von sechs Wochen zu widersprechen. Die Bank storniert daraufhin valutagerecht die Kontobelastung und gibt diese an die erste Inkassostelle zurück. Daraufhin wird der ursprünglich gutgeschriebene Betrag auf dem Konto des Zahlungsempfängers belastet. Wird erst nach dieser Zeit widerrufen ist nicht sichergestellt, dass beispielsweise unberechtigt abgebuchtes Geld zurückerstattet werden kann.

Weiterhin löst der Zahlungsempfänger die Buchung aus, ohne das der Zahlungspflichtige dies bei seiner Bank veranlassen muss.

Obwohl Kriminelle durch Kenntnis der Bankverbindung und Erschleichung eines Lastschrifteinkassos bei einer Bank von anderen Konten Beträge einziehen lassen könnten, sind die Zahlungspflichtigen durch eine sechswöchige Widerspruchsfrist, ohne Angabe von Gründen, vor unrechtmäßigen Abbuchungen geschützt [VNR08]. Die Bank selbst ist dabei nicht zu einer Überprüfung der Identität verpflichtet. Diese Zahlmethode ist vor allem bei stetigen oder in der Höhe unterschiedlichen Zahlungen wie beispielsweise Miete, Versicherungsbeiträgen oder Telefonrechnungen sehr verbreitet.

3.1.3 Abbuchungsauftragsverfahren

Beim Abbuchungsauftragsverfahren (AAV) hinterlegt der Zahlungspflichtige seiner Hausbank eine schriftliche Vollmacht, so dass der Zahlungsempfänger von seinem Konto abbuchen darf. Der Zahlungsempfänger erteilt daraufhin seiner Bank den Auftrag den entsprechenden Betrag vom Konto des Zahlungspflichtigen einzuziehen. Durch die Erteilung des Abbuchungsauftrages stimmt der Zahlungspflichtige der Belastung seines Kontos zu [Göß04, Seite 159].

Dadurch ist eine Rückbuchung wegen Widerspruchs normalerweise nicht möglich. Man kann jedoch, unter Einhaltung sehr kurzer Fristen (zumeißt binnen eines Bankarbeitstages), den Abbuchungsauftrag widerrufen.

Das Abbuchungsauftragsverfahren ist in Deutschland weitaus weniger verbreitet als das Einzugsermächtigungsverfahren. Abbuchungsaufträge kommen häufig in Bereichen vor, in denen der Zahlungsempfänger auf rasche Zahlungssicherheit aus ist. Es wird hauptsächlich zwischen Unternehmen im sogenannten Business to Business (B2B)-Bereich in bestimmten Branchen eingesetzt.

Vorteile sind bei diesem Verfahren vor allem beim Zahlungsbegünstigten (Gläubiger) zu sehen. Analog zur Überweisung kann jedoch der belastete Betrag nicht mehr zurückgefordert oder der Abbuchung widersprochen werden. Diese Variante wird insbesondere von Zahlungsempfängern gewählt, die sicherstellen wollen, dass ein einmal gutgeschriebener Betrag nicht wieder zurückgebucht werden kann. Zusätzlich werden dem Schuldner die Überwachung seiner Zahltermine abgenommen.

3.1.4 Lastschriftinkasso

Um das Lastschriftverfahren nutzen zu können, muss ein Kontoinhaber zuerst eine Inkassovereinbarung mit seiner Bank abschließen. Ein Anspruch auf Zulassung zum Lastschriftverfahren besteht dabei jedoch nicht. Erst nach Prüfung der Bonität schaltet die Bank den Kunden für das Lastschriftverfahren frei.

In der Inkassovereinbarung *Vereinbarung über den Einzug von Forderungen durch Lastschriften* werden laut [Zah08] folgende Punkte geregelt:

- Es dürfen nur fällige Forderungen eingezogen werden.
- Lastschriften sind grundsätzlich bei Vorlage fällig und es gibt keine festgelegten Wartezeiten.
- Nicht eingelöste Lastschriften werden mittels eines gesondert vereinbartem Entgeltes zurückbelastet.

3.1.5 Lastschriftrückgabe

Eine Lastschriftrückgabe kann aus mehreren Gründen erfolgen. Die Rückgabegründe werden dabei als R-Messages bezeichnet und können folgende sein:

- Für den Einzug existiert kein Konto.
- Die Kontodaten sind ungültig. Beispielsweise sind Kontonummer und Kontoname nicht identisch oder es handelt sich um fingierte Kontodaten.
- Es liegt keine Einzugsermächtigung bzw. kein Abbuchungsauftrag vor. Beispielsweise ist das Konto für Lastschriften gesperrt.
- Die Lastschrift wird zurückgerufen
- Es liegt ein Widerspruch für den Einzug vor. Die Lastschrift muss deshalb zurückgegeben werden.

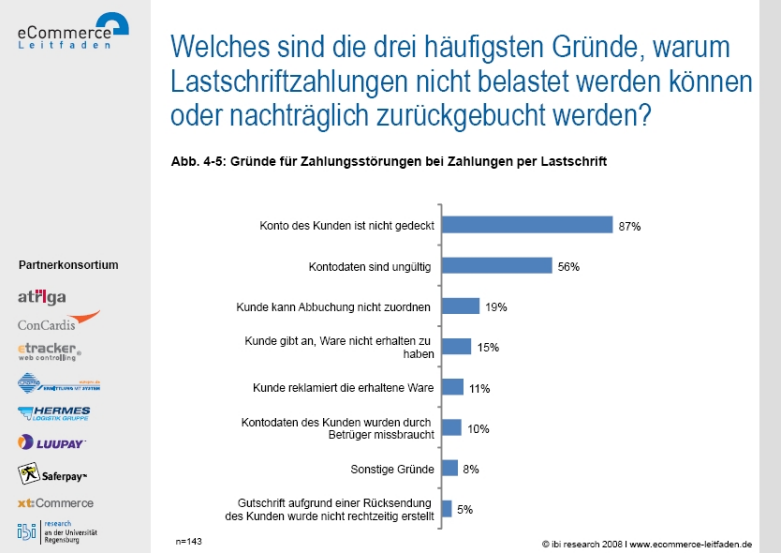


Abbildung 3.1: Gründe für Zahlungsstörungen bei Zahlungen per Lastschrift [IBI08]

3.1.6 Europäische Lastschriftzahlungen per Internet im Vergleich

Die Lastschrift ist mit einem Anteil von 64% in Deutschland das zurzeit wichtigste Bezahlverfahren im Internet. Voraussetzung für diese Variante ist lediglich ein Bankkonto in Deutschland sowie die schriftliche Einwilligung des Zahlungspflichtigen mit seiner Hausbank für die Teilnahme am Lastschrift-Einzugsverfahren [Lei08]. Nach Deutschland ist Spanien mit 56% der Internetzahlungen per Lastschrift an zweiter Stelle. Im Unterschied zu den anderen Ländern muss bei einer Bezahlung zusätzlich noch die Ausweisnummer mit angegeben werden, um so die Zahlung vom Endverbraucher autorisieren zu lassen. In den Niederlanden werden etwa 42% aller Internetzahlungen per Lastschrift abgewickelt. Dabei gewähren niederländische Banken, im Gegensatz zu anderen Ländern, eine Widerrufsfrist von 35 Tagen. In Österreich und in Großbritannien gewinnt die Bezahlung von Internetbestellungen per Lastschrift immer mehr an Bedeutung. Dafür muss das Konto durch den Kontoinhaber einmalig bei der Bank freigeschaltet werden, welches mittlerweile standardmäßig bei der Kontoeröffnung geschieht. Während in den USA die Zahlung per Lastschrift als ACH oder Direct Payment bekannt ist und von knapp 43% der Haushalte genutzt wird, werden in Norwegen und der Schweiz Lastschriftverfahren per Einzugsermächtigung nicht angeboten. Dies alles verdeutlicht, welchen Stellenwert die Bezahlung per Lastschrift momentan einnimmt und zeigt einen zunehmenden Trend auf.

3.2 SEPA Direct Debit

Unterschiedliche rechtliche Rahmenbedingungen und verschiedene nationale Regelungen behindern heutzutage einen grenzüberschreitenden Geldtransfer per Lastschriftverfahren. Ein reibungsloser Ablauf lässt sich derzeit nur mit einem gewissen Aufwand erreichen. Aus diesem Grund entschied sich das EPC schon frühzeitig gegen die Harmonisierung von bestehenden

nationalen Lastschriftverfahren und entwickelte ein völlig neues Lastschriftverfahren namens SDD.

Hierbei orientierte man sich beispielsweise an die vom deutschen Einzugsermächtigungsverfahren bekannte Autorisierung zwischen Schuldner und Gläubiger. Ebenso übernahm man das Widerspruchsrecht und passte es an. Außerdem wurden bestehende Regelungen der EU-Standardüberweisung ausgebaut. Überweisende (Debtors) und Begünstigte (Creditors) müssen sich in diesem neuen Verfahren mittels IBAN und BIC identifizieren. Grundsätzlich gibt es auch Neuerungen, wie beispielsweise eine fest definierte Vorlaufzeit zur Vorlage einer Lastschrift bei der Zahlstelle.

Folgender Überblick stellt die wichtigsten Neuerungen in der SEPA-Lastschrift dar:

- Die SEPA-Lastschrift ist europaweit einheitlich standardisiert.
- Der Zahlungsempfänger bestimmt einen exakten Fälligkeitstermin (Due Date).
- Zur Identifizierung der Kundenkonten dient die Verwendung von IBAN und BIC.
- Alle Lastschriften müssen einheitlich in der Währung Euro erfolgen.
- Das Verfahren erlaubt eine eindeutige Identifizierung des Lastschriftanreichers.
- Der Zahlungspflichtige hat längere Widerspruchsfristen und ist damit besser geschützt.
- Die Autorisierungsinformationen müssen bei jeder Lastschrift in Form eines Mandats mit übermittelt werden.

Durch fehlende rechtliche Bestimmungen sind in machen Ländern jedoch noch keine einheitlichen juristischen Grundlagen für ein Lastschriftverfahren gegeben. Deshalb hat die EU im Dezember 2005 einen Vorschlag für einen neuen einheitlichen Rechtsrahmen vorgelegt, welcher in die *Richtlinie für Zahlungsdienstleistungen* (Richtlinie 2007/64/EG) mündete. Dieser wird auch als Payment Services Directive (PSD) bezeichnet und muss von den Mitgliedstaaten der Europäischen Union und des Europäischen Wirtschaftsraums bis zum 1. November 2009 in nationales Recht umgesetzt werden. Für Direct Debits hat das EPC im Juni 2008 eine Empfehlung zur Umsetzung in Form eines Regelwerkes herausgegeben, welches momentan in der Version 3.1 unter dem Namen SEPA Core Direct Debit Scheme Rulebook zu finden ist.

3.2.1 Die Teilnehmer im SDD-Verfahren

Die Funktionsweise der SEPA Lastschrift lässt sich nach [Eur08b] an folgendem Diagramm erläutern. Dabei bilden der Zahlungspflichtige (Debtor) und der Zahlungsempfänger (Creditor) zusammen mit ihrer jeweiligen Bank die Teilnehmer. Zwischen beiden Banken befinden sich die Clearing-Stellen. Der Clearing and Settlement Mechanism (CSM) stellt dabei das genutzte Verfahren zwischen den Clearing-Häusern dar.

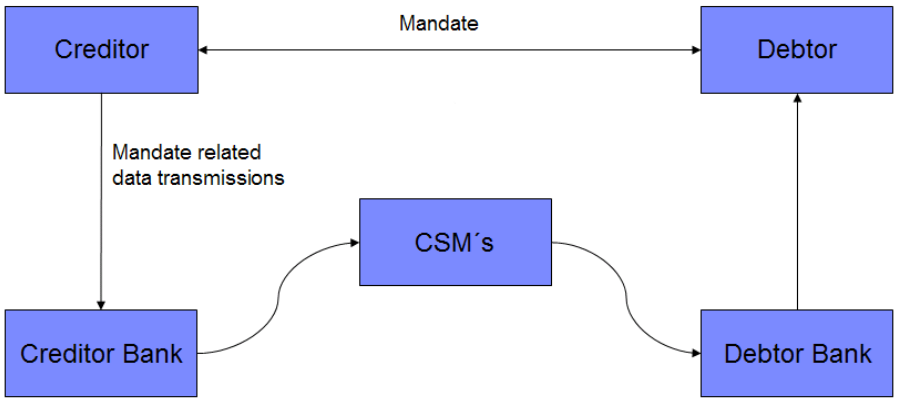


Abbildung 3.2: Vier-Ecken-Modell von SEPA Direct Debit [Eur08b, Seite 21]

Für den Creditor ergeben sich durch die Nutzung von SEPA Direct Debit viele Vorteile. So kann er sich durch die Angabe eines exakten Fälligkeitsdatum genau auf den Tag des Geldeingangs einstellen. Weiterhin stellt die SEPA-Lastschrift einen einfachen und kostengünstigen Weg für den Geldeinzug dar, welcher ab diesem Zeitpunkt mit einem einzelnen Zahlungsinstrument in 31 Ländern getätigt werden kann. Der Debtor bekommt ebenfalls ein zuverlässiges Verfahren bereitgestellt, welches von allen SEPA-Creditoren akzeptiert ist. Innerhalb von acht Wochen kann ein Debtor autorisierten Lastschriften widersprechen und unerlaubte Abbuchungen können sogar bis zu 13 Monaten nach dem Tag der Abbuchung zurückgefordert werden.

3.2.2 Voraussetzungen

Um das SEPA-Lastschriftverfahren nutzen zu können, ist eine *Vereinbarung zur Teilnahme am SEPA-Lastschriftverfahren* des Debtors mit seiner Bank notwendig. Hier hat der Debtor laut Regelwerk nur die Möglichkeit entweder komplett oder gar nicht diesem Verfahren zuzustimmen.

3.2.2.1 Gläubiger-Identifikationsnummer

Zusätzlich benötigt der Creditor eine SEPA-konforme Gläubiger-Identifikationsnummer, anhand dessen er eindeutig identifizierbar ist und die jeder Lastschrift mitgegeben werden muss. Die Gläubiger-Identifikationsnummer wird in Deutschland zentral durch die Deutsche Bundesbank vergeben und ist für Deutschland mit genau 18 Stellen festgelegt. Die Ausgabe erfolgt unabhängig von den rechtlichen Eigenschaften und der wirtschaftlichen Situation des Antragstellers und enthält diesbezüglich auch keine Aussagen oder Bewertungen der Deutschen Bundesbank über ihn. Sinn und Zweck des Unique Creditor Identifier (UCI) ist die eindeutige Identifikation des Lastschrifteinreichers. Unter `http://www.glaeubiger-id.bundesbank.de/` kann derzeit ausschließlich per Webformular eine UCI beantragt werden. Alternative Wege, beispielsweise auf schriftlichen oder telekommunikativen Wege, sind dafür bisher nicht vorgesehen.

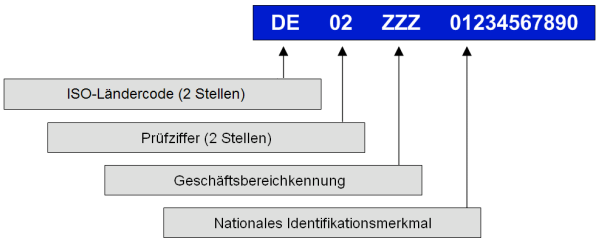


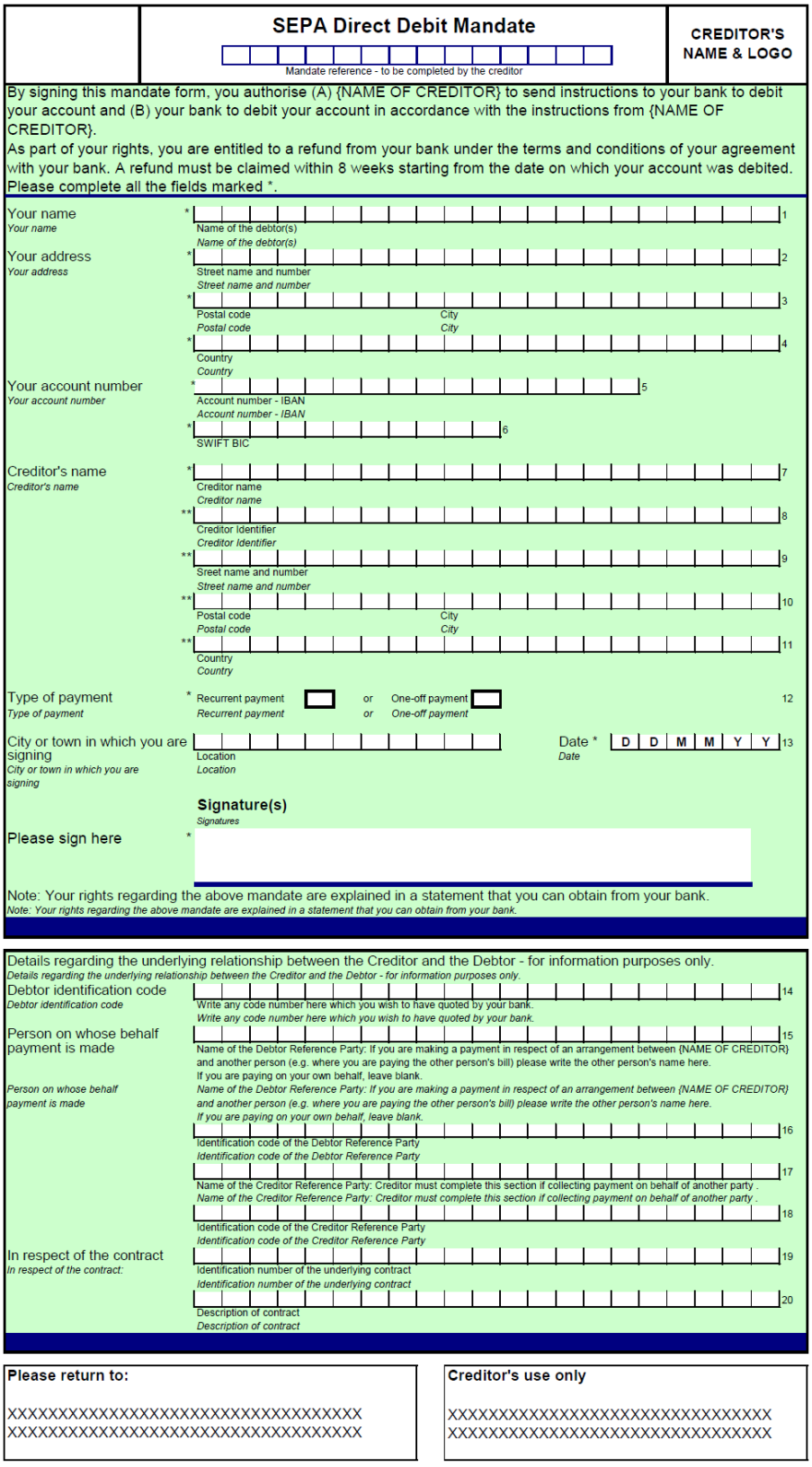
Abbildung 3.3: Aufbau der Gläubiger-Identifikationsnummer [Deu08c]

Der Aufbau der Identifikationsnummer ist hingegen SEPA-weit einheitlich geregelt. Sie wird aus einem ISO-Ländercode, einer zweistelligen Prüfziffer, der Geschäftsbereichkennung (Business Area Code) und einem nationalen Identifikationsmerkmal zusammengesetzt. Dabei kann das nationale Identifikationsmerkmal in der Länge variieren, darf jedoch höchsten 28 Stellen aufweisen. Bis auf weiteres wird deshalb die achte Stelle der Gläubiger-Identifikationsnummer mit dem Wert 0 belegt. Außerdem variiert die Länge der Gläubiger-Identifikationsnummer von Land zu Land, ist jedoch auf maximal 35 Stellen begrenzt.

3.2.2.2 Das Mandat

Unter dem Begriff Mandat ist eine Vollmacht zu verstehen. Diese ist notwendig, damit der Lastschriftempfänger (Creditor) Verbindlichkeiten vom Konto des Lastschriftschuldners (Debtor) einziehen darf. Im Gegensatz zum bisher in Deutschland etablierten Verfahren wurde vom EPC ein neuer Prozessablauf für den Umgang mit Mandaten entworfen. Voraussetzung für den Einzug einer SEPA-Lastschrift, ist das Vorliegen eines gültigen Mandats, dass über die heute aus dem deutschen Lastschriftverfahren bekannte Einzugsermächtigung hinausgeht. Das Mandat muss in Papierform oder in elektronischer Form erteilt werden und ist bei jeder Einreichung einer SEPA-Lastschrift im Datensatz an die zahlende Bank mit zu übermitteln. Weiterhin verlieren Mandate, wenn sie innerhalb von 36 Monaten nicht genutzt werden, automatisch ihre Gültigkeit. Außer der Ermächtigung des Creditors zum Einzug der Lastschrift, enthält das Mandat auch eine Weisung zur Bezahlung an die Bank des Debtors. Eine eindeutige Identifikation eines Mandats ergibt sich durch die Kombination der Unique Mandate Reference mit dem Creditor Identifier. Das Mandatsformular ist in Abbildung 3.4 dargestellt und die dabei genutzten Felder sind in Tabelle 3.1 und 3.2 abgebildet.

Die technische Beschreibung ist vom EPC durch ein Regelwerk definiert. Hierfür bildet das SEPA Direct Debit Core Schema in der Version 3.1 die Grundlage. Für die speziellen Anforderungen im Business-To-Business-Umfeld, also der ausschließlichen Nutzung durch Firmenkunden, wurde ein eigenes Regelwerk spezifiziert. Es setzt dabei auf dem Core-Regelwerk auf und ist dem bisherigen Abbuchungsauftragsverfahren ähnlich. Das B2B-Regelwerk liegt zum Zeitpunkt der Erstellung dieser Arbeit in der Version 1.1 vor und wird im Abschnitt 3.2.5 näher erläutert.



SEPA Direct Debit Mandate

Mandate reference - to be completed by the creditor

CREDITOR'S NAME & LOGO

By signing this mandate form, you authorise (A) {NAME OF CREDITOR} to send instructions to your bank to debit your account and (B) your bank to debit your account in accordance with the instructions from {NAME OF CREDITOR}.
As part of your rights, you are entitled to a refund from your bank under the terms and conditions of your agreement with your bank. A refund must be claimed within 8 weeks starting from the date on which your account was debited.
Please complete all the fields marked *.

Your name / *Your name*
* Name of the debtor(s) / *Name of the debtor(s)* 1

Your address / *Your address*
* Street name and number / *Street name and number* 2
* Postal code / *Postal code* City / *City* 3
* Country / *Country* 4

Your account number / *Your account number*
* Account number - IBAN / *Account number - IBAN* 5
* SWIFT BIC 6

Creditor's name / *Creditor's name*
* Creditor name / *Creditor name* 7
** Creditor Identifier / *Creditor Identifier* 8
** Sreet name and number / *Street name and number* 9
** Postal code / *Postal code* City / *City* 10
** Country / *Country* 11

Type of payment / *Type of payment*
* Recurrent payment / *Recurrent payment* ☐ or / *or* One-off payment / *One-off payment* ☐ 12

City or town in which you are signing / *City or town in which you are signing*
Location / *Location*
Date * D D M M Y Y / *Date* 13

Signature(s)
Signatures

Please sign here *

Note: Your rights regarding the above mandate are explained in a statement that you can obtain from your bank.
Note: Your rights regarding the above mandate are explained in a statement that you can obtain from your bank.

Details regarding the underlying relationship between the Creditor and the Debtor - for information purposes only.
Details regarding the underlying relationship between the Creditor and the Debtor - for information purposes only.

Debtor identification code / *Debtor identification code*
Write any code number here which you wish to have quoted by your bank. / *Write any code number here which you wish to have quoted by your bank.* 14

Person on whose behalf payment is made / *Person on whose behalf payment is made*
Name of the Debtor Reference Party: If you are making a payment in respect of an arrangement between {NAME OF CREDITOR} and another person (e.g. where you are paying the other person's bill) please write the other person's name here. If you are paying on your own behalf, leave blank. / *Name of the Debtor Reference Party: If you are making a payment in respect of an arrangement between {NAME OF CREDITOR} and another person (e.g. where you are paying the other person's bill) please write the other person's name here. If you are paying on your own behalf, leave blank.* 15
Identification code of the Debtor Reference Party / *Identification code of the Debtor Reference Party* 16
Name of the Creditor Reference Party: Creditor must complete this section if collecting payment on behalf of another party . / *Name of the Creditor Reference Party: Creditor must complete this section if collecting payment on behalf of another party .* 17
Identification code of the Creditor Reference Party / *Identification code of the Creditor Reference Party* 18

In respect of the contract / *In respect of the contract:*
Identification number of the underlying contract / *Identification number of the underlying contract* 19
Description of contract / *Description of contract* 20

Please return to:

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

Creditor's use only

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

Abbildung 3.4: Das Mandatsformular im Core-Prozess [Eur08b, Seite 68]

Feldname	Englische Bezeichnung	Deutsche Bezeichnung
AT-01	Unique Mandate reference	Eindeutige Mandatsreferenz (Identifikationsnummer), welche vom Creditor vergeben wird
AT-14	Name of Debtor	Name des Debtors
AT-09	Address of Debtor	Adresse des Debtors
AT-07	Account number - IBAN of Debtor	IBAN des Debtors
AT-13	SWIFT BIC of the bank of Debtor	SWIFT BIC der Bank des Debtors
AT-03	Name of the Creditor	Name des Creditors
AT-02	Creditor Identifier	Die eindeutige Identifikationsnummer des Creditors
AT-05	Address of Creditor	Adresse des Creditors
AT-21	Type of Payment (Recurrent/One-off payment)	Zahlungsart (wiederkehrende/einmalige Lastschrift)
AT-25	Date of Signing	Datum, wann das Mandat unterschrieben wurde
AT-33	Signature(s) of the Debtor	Unterschrift des Debtors

Tabelle 3.1: Notwendiger Inhalt eines Mandats [Eur08b, Seite 71]

Feldname	Englische Bezeichnung	Deutsche Bezeichnung
AT-27	Debtor identification code	Der Identifikationscode des Debtors
AT-15	Name of the Debtor Reference Party	Name des Debtor-Stellvertreters für Zahlungen
AT-37	Identification code of Debtor Reference Party	Identifikationscode des Debtor-Stellvertreters für Zahlungen
AT-38	Name of the Creditor Reference Party	Name des Debtor-Stellvertreters für Zahlungen
AT-39	Identification code of the Creditor Reference Party	Identificationscode des Creditor-Stellvertreters für Zahlungen
AT-08	Identifier of the underlying contract	Vertragsreferenz

Tabelle 3.2: Optionaler Inhalt eines Mandats [Eur08b, Seite 71]

Weiterhin kann ein Creditor mehr als einen UCI benutzen. Zur genaueren Unterscheidung seiner verschiedenen Geschäftsaktivitäten ist ein dreistelliger Creditor Business Code (Geschäftsbereichkennung) vorgesehen.

3.2.3 Ablauf von SEPA Direct Debits

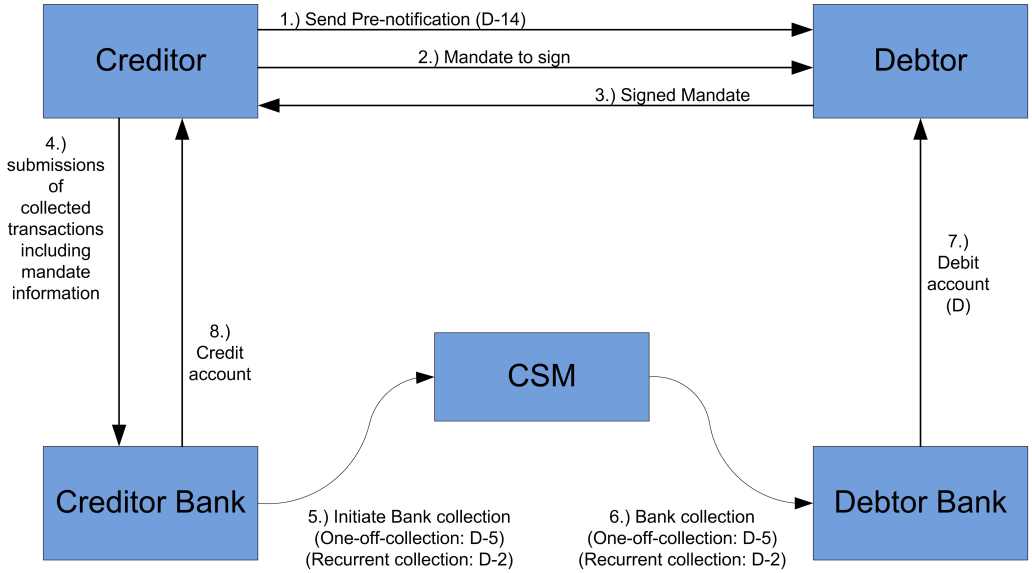


Abbildung 3.5: Darstellung einer SEPA-Lastschriftzahlung [SN08]

Nachdem der Debtor eine Leistung des Creditors in Anspruch genommen hat, ist er zur Zahlung der Leistung verpflichtet. Diese möchte er in Form einer Lastschrift tätigen, welche im Folgenden in Form eines Ablaufprozesses näher erläutert wird.

Begonnen wird eine Lastschrift, sofern nicht anders vereinbart, 14 Tage vor Belastung des Debtors durch das Senden einer Vorabinformation (beispielsweise einer Rechnung) vom Creditor zum Debtor. Hat der Creditor noch keine Autorisierung des Debtors in Form eines Mandats vorliegen, schickt er ein Mandatsformular zum Debtor. Dieser füllt es aus, unterzeichnet es schriftlich oder nutzt dafür ein signiertes elektronisches Dokument und sendet das erteilte Mandat zurück an den Creditor. Das Papiermandat wird elektronisch erfasst und digitalisiert (dematerialisiert). Der Creditor muss das erstellte und unterschriebene Mandat speichern und entsprechend den gesetzlichen Bestimmungen aufbewahren. Mit diesem Mandat ist der Creditor nunmehr berechtigt per Lastschrift die Belastung des Debtors auszulösen und sendet eine Belastungsankündigung an den Debtor. Im nächsten Schritt werden die Mandatsinformationen in Form einer Zahlungsanweisung (Payment Instruction) an die Creditor-Bank mitgegeben. Hierbei sind Vorlauffristen zu beachten, die sich nach der Art der Lastschrift unterscheiden. Für eine Erst- bzw. Einmallastschrift (First Direct Debit) muss die Transaktion mindestens fünf Tage vor Belastung des Debtors bei der Creditor-Bank eingereicht werden. Die Mindestlaufzeit verkürzt sich bei wiederkehrenden Lastschriften (Recurrent Direct Debits) auf zwei Tage. Beide

Transaktionsarten müssen dementsprechend gekennzeichnet werden. Danach schickt die Creditor-Bank eine PACS-Nachricht (per Clearing & Settlement Mechanism) an die Debtor-Bank, welche eine Belastung des Kontos vornehmen muss und den fälligen Betrag mittels einer PACS-Nachricht an den Creditor übermittelt. Hierbei ist sicherzustellen, dass die Lastschriftdaten spätestens zwei respektive fünf Tage vor der Belastungsbuchung bei der Creditor-Bank eintreffen. An dieser Stelle kann die Debtor-Bank innerhalb der Vorlauffrist ihren Kunden (den Debtor) kontaktieren und die Belastungsbuchung genehmigen lassen. Dies ist jedoch nicht verpflichtend im Regelwerk festgelegt, sondern nur als optionaler Service seitens der Banken definiert. Am vorgegebenen Belastungstag (due date) wird das Konto des Debtors durch dessen Bank mit dem in der Lastschrift vorgegebenen Betrag belastet. Weiterhin gelten Lastschriften solange als autorisiert, bis der Debtor gegenüber dem Creditor sein Mandat widerruft. Ebenfalls wird der Debtor-Bank keine Pflicht zur Administration der Mandate auferlegt. Allerdings kann auch hier eine Mandatsadministration und Mandatsprüfung durch die Bank eine Mehrwertdienstleistung darstellen.

Im Gegensatz zur bisherigen Einzugsermächtigung erteilt der Debtor nun zwei Weisungen:

- Die erste Weisung an den Creditor, dass Zahlungen mittels Lastschrift vom Bankkonto des Debtors einzuziehen sind.
- Die zweite Weisung an die Debtor-Bank (Zahlstelle), welche die vom Creditor gezogene Lastschrift einlöst.

Somit liegt eine Autorisierung der Debtor-Bank zur Belastung des Debtor-Kontos vor. Die Weisung an die Debtor-Bank wird dabei nicht direkt vom Debtor an die Debtor-Bank übermittelt, sondern werden vom Creditor mittels SEPA-Lastschriftdatensatz an die Creditor-Bank gesendet, welche diese im Anschluss zur Debtor-Bank weiterleitet. Folglich ist damit eine Belastungsbuchung auf dem Debtor-Konto auch gegenüber der Debtor-Bank genehmigt.

Als Unterschied lässt sich feststellen, dass Zahlungen bereits vorab gegenüber der Debtor-Bank genehmigt sind und nicht, wie bei der bisherigen Einzugsermächtigung, erst nachträglich von ihr genehmigt werden. Somit muss auch der Creditor dem Debtor den genauen Tag der Belastung zusenden. Dieser Belastungstag ist dabei im Einzugsverfahren genau einzuhalten, so dass sich der Debtor auf die folgende Kontodisposition einstellen kann.

3.2.4 Abbruchbedingungen

Der Debtor hat das Recht nicht am SEPA-Lastschriftverfahren teilzunehmen und somit der Debtor-Bank das Ausführen von Lastschriften zu Lasten seines Kontos zu verbieten. Ebenso kann der Debtor sein Mandat zu jeder Zeit widerrufen. Hierfür sollten mögliche Sperrvermerke zu SEPA-Lastschriften überprüft werden. Weiterhin können auch die Banken, unter Angaben von

bestimmten Gründen, Lastschriften zurückweisen. Dies kann sowohl vor, als auch nach der Interbankenverrechnung (Clearing) geschehen. Nachfolgend sind die Rückgabegründe, welche in Form von R-Transaktionen getätigt werden, aufgelistet und in der Abbildung 3.6 zusasammengefasst.

- **Vor der Interbankenverrechnung**
  1. **Rejects (Abweisungen):**
     - Auslöser ist die Creditor-Bank
     - Lastschriften, die vom normalen Prozessablauf abweichen
       * Rückweisungen wegen Datenfehler
       * Validierungsfehler (falsches Format, falsche IBAN Prüfsumme)
       * Einspruch des Debtors vorab
  2. **Refusals (Ablehnungen):**
     - Auslöser ist der Debtor
     - Verweigerung der Abbuchung durch den Creditor nach Vorabinformation (noch vor der Buchung)
     - Refusals sind bei Erstlastschrift innerhalb von fünf Tagen, bei Folgelastschrift innerhalb von zwei Tagen vor der Buchung möglich
  3. **Request for Cancellations (Annulierungsanforderung):**
     - Auslöser ist die Creditor-Bank
     - Widerruf der Anweisung für eine Lastschrift

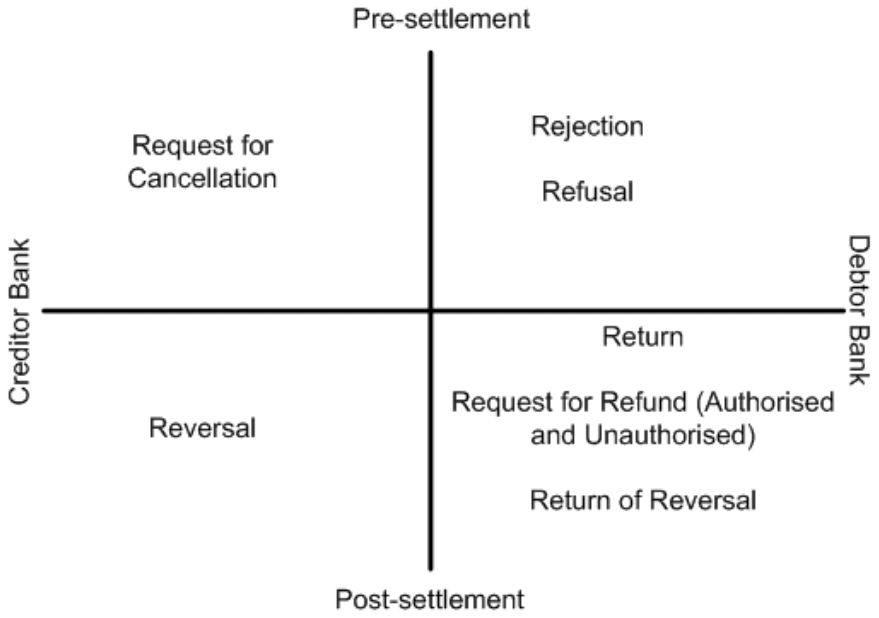


Abbildung 3.6: Zusammenfassung der R-Messages [EBA08a, Seite 11]

- **Nach der Interbankenverrechnung**
  1. **Returns (Rückgaben):**
     - Initiator ist die Debtor-Bank
     - Vom normalen Prozessablauf abweichende Lastschriften
       * Rückweisungen aufgrund technischer und fachlicher Ursachen
       * Technische Gründe (falsches Format, falsche IBAN Prüfsumme)
       * Kein SEPA Konto, Konto geschlossen, mangelnde Deckung
  2. **Reversals (Rückrechnungen):**
     - Auslöser ist der Creditor
     - Korrektur eines gebuchten Lastschrifteinzuges (beispielsweise stellt der Creditor fest, dass eine Lastschrift nicht hätte durchgeführt werden dürfen)
  3. **Refunds (Widersprüche):**
     - Auslöser ist der Debtor
     - Widerspruch des Debtors nach Beteiligtenbuchung
     - Autorisierte und nicht autorisierte Widersprüche sind möglich
     - Im B2B-Schema ist kein Widerspruch bei einer autorisierten Lastschrift möglich

3.2.5 Besonderheiten im B2B-Umfeld

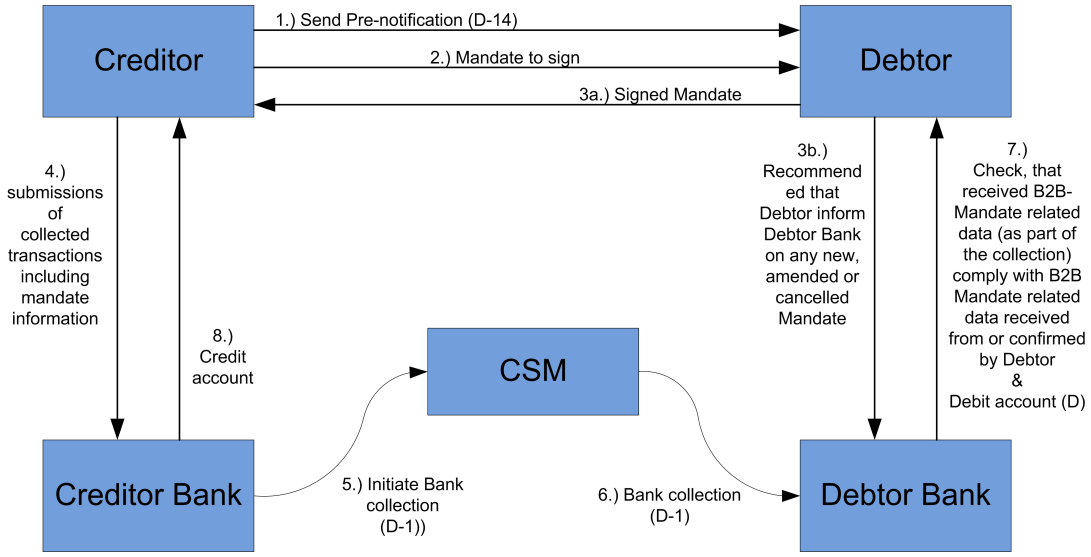


Abbildung 3.7: Zahlungsablauf im B2B-Verfahren [Eur08a, Seite 29]

Speziell für die Nutzung durch Firmenkunden wurde ein B2B-Schema entwickelt und spezifiziert. Dieses bildet durch kürzere Ablaufzeiten die Basis für Businesskunden, welche Lastschriften als Teil ihres Geschäftsvorganges nutzen. Die Einreichungszeit beträgt für Lastschriften nur noch einen Interbanken-Tag, anstatt zwei oder fünf Tagen beim Core-Regelwerk. Ein weiterer wichtiger Unterschied lässt sich im Widerspruchsrecht finden. So ist es dem Debtor nicht gestattet erlaubte Transaktionen, d.h. Transaktionen welche durch ein gültiges Mandat gedeckt sind, zurückzuweisen. Dies bringt neben einer höheren Zahlungssicherheit auch eine schnellere Verarbeitung. Um die Sicherheit der Zahlung zu gewährleisten, wird eine zusätzliche Überprüfung vor der eigentlichen Belastung des Debtor-Kontos vorgenommen.

Die Debtor-Bank ist vor der Kontobelastung verpflichtet die B2B-Mandatsdaten, welche sowohl mit der eingehenden Lastschrift aus dem CSM empfangen werden, als auch mit den Mandatsdaten die sie vom Debtor bestätigt bekommt, gegenzuprüfen. Weiterhin ist zu prüfen, ob die B2B-Mandatsdaten vom Debtor ordnungsgemäß vorliegen und autorisiert sind.

Beispielsweise könnte hierfür der Debtor eine Kopie des Mandates an seine Bank schicken. Eine andere Variante stellt die selbstständige Rücksprache der Bank mit ihm dar. Somit kann er sichergehen, dass der Einzug erst von seiner Bank überprüft wird, bevor eine Belastung des Kontos stattfindet.

Das B2B-Schema unterscheidet sich vom Core-Schema insbesondere durch folgende Punkte:

- Die Nutzung des B2B-Verfahrens ist ausschließlich Firmenkunden gestattet.
- Bevor der Debtor belastet werden kann, muss die Debtor-Bank die Mandatsdaten aus der Lastschrift mit dem vom Debtor bestätigten Mandat gegenprüfen.
- Der Debtor ist nicht berechtigt eine autorisierte Lastschrift zurückzuweisen.
- Der Debtor kann eine nichtautorisierte Lastschrift innerhalb von 13 Monaten nach Belastung gegen seine Debtor-Bank widerrufen, jedoch nur wenn die Lastschrift nicht durch ein gültiges B2B-Mandat gedeckt ist.
- Die Lastschrift muss spätestens einen Tag (anstatt zwei Tagen im Core-Schema) vor dem Buchungsdatum bei der Debtor-Bank eingehen und darf frühestens 14 Tage vorher gesendet werden.
- Das B2B-Mandat nutzt ein eigenes Formular namens SEPA Business-to-Business Direct Debit Mandate.
- Formale Unterschiede zwischen Core- und B2B-Mandat sind laut Regelwerk lediglich im Feld Identification Code und in der Farbgestaltung zu finden.

3.2.6 Mandatsänderung und -löschung

Für die Änderung eines Mandats ist ebenfalls ein Prozessablauf vom EPC entwickelt worden. Im Core-Schema wird festgelegt, dass der Debtor die Änderungen am Mandat mit dem Creditor abstimmt, jedoch ohne seine Debtor-Bank hierüber direkt zu informieren. Der Creditor dematerialisiert und archiviert die Daten und hängt diese einer Lastschrift an. Für diesen Zweck sind bereits spezielle Felder im Mandat vorgesehen. Anschließend leitet er den Einzug über seine Creditor-Bank und CSM weiter zur Debtor-Bank. Diese ist jetzt ebenfalls über die Änderung informiert und kann ihrem Debtor noch weitere Services anbieten, ohne jedoch verpflichtet zu sein. Analog zur Änderung verhält sich der Prozess zur Löschung des Mandats.

Anders ist der Sachverhalt im B2B-Verfahren geregelt. Der Debtor ist verpflichet, bei einer Änderung oder Löschung des Mandats seine Debtor-Bank direkt zu informieren. Dies ist notwendig, damit eine Überprüfung der Lastschrift auf Seite der Debtor-Bank nicht durch veraltete Mandatsdaten abgewiesen wird. Der Creditor wird, ebenso wie im Core-Schema, vom Debtor über Mandatsänderungen oder -löschungen informiert und ist verpflichtet diese zu archivieren. Der anschließende Prozessablauf ist mit dem oben genannten Prozess identisch.

3.2.7 Mandatskopie anfordern

Hat die Debtor-Bank kein entsprechendes Mandat vorliegen, kann sie das Mandat vom Creditor anfordern. Hierfür stellt sie einen Request an die Creditor-Bank, welcher die Anforderung an den Creditor weitergibt. Dieser sendet daraufhin eine Kopie des Mandats an seine Creditor-Bank zurück, welche diese anschließend zur Debtor-Bank weiterleitet. Das Core- und B2B-Verfahren haben hierbei den gleichen Prozessablauf.

3.2.8 EBA-Beispiel

Nachdem der logische Weg einer Zahlung bekannt ist, soll im Folgenden Abschnitt die technische Umsetzung näher beleuchtet werden. Viele Zahlungen (Transactions) sind in einen Bulk zusammengefasst und mit Headerinformationen versehen, beispielsweise die Summe aller Zahlungen in Euro. Mit diesen Informationen lässt sich die Datenintegrität überprüfen. Die Bulks werden in einem Container transportiert, der weitere Attribute zur Verfügung stellt um die Datenkonsistenz der Bulks zu sichern. In der Kommunikation zwischen Kunde und Bank werden, wie bereits erläutert, PAIN-Nachrichten eingesetzt. Diese können vom jeweiligen Kreditinstitut angepasst werden und sind nicht einheitlich festgelegt, lediglich die Grundstruktur gibt das EPC vor. Anders verhält es sich bei den PACS-Nachrichten. Diese sind vom EPC genau spezifiziert und können schon jetzt umgesetzt werden. Für die verschiedenen PACS-Zahlungsnachrichten (Einlieferung, Rückweisung) gibt es fest definierte Containerformate (IDF, RSF, DVF, etc.). Diese sind insbesondere fürs Routing der Zahlungsnachrichten von Bedeutung, welches sich am Beispiel des international tätigen Clearing-und-Settlement-Dienstleister EBA Clearing verdeutlichen

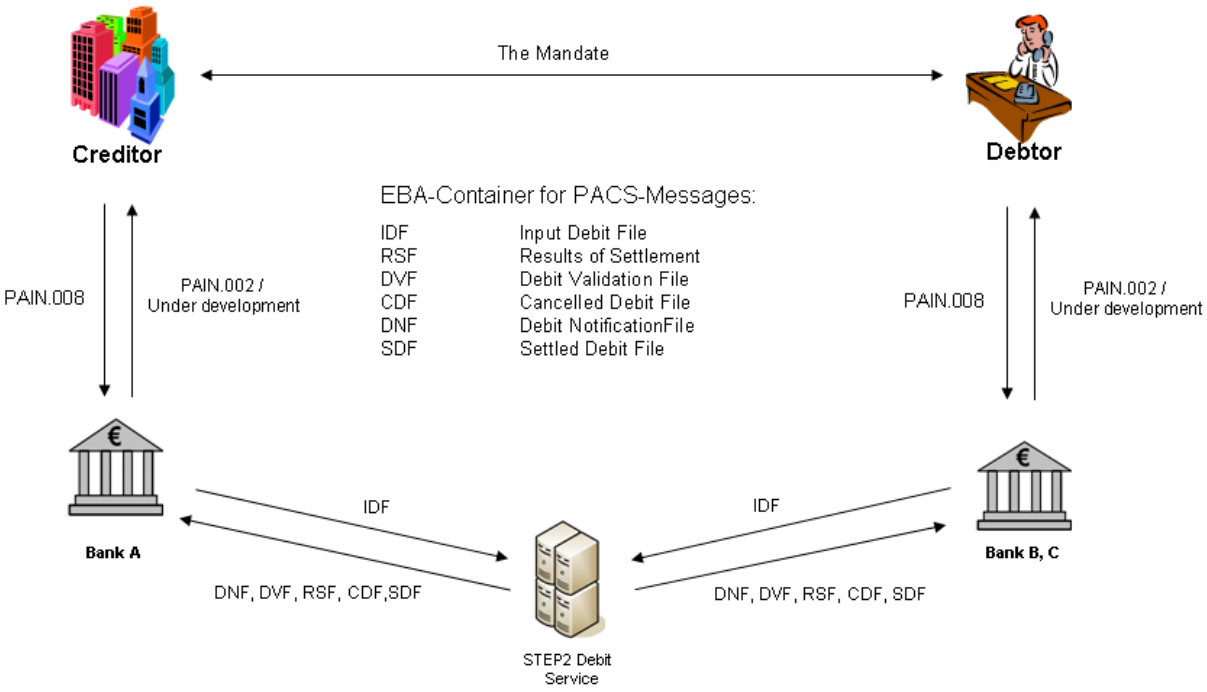


Abbildung 3.8: Beispiel des Zahlungsablaufes am EBA-Clearing [EBA08a, Seite 21]

lässt und in Abbildung 3.8 sowie 3.9 dargestellt ist. Der genaue Ablaufplan mit Einreichung und Rückweisungen der Nachrichten ist im Anhang unter Abbildung A.1 zu finden. Dabei sind sowohl die speziellen Container als auch ihre enthaltenen PACS-Nachrichten berücksichtigt.

3.2.9 Zusammenfassung SDD

Durch das neue SEPA-Lastschriftverfahren können Bankverbindungen gestrafft und das Liquiditätsmanagement vereinfacht werden. Kostensenkungspotentiale für Unternehmen ergeben sich insbesondere durch die Möglichkeit der internen Konzentration der Zahlungsverkehrsabwicklung. Dies ist insbesondere im Zeitalter der Globalisierung ein ernstzunehmender Faktor, auch wenn die Umstellung auf SEPA-Lastschrift nicht sofort stattfinden wird, jedoch in den nächsten Jahren unumgänglich scheint.

Der Grund für eine schrittweise Umstellung auf SEPA ergibt sich vor allem daraus, dass sich in Deutschland ein gut ausgebautes und funktionierendes Lastschriftverfahren etabliert hat. Dabei ist die Einfachheit und Schnelligkeit dieses Lastschriftverfahren hervorzuheben. So hat es laut [Mit08] im Jahre 2006 in Deutschland 7,36 Milliarden Lastschrifttransaktionen gegeben, während es europaweit 15,09 Milliarden waren. Fast die Hälfte aller europäischen Lastschrifttransaktionen sind demnach in Deutschland abgewickelt wurden. Allerdings ergeben sich durch die schnelle Abwicklung auch Defizite in der Rechtssicherheit des Verfahrens, welche mit dem SEPA-Lastschriftverfahren verbessert werden sollen.

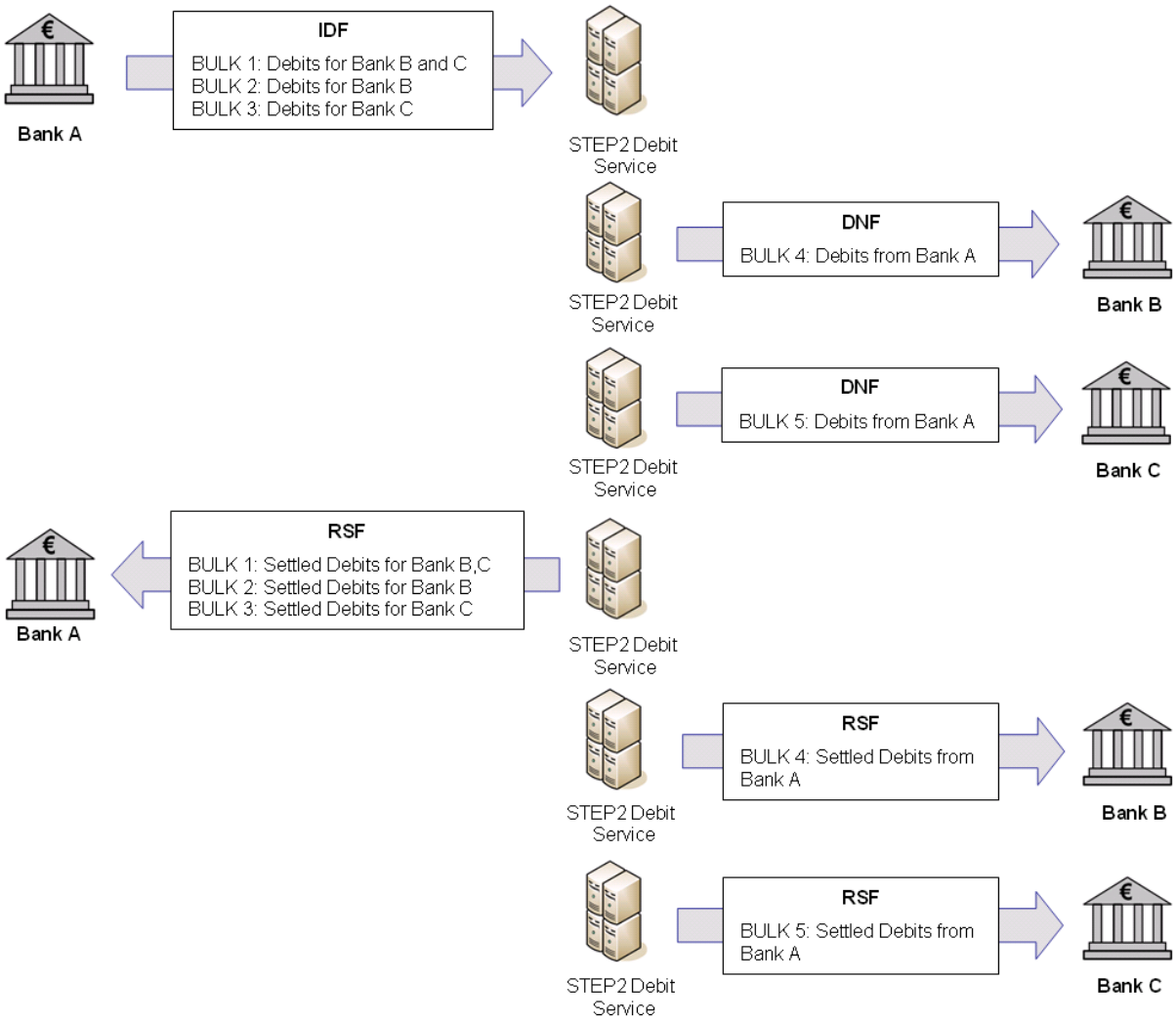


Abbildung 3.9: Schematische Darstellung der Nachrichtenverrechnung [EBA08b, Seite 19]

4 Spezifikation des Mandatsmanagements

4.1 Definition Mandatsmanagement

Durch die Einführung der SEPA-Lastschrift wird es notwendig, jede Lastschrift um die entsprechenden Mandatsinformationen zu ergänzen. Ein Mandat ist, wie bereits im Kapitel 3 unter Abschnitt 3.2.2.2 aufgeführt, die Erlaubnis des Creditors vom Debtor Geld einzuziehen. Dafür müssen Mandate, respektive die Inhalte der Mandate, erfasst, gespeichert und aufbewahrt werden.

Eine computergestütze Verwaltung ist besonders bei einem hohen Aufkommen von Lastschriften und der damit verbundenen Mandate notwendig. Hier bietet das System, welches die Mandate verwaltet, eine höhere Effizienz und eine schnellere Abwicklung von Lastschriften an. Mandate können durch ein solches System elektronisch archiviert werden, was einen geringeren Aufwand als die händische Verwaltung der Dokumente darstellt. Eine bessere und detailiertere Übersicht über die Mandate und deren Nutzung ist ebenfalls damit verbunden.

Für diese Verwaltungstätigkeit wird das Mandatsmanagement benötigt. Dieses Verfahren ist noch neu und befindet sich in der Entwicklung. Dementsprechend gibt es bisher auch kein fertiges System auf dem Markt. Ein Vergleich zu vorhandenen Alternativen kann demnach nicht gemacht werden. Die Anforderungen an ein solches System ergeben sich ausschließlich aus den vom Standard vorgeschriebenen Prozessen und einigen Kundenanforderungen (siehe Abschnitt 4.3).

4.2 Analyse der Nutzer

Insbesondere Unternehmen mit einer großen Kundenzahl, die auf Bezahlung mittels Lastschrift zurückgreifen, und Unternehmen die zusätzlich viele Kunden im europäischen Ausland haben, können von einem computergestützten System profitieren. Unternehmenszweige, auf welche die aufgeführten Bedingungen zutreffen, sind folgende:

- Telekommunikationsunternehmen
- Versicherungsunternehmen
- Immobilienwirtschaft (Mietenmanagement, Betriebskostenmanagement)
- Versorgungsunternehmen, Abfallwirtschaft, Energie

- Leasingfirmen
- Versandhandel

Entsprechend der jeweiligen Branche kann die Anzahl der Mandate stark variieren. So hat ein großer Energieversorger wesentlich mehr Kunden, und somit auch Lastschriften, als ein mittelständisches Unternehmen. Dieses ist beim Entwurf mit zu berücksichtigen.

4.3 Anforderungsspezifikation

Die Anforderungen an ein Mandatsmanagement lassen sich anhand zweier Kategorien unterscheiden. Einerseits werden Prozesse durch den Standard festgelegt und andererseits durch den Kunden selbst genauer spezifiziert.

4.3.1 Anforderungen seitens des Standards

Die folgende Übersicht gibt die Anforderungen seitens des Standards wieder:

- Für die rechtmäßige Archivierung der Mandate ist der Creditor verpflichtend festgelegt.
- Der Creditor muss das Mandat dematerialisieren und in der entsprechenden Lastschrift mitgeben.
- Jede SEPA-Lastschrift muss ein gültiges Mandat beinhalten.
- Die Debtor-Bank ist im B2B-Verfahren zur Überprüfung des Mandats verpflichtet.

Hierbei wird deutlich, dass insbesondere der Creditor für die Verwaltung der Mandate zuständig ist. Ebenso ist zu klären, wie Mandate erfasst werden sollen, um diese anschließend elektronisch weiterverarbeiten zu können. Da jede SEPA-Lastschrift ein gültiges Mandat beinhalten muss, ergibt sich die Funktionalität des automatischen Anreicherns von Lastschriften mit Mandaten (siehe Kapitel 5). Einer Lastschrift ohne Mandat werden automatisch an der richigen Stelle die benötigten Informationen eingefügt. Für diesen Vorgang sind in der Lastschrift bereits Platzhalter für das Mandat vorgesehen.

4.3.2 Anforderung der Banken

Insbesondere im B2B-Verfahren muss die Bank Mandate verwalten können, um die erforderliche Validierung durchführen zu können. Weiterhin gilt es zu beachten, dass die Debtor-Bank auch die Funktion der Creditor-Bank ausübt, da vermutlich nicht nur Lastschriften entgegengenommen, sondern auch verschickt werden sollen.

Als besonderen Service könnten die Banken das komplette Mandatsmanagement für ihre Kunden übernehmen. Dabei wird dem Kunden die Administration seiner Mandate, beispielsweise über

eine gesicherte Webverbindung zur GUI, bereitgestellt. Die Bank selbst könnte weitere Funktionen übernehmen, wie die Prüfung des Mandats auf Gültigkeit. Auf diese Weise könnte die Debtor-Bank ihren Kunden bei ungültigen Mandaten informieren, obwohl der Creditor für die revisionssichere Archivierung der Mandate verpflichtet ist. Inwieweit diese Variante rechtlich zulässig ist, ist jedoch zum Zeitpunkt der Erstellung dieser Arbeit noch völlig offen.

Anders stellt sich dies im B2B-Prozess dar. In diesem Fall ist die Debtor-Bank verpflichtet die Mandatsinformation in der Lastschrift mit den verifizierten Mandatsinformationen vom Debtor gegenzuprüfen. Dafür sind zwei Varianten möglich. Zum Einen könnte die Bank die Mandatsdaten aus der Lastschrift zum Debtor schicken und sich diese von ihm bestätigen lassen. Die andere Variante ist, dass der Debtor bei der Ausstellung eines Mandats eine Kopie an seine Debtor-Bank schickt. Die Kopie wird daraufhin von der Debtor-Bank dematerialisiert und gespeichert. Wenn beide Mandate nicht übereinstimmen, dann muss die Bank einschreiten und den Fehler entsprechend verarbeiten, beispielsweise durch die Ablehnung der Transaktion.

Die Zahlungen selbst werden bereits von den Banken kontrolliert. Deshalb ist es an dieser Stelle zweckmäßig, die Mandatsdaten auf bestimmte Kriterien mit zu überprüfen. Diese Prüfungen sind jedoch noch nicht genau festgelegt und müssen Schritt für Schritt mit dem Kunden erarbeitet werden. Wieviele Mandate das Mandatsmanagement zu verwalten hat, hängt dabei hauptsächlich vom Lastschriftvolumen des jeweiligen Kunden ab.

4.3.3 Anforderung der Corporates

Viele große Firmen nutzen heutzutage eigene Zahlungssysteme. Diese sind bis jetzt jedoch noch nicht für die zukünftige Lastschriftabwicklung via SEPA vorbereitet. Die Corporates (große Unternehmen mit vielen Zahlungsaus- und eingängen) sind für den Fall des Zahlungseingangs entsprechend dem Standard [Eur08b] verpflichtend festgelegt, dass Mandat zu prüfen und sicher zu archivieren.

Dabei können die Direct Debits sowohl in proprietären Formaten als auch schon SEPA-konform von einem bestehenden Zahlungssystem angeliefert werden. Handelt es sich um proprietäre Formate, sind diese zuerst noch in SEPA-konforme Zahlungsnachrichten umzuwandeln. Anschließend kann das Mandatsmanagement auf die Mandatsfelder in der Nachricht zugreifen, diese nutzen und weiterverarbeiten. Weiterhin sollen externe Systeme per Schnittstelle mit dem Mandatsmanagement kommunizieren können. So soll es den Corporates möglich sein, SEPA-konforme Zahlungsnachrichten ohne Mandat direkt an das Mandatsmanagement zu schicken und mit den notwendigen Mandatsinformationen füllen zu lassen.

Wie im vorigen Kapitel unter 3.2.4 beschrieben, werden Zahlungen, welche nicht ausgeführt werden können, in Form einer R-Message retourniert. Somit muss das System ebenfalls R-Messages erstellen können. Zum Administrieren des Mandatsmanagements soll eine grafische Oberfläche verwendet werden. Des Weiteren benötigt das Mandatsmanagement eine Import- und Exportfunktionalität, um Mandate schnell und einfach mit anderen Systemen austauschen zu

können. Ebenso sollte es sich gut in eine bestehende Umgebung einbetten lassen. Abschließend müssen die fertigen Zahlungsnachrichten an die jeweiligen Empfänger bzw. an das angebundene Zahlungssystem weitergeleitet werden.

4.3.4 Zusammenfassung der Anforderungen

Anhand der fachlichen Analyse können die folgenden drei Aufgabenbereiche an das Mandatsmanagement identifiziert werden.

- Für die Administration und den Im- und Export von Mandaten wird eine Mandatsverwaltung benötigt.
- Die Lastschriften müssen mit den zugehörigen Mandatsinformationen auf Seiten der Corporates angereichert werden.
- Die Mandate sind auf Gültigkeit zu prüfen und gegebenenfalls sind R-Messages zu erstellen.

4.3.4.1 Verwalten von Mandaten

Der erste Aufgabenbereich bildet die Administration von Mandaten. Darunter ist das Anlegen, Ändern und Deaktivieren (Löschen) von Mandaten zu verstehen. Wichtig ist, dass die Mandatsinformationen revisionssicher in einer Datenbank gespeichert werden. Für die Frage, wie die relevanten Daten in das System gelangen, sind verschiedene Lösungsansätze vorstellbar. Eine Möglichkeit stellt die manuelle Dateneingabe mittels einer Maske im Frontend dar. Dabei sollte sich das Eingabefenster an dem vom EPC festgelegten Formular orientieren. Eine andere Variante ist die Nutzung einer Schnittstelle, welche die relevanten Daten dem Mandatsmanagement zur Verfügung stellt.

Der Prozess für das Anlegen eines Mandats ist im Folgenden kurz dargestellt. Nachdem die Felder eines Mandats erfasst worden sind, müssen die Felder auf syntaktische und formelle Korrektheit geprüft werden. Tritt ein Fehler auf, ist mit einer entsprechenden Fehlermeldung der Vorgang zu unterbrechen. Bei einer fehlerfreien Verarbeitung folgt eine Prüfung auf vorhandene Duplikate im System. Existiert das Mandat nicht in der Datenbank, wird es neu angelegt und mit dem aktuellem Datum versehen. Dieses ist wichtig, um mittels eines Gültigkeitschecks prüfen zu können, ob und wann ein Mandat abläuft und inaktiv gesetzt werden muss. Ist bereits ein Duplikat vorhanden, ist dieser Fehler entsprechend in einem Bericht zu vermerken. Anschließend wird das Mandat in die Datenbank gespeichert.

Ist eine große Anzahl von Mandaten in die Datenbank zu speichern, stellt die manuelle Eingabe eine Unzumutbarkeit in der Bedienungsfreundlichkeit für den Nutzer dar. Deshalb sollen sich die Daten über eine Schnittstelle ins Mandatsmanagement importieren lassen. Denkbar ist dies beispielsweise in Form von einzelnen Dateien. Für große Firmen, mit einem entsprechend großen Aufkommen an Zahlungen, ist die Anbindung an ein Dokumenten-Management-System (DMS)

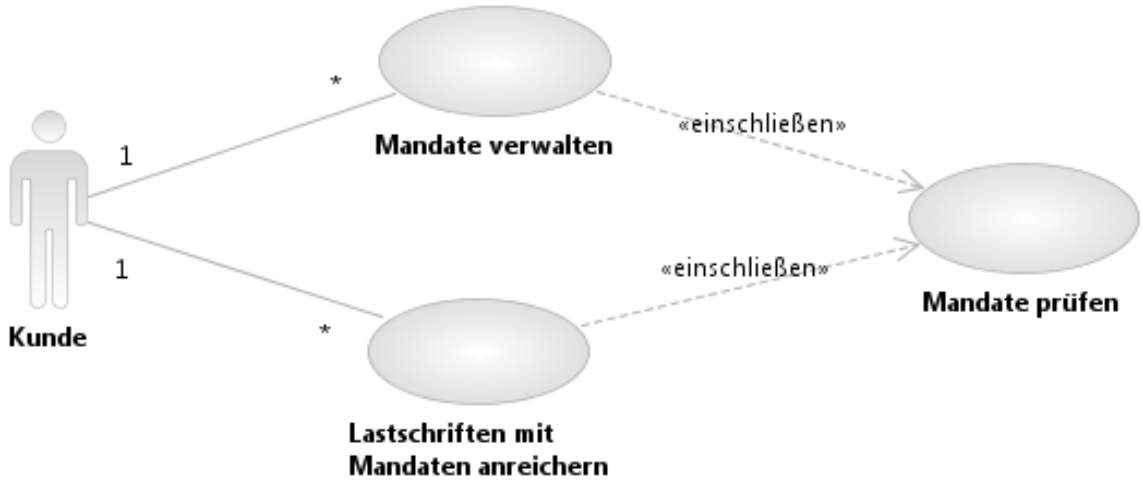


Abbildung 4.1: Anforderungsmodell des Mandatsmanagements

vorteilhaft. Hierbei erfasst ein DMS diese Daten und liefert sie unter anderem in Form von XML-Dateien an. Dabei ist auf eine eindeutige Zuordnung der Papierdokumente zu den elektronischen Zahlungen zu achten. Eine Verknüpfung zum Original lässt sich in Form einer Referenz im Mandatsmanagement mitgeben.

4.3.4.2 Anreichern von Lastschriften mit den zugehörigen Mandaten

In diesem Aufgabenbereich ist die automatische Auffüllung von Zahlungsnachrichten mit den zugehörigen Mandaten definiert. Dieser besonders zeitkritische Aspekt soll deshalb im nächsten Kapitel ausführlich analysiert und anhand eines Prototypen später umgesetzt werden. Da es sich hierbei um die Kunde-Bank-Schnittstelle handelt, werden hierfür nur die PAIN-Nachrichten betrachtet.

4.3.4.3 Prüfen von Mandaten

Die Prüfungen für die Mandate bilden den dritten Aufgabenbereich. Geprüft werden muss zwingend, ob das Mandat valide und gültig ist. Dabei sind weitere Prüfungen mit dem Kunden genau abzustimmen. Ebenfalls könnten Prüfungen auf bestimmte Limits eine sinnvolle Erweiterung darstellen. So könnte unter anderem das Mandat markiert und eventuell gesperrt werden, wenn eine Lastschrift eine bestimmte Höhe übersteigt. Durch derzeit unscharfe Formulierungen der Prüfungen und unbekannten Formaten der R-Messages, kann in dieser Arbeit auf diesen Aspekt nicht näher eingegangen werden.

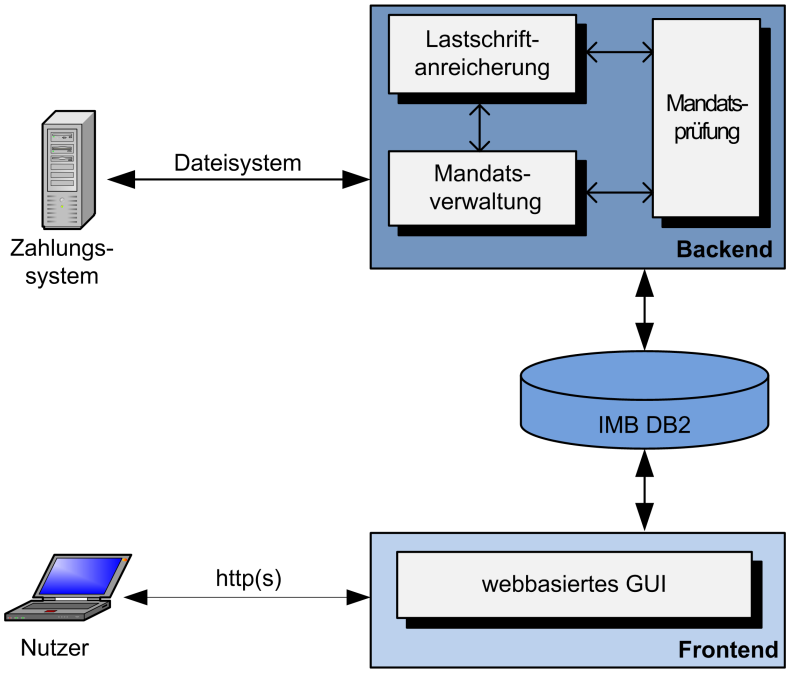


Abbildung 4.2: Architektur des Mandatsmanagement

4.4 Architektur des Gesamtsystems

Nachdem die fachlichen Anforderungen analysiert sind, ist dafür ein Architekturentwurf zu erstellen. Hier ist insbesondere auf eine gute Integrationsfähigkeit und Portabilität zu achten, um es gut in bestehende Systemlandschaften einbetten zu können. Deshalb soll das Mandatsmanagement als Serveranwendung mit einer webbasierten Benutzeroberfläche entwickelt werden. Java ist als Programmiersprache vorgegeben und bietet den Vorteil einer Plattformunabhängigkeit. Somit kann es anschließend auf verschiedenen Systemplattformen, beispielsweise AIX und Windows, eingesetzt werden. Des Weiteren wird eine Datenbank benötigt, um die Zahlungsdateien, und die mit ihr verbundenen Informationen, zu verwalten und zu speichern. Für den Datenzugriff ist Java Database Connectivity (JDBC) festgelegt. Als Laufzeitumgebung wird für alle Java-Anwendungen ein Java runtime environment benötigt.

Eine ressourcenschonende und übersichtliche Abgrenzung der Funktionalitäten erfolgt durch die Aufteilung in Module. Dabei bilden die drei identifizierten Aufgabenbereiche je ein Modul und kapseln die Funktionalitäten voneinander ab. Durch diese Modularisierung ist eine gute Systempflege und eine hohe Portabilität gewährleistet. Nutzerinteraktionen, unter anderem die Administration des Mandatsmanagements, sind ausschließlich im Frontend möglich. Durch die Trennung in Front- und Backend wird neben einer besseren Ressourcenverteilung zusätzlich noch die Sicherheit erhöht. Im Backend werden die einzelnen Module ausgeführt, kommunizieren untereinander und greifen auf eine Datenbank zu. Diese ist in Form einer IBM DB2 in der Version 9 vorgegeben.

4.5 Abgrenzung

Anhand der im vorigen Abschnitt festgestellten Anforderungen, kann man die hohe Komplexität einer computergestützen Mandatsverwaltung erkennen. Durch den Umstand, dass sich das SDD-Verfahren noch in der Entwicklung befindet, ist eine prototypische Umsetzung für die Verwaltungsaufgaben und die Prüfungen schwierig. Weiterhin sind viele Verwaltungsaufgaben im Detail sehr kundenspezifisch, weshalb es für eine finale Lösung unerlässlich ist einen konkreten Kunden einzubeziehen. Aus diesem Grund setzen sich die nächsten Kapitel ausschließlich mit dem Anreicherungsmodul auseinander. Dazu wird die automatische Anreicherung prototypisch implementiert und getestet.

5 Technische Lösung des Moduls Anreicherung

5.1 Funktionale Anforderung

Die Anreicherung von Lastschriften mit Mandatsinformationen stellt sich als besonders zeitkritischer Prozess heraus. Darunter ist das Einfügen eines gültigen, und zur Zahlung zugehörigen Mandats in eine Zahlung zu verstehen. Das Mandatsmanagement soll SEPA-konforme Lastschriften ohne Mandatsinformationen automatisch um diese notwendigen Daten ergänzen. Hierfür muss zunächst die Zahlung analysiert und auf ihre Validität geprüft werden. Im nächsten Schritt ist das Mandat anhand festgelegter Parameter im Mandatsmanagement zu suchen, zu prüfen, und an der jeweiligen Stelle in die Zahlung einzufügen. Zu beachten gilt, dass es sich hierbei um die Kunde-Bank-Schnittstelle handelt und deswegen der Corporate die Zahlungen anreichert. Als Zahlungsformat sind somit ausschließlich PAIN-Nachrichten zu verarbeiten. Können Zahlungen nicht angereichert werden, so sollen diese für eine manuelle Bearbeitung, beispielsweise durch einen Bankmitarbeiter, ausgesteuert werden. Weiterhin setzen sich SEPA-konforme Lastschriften nicht nur aus einer Zahlung zusammen, sondern können sehr viele Transaktionen beinhalten. Aus diesem Grund muss das System Massendaten verarbeiten können.

5.2 Nichtfunktionale Anforderung

Für die Implementierung des Anreicherungsprozesses sind weiterhin wichtige technische Parameter ausschlaggebend. So sind neben der geforderten XML-Verarbeitung und einer Plattformunabhängigkeit vor allem eine sehr schnelle Verarbeitung der Massendaten gefordert. Dies setzt eine sichere und fehlerfreie Verarbeitung voraus. Aus diesem Grund muss sichergestellt sein, dass keine Zahlung doppelt verarbeitet wird oder gar verloren geht. Weiterhin soll dieser Prozess automatisch im Backend ablaufen und somit ohne eine Interaktion des Nutzers auskommen. Somit benötigt der Anreicherungsprozess auch keine grafische Oberfläche. Um die Datenkonsistenz gewährleisten zu können, ist es notwendig die Zahlungen an mehreren Stellen zu sichern. Deshalb sind diese Daten sowohl vor der Anreicherung, als auch danach zu persistieren. Dieser Schritt ist notwendig, damit im Fehlerfalle nicht noch einmal alle Schritte durchlaufen werden müssen und somit Verarbeitungszeit eingespart werden kann. Des Weiteren bietet diese Vorgehensweise die Möglichkeit, dass Verarbeitungsschritte auf anderen Systemen ausgeführt werden können und sich somit insgesamt eine bessere Lastverteilung ergibt. Diese Skalierung wird jedoch aus Komplexitätsgründen bei dem zu entwerfenden Prototyp nicht mit berücksichtigt.

5.3 Das PAIN-Format

Der Ausgangspunkt für die Zahlungsdateien ist das Dateisystem. Ein weiterer wichtiger Punkt ist die eindeutige Identifizierung dieser Zahlungsdateien. Dabei wurde die Annahme getroffen, dass sich jede PAIN-Zahlungsdatei aus einem eindeutigen Namen zusammensetzt. In diesem Abschnitt wird deshalb zunächst der grundlegende Aufbau einer PAIN-Zahlungsnachricht beschrieben.

Die PAIN-Zahlungsnachrichten sind nach den Vorgaben der Extensible Markup Language (XML) aufgebaut und enthalten alle Zahlungsinformationen. Die Schwierigkeiten bereitet das nicht komplett standardisierte Format.

Der grundsätzliche Aufbau ist durch ISO 20022 spezifiziert [Int09] und somit ausschlaggebend. Bis zum jetzigen Zeitpunkt bietet beispielsweise die WestLB AG ein Beispiel für eine PAIN.008-Zahlungsnachricht inklusive dem dazugehörigen XML-Schema an [Wes08]. Eine genauere Auflistung über die genutzten Felder und deren Inhalte sind weiterhin auf der Webseite des Zentralen Kreditausschusses [Zen08] zu finden. Seit Januar 2009 ist unter [Eur09] ein Dokument zur Implementierung vom EPC freigegeben und beschreibt den Aufbau der PAIN.008-Zahlungsnachricht detaillierter. In diesem ist die Reihenfolge und die Häufigkeit der XML-Elemente definiert.

```
<?xml version="1.0" encoding="UTF-8"?>
<Document xmlns="urn:sepade:xsd:pain.008.001.01.grp" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
    instance" xsi:schemaLocation="urn:sepade:xsd:pain.008.001.01.grp pain.008.001.01.grp.xsd">
  <pain.008.001.01>
    <GrpHdr>
      <MsgId>Message-ID</MsgId>
      <CreDtTm>2008-12-21T09:30:47.000Z</CreDtTm>
      <NbOfTxs>3</NbOfTxs>
      <CtrlSum>6966.54</CtrlSum>
      <Grpg>MIXD</Grpg>
    </GrpHdr>
    <PmtInf>
      <PmtMtd>DD</PmtMtd>
      <PmtTpInf>
        <SeqTp>FRST</SeqTp>
      </PmtTpInf>
      <DrctDbtTxInf>
        ...
        <DrctDbtTx>
          ...
        </DrctDbtTx>
        ...
      </DrctDbtTxInf>
    </PmtInf>
    <PmtInf>
      ...
    </PmtInf>
  </pain.008.001.01>
</Document>
```

Listing 5.1: Prinzipieller Aufbau einer PAIN.008-Zahlungsnachricht

Das Listing 5.1 stellt einen kurzen Auszug aus so einer PAIN.008-Zahlungsdatei dar. Da an dieser Stelle die Transaktionen in ihrer Häufigkeit nicht begrenzt werden, sind theoretisch unendlich große Zahlungen möglich. Aus diesem Grund können PAIN-Nachrichten mehrere Gigabyte groß werden. Weiterhin lassen sich diese XML-Dateien sehr gut komprimieren, um sie anschließend, in Relation zur Ausgangsgröße, in einer sehr kleinen Datei versenden zu können. So lässt sich nach [Ste09, Seite 10] eine 5 GB XML-Zahlungsdatei auf eine Größe von 100 MB komprimieren.

Grundsätzlich gliedert sich eine PAIN.008-Nachricht in den GroupHeader (GrpHdr) und in die PaymentInformation (PmtInf). Im GroupHeader stehen zusätzliche validierende Informationen, beispielsweise die Anzahl aller Transaktionen die in dieser Zahlungsdatei enthalten sind. Unter dem Tag *<PmtInf>* sind die Zahlungsinformationen, die *<DrctDbtTxInf>*, zu finden. Diese können laut Standard dabei 1 bis n-mal auftreten.

```
1  ...
2 <PmtInf>
3   ...
4   <DrctDbtTxInf>
5   <PmtId>
6     <EndToEndId>Mandate ID</EndToEndId>
7   </PmtId>
8   <InstdAmt Ccy="EUR">6543.14</InstdAmt>
9   <DrctDbtTx>
10    <MndtRltdInf>
11    <MndtId>Mandate ID</MndtId>
12    <DtOfSgntr>2008-11-20</DtOfSgntr>
13    </MndtRltdInf>
14    <CdtrSchmeId>
15    <Id>
16      <PrvtId>
17      <OthrId>
18        <Id>UCI</Id>
19        <IdTp>SEPA</IdTp>
20      </OthrId>
21      </PrvtId>
22    </Id>
23    </CdtrSchmeId>
24  </DrctDbtTx>
25  </DrctDbtTxInf>
26 </PmtInf>
27 ...
```

Listing 5.2: Auszugsweise Darstellung der Mandatsinformationen in der Zahlungsnachricht

Die Zahlungen selbst sind unter dem Tag *<DrctDbtTx>* zu finden und können nur genau einmal vorkommen. Zu diesen DrctDbtTx-Tags gehört je ein Mandat. Im Listing 5.2 ist dieses Mandat auszugsweise dargestellt. Um die Zahlungsinformationen korrekt verarbeiten zu können, müssen die XML-Dateien valide sein. Diese Validierung übernimmt üblicherweise ein XML-Parser. Ein solch validierender Parser benötigt dafür in jedem Falle ein Dokument (XML-Schema oder eine Document Type Definition (DTD)) gegen welches er validieren kann. Im Gegensatz zu einem nichtvalidierenden Parser, hat ein validierender Parser einen Geschwindigkeitsnachteil zur Folge.

5.4 XML-Verarbeitung

In dieser Arbeit sind laut Kundenaussage mehrere Millionen von Transaktionen zu erwarten. Weiterhin können durch diese große Anzahl die Zahlungsdateien theoretisch mehrere Gigabyte groß werden. Dies stellt die Anforderung an eine sehr schnelle und speichereffiziente Verarbeitung.

Wie im vorigen Abschnitt beschrieben, liegen die Zahlungsdateien im XML-Format vor. Um diese zu verarbeiten, wird ein entsprechender XML-Prozessor benötigt. Für diesen Zweck existieren bereits mehrere Verarbeitungsvarianten, die wichtigsten sollen nachfolgend dargestellt und kurz analysiert werden. In der Literatur wird für XML-Prozessor oft auch der Begriff XML-Parser genutzt, obwohl in der XML-Spezifikation der Begriff XML-Prozessor für ein Programm verwendet wird, welches eine syntaktische und lexikalische Analyse vornimmt [sof09]. Erst dessen Ergebnis verarbeitet anschließend eine XML-Applikation weiter. Ein Parser hingegen ist per Definition ein Programm zur Syntaxanalyse.

Für die XML-Verarbeitung in Java existieren unterschiedliche Konzepte. Die derzeit populärsten Vertreter sind dabei Document Object Model (DOM) und Simple API for XML (SAX). Im folgenden Abschnitt sollen diese verschiedenen Parser-Konzepte und weitere Varianten davon miteinander verglichen und schließlich der zu nutzende Parser für die prototypische Implementierung ausgewählt werden.

5.4.1 Simple API for XML

Die SAX stellt eine weit verbreitete Möglichkeit dar, XML-Dokumente zu verarbeiten. Hierbei ist SAX jedoch nicht als konkreter Parser zu verstehen, sondern besteht aus einer Menge von Empfehlungen, welche die Arbeitsweise eines SAX-Parsers beschreiben. Somit ist der SAX-Standard auch nicht auf eine bestimmte Plattform oder Programmiersprache beschränkt. Für Java stellt die Apache Software Group mit Xerces die geläufigste SAX-Implementierung zur Verfügung.

Die Verarbeitung mit SAX erfolgt nach einem ereignisorientierten Prinzip [Nie06, Seite 119]. Hierbei wird das gesamte Dokument als sequentieller Datenstrom gelesen. Immer wenn der Parser auf ein bestimmtes XML-Konstrukt (Element, Kommentar, etc.) trifft, wird ein zugehöriges Ereignis ausgelöst. Für die im Standard definierten Ereignisse werden daraufhin vorgegebene Rückrufmethoden aufgerufen. Anwendungen, die SAX nutzen, können hierbei eigene Unterprogramme als Rückruffunktion registrieren und auf diese Weise anschließend die XML-Daten auswerten. Die Kommunikation wird dabei über sogenannte Handler abgewickelt, für dessen Registrierung SAX eine Reihe von Schnittstellen zur Verfügung stellt.

- ContentHandler - Identifizierung der XML-Konstrukte im Dokument
- ErrorHandler - Reaktion auf Fehler während des Parsens
- DTD-Handler - Validierung gegen eine DTD

- EntityResolver - Behandlung von externen Entities

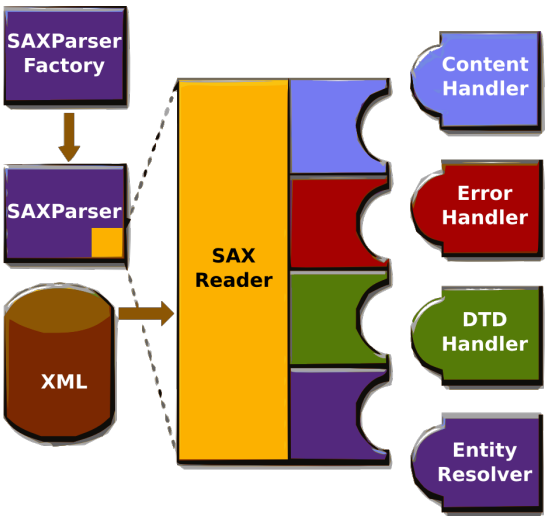


Abbildung 5.1: Schematische Darstellung der SAX2 API [jav09b]

Während des Parsens hält der SAX-Parser nur Informationen zum aktuellen Ereignis im Speicher. Ist die Bearbeitung eines Events abgeschlossen, wird der verwendete Speicher wieder freigegeben. Mit Hilfe dieser Funktionalität können auch sehr große Dokumente verarbeitet werden, da stets nur das aktuelle Ereignis im Speicher gehalten wird. Allerdings muss eine interne Struktur zur Navigation für das XML-Dokument selbst implementiert werden, da es dem Parser nicht möglich ist, Vorgänger oder Nachfolger von Ereignissen zu bestimmen. Durch das Einlesen des Dokuments mittels eines Streams wird eine hohe Verarbeitungsgeschwindigkeit bei gleichzeitig geringem Speicherverbrauch erzielt. Somit ist diese Methode insbesondere für die Verarbeitung von großen Dokumenten geeignet.

5.4.2 Document Object Model

Eine DOM-Programmierschnittstelle liefert einen grundsätzlich anderen Ansatz. DOM stellt ebenfalls ein Application Programming Interface (API) für den Zugriff auf XML-Dokumente bereit und wurde vom World Wide Web Consortium (W3C) definiert [Nie06, Seite 67]. Ursprünglich aus einer Erweiterung aus Dynamic HTML (DHTML) hervorgegangen, stellt DOM Schnittstellen für die Implementierung bereit und ist somit auch nicht an eine konkrete Plattform oder Programmiersprache gebunden. Für Java wird ebenfalls von der Apache Software Group mit Xerces eine Implementierung bereitgestellt.

Die Verarbeitung bei DOM erfolgt nach einem modellorientierten Prinzip. Der DOM-Parser liest ein XML-Dokument vollständig in den Speicher und generiert aus dessen Struktur ein Baummodell. Durch diese Baumstruktur ist ein wahlfreier Zugriff auf beliebige Knoten im Dokument möglich. Auf diese Knoten kann direkt zugegriffen und ihr Inhalt manipuliert werden. Die Vorzüge von DOM stellen auch gleichzeitig seine Nachteile dar. Durch das vollständig im

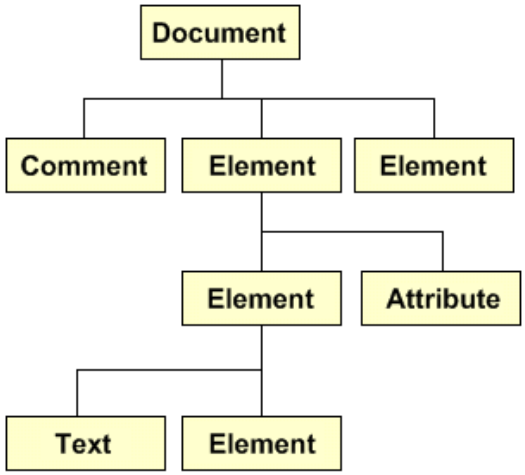


Abbildung 5.2: Baumdarstellung mittels DOM [Zim08]

Speicher gehaltene Dokument ist DOM extrem speicherlastig [Mar02, Seite 120]) und somit für diese Arbeit nicht geeignet. Weiterhin kann DOM nur vollständig vorliegende Dokumente verarbeiten und nicht, wie beispielsweise SAX, an einen Datenstrom gekoppelt werden.

Seit DOM im Jahre 1998 ein Standard des W3C wurde, ist es ständig aktualisiert und dabei in mehrere Versionen (Levels) und Module unterteilt wurden. Diese sind nach [Nie06, Seite 68 ff.] gegliedert in:

- DOM Level 1 mit dem Modul Core (definiert alle Basis-DOM-Objekte und legt die Art und Weise fest, wie in einem Dokument navigiert werden kann)
- DOM Level 2 erweitert das Level 1 um die Module View, Style, Event, Traversal und Range
- DOM Level 3 ergänzt Level 2 um die Module Load und Save, XPath und Validation

5.4.3 Streaming API for XML

Eine weitere Variante zur Verarbeitung von XML-Dokumenten ist die Streaming API for XML (StAX). Diese API wurde von SUN (JSR 173) als eine Art Mittelweg zwischen den beiden Extrempositionen von SAX und DOM entwickelt und arbeitet nach dem Pull-Prinzip [Nie06, Seite 271]. Den programmtechnischen Einstiegspunkt von StaX stellt ein Cursor dar und repräsentiert einen Punkt innerhalb des XML-Dokumentes. Eine Anwendung bewegt diesen Cursor vorwärts und fordert Informationen vom Parser an. Der Cursor zeigt dabei auf den Inhalt im XML-Dokument.

Im Gegensatz zu einem Push-Parser, wie es beispielsweise bei SAX der Fall ist, spielt bei einem Pull-Parser die Applikation die aktive Rolle und behält die Kontrolle über den Verarbeitungsprozess. So liefert der Parser nur nach Aufforderung einer Applikation das Ereignis zurück und die Applikation kann jederzeit darüber entscheiden, ob die Verarbeitung angehalten, fortgesetzt oder abgebrochen werden soll. Deshalb lässt sich sagen, dass der Entwickler bei StAX die Steuerung in der Hand hat und den Cursor programmtechnisch selbst weitersetzen muss, während hingegen bei SAX nur die Rückgabeereignisse ausgewertet werden können. Den Kern für die Implementierung bieten bei StAX zwei APIs, eine Cursor-API für eine streambasierte Verarbeitung und eine Event-Iterator-API für die eventbasierte Verarbeitung.

5.4.3.1 Cursor API

Im Parse-Vorgang wird der Cursor vom Parser nacheinander über das XML-Dokument bewegt. Dies wird durch die Klasse XMLStreamReader erreicht. Anschließend kann mit entsprechenden Zugriffsmethoden des Parsers auf die Information unter dem Cursor zugegriffen werden. Dabei ist das Cursor-API dem SAX-API recht ähnlich. Beispielsweise sind Methoden für den direkten Zugriff auf Zeichen und Zeichenketten verfügbar. Ebenso wie SAX gibt das Cursor-API die gewünschten XML-Informationen als String zurück. Allerdings muss bei StAX die Anwendung selbst über die Ereignisse iterieren. Mit der *next()*-Methode wird der Cursor auf das nächste Element gesetzt und mit der Verarbeitung fortgefahren, solange bis das Ende des Dokumentes erreicht ist. Zum Schreiben eines XML-Dokumentes, dem Serialisieren, bietet das Cursor API die XMLStreamWriter-Klasse an. Für jeden Ereignistyp existiert dabei eine entsprechende *write()*-Methode.

Eine weitere Besonderheit stellt der Anfang des Parsens dar. Bereits nach der Initialisierung durch das XMLReader-Objekt, zeigt der Cursor auf das erste Element. Daraus folgt, dass mit dem ersten Aufruf den Cursor zu versetzen, bereits das zweite Element betrachtet wird.

5.4.3.2 Event Iterator API

Die Event Iterator API bietet ebenfalls zwei Schnittstellen, die XMLEventReader-Klasse zum Parsen und die XMLEventWriter-Klasse zum Serialisieren von XML-Dokumenten an. Weiterhin offeriert es einen höheren Programmierkomfort, was sich jedoch durch zusätzlichen Verwaltungsaufwand (Overhead) negativ auf die Geschwindigkeit auswirkt. Im Gegensatz zum Cursor API, muss nicht mehr mit Abfragemethoden auf einzelne XML-Elemente zugegriffen werden, sondern die Event Iterator API liefert diese Events in Form von Objekten zurück. Entsprechend ihres Inhaltstyps besitzen sie einen dazugehörigen Java-Typ. Den Stream eines XML-Dokumentes stellt die StAX-API als Gruppe von Eventobjekten dar. Diese Events werden durch eine Anwendung angefordert und durch den Parser in der Reihenfolge, wie sie im XML-Dokument auftreten, bereitgestellt.

Für die Serialisierung stellt die Event Iterator API die *add()*-Methode bereit. Diese nimmt, ähnlich den Listen und Mengen der Java-Collection-Klasse, nur noch ein Objekt vom Typ XMLEvent entgegen. Somit erzeugt dieses API für jedes Ereignis ein passendes XMLEvent-Objekt und übergibt es der Anwendung.

API-Variante	Cursor API	Event Iterator API
Zugriff	Über dem Dokument verschiebbarer Cursor Gemeinsame Abfragemethoden für jeden Ereignistyp	Pro Ereignis ein Event-Objekt und Zugriff nach Iterator-Prinzip Pro Ereignistyp ein eigener Javatyp
Geschwindigkeit	Schnell	Langsamer durch höheren Verwaltungsaufwand
Programmierkomfort	Gering Funktion der Abfragemethoden je nach Ereignistyp unterschiedlich Eigene Methode pro Typ bei Serialisierung	Hoch Eigene angepasste Schnittstelle pro Ereignistyp Nur eine Methode bei Serialisierung

Tabelle 5.1: Vor- und Nachteile von Cursor und Event Iterator API [Nie06, Seite 287]

5.4.4 Java API for XML Binding

Mit Java API for XML Binding (JAXB) existiert eine weitere Variante zur Verarbeitung von XML. Dabei handelt es sich primär nicht um einen neuen Parser, sondern genauer um eine Schnittstelle, die eine Interaktion von Java-Anwendungen mit XML ermöglicht.

Mit JAXB können Daten aus einer XML-Schema-Instanz automatisch an Java-Klassen gebunden werden. Zur Erzeugung dieser Klassen wird auf einem XML-Binding-Prozessor zurückgegriffen. Diese Objekt-zu-XML-Transformation erlaubt eine abstrakte Programmierung. Für den lesenden Zugriff auf das XML-Dokument (Parsen) stellt JAXB Hilfsmittel zur Verfügung, welche von einem gegebenen XML-Dokument Java-Objekte zum Zugriff auf diese XML-Daten erzeugen. Dieser Vorgang wird unter JAXB als Unmarshalling bezeichnet und stellt eine spezielle Form der Deserialisierung dar.

Mit den erzeugten Java-Objekten ist ein einfacher Zugriff (per Getter- und Setter-Methoden) auf die XML-Daten möglich. Um gewünschte Informationen wieder in ein XML-Dokument zu überführen, muss die Objektstruktur in eine serielle Darstellungsform umgewandelt werden. Für diese Serialisierung, unter JAXB auch als Marshalling bezeichnet, werden ebenfalls Methoden angeboten.

JAXB stellt somit eine Alternative zu den vorher betrachteten Technologien dar. Während SAX, DOM und auch StAX eine sehr direkte Programmierung erlauben, stellt JAXB einen komfortableren Weg bereit. JAXB beinhaltet intern die Funktionalität von DOM und SAX,

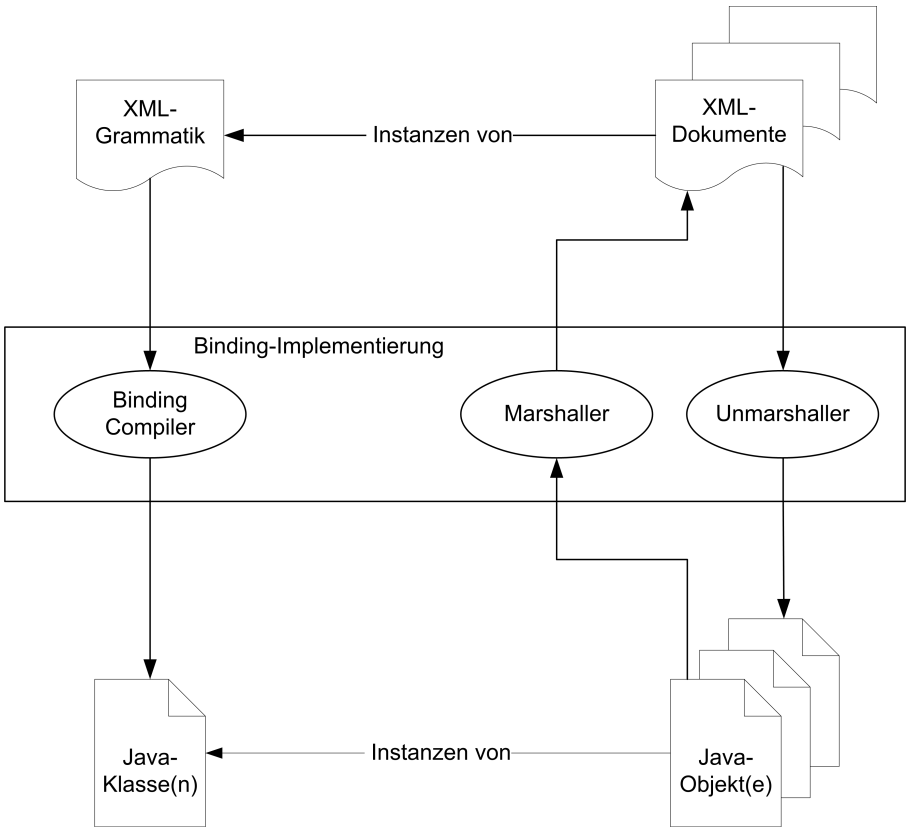


Abbildung 5.3: Prinzip der XML-Bindung in JAXB [Nie06, Seite 372]

allerdings wird diese durch die API verdeckt. So bleiben die Details wie JAXB XML-Dokumente liest und schreibt für den Entwickler weitestgehend verborgen. Anders ausgedrückt können die APIs von SAX, StAX und DOM als Low-Level-APIs [Nie06, Seite 367] bezeichnet werden, während JAXB hingegen eine High-Level API anbietet.

5.4.5 Fazit XML-Verarbeitung

Anhand der bereits genannten Anforderungen, soll der momentan schnellste und dabei ressourcenschonendste Parser gewählt werden. Dabei bilden vorangegangene Untersuchungen von [Ste09] die Basis. Diese Diplomarbeit setzte sich detailiert mit den einzelnen Parser-Implementierungen und der darauf folgenden Persistierung auseinander. Verglichen wurden hierbei die XML-Verarbeitungen mittels SAX, DOM, StAX, VTD und XMLBeans. Als Ergebnis dieser Arbeit konnte der Woodstox-Parser der Firma Codehaus [Cod09] klar empfohlen werden. Dieser ist eine freie StAX-Implementierung und stellt momentan, in Bezug auf Verarbeitungsgeschwindigkeit und Speichergröße, das Optimum dar. Die Abbildung 5.4 verdeutlicht diese Aussage noch einmal.

Mit JAXB wird ebenfalls ein komfortabler Zugriff zur Manipulation von XML-Dokumenten angeboten. Diesen Komfort erkauft man sich jedoch mit einem höheren Speicherverbrauch und einer geringeren Performance. In eigenen Tests (siehe Anhang A.2) konnte herausgefunden

XML-API	Speicherverbrauch [MB]			Zeit [s]		
	Ohne NS+Val	nur NS	mit NS+Val	Ohne NS+Val	nur NS	mit NS+Val
SAX	187,3	187,4	197,5	14,001	14,972	28,154
SAX Mapper	-	194,4	208,6	-	22,044	35,828
StAX(Cursur)	187,0	187,0	-	14,363	14,754	-
StAX(Iterator)	188,1	188,8	-	189,903	793,541	-
Woodstox(Cur.)	186,1	188,4		13,583	13,953	23,235
VTD	-	342,9	-	-	18,962	-
VTD(XPath)	-	-	-	-	-	-
XMLBeans	-	428,7	451,5	-	9,992	21,631

Vergleich der XML-Parser bei 100000 Transaktionen

Abbildung 5.4: Vergleich von XML-Parsern [Ste09, Seite 33]

werden, dass bereits ab 100 000 Transaktionen ein Java-Heapspace-Fehler auftritt. Selbst nach einer Erhöhung des Heapspace auf 1024 MB, kann eine Zahlungsdatei mit einer 1 000 000 Transaktionen nicht mehr ohne Fehler verarbeitet werden. Für sehr große Dokumente ist JAXB deshalb nicht empfehlenswert und wird aus der weiteren Betrachtung ausgeschlossen.

Für die Implementierung des Prototyps wird deshalb der Woodstox-Parser festgelegt. Dieser bietet sowohl die Serialisierung als auch die Deserialisierung an und stellt derzeit bezüglich Geschwindigkeit und Speicherverbrauch die beste Möglichkeit dar.

5.5 Effizienzbestimmung der Persistierung mittels JDBC

Neben dem Parsen gehört auch das Speichern der gewonnenen Informationen zur XML-Verarbeitung. Vorgeschrieben ist in diesem Falle der Einsatz einer DB2-Datenbank. In Java erfolgt der Zugriff über die Datenbankschnittstelle JDBC. Zu den Aufgaben von JDBC zählen die Datenverbindung auf- und abzubauen, zu verwalten und SQL-Anfragen gegen die Datenbank auszuführen. Anschließend können die erzeugten Ergebnisse mit Java weiterverarbeitet werden. Für diesen Zweck ist die Einbindung eines JDBC-Treibers notwendig.

Weiterhin können sogenannte Persistenzframeworks dem Entwickler Arbeit mit JDBC abnehmen. Ein bekanntes Framework ist hier vor allem Hibernate. Dieses ermöglicht es, den Zustand eines Objektes in eine relationale Datenbank zu speichern und daraus Objektstrukturen zu erzeugen, ohne dabei Datenbankzugriffe explizit in Structured Query Language (SQL) programmieren zu müssen. Hibernate stellt dafür Mittel zur Verfügung, mit welchen die Transaktionssicherheit für den Entwickler leichter zu implementieren ist.

Bezugnehmend auf den Untersuchungen von [Ste09] stellt sich der Einsatz von JDBC als schnellste Variante heraus, insbesondere wenn es sich um mehr als 100 000 Transaktionen handelt. Ab einer Größe von 500 000 Transaktionen ist der Einsatz von Hibernate nicht zu empfehlen, weil ab dieser Größe der Speicherverbrauch zu hoch wäre. Diese Aussage wird durch Abbildung 5.5 gestützt.

Persistenzschicht	Speicherverbrauch [MB]	Zeit [s]
Hibernate	91,3	136,071
iBATIS	26,4	384,609
JPA	400,3	364,625
JDBC	16,1	103,509

Vergleich beim Einfügen von 100000 Transaktionen

Persistenzschicht	Speicherverbrauch [MB]	Zeit [s]
Hibernate	-	-
iBATIS	107,3	4971,740
JPA	-	-
JDBC	62,5	1041,246

Vergleich beim Einfügen von 500000 Transaktionen

Abbildung 5.5: Vergleich der Persistenzschichten (aus [Ste09, Seite 55])

Anhand der gewonnenen Ergebnissen wird für diese Arbeit ausschließlich auf JDBC, ohne den Einsatz von Persistenzframeworks, zurückgegriffen. Dies stellt momentan die schnellste und speicherschonendste Variante dar. Allerdings muss hierbei die Transaktionssicherheit selbst implementiert werden, was somit einen höheren Entwicklungsaufwand nach sich zieht.

5.5.1 Ablauf der Effizienzbestimmung

Für die Anreicherung der Mandate sind nicht alle Zahlungsinformationen relevant. Trotzdem ist es wichtig, diese Informationen zu speichern oder zu referenzieren, damit später die Zahlungen eindeutig nachvollzogen werden können. Hierbei ist, entsprechend der nichtfunktionalen Anforderungen, auf eine schnelle und sichere Datenspeicherung zu achten. Einen weiteren Punkt stellt die Größe der Zahlungsdateien dar. Während kleine Dateien mit wenigen Transaktionen keine besonderen Ansprüche an die Verarbeitung stellen, ist dies bei großen Dateien (größer als 1 GB) besonders zu berücksichtigen. So ist es beispielsweise nicht optimal, 100 000 Transaktionen komplett und strukturiert in eine Datenbank zu speichern. Vermutlich wäre mit dieser Variante die Verarbeitungsgeschwindigkeit nicht hoch genug, was nachfolgend in Persistierungstests mittels JDBC überprüft werden soll.

5.5.2 Erstellung von Testdaten

Um die Größe der Zahlungsdatei in Abhängigkeit zur Anzahl der Transaktionen zu ermitteln, werden zunächst eigene Beispielzahlungen generiert. Mit Hilfe des bisher gegebenen Schemas (vgl. [Wes08]) wird ein Minimal- und ein Maximalbeispiel mit je einer Transaktion generiert. Dabei setzt sich das Minimalbeispiel nur aus den notwendigen Plichtfeldern (mandatory elements) zusammen. Um eine abschätzbare Maximalgröße zu ermitteln, werden für das Maximalbeispiel alle optionalen Elemente generiert und mit Daten gefüllt. Anschließend lassen sich die Dateigrößen

ermitteln. Die Integrität der Beispieldateien ist durch eine Validierung gegen das gegebene Schema sichergestellt.

Folgendes Testsystem wurde für die Messungen verwendet:

- Notebook IBM ThinkPad T41
- Intel Pentium M mit 1,7 GHz
- 1,5 GB Arbeitsspeicher
- JDK1.6.0_06 als Java-Laufzeitumgebung
- IBM DB2 Datenbanksystem in der Version 9

Anzahl der Transaktionen	Minimalbeispiel [Byte]	Maximalbeispiel [Byte]
1	1381	3341
10	9103	24 671
100	86 323	237 971
1000	858 523	2 370 971
10 000	8 580 523	23 700 971
100 000	85 800 523	237 000 971
1 000 000	858 000 523	2 370 000 971

Tabelle 5.2: Ermittlung der Dateigrößen in Abhängigkeit zur Anzahl der Transaktionen

Die Tabelle 5.2 stellt die ermittelten Werte dar. Es gilt jedoch zu beachten, dass diese Werte nur Richtwerte sind und nicht als Minimal- oder Maximalwerte festgelegt werden sollten. Des Weiteren kann die benötigte Anzahl der Elemente für eine strukturierte Speicherung davon abgeleitet werden.

5.5.3 Strukturierte Speicherung der Testdaten

Für die Tests werden je eine Tabelle mit 18 Spalten (Minimalbeispiel) und eine Tabelle mit 61 Spalten (Maximalbeispiel) erstellt und unter Nutzung von Prepared Statements mit Daten gefüllt. Die Datentypen der einzelnen Spalten sind dabei so gewählt, dass diese zu den vorgegebenen Werten aus dem Schema passen. Um weitere Abweichungen durch Berechnungen zu minimieren, wird auf spezielle Datentypen wie Timestamp oder Date verzichtet.

Zum Einfügen der Testdaten in die Tabellen wurde ein Java-Programm mit den entsprechenden Methoden erstellt. Die Laufzeit der Testmethoden wird anhand der Differenz zweier Zeitstempel, vor und nach dem Ausführen der Testmethoden, berechnet.

Anzahl der Transaktionen	Minimalbeispiel [Byte]	Maximalbeispiel [Byte]
500	0,7	1,6
1000	1,6	3,2
5000	6,5	12
10 000	13,1	26,7
50 000	70,3	146,0
100 000	148,6	274,1
500 000	736,7	1389,2
1 000 000	1385,2	3327,3

Tabelle 5.3: Zeitmessung für den Testfall Strukturierte Speicherung

In Tabelle 5.3 sind die Messergebnisse der strukturierten Speicherung dargestellt. Hierbei ist ein linearer Verlauf in Abhängigkeit von der Zeit und der Anzahl der Transaktionen erkennbar. Ab einer Größe von einer 1 000 000 Transaktionen stellt sich die strukturierte Speicherung als nicht sinnvoll heraus.

5.5.4 Speicherung der Testdaten in Form von Blobs

Um dennoch große Datenmengen verarbeiten zu können, wird im Folgenden der Weg über die Speicherung in Form von einem Binary Large Object (Blob) untersucht. Blob ist die Bezeichnung für große Datenobjekte (beispielsweise Bitmap-Grafiken, Audio- oder Videodaten), die in relationalen Datenbanken als Binärdaten abgelegt werden. Das Datenbankmanagementsystem (DBMS) speichert den konkreten Wert gesondert ab und vermerkt innerhalb der Tabelle nur eine Referenz auf diesen Wert, den sogenannten Blobkey. Die API von JDBC bietet eine Schnittstelle mit dem Namen Blob an, welches den Zugriff auf diese Datenbankobjekte durch Java ermöglicht.

Den Ausgangspunkt für die festzulegende Blobgröße stellen die ermittelten Dateigrößen aus der Tabelle 5.2 dar. So benötigt eine einzelne Transaktion des Maximalbeispiels 3341 Bytes. Deshalb wird eine Blobgröße von 5 KB pro Transaktion festgelegt und bietet damit auch für eventuelle Abweichungen noch genügend Spielraum. Die abgeleiteten Blobgrößen in Abhängigkeit zur Gesamttransaktionszahl sind im Kopf der Tabelle 5.5 dargestellt. Um 1000 Zahlungen in einem Blob zu speichern sind auch mindestens 1000 Zahlungen notwendig. Daraus ergibt sich der Begriff *nicht realisierbar*, da mit 1000 gegebenen Transaktionen nicht 500 benötigte Transaktionen gemessen werden können.

Transaktionszahl	Benötigte Zeit, um Zahlungen in einem Blob zu speichern [s] (Anzahl der Zahlungen / Blobgröße in MB)			
	(10 / 0,05)	(100 / 0,5)	(1000 / 5)	(10 000 / 50)
500	0,3	0,1	nicht realisierbar	nicht realisierbar
1000	0,4	0,1	0,2	nicht realisierbar
5000	2,2	0,9	0,7	nicht realisierbar
10 000	4,2	1,8	1,7	2,3
50 000	22,3	8,2	9,1	8,7
100 000	48,9	19,5	19,6	15,7
500 000	249,1	83,1	108,7	71,5
1 000 000	473,3	158,9	174,2	151,9

Tabelle 5.4: Zeitmessung für den Testfall Minimalbeispiel und Speicherung als Blobs

Transaktionszahl	Benötigte Zeit, um Zahlungen in einem Blob zu speichern [s] (Anzahl der Zahlungen / Blobgröße in MB)			
	(10 / 0,05)	(100 / 0,5)	(1000 / 5)	(10 000 / 50)
500	0,4	0,5	nicht realisierbar	nicht realisierbar
1000	0,7	0,9	0,8	nicht realisierbar
5000	3,6	4,1	3,6	nicht realisierbar
10 000	6,6	8,5	6,2	7,2
50 000	29,1	42,6	23,6	23,5
100 000	54,9	76,8	51,4	44,3
500 000	273,3	456,2	217,6	229,0
1 000 000	554,9	792,5	464,8	423,9

Tabelle 5.5: Zeitmessung für den Testfall Maximalbeispiel und Speicherung als Blobs

5.5.5 Fazit der Effizienzbestimmung

Die Speicherung der Daten in Form von Blobs zeigt einen deutlichen Geschwindigkeitszuwachs mit steigender Größe, insbesondere im Gegensatz zur strukturierten Speicherung. Die ermittelten Messergebnisse sind in den Tabellen 5.4 und 5.5 dargestellt. Außerdem bietet diese Variante den Vorteil, dass nicht alle Felder vorab bekannt sein müssen, um in der Tabelle angelegt zu werden.

Nachteilig ist jedoch der Zugriff auf die einzelnen Elemente. Zum Einen müssen die Zahlungsinformationen zunächst in binäre Daten umgewandelt werden, um sie überhaupt als Blob speichern zu können. Im Gegenzug müssen die Daten für ihre anschließende Verarbeitung wieder in ihre ursprüngliche Form gebracht werden. Hierfür stellt Java verschiedene Methoden bereit, wie beispielsweise die Umwandlung von Zeichenketten in ein ByteArray mittels der *getBytes()*-Methode. Aufgrund der recht großen Zeitersparnis bei einer hohen Anzahl an Transaktionen, wird die Speicherung in Form von Blobs für die weitere Implementierung festgelegt.

5.6 Ablauf des Anreicherungsprozesses

Der schematische Ablauf ist nachfolgend in Abbildung 5.6 dargestellt. Der Anreicherungsprozess setzt sich aus drei Teilschritten zusammen, die je in ein eigenes Modul gegliedert sind. Als Ausgangspunkt dienen valide Zahlungsdateien im PAIN.008-Format, welche vom Dateisystem eingelesen werden. Anschließend sind die Zahlungen auf fehlende Mandatsinformationen zu prüfen. Daraufhin werden die Mandatsinformationen anhand bestimmter Parameter aus der Datenbank ausgelesen und an der entsprechenden Stelle in der Zahlungsdatei eingefügt.

Können Zahlungen nicht angereichert werden, sollen diese entsprechend ausgesteuert werden. Im letzten Schritt erfolgt die Aufbereitung der Zahlungsdateien für die weitere Verarbeitung. Ist der Prozess abgeschlossen, liegen sowohl valide und mit dem entsprechenden Mandat angereicherte Zahlungsdateien für die weitere Verarbeitung vor, als auch die ausgesteuerten Zahlungen.

Ausgehend von diesen Untersuchungen, sollen im nächsten Kapitel die entsprechenden Module für den Anreicherungsprozess prototypisch umgesetzt werden.

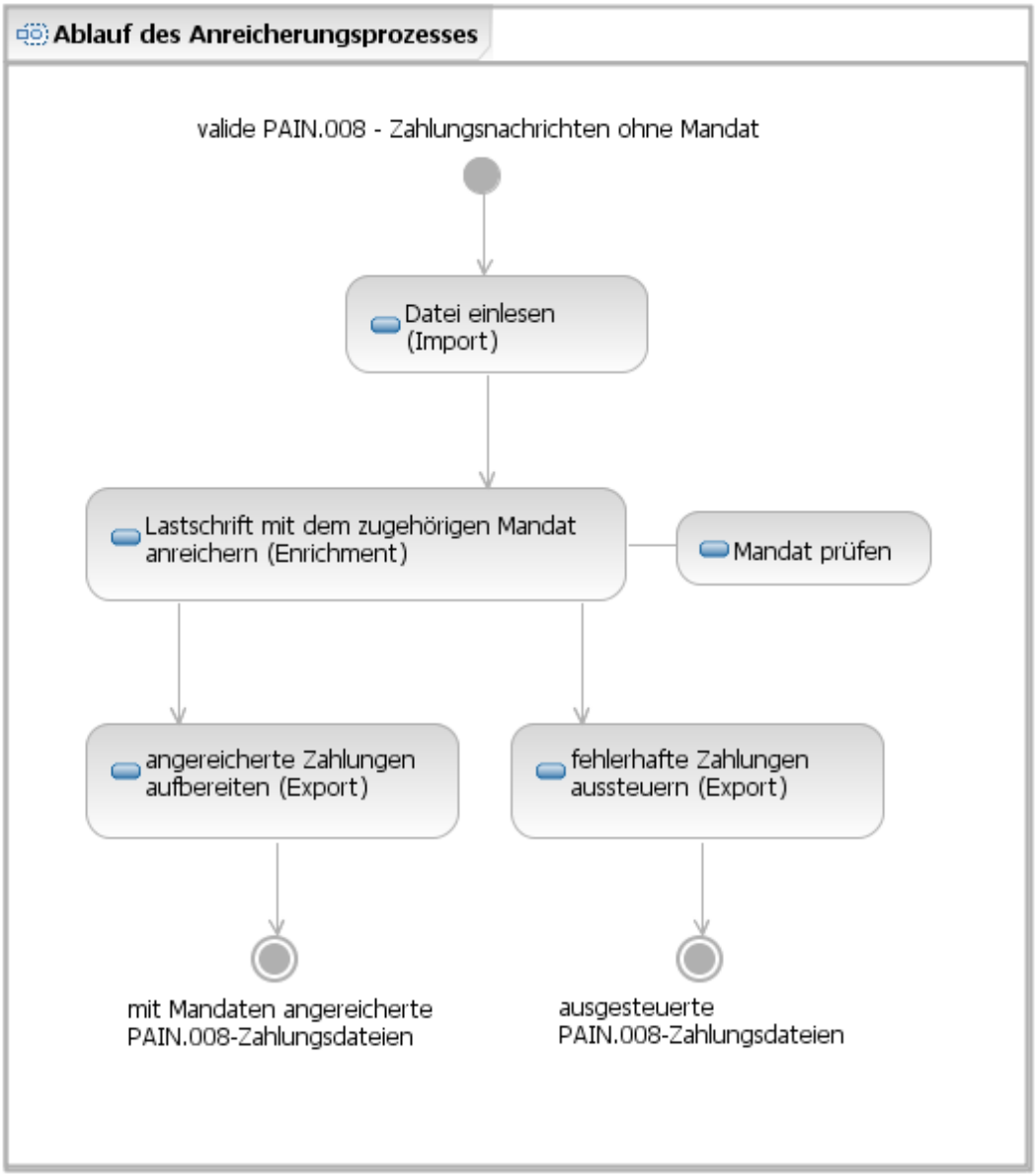


Abbildung 5.6: Ablauf des Anreicherungsprozesses

6 Erstellung des Prototypen

6.1 Das Modul Import

Den ersten Schritt im Ablaufdiagramm stellt der Importprozess dar, mit welchem die Zahlungen eingelesen werden. Durch den Umstand, dass Zahlungsdateien sehr groß sein können, muss eine Möglichkeit zur Verkleinerung dieser Dateien gefunden werden. Eine erste Variante wäre die Komprimierung dieser Dateien. Dies ist jedoch nicht sinnvoll, da durch Komprimierung und Dekomprimierung viel Zeit zur Verarbeitung beansprucht wird. Da es sich beim Anreichern um einen zeitkritischen Prozess handelt, entfällt diese Variante.

Die nächste Variante ist die Aufsplittung einer großen Datei in mehrere kleine Dateien. Hierbei könnten sehr viele kleine Dateien entstehen und unter Umständen das Dateisystem fragmentieren, was sich negativ auf die Performance des Gesamtsystems auswirken würde. Außerdem ist ohne eine Indizierung der Dateien eine Suche sehr mühselig und nicht effizient.

Eine wesentlich bessere Variante bietet die Nutzung einer relationalen Datenbank. Diese bietet den Vorteil, relevante Daten strukturiert zu speichern, zu indizieren und miteinander zu verknüpfen. Anschließend lassen sich diese Daten nach bestimmten Kriterien durchsuchen und entsprechend ändern.

6.1.1 Prototypischer Ablauf des Imports

Der Ablauf für den Import ist in Abbildung 6.1 als Anwendungsfall dargestellt. Zunächst bekommt das Modul die zu verarbeitenden Daten übergeben. Der Prototyp liest hierbei die Zahlungsdateien direkt vom Dateisystem ein. Weiterhin ist der GroupHeader strukturiert in eine eigene Tabelle zu speichern, da er für den Export neu generiert und entsprechend den angereicherten Zahlungen angepasst werden muss. Um nicht alle Transaktionen in einem Blob zu speichern, muss eine Möglichkeit gefunden werden, diese in Blöcke einer fest definierten Größe aufzuteilen. Dieses Splitting ist im Abschnitt 6.1.3.3 genauer erläutert. Eine Referenz soll sicherstellen, dass die Transaktionen eindeutig und in der richtigen Reihenfolge wiederhergestellt werden können. Ebenfalls ist eine entsprechende Fehlerbehandlung zu implementieren.

Beschreibung Anwendungsfall	
Name	Import von Zahlungsnachrichten
Kurzbeschreibung	Zahlungsnachrichten sind vom Dateisystem einzulesen und als Blobs in die Datenbank zu speichern.
Akteure	System
Auslöser	Das Programm bekommt neue, zu verarbeitende Daten übergeben.
Ergebnis(se)	Die verarbeiteten Dateien liegen in der Datenbank. - GroupHeader ist strukturiert gespeichert. - Transaktionen sind gesplittet und als Blobs gespeichert.
Eingehende Daten	PAIN.008 Dateien vom Dateisystem.
Vorbedingungen	Valide PAIN.008 Dateien liegen vor und die Datenbank muss verfügbar sein.
Nachbedingungen	Das Statusflag muss gesetzt werden.
Essentielle Schritte	- Ankommende PAIN.008 werden eingelesen. - PAIN.008 müssen geparst werden. - GroupHeader ist strukturiert in die Datenbank zu speichern. - PmtInfs werden als Blobs mit einer Referenz zum GroupHeader gespeichert. - PmtInfs sind anhand einer vorgegebenen Anzahl von Transaktionen zu splitten und als Blob in die Datenbank zu speichern. - Referenz für ordnungsgemäßes Zusammensetzen der Blobs muss erstellt und gespeichert werden. - Die Fehler sind abzufangen und entsprechend zu behandeln.

Abbildung 6.1: Anwendungsfall Import - Prototypischer Ablauf

6.1.2 Datenmodell des Imports

Anhand der im vorigen Abschnitt gewonnenen Erkenntnisse galt es ein Datenbankschema zu erstellen, welches eine sinnvolle Speicherung der PAIN.008-Zahlungsnachrichten ermöglicht. Die Identifizierung der Nachrichten erfolgt für den Prototypen anhand des Dateinamens der Zahlungsdatei. Somit lassen sind alle Zahlungsdateien eindeutig zuordnen.

Den Ausgangspunkt für die Persistierung stellt die Tabelle P_TRANSACTION dar. In dieser Tabelle wird für jede zu verarbeitende Datei ein eindeutiger Eintrag anhand des Dateinamens erstellt. Ist diese Datei noch nicht in der Tabelle vorhanden, so wird ein neuer Eintrag eingefügt und eine fortlaufende Nummer erzeugt. Diese ist im Folgenden als FID bezeichnet und stellt die Referenz zur Zahlungsdatei her. Außerdem werden noch Statusfelder für die einzelnen Schritte des Anreicherungsprozesses initialisiert, die insbesondere für die Transaktionssicherheit und den weiteren Ablauf der Prozesse benötigt werden.

Weiterhin ist im Anwendungsfall 6.1 gefordert, den GroupHeader strukturiert abzuspeichern. Für diese Aufteilung ist die Tabelle P_GRPHDR vorgesehen. Die Datentypen der einzelnen Spalten sind so gewählt, dass diese den zugehörigen Werten in der Zahlungsnachricht entsprechen. Die Verknüpfung der Tabelle P_GRPHDR erfolgt über eine Fremdschlüsselbeziehung zur Tabelle P_TRANSACTION.

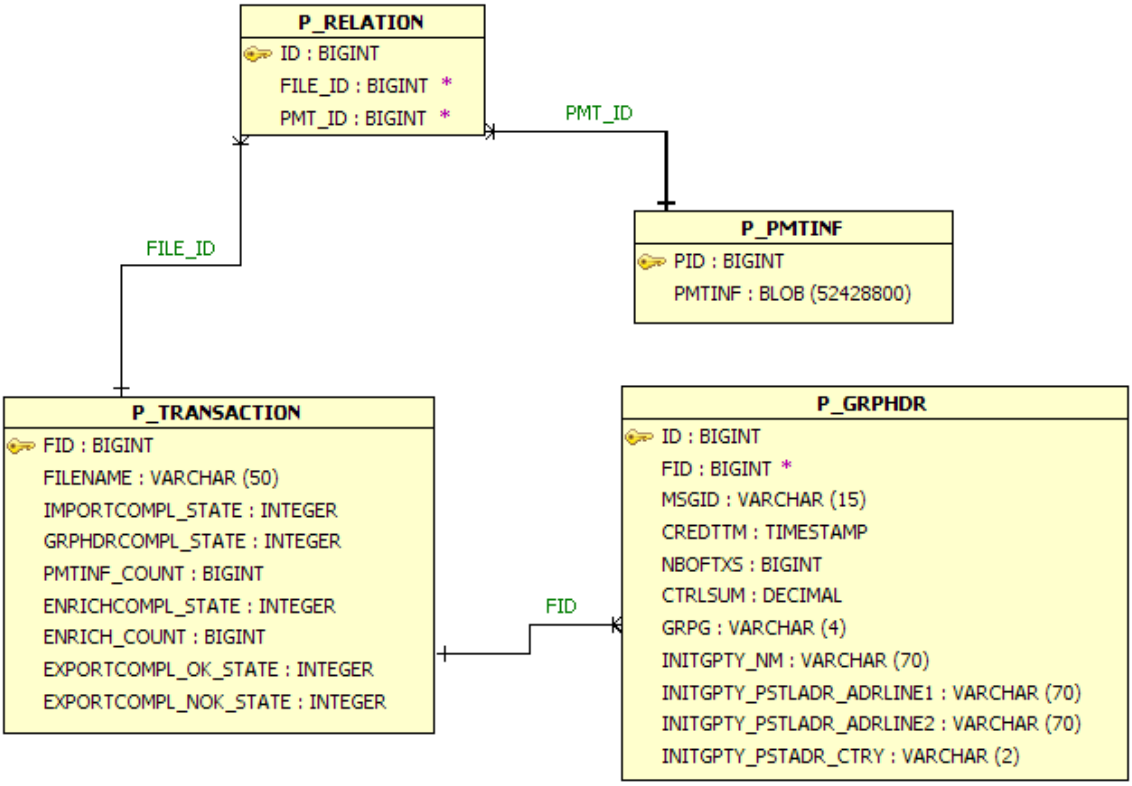


Abbildung 6.2: ER-Diagramm für das Importmodul

Des Weiteren können theoretisch beliebig viele Zahlungen in einer Zahlungsdatei vorkommen. Im Abschnitt 5.5 wurde herausgefunden, dass diese am Besten in Form von Blobs zu speichern sind. Für diesen Zweck ist die Tabelle P_PMTINF vorgesehen. Ihr dient als Primärschlüssel die Spalte ID, die einer fortlaufend nummerierten Zahl mit dem DB2-Datentyp BigInt entspricht und in Java durch den Datentyp Long repräsentiert ist.

Die Blobs aus dieser Tabelle sind mittels Fremdschlüsselbeziehung über eine Relationstabelle P_RELATION mit der Tabelle P_TRANSACTION verknüpft. Die Beziehungen sind in Abbildung 6.2 dargestellt.

6.1.3 Implementierung des Imports

Das Parsen der Zahlungsdateien soll, wie im Abschnitt 5.4.5, mit dem Woodstox-Parser erfolgen und ist nachfolgend beschrieben.

Im ersten Schritt des Importprozesses übergibt die *main()*-Methode die zu parsenden Zahlungsdateien an eine Parse-Klasse. In dieser Klasse sind die Methoden für den Parse-Vorgang implementiert. Um den Woodstox-Parser nutzen zu können, muss zunächst eine Factory-Klasse aufgerufen und davon je ein Objekt zum Lesen und Schreiben erzeugt werden. Dieses ist notwendig, weil Reader und Writer nur aus der Factory entstehen können. Anschließend stellen die

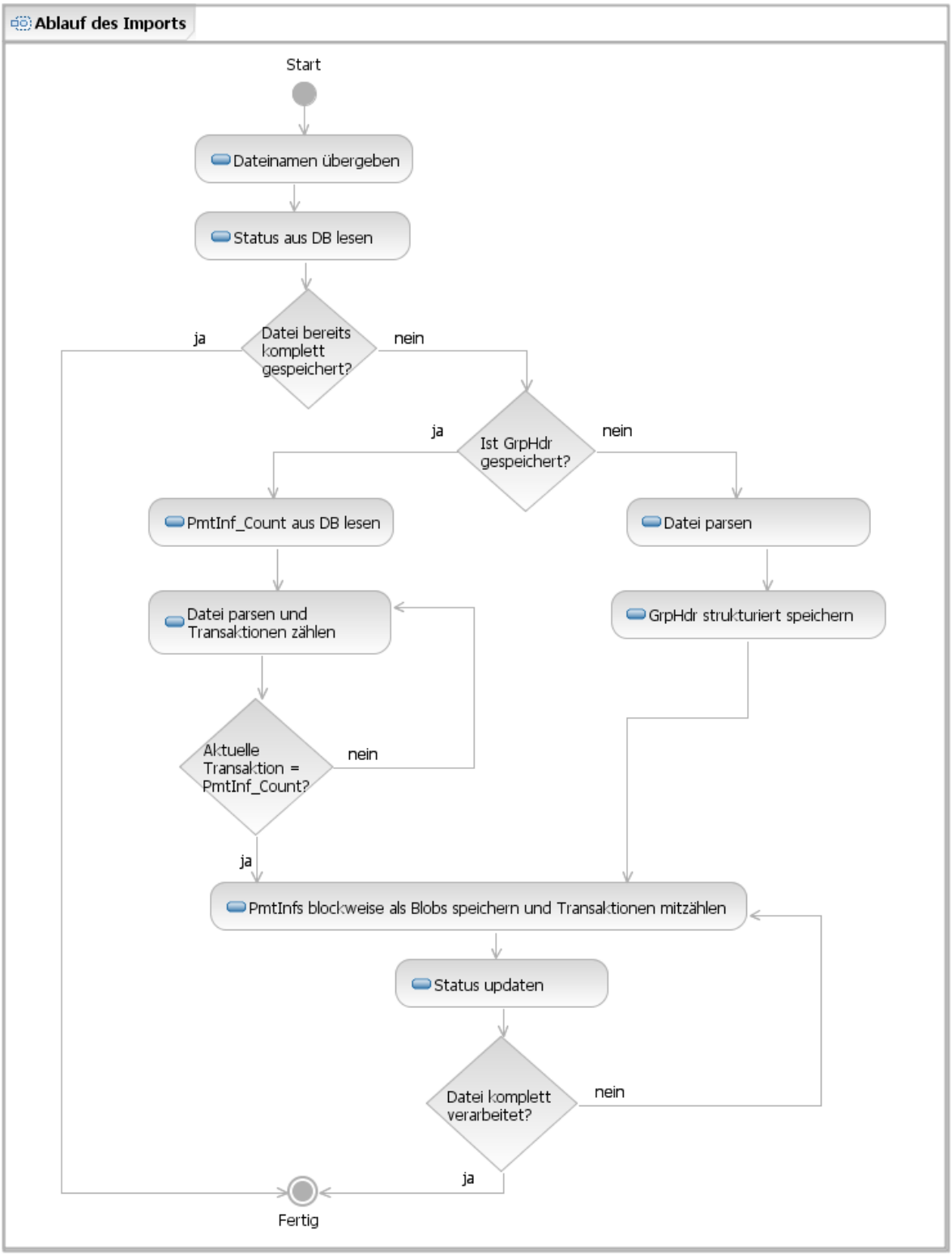


Abbildung 6.3: Schematischer Ablauf des Imports

XMLStreamReader2-Klasse, respektive die XMLStreamWriter2-Klasse die Methoden für die XML-Verarbeitung bereit.

Der Reader bekommt seine Daten in Form eines InputStreams geliefert. Eine Besonderheit ergibt sich bei der Konfiguration des Readers. Dieser muss so konfiguriert werden, dass er mehr als ein Root-Tag in der Ausgabe erlaubt und somit mehrere *PmtInf*-Tags geschrieben werden können (siehe Kapitel 5.3).

```
1  public void parse(String filename) throws XMLStreamException, IOException {
2          // create input and output factory
3          XMLInputFactory2 infactory = null;
4          XMLOutputFactory2 outfactory = null;
5          infactory = (XMLInputFactory2)XMLInputFactory2.newInstance();
6          outfactory = (XMLOutputFactory2)XMLOutputFactory2.newInstance();
7
8          // disable only one root tag
9          outfactory.setProperty(WstxOutputProperties.P_OUTPUT_VALIDATE_STRUCTURE, Boolean.
               FALSE);
10
11         // create input and output stream
12         InputStream inputStream;
13         inputStream = new FileInputStream(filename);
14         ByteArrayOutputStream baos = new ByteArrayOutputStream();
15
16         // create reader and writer
17         XMLStreamReader2 reader = (XMLStreamReader2)infactory.createXMLStreamReader(inputStream);
18         XMLStreamWriter2 writer = (XMLStreamWriter2)outfactory.createXMLStreamWriter(baos);
19          ...
20 }
```

Listing 6.1: Erstellung des Readers und Writers

6.1.3.1 Verarbeitung des GroupHeaders

Im nächsten Schritt werden die Statusfelder der Tabelle P_TRANSACTION überprüft, welche man für die weiter unten beschriebene Transaktionssicherheit benötigt. Zunächst wird von einem neuen, noch nicht verarbeiteten PAIN-File ausgegangen und die Statusfelder initialisiert. Anschließend folgt die strukturierte Speicherung des GroupHeaders. Der Prototyp persistiert dabei nur die notwendigen Elemente, die sich anhand der vorgegebenen Pflichtfelder aus dem XML-Schema ergeben. Um an die Inhalte der einzelnen Tags zu kommen, ist nachfolgend der Parse-Vorgang näher erläutert.

Der Woodstox-Parser funktioniert eventgesteuert und reagiert dabei auf folgende Events.

- *Start_Document*
- *Start_Element*
- *Characters*
- *End_Element*

- *End_Document*

Mittels eines Zeigers (Cursor) wird das aktuelle Element im XML-Dokument gekennzeichnet. Dabei wird, je nach Eventtyp, ein entsprechender Verarbeitungsschritt aufgerufen. Mit der Methode *next()* wird der Zeiger auf das nächste Element im XML-Dokument gesetzt. Hierbei gilt zu beachten, dass sich der Zeiger nach der Initialisierung bereits auf dem ersten Element befindet und er deshalb durch den ersten *next()*-Aufruf bereits auf das zweite Element im Dokument zeigt.

Folgende Methoden der XML-StreamReader2-Klasse werden zum Einlesen des XML-Dokuments eingesetzt.

- *hasNext(), next()*
- *getLocalName()*
- *getAttributeCount(), getAttributeLocalName(), getAttributeValue()*
- *getText()*

Für das Serialisieren stellt der Woodstox-Parser die XMLStreamWriter2-Klasse mit den folgenden Methoden zur Verfügung.

- *writeStartDocument()*
- *writeStartElement()*
- *writeAttribute()*
- *writeCharacters()*
- *writeEndElement()*
- *writeEndDocument()*

Im Kapitel 5 im Abschnitt 5.4 wurde herausgefunden, dass ein StaX-Parser nicht seinen Verarbeitungszustand speichert. Aus diesem Grund muss eine Navigation eigenständig implementiert werden. Für diesen Zweck wird in dieser Arbeit auf den Einsatz von Flags zurückgegriffen. Diese Flags sind mit dem Datentyp *boolean* definiert und können somit nur zwei Werte, nämlich *true* oder *false*, annehmen. Durch die Vorgabe, dass alle zu verarbeitenden Zahlungen valide sind, kann an dieser Stelle auf eine erneute Validierung verzichtet werden. Fehler, die durch falsch verschachtelte Tags entstehen, sind damit ausgeschlossen. Der Name des XML-Elementes wird mit der Methode *getLocalName()* ausgelesen und mit dem Knotennamen verglichen.

Findet der Parser mit der *Start_Element*-Methode anschließend ein bestimmtes XML-Element, beispielsweise *<GrpHdr>*, so bekommt das Flag *isGrpHdr* den Wert *true* zugeteilt. Dabei behält es solange diesen Wert, bis das zugehörige End-Tag *</GrpHdr>* gefunden wird. Anschließend wird die Variable *isGrpHdr* auf *false* zurückgesetzt.

Im weiteren Verlauf verarbeitet der Parser das *Start_Element <MsgId>*. An dieser Stelle wird erneut geprüft, ob das Flag *isGrpHdr* true ist. Ist dem so, wird das Flag *isMsgId* auch *true* gesetzt, anderenfalls behält es seinen derzeitigen Wert. Mit diesem Vorgehen wird sichergestellt, dass das Tag *MsgId* unbedingt ein Unterknoten von *GrpHdr* ist. Diese Vorgehensweise kommt für jede Prüfung auf Unterknoten in abgewandelter Form zum Einsatz.

Das *Characters*-Event ist für die Verarbeitung des Inhalts der XML-Elemente zuständig. Hierzu wird ebenfalls geprüft, ob der Elementname des Inhalts mit dem Elementnamen des Startelementes übereinstimmt. Wenn diese Prüfung positiv ist, wird der Inhalt als String an ein Data Transfer Object (DTO) übergeben. Mit dem *EndElement*-Event wird das gleiche Flag wieder auf *false* gesetzt, wenn der ausgelesene Name mit dem gerade verarbeiteten Elementnamen übereinstimmt. Ein DTO bündelt mehrere Daten in ein Objekt, so dass ein einziger Programmablauf alle Daten überträgt. Auf diese Weise lassen sich mehrere zeitintensive Zugriffe durch einen einzigen Zugriff ersetzen.

Für alle anderen Tags ist der Ablauf des Parsens gleich. Der Vorteil bei der Arbeit mit Flags ist eine verhältnismäßig einfache Implementierung. Weiterhin ist diese Variante recht schnell, da der Compiler nur die Flags auf ihre Werte überprüft und dementsprechend die Verarbeitung ausführt. Nachteilig bei dieser Variante sind die vielen Zeilen Programmcode und dessen aufwendige Wartbarkeit. Um das Problem der Wartbarkeit zu minimieren, sind alle Konstanten in eine eigene Klasse namens `ParserConstants` ausgelagert. Diese Klasse definiert alle genutzten Knotennamen und Flags und kann somit für eventuell ander PAIN-Formate angepasst werden.

```
1  case XMLStreamReader2.START_ELEMENT:
2    if (reader.getLocalName().equals(ParserConstants.PMTINF)) {
3      ParserConstants.isPmtInf = true; // Flag true setzen
4    }
5    if (ParserConstants.isPmtInf) {
6      writer.writeStartElement(reader.getLocalName());
7      for ( int _attrIdx = 0; _attrIdx < reader.getAttributeCount(); _attrIdx++) {
8        writer.writeAttribute(reader.getAttributeLocalName(_attrIdx), reader.getAttributeValue(_attrIdx));
9      }
10   }
11
12 case XMLStreamReader2.CHARACTERS:
13   if (ParserConstants.isGrpHdr) { dto.setGrpHdr(reader.getText().trim()); } // Übergabe an DTO
14   if (ParserConstants.isPmtInf) {
15     writer.writeCharacters(reader.getText().trim());
16   }
17
18 case XMLStreamReader2.END_ELEMENT:
19   if (ParserConstants.isPmtInf) {
20     writer.writeEndElement();
21   }
22   if (reader.getLocalName().equals(ParserConstants.PMTINF)) {
23     ParserConstants.isPmtInf = false; // Flag false setzen
24   }
```

Listing 6.2: Auszug zur Speicherung mittels Flags

6.1.3.2 Verarbeitung der PmtInfs

Für die Speicherung der PmtInfs kommt ebenfalls das Flag-Prinzip zum Einsatz. Anders als beim GroupHeader, sind hier die Daten nicht einzeln und strukturiert zu speichern, sondern sollen in Blobs umgewandelt werden. Dadurch hält man die Daten einerseits von der Größe her gering und ermöglicht andererseits die schnelle Persistierung in eine Datenbank. Die Vorgehensweise für die PmtInfs ist analog dem Ansatz für das Einlesen des GroupHeaders. Anstatt die Elemente einzeln zu speichern, wird nur wenn sich das Flag *isPmtInf* im Zustand *true* befindet, mittels der XMLStreamWriter2-Klasse, in einen Augabestrom geschrieben. Dadurch werden alle in diesem Knoten vorkommende Elemente und Unterknoten nacheinander geschrieben. Ein ByteArrayOutputStream-Objekt stellt in diesem Falle den Ausgabestrom dar und ermöglicht anschließend eine Speicherung in Form von Blobs.

6.1.3.3 Splitting

Weiterhin muss eine Möglichkeit gefunden werden, wie die *PmtInf*-Knoten in mehrere Blöcke aufgeteilt werden können. Für diesen Zweck wird eine Zählvariable *PmtInf_Count* angelegt, welche sich jedem End-Tag *</PmtInf>* um den Faktor eins inkrementiert. Mit einem konfigurierbaren Parameter *maxTrans* wird eingestellt, wieviele *PmtInf*-Blöcke maximal in einem Blob gespeichert werden sollen. Anschließend vergleicht man die Anzahl der aktuell verarbeiteten PmtInfs mit dem vorher festgelegten Maximalwert. Ist dieser Vergleich wahr, es sind also genau so viele *PmtInf*-Knoten vorhanden wie maximal angegeben, schreibt die Methode *flush()* den Ausgabestrom fest. Somit ist sichergestellt, dass die Zahlungen nicht getrennt werden und logisch immer zusammenhängen. Im Anschluss daran folgt die Umwandlung des ByteArrayOutputStream in ein ByteArray, welches für die Erzeugung von Blobs zwingend benötigt wird. Eine Methode übergibt danach das ByteArray zur Persistierung in die Datenbank. Zum Schluss wird der ByteArrayOutputStream geleert und der Ablauf beginnt wieder von vorn. Erreicht der Parser das Dokumentende, dann speichert der Prototyp alle bisher noch nicht berücksichtigten Transaktionen in ein eigenes Blob.

6.1.3.4 Transaktionssicherheit

Eine erfolgreiche Umwandlung des ByteArrays in ein Blob erfolgt immer, wenn der Vergleich mit dem festgelegten Maximalwert wahr ist. Jede erfolgreich abgeschlossene Transaktion beinhaltet daraufhin den Status *PmtInf_Count* in der Tabelle P_TRANSACTION mit der aktuellen Anzahl der erfolgreich verarbeiteten PmtInf-Tags.

Bricht die Verarbeitung zwischendrin ab und die gleiche Datei wird erneut verarbeitet, dann liest das Programm den aktuellen *PmtInf_Count* aus der Datenbank und setzt erst ab dieser Stelle mit der Persistierung fort. Somit lässt sich sicherstellen, dass keine Zahlung doppelt verarbeitet wird.

Erst wenn die komplette Datei geparst ist, setzt das Programm einen *ImportComplete_State* auf *1* und gibt die erfolgreiche Importierung dieser Datei in die Datenbank an.

Einen weiteren Faktor für die Transaktionssicherheit bildet die Reihenfolge der Eintragungen in die genutzten Tabellen. Beispielsweise darf eine Eintragung in der Tabelle P_GRPHDR nur dann vorgenommen werden, wenn eine FID in der Tabelle P_TRANSACTION angelegt werden konnte. Um dieses Prinzip sicherzustellen, wird auf eine Rollback-Funktionalität der Datenbank zurückgegriffen. Vor der eigentlichen Datenbankanfrage legt die *setSavepoint()*-Methode ein Wiederherstellungspunkt an. Tritt während dieser Verarbeitung ein Fehler auf, so stellt die Datenbank den Anfangszustand durch ein Rollback wieder her. Nach der fehlerfreien Verarbeitung gibt die Methode *releaseSavepoint()* die reservierten Ressourcen wieder frei.

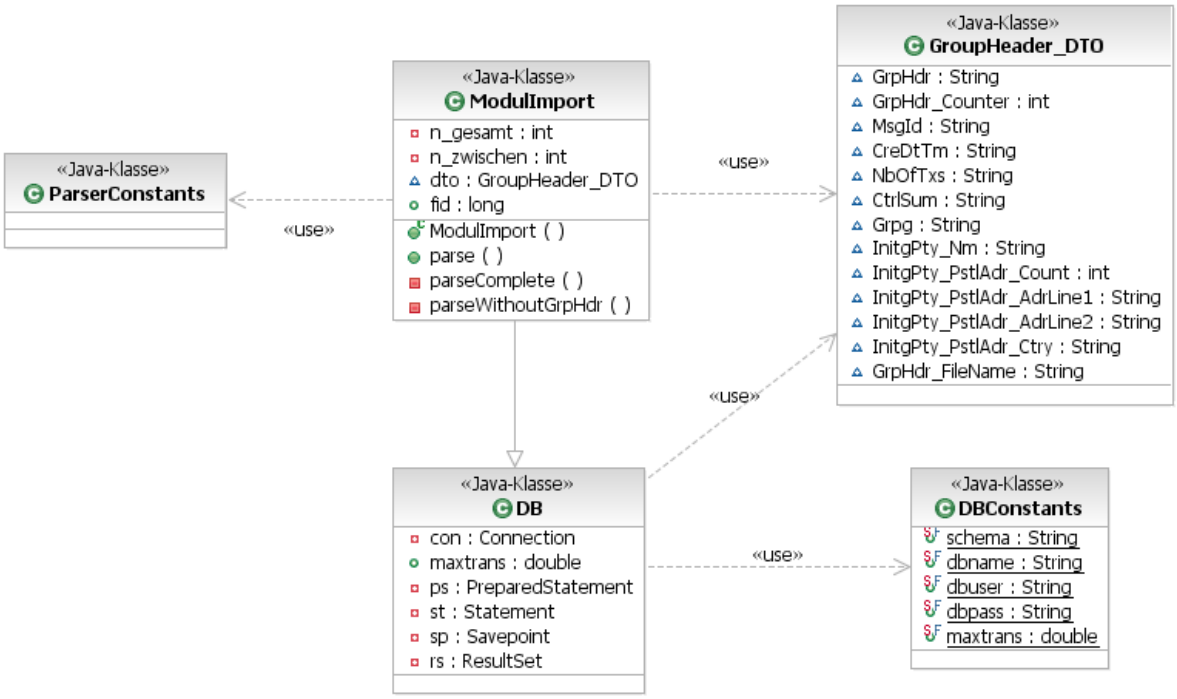


Abbildung 6.4: Klassendiagramm für das Modul Import

6.2 Das Modul Enrichment

Das Modul Enrichment stellt den nächsten Punkt im Ablauf des Anreicherungsprozesses dar. Hierbei werden die im vorigen Schritt in Blobs zerlegten Zahlungen aus der Datenbank gelesen und sind mit dem entsprechenden Mandat anzureichern. Anschließend sind die fertigen Zahlungen in die Datenbank zu persistieren. Die nicht angereicherten Zahlungen sind entsprechend abzusplitten und auszusteuern. Dabei darf jedoch der Zusammenhang zur Ursprungsdatei nicht verloren gehen. Eine Übersicht für das Anreichern stellt die Abbildung 5.6 dar.

Beschreibung Anwendungsfall	
Name	Anreichern der Zahlungen in Form von Blobs
Kurzbeschreibung	Die Blobs sind aus der Datenbank zu lesen und mit entsprechenden Mandaten anzureichern, die angereicherten PAINs werden anschließend wieder in Datenbank gespeichert, nicht angereicherte Zahlungen sind auszusteuern.
Akteure	System
Auslöser	Programm bekommt die zu verarbeitenden Daten übergeben.
Ergebnis(se)	- Angereicherte Zahlungen mit dem zugehörigen Mandat liegen in der Datenbank. - Zahlungen, die nicht angereichert werden konnten (z.B. liegt kein gültiges Mandat vor) werden ausgesteuert und in eigene Tabelle gespeichert.
Eingehende Daten	Blobs und Mandate aus der Datenbank.
Vorbedingungen	Neue Blobs zum Anreichern liegen vor, Datenbank muss vorhanden sein
Nachbedingungen	Das Statusflag muss entsprechend auf Angereichert/Ausgesteuert gesetzt werden und das Last_Used_Date ist in der Mandatsmanagement-Datenbank zu aktualisieren.
Essentielle Schritte	- Blobs werden aus der Datenbank gelesen. - Blobs müssen geparst und eine Referenz zum Zuordnen der Originalzahlung muss beibehalten werden. - Zu jeder Transaktion ist das enstprechende Mandat in der Datenbank zu suchen. - Wird ein Mandat gefunden, ist dieses Mandat an vorgegebener Stelle in die Zahlung einzufügen. - Ist es nicht gültig, dann wird der Status Not Valid für dieses Mandat in der Datenbank gesetzt. - In der Datenbank wird das Flag Angereichert für den Erfolgsfall gesetzt und das Last_Used_Date wird mit der aktuellen Verarbeitungszeit aktualisiert. - Wenn kein Mandat gefunden wurde, dann ist die Zahlung abzusplitten, als Ausgesteuert zu kennzeichnen und in die Datenbank zu speichern. - Die fertig verarbeiteteten Blobs werden in die Datenbank gespeichert und das Flag Angereichert gesetzt. - Die Fehler sind abzufangen und entsprechend zu behandeln.

Abbildung 6.5: Anwendungsfall Anreicherung - Prototypischer Ablauf

6.2.1 Prototypischer Ablauf des Anreicherns

Zuerst werden die Zahlungen in Form von Blobs aus der Datenbank gelesen. Anschließend werden diese erneut geparst, um einen Zugriff auf die einzelnen Elemente zu bekommen und herauszufinden, ob bereits ein Mandat in der Zahlung existiert. Ist dem so, wird die Anreicherung übersprungen. Enthält die Zahlung noch kein Mandat, dann soll das Modul Enrichment für jede einzelne Zahlung die zugehörigen Mandatsinformationen aus der Datenbank heraussuchen und an die entsprechende Stelle in die Zahlung einfügen. Dies ist jedoch nur möglich, wenn das Mandat in der Datenbank vorliegt und gültig ist. Konnte kein Mandat gefunden werden, so ist die komplette Zahlung abzusplitten und auszusteuern. Konkret wird dabei der Knoten *DrctDbtTxInf* aus der Zahlung herausgelöst, wenn es für diese Zahlung kein gültiges Mandat gibt. Schlussendlich müssen angereicherte Zahlungen, wie auch ausgesteuerte Zahlungen, wieder in die Datenbank persistiert werden.

6.2.1.1 Zuordnung der Mandate zu den Transaktionen

Um das Mandat aus der Datenbank herauszusuchen, ist eine eineindeutige Zuordnung der Mandate zu den Zahlungen notwendig. Hierfür sind im Regelwerk keine genauen Angaben gemacht und müssen deshalb erst definiert werden. Das Regelwerk gibt lediglich die eindeutige Identifizierung mittels Mandatsreferenz und UCI vor. Für den Prototyp wurden deswegen, ausgehend von den Implementation Guidelines für PAIN.008-Zahlungsnachrichten (siehe [Eur09]), folgende Zuordnungen festgelegt.

Feldname im Mandat	Tagname in der Zahlungsnachricht
Unique Mandate reference (AT-01)	*<EndToEndId>*
Unique Creditor identifier (AT-02)	*<Id>* (entweder unter *<OrgId>* oder unter *<PrvtId>* vorhanden)

Tabelle 6.1: Zuordnung der Mandatsfelder zur PAIN-Zahlungsnachricht

Eine weitere Schwierigkeit stellt die Angabe der UCI an mehreren Stellen in der Zahlung dar. So ist für Firmenkunden die UCI unter dem Knoten *<OrgId>* zu finden, während hingegen bei Privatpersonen der Knoten *<PrvtId>* genutzt wird. Im Listing 6.3 ist die genutzte UCI unter Angabe ihres Pfades auszugsweise dargestellt.

6.2.2 Datenmodell des Anreicherns

Zusätzlich zu den unter Abschnitt 6.1.2 genannten Datenmodell, benötigt das Modul Enrichment weitere Tabellen. In der Tabelle P_ENRICHED sind alle erfolgreich angereicherten Zahlungen in Form von Blobs, mit einer Referenz zur Ausgangszahlung, gespeichert. Weiterhin werden für

```
<pain.008.001.01>
  <PmtInf>
    <DrctDbtTxInf>
      <PmtId>
        <EndToEndId>Mandate ID</EndToEndId>
      </PmtId>
      <DrctDbtTx>
        <CdtrSchmeId>
          <Id>
            <OrgId>
              <PrtryId>
                <Id>UCI von einer Firma</Id>
              </PrtryId>
            </OrgId>
            ...
            <PrvtId>
              <OthrId>
                <Id>UCI von einer Privatperson</Id>
              </OthrId>
            </PrvtId>
          </Id>
        </CdtrSchmeId>
      </DrxtDbtTx>
    </DrctDbtTxInf>
  </PmtInf>
</pain.008.001.01>
```

Listing 6.3: Auszugsweise Darstellung der benötigten Felder für die Zuordnung

das nächste Modul noch die Summe der einzelnen Zahlungen und die Zahlungsanzahl je Blob benötigt. Für diesen Zweck dienen die Spalten NBOFTXS und CTRLSUM. Alle ausgesteuerten Zahlungen werden in der Tabelle P_NOTENRICHED ebenfalls als Blobs gespeichert.

Eine weitere Voraussetzung ist der Zugriff auf eine gegebene Mandatsdatenbank. Aus Performance- und Sicherheitsgründen soll hierfür auf eine replizierte Datenbank zugegriffen werden. Somit können Kollisionen durch Zugriffe, beispielsweise während der Administration der Mandate durch ein anderes Modul, vermieden werden.

6.2.3 Implementierung des Anreicherns

Für das Anreichern kommt ebenfalls der Woodstox-Parser zum Einsatz. Der schematische Ablauf dieses Moduls ist in Abbildung 6.7 dargestellt.

Im ersten Schritt wird der Dateiname der zu verarbeitenden Zahlung übergeben. Daraufhin sucht die Methode *getFID()* die dazugehörige FID aus der Tabelle P_TRANSACTION heraus. Anhand dieser FID werden die zugehörigen Blobs aus der Tabelle P_PMTINF ausgewählt.

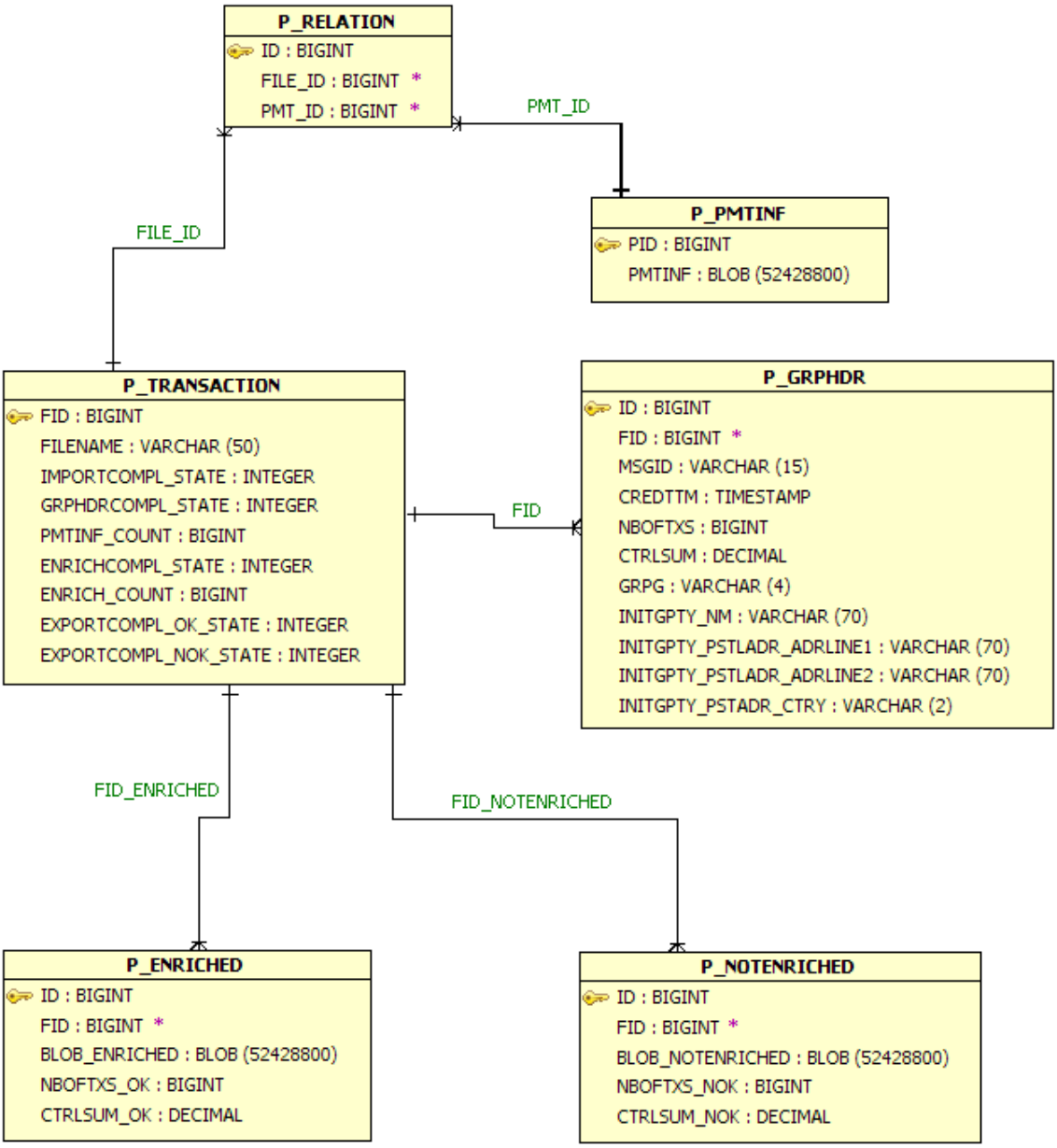


Abbildung 6.6: ER-Diagramm für das Modul Enrichment

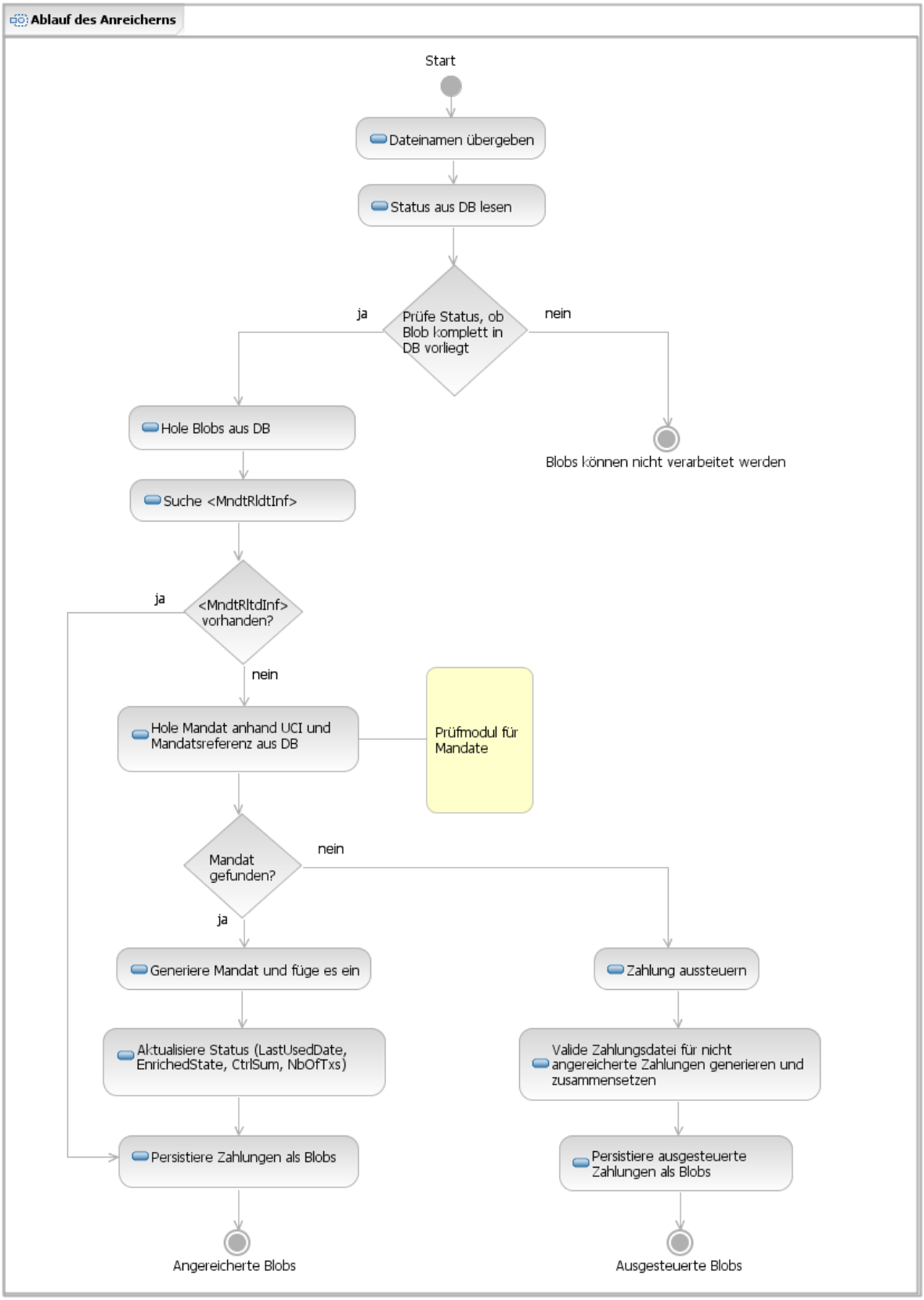


Abbildung 6.7: Schematischer Ablauf des Anreicherns

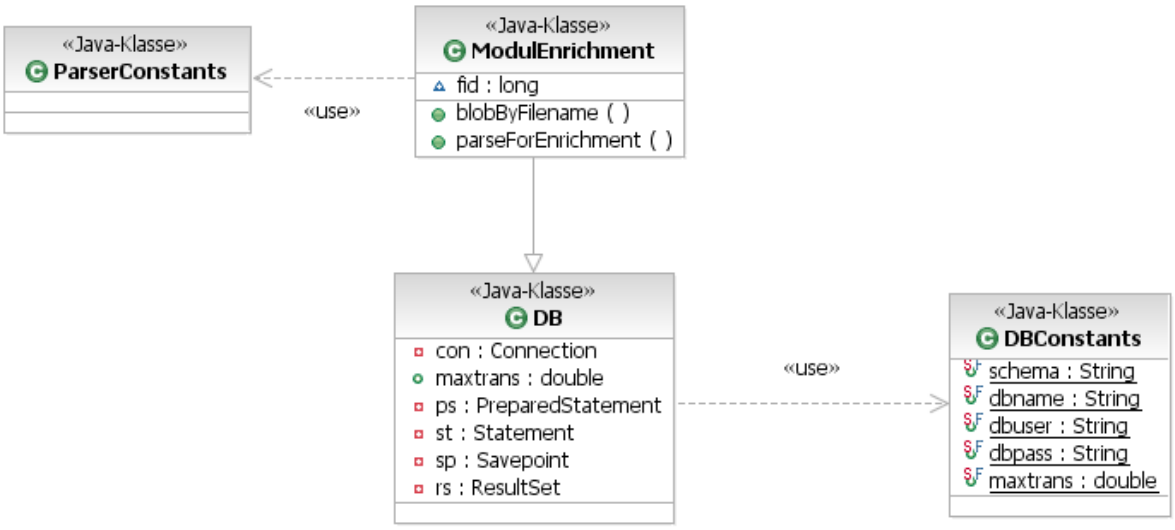


Abbildung 6.8: Klassendiagramm für das Modul Enrichment

6.2.3.1 Mandat ermitteln

Im nächsten Schritt wird das Blob geparst und auf das Vorkommen der Mandatsinformationen überprüft. Ist das Tag *<MndtRltdInf>* vorhanden, ein entsprechendes Mandat ist demnach schon eingefügt, wird diese Zahlung nicht weiter untersucht und übersprungen. Findet der Prototyp dieses Tag nicht, so muss das entsprechende Mandat für diese Zahlung aus der Datenbank geholt und an dieser Stelle eingefügt werden.

Zur Identifizierung des Mandats benötigt man zwei Informationen, die UCI und die Mandatsreferenz. Eine Schwierigkeit stellt das Herausfinden der UCI dar. Die UCI kann an zwei verschiedenen Stellen vorkommen, je nachdem ob das Feld *<PrvtId>* oder *<OrgId>* belegt ist. Weiterhin ist die Information erst unterhalb der Stelle, an der das Mandat eingefügt werden soll, zu finden.

Um eine schnelle Verarbeitung zu ermöglichen, werden alle nicht für die Mandatsgenerierung benötigten Daten gelesen und sofort mittels der XMLStreamWriter2-Klasse in einen ByteArrayOutputStream geschrieben. Da der Woodstox-Parser streamingbasiert ist und deshalb nicht während des Parsens vor- oder zurückspringen kann, wird der Inhalt des Knotens, welcher eine komplette Zahlung enthält, in einem Puffer zwischengespeichert und nicht sofort mit dem XMLStreamWriter2 geschrieben. Im Anschluss daran muss nur dieser Puffer auf das Mandat untersucht werden. Ist es nicht vorhanden, sucht es eine Methode anhand der UCI und der Mandatsreferenz in der Datenbank.

6.2.3.2 Mandat generieren

Wird ein Mandat in der Datenbank für die übergebene UCI und Mandatsreferenz gefunden, so findet anschließend eine Überprüfung der Gültigkeit des Mandats statt.

Hierbei überprüft eine Methode das Datum für die Lastschrifteinreichung *<ReqdColltnDt>* mit dem Datum der letzten Nutzung des Mandats *<Last_Used_Date>*. Nur wenn diese Differenz innerhalb von drei Jahren liegt ist das Mandat gültig. Trifft dieser Fall zu, wird daraufhin das Mandat generiert und an der richtigen Stelle in den Puffer eingefügt, sowie das Datum der letzten Mandatsnutzung, das *Last_Used_Date*, mit dem aktuellen Zeitstempel in der Datenbank aktualisiert.

Ist für die Zahlung kein gültiges Mandat vorhanden, so muss diese ausgesteuert werden. Erst nachdem die komplette Zahlung fertig geparst ist, schreibt der Parser mit der XMLStreamWriter2-Methode *writeRaw()* auf den ByteArrayOutputStream und fährt mit der oben beschriebenen Verarbeitung fort.

Schlussendlich sind in der Tabelle P_ENRICHED alle erfolgreich angereicherten Zahlungen vorhanden, sowie alle Zahlungen die bereits ein Mandat enthalten.

6.2.3.3 Zahlung aussteuern

Eine weitere Besonderheit stellen Zahlungen dar, welche nicht angereichert werden konnten. Diese sind in einem Puffer zwischengespeichert, beinhalten dabei jedoch nur die konkreten Zahlungen *<DrctDbtTxInf>* ohne die dazugehörige Zahlungsinformation *<PmtInf>*.

Aus diesem Grund wird während des Parsens eine Kopie vom Anfang der *<PmtInf>* bis zur Zahlung *<DrctDbtTxInf>* auf einem separaten Puffer abgelegt. Anschließend kann dieser Puffer mit dem Puffer für die ausgesteuerten Zahlungen zusammengefasst werden, um daraufhin wieder eine valide Zahlungsdatei generieren zu können.

Erreicht der Parser das Ende des Blobs, wird dieser zusammengefasste Puffer in ein ByteArray umgewandelt, um es als Blob in eine eigene Tabelle für die ausgesteuerten Zahlungen zu speichern.

Um im nächsten Modul einen gültigen GroupHeader erzeugen zu können, wird noch die Summe je Blob für alle erfolgreich angereicherten Zahlungen, sowie auch alle Zahlungen die bereits ein Mandat enthalten, gebildet und in die Tabelle P_ENRICHED als *CtrlSum* und *NbOfTxs* eingetragen.

6.2.3.4 Transaktionssicherheit

Im Modul Enrichment dürfen, ebenso wie im Importmodul, keine Zahlungen mehrfach bearbeitet oder ausgelassen werden. Aus diesem Grund muss auch der Anreicherungsprozess transaktionssicher ablaufen. Hierfür existiert eine Zählvariable *Enrich_Count*, welche die Anzahl der bereits verarbeiteten Blobs speichert. Mit dem aus dem Import-Modul bekannten Prinzip, wird auch hier eine doppelte Verarbeitung ausgeschlossen und die bereits angereicherten Blobs übersprungen.

Um fehlerhafte Datenbankanfragen abzufangen und somit inkonsistente Datenbeziehungen zu verhindern, wird ebenfalls auf die Wiederherstellungspunktmethode zurückgegriffen. Der Ablauf ist identisch mit dem unter Abschnitt 6.1.3.4 erläuterten Verfahren.

6.3 Das Modul Export

6.3.1 Prototypischer Ablauf des Exports

Beschreibung Anwendungsfall	
Name	Generieren einer neuen Zahlungsdatei mit den angereicherten Lastschriften
Kurzbeschreibung	Die angereicherten Blobs sollen als valide Zahlungsdatei zusammengesetzt werden, um sie danach weiterverarbeiten zu können.
Akteure	System
Auslöser	Programm bekommt die zu verarbeitenden Daten übergeben, die den Status Angereichert bzw. Ausgesteuert haben.
Ergebnis(se)	Eine Valide Zahlungsdatei ist zur weiteren Verarbeitung mit den angereicherten Zahlungen entstanden.
Eingehende Daten	Angereicherte Blobs aus der Datenbank liegen vor.
Vorbedingungen	Status der Blobs ist auf Angereichert/Ausgesteuert gesetzt.
Nachbedingungen	Das Statusflag muss gesetzt werden.
Essentielle Schritte	- Blobs werden aus der Datenbank gelesen. - Eine neue Zahlungsdatei muss erzeugt werden. - Der GroupHeader ist entsprechend der angereichterten Zahlungen anzupassen. - Die Blobs müssen zusammengesetzt werden. - Anschließend wird eine gültige PAIN-Datei erzeugt und an einem vorgegebenen Pfad gespeichert. - Die Fehler sind abzufangen und zu behandeln.

Abbildung 6.9: Anwendungsfall Export - Prototypischer Ablauf

Im letzten Schritt des Anreicherungsprozesses sind die fertig angereicherten Zahlungen in eine neue valide Zahlungsdatei zu exportieren. Dafür ist der GroupHeader, entsprechend des neuen Inhalts, anzupassen und neu zu erstellen. Anschließend werden daraus die fertigen und als Blobs in der Datenbank vorliegenden angereicherten Zahlungen generiert.

Für den Export sind insbesondere die Felder *CtrlSum* und *NbOfTxs* im neuen GroupHeader anzupassen. So beinhaltet das Feld *CtrlSum* die Buchungssumme aller angereicherten Payments, die in dieser Datei vorkommen. Das Feld *NbOfTxs* enthält hingegen die Gesamtzahl aller Transaktionen in dieser Datei.

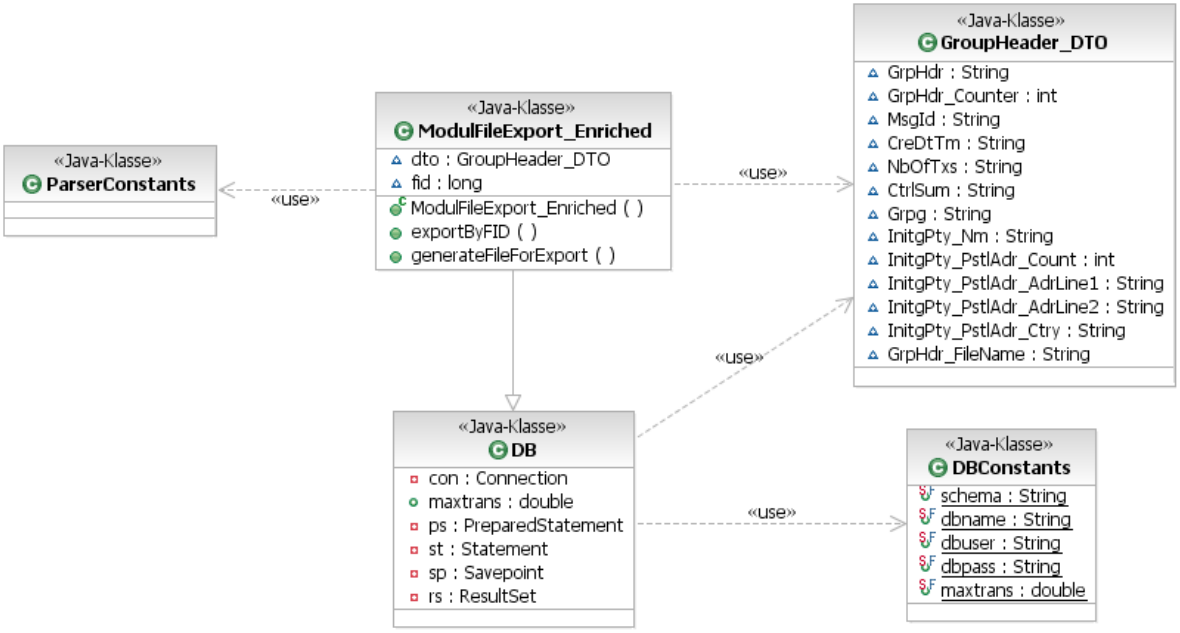


Abbildung 6.10: Klassendiagramm für das Modul Export_Enriched

6.3.2 Datenmodell des Exports

Das Export-Modul greift auf die bisher bekannten Tabellen zu und benötigt kein weiteres Datenmodell. Ist der Export abgeschlossen, wird außerdem ein Status für die erfolgreich durchlaufene Verarbeitung in der Tabelle P_TRANSACTION gesetzt.

6.3.3 Implementierung des Exports

Entsprechend den anderen Modulen, beginnt auch dieses Modul mit der Übergabe des Dateinamens, der Ermittlung der FID und der anschließenden Auswahl der zu verarbeitenden Blobs. Im nächsten Schritt wird der *Status* aus der Tabelle P_TRANSACTION überprüft, ob diese Blobs bereits verarbeitet worden sind. Ist dem nicht so, wird im Folgenden eine neue Zahlungsdatei erstellt, welche alle angereicherten Zahlungen für den übergebenen Dateinamen beinhaltet.

Hierbei ist in zwei Fällen zu unterscheiden, in den Export der angereicherten Zahlungen, und in den Export der ausgesteuerten Zahlungen. Die angereicherten Zahlungsdateien befinden sich in der Tabelle P_ENRICHED, allerdings nur in Form von Blobs und ohne ihren GroupHeader. Um eine neue valide Zahlungsdatei zu erstellen, muss zuerst der GroupHeader und hinterher die dazugehörigen Blobs geschrieben werden.

Mit den Woodstox-eigenen Methoden ist es bisher nicht möglich, den Namespace korrekt in die Ausgabedatei zu schreiben. Für den Prototyp wird deshalb der Dateikopf mittels *writeRaw()*-Methode so geschrieben, wie er in der Ausgangsdatei angegeben ist.

```
1 ...
2   writer.writeStartDocument("UTF-8", "1.0");
3   writer.writeRaw("<Document xmlns=\"urn:sepade:xsd:pain.008.001.01.grp\" xmlns:xsi=\"http://www.w3.
        org/2001/XMLSchema-instance\" xsi:schemaLocation=\"urn:sepade:xsd:pain.008.001.01.grp pain
        .008.001.01.grp.xsd\">");
4   writer.writeStartElement("pain.008.001.01");
5   writer.writeStartElement("GrpHdr");
6 ...
```

Listing 6.4: Auszugsweise Darstellung um den XML-Header korrekt zu schreiben

Anschließend werden die GroupHeader-Felder aus der Tabelle P_GRPHDR gelesen und in einem DTO gespeichert. Dies ermöglicht den Zugriff auf alle Felder mit nur einer Datenbankabfrage. Im nächsten Schritt serialisieren die XMLStreamWriter2-Methoden *writeStartElement()*, *writeCharacter()* und *writeEndElement()* diese Informationen in eine neue Datei.

Als nächstes ist eine Anpassung des GroupHeaders auf den neuen Inhalt notwendig. Für diesen Zweck bildet das vorige Modul eine Summe für alle pro Blob enthaltenen Zahlungen, welche einmal durch die Buchungssumme der einzelnen Transaktionen (*<CtrlSum>*) und einmal die die Summe Transaktionen (*NbOfTxs*) generiert wird. Da mehrere Blobs eine Zahlungsdatei bilden, werden diese Teilsummen, unter Nutzung der Datenbankfunktion *SUM()*, aufsummiert und in Variablen gespeichert. Als Ergebnis erhält man für die komplette Zahlungsdatei die Anzahl aller darin enthaltenen Transaktionen sowie deren Buchungssumme.

```
1  <GrpHdr>
2    <MsgId>Message-ID</MsgId>
3    <CreDtTm>2008-12-21T09:30:47.000Z</CreDtTm>
4    <NbOfTxs>3</NbOfTxs>     <!-- Anzahl aller Transaktionen -->
5    <CtrlSum>6966.54</CtrlSum>   <!-- Summe aller Transaktionen -->
6    <Grpg>MIXD</Grpg>
7    <InitgPty>
8      <Nm>Initiator Name</Nm>
9      </InitgPty>
10    ...
11 </GrpHdr>
```

Listing 6.5: Auszugsweise Darstellung des GroupHeaders

Danach liest der Prototyp alle angereicherten und dazugehörigen Blobs aus P_ENRICHED und schreibt diese mit der *writeRaw()*-Methode in die Ausgabedatei. Um die Wohlgeformtheit der Zahlungsdatei sicherzustellen, müssen noch alle schließenden Tags geschrieben werden. Schlussendlich ist eine neue valide Zahlungsdatei entstanden, welche alle gültigen und mit einem Mandat angereicherten Zahlungen enthält.

Für alle Zahlungen die ausgesteuert wurden, ergibt sich eine ähnliche Vorgehensweise. Hierbei wird jedoch die Tabelle P_NOTENRICHED verwendet und für die Ausgabe müssen die Pfade dementsprechend angepasst werden.

6.3.3.1 Transaktionssicherheit

Im Gegensatz zu den vorigen Modulen, benötigt das Modul Export keine speziellen Verfahren für die Transaktionssicherheit. Ist die Verarbeitung der angereicherten Zahlungen nicht erfolgreich und vollständig abgeschlossen, steht im Feld EXPORTCOMPL_OK_STATE der Wert *0*. Erst nach dem erfolgreichen Export wird dieser Wert auf *1* gesetzt. Wird die Verarbeitung zwischendrin unterbrochen, so überschreibt das Exportmodul die nur teilweise erzeugte Datei bei einer erneuten Ausführung. Der Grund ist, dass es mehr Zeit kosten würde eine fehlerhaft geschriebene Datei zu analysieren, anstatt diese zu überschreiben.

Für ausgesteuerte Zahlungen erfolgt die gleiche Vorgehensweise, jedoch wird hierbei das Feld EXPORTCOMPL_NOK_STATE genutzt.

6.4 Optimierung des Prototypen

6.4.1 Datenbankspezifische Optimierungen

Für die spätere Verwendung des Anreicherungsprozesses als Modul, stellen insbesondere die Datenbankzugriffe eine Optimierungsmöglichkeit dar. Aus diesem Grund sind alle Datenbankanfragen in eine eigene Klasse `DB` ausgelagert. Die Klasse `DBConstants` beinhaltet die Zugriffsdaten für die Datenbank sowie das verwendete Schema. Weiterhin stellen bei einer hohen Anzahl an Datenbankanfragen nicht geschlossene Datenbankressourcen nicht nur ein Problem, sondern häufig auch eine Absturzursache (Memory Leaks) dar. Um dieses Problem zu minimieren, ist im Folgenden die Nutzung eines ConnectionPools beschrieben.

6.4.1.1 Nutzung eines Connection Pools

Bevor auf eine Datenbank zugegriffen werden kann, muss eine Applikation einen Kommunikationskanal zwischen der Applikation und der Datenbank herstellen. Dieser Kanal wird als Datenbankverbindung bezeichnet und ihr Aufbau ist relativ zeitintensiv. Der Vorteil eines Connection Pools liegt darin, dass er nur einmal die Verbindung zur Datenbank aufbaut, diese jedoch bei einem *close()* nicht schließt, sondern sie in einem Pool freigibt [jav09a]. Der Connection Pool enthält dabei eine definierte Anzahl von physischen Verbindungen. Wird von der Applikation eine neue Verbindung angefordert, so gibt der Connection Pool eine Verbindung aus seinem Pool an die Applikation frei [Mar02, Seite 648 ff.]. Mit diesem Konzept kann eine optimale Ausnutzung der Ressource Verbindung erreicht werden.

6.4.1.2 Nutzung eines Datenbankindex

Um sehr viele Mandate verwalten zu können, ist die Optimierung der Zugriffe auf die Datenbank wichtig. Der Prototyp sucht die Mandatsinformationen anhand der REFERENCE_ID und des CREDITOR_ID_CODE aus der Tabelle P_MANDATE heraus. Da speziell die Suche bei sehr vielen Datensätzen sehr zeitaufwendig ist, besteht an dieser Stelle eine Möglichkeit der Optimierung. Ein DBMS stellt dafür eine Datenstruktur namens Index zur Verfügung. Dieser Datenbankindex ist eine Indexstruktur, welche von der Datenstruktur getrennt ist und die Suche sowie das Sortieren nach bestimmten Feldern beschleunigt. Ein Index besteht aus einer Ansammlung von Zeigern, welche eine Ordnungsrelation auf ein oder mehrere Spalten in einer Tabelle definieren. Wird eine indizierte Spalte als Suchkriterium herangezogen, so sucht das DBMS die gewünschten Datensätze anhand dieser Zeiger. Ohne einen Index müsste die Spalte sequentiell durchsucht werden, was vor allem bei sehr großen Datenmengen viel Zeit in Anspruch nimmt. Aus diesem Grund wird ein Index auf die Spalten REFERENCE_ID und CREDITOR_ID_CODE gelegt.

6.4.2 Javaspezifische Optimierung

Um die Mandatsinformationen anhand der UCI schnell analysieren zu können, wird nur ein Teilstring geparst. Damit kann, im Gegensatz zum Parsen der kompletten Zeichenkette, Verarbeitungszeit eingespart werden. Weiterhin wird für sämtliche Zeichenketten-Konkatenationen auf die StringBuilder-Klasse zurückgegriffen. Diese ist seit Java 5 enthalten und bietet, im Gegensatz zur standardmäßig genutzten String-Klasse mit dem + -Operator, eine viel höhere Geschwindigkeit. Nach [Ste09, Seite 19] benötigt die Verkettung von 50 000 String-Objekten mit der standardmäßigen String-Klasse ungefähr 9200 Millisekunden, während die gleiche Anzahl von Objekten mittels StringBuilder nur knapp 4 Millisekunden benötigt.

Eine weitere Optimierungsmöglichkeit stellt die sparsame Verwendung von Ressourcen in Java dar. So werden im Prototypen nicht nur die Puffer mit einer definierten Anfangsgröße initialisiert, sondern auch unmittelbar nach ihrer Verwendung gelöscht. Daraufhin ist der Java-GarbageCollector in der Lage, die nicht mehr genutzten Ressourcen schnell wieder freizugeben.

7 Test des Prototypen

Im Kapitel 5 wurden die Anforderungen an den Anreicherungsprozess definiert. Dabei soll der Prozess vor allem transaktionssicher und fehlerfrei ablaufen. Des Weiteren ist eine schnelle Verarbeitung von Massendaten gefordert. Die nachfolgenden Tests sollen die ordnungsgemäße Funktionalität des Prototypen überprüfen.

7.1 Testkonzept

Für diesen Test müssen zunächst Mandate in der Datenbank angelegt werden. Weiterhin werden Beispielzahlungen mit darin enthaltenen unterschiedlichen Informationen zur Identifikation der Mandate benötigt. Da es noch keiner vorgefertigten Beispiele gibt, müssen diese zunächst erzeugt werden.

Den Ausgangspunkt für die Beispielzahlungen bildet das XML-Schema `pain.008.001.01.xsd`. Anhand dessen werden nachfolgend verschiedene Varianten von Beispielzahlungen generiert. Hierbei variiert sowohl der Aufbau der Zahlungsdateien als auch deren Transaktionszahl. Um die Ergebnisse besser einordnen zu können, bieten sich folgende drei Szenarien an.

- `enrich10.xml` - Alle Zahlungen sind gültig und können angereichert werden.
- `disqualify10.xml` - Keine Zahlung davon ist gültig. Deshalb sind alle Zahlungen auszusteuern.
- `average10.xml` - In dieser Datei befinden sich drei Zahlungen für die Anreicherung, drei Zahlungen zur Aussteuerung und vier Zahlungen die bereits ein Mandat enthalten und in die fertige Datei für gültige Zahlungen mit übernommen werden sollen.

7.1.1 Generierung von Beispielmandaten

Um einen realistischen Test durchführen zu können, werden Beispielmandate benötigt. Für diesen Zweck trägt die Klasse `SampleMandates` 50 000 Mandate mittels einer Schleife in die Tabelle P_MANDATE ein. Der Aufbau dieser Tabelle ist im Anhang unter Abbildung A.2 dargestellt. Die für den Prototyp notwendigen Felder werden mit Beispieldaten belegt, u. a. das Feld *Last_Used_Date*. Für dieses Feld wird ein Zeitstempel mit der aktuellen Uhrzeit ermittelt und darin eingetragen.

7.1.2 Generierung von Beispielzahlungen

Jede generierte Beispieldatei enthält eine festgelegte Anzahl von Zahlungen *<DrctDbtTxInf>* in genau einem Zahlungsblock *<PmtInf>*. Zur Generierung von angemessenen Beispieldaten, werden modifizierte Zahlungen aus der Datei `pain.008.001.01.xml` für die neuen Beispieldateien genutzt. Um die Zahlungen besser unterscheiden und kontrollieren zu können, wird jeder Beispielzahlung eine andere Transaktionssumme *<InstdAmt>* zugewiesen. Der aufsummierte Geldbetrag *CtrlSum* und die Anzahl aller in dieser Beispieldatei vorhandenen Transaktionen *NbOfTxs* geben so einen Aufschluss über die Korrektheit des Anreicherungsprozesses. Für den ersten Test werden zehn Zahlungen pro Zahlungsblock festgelegt. Aus der Variante und der Anzahl der Zahlungen je Zahlungsblock, ergibt sich auch der Name der Beispielzahlung. So stellt die Datei `enrich10.xml` einen Zahlungsblock mit je zehn Zahlungen bereit. Diese Datei ist im Listing 7.1 auszugsweise dargestellt. In den Tests unter 5.5.4 konnte herausgefunden werden, dass sich eine Blobgröße von 50 MB gut für die weiteren Messungen eignet. Deshalb wird sie auf diese Größe festgelegt. Um vernünftige und aussagekräftige Ergebnisse zu bekommen, nutzen die Beispielzahlungsdateien zufällige Mandatsreferenzen zwischen 1 und 50 000. Durch die im Abschnitt 6.4.1.2 erfolgte Indizierung ist eine schnelle Suche der Daten sichergestellt.

Im weiteren Verlauf werden die Zahlungsblöcke *<PmtInf>* schrittweise erhöht. Bei der manuellen Erstellung der Beispieldateien werden die Speichergrenzen eines gebräuchlichen XML-Editors schnell erreicht.

Um dennoch große Zahlungsdateien zu erstellen, wird auf die Klasse `PainGenerator` zurückgegriffen. Es liest eine Zahlungsdatei ein und schreibt, mittels einer Schleife, alles von *<PmtInf>* bis *</PmtInf>* in einem FileOutputStream. Ein Parameter *amountTransaction* gibt dabei die Anzahl der Schleifendurchläufe an. Als Ergebnis bekommt man eine valide Zahlungsdatei mit der unter *amountTransaction* angegebenen Zahl als *PmtInf*-Blöcke.

```
1  <pain.008.001.01>
2    <GrpHdr>
3    <PmtInf>
4      <ReqdColltnDt>2009-12-03</ReqdColltnDt>
5      <DrctDbtTxInf>                          <!-- 1. Zahlung -->
6        <InstdAmt Ccy="EUR">8735.78</InstdAmt>
7      </DrctDbtTxInf>
8      <DrctDbtTxInf>...</DrctDbtTxInf>        <!-- 2. Zahlung -->
9      <DrctDbtTxInf>...</DrctDbtTxInf>        <!-- 3. Zahlung -->
10     ...
11     <DrctDbtTxInf>...</DrctDbtTxInf>        <!-- 10. Zahlung -->
12   </PmtInf>
13 </pain.008.001.01>
```

Listing 7.1: Auszugsweise Darstellung der Beispielzahlungsdatei `enrich10.xml`

7.2 Testablauf

In der Klasse `TestComplete` wird den einzelnen Modulen eine Beispielzahlungsdatei übergeben. Die Laufzeit der Testmethoden wird anhand der Differenz zweier Zeitstempel, vor und nach dem Ausführen der Testmethoden, berechnet. Ist der Test beendet, sind die angereicherten und die ausgesteuerten Zahlungen in einem separaten Ordner vorhanden. Dieser Ablauf wird für alle generierten Beispielzahlungen durchlaufen. Dabei unterscheiden sich die Dateien hinsichtlich ihrer darin enthaltenen Transaktionen. Ebenfalls sollen alle drei verschiedenen Varianten berücksichtigt werden, um die Auswirkungen auf den Prototypen zu erkennen. Im Anhang A.3 ist das Klassendiagramm für den Test zu finden.

Weiterhin wird die Verarbeitung in jedem der drei Module unterbrochen, um in der Tabelle P_TRANSACTION den Status überprüfen zu können und somit die Transaktionssicherheit zu testen. Im Anschluss soll mit der Bearbeitung der Datei fortgefahren und die korrekte Weiterverarbeitung überprüft werden.

Folgendes Testsystem wurde für die Messungen verwendet:

- Notebook IBM ThinkPad T41
- Intel Pentium M mit 1,7 GHz
- 1,5 GB Arbeitsspeicher
- JDK1.6.0_06 als Java-Laufzeitumgebung
- IBM DB2 Datenbanksystem in der Version 9

7.3 Testergebnisse

Um korrekte Testergebnisse zu erhalten, wurden je zehn Messungen durchgeführt und ihre Ergebnisse gemittelt, welche in den Tabellen 7.1, 7.2 und 7.3 übersichtlich dargestellt sind. Das Modul Enrichment benötigt von allen drei Modulen die längste Zeit. Dies lässt sich durch die vielen Datenbankzugriffe erklären, welche die meiste Zeit beanspruchen. Das Modul Import ist schneller als das Modul Enrichment, benötigt jedoch noch deutlich länger in der Verarbeitung als das Modul Export. Sind keine auszusteuernden Zahlungen vorhanden, wie im Fall `enrich10.xml`, dann erzeugt der Export für auszusteuernde Zahlungen keine Datei. Deshalb ist die Laufzeit für diesen Fall unter einer Sekunde. Weiterhin zeigen alle drei Varianten pro Modul und Transaktionszahl recht gleichmässige Laufzeiten. Die Unterschiede ergeben sich vor allem aus der Tatsache, dass der Prototyp und die Datenbank auf dem gleichen System liefen und sich somit die Laufzeiten nicht exakt bestimmt werden konnten. Anhand der Tabellen 7.1, 7.2 und 7.3 lässt sich erkennen, dass der Prototyp für eine Zahlungsdatei mit 1 000 000 an Transaktionen ungefähr 45 Minuten benötigte. Die Anforderung an eine schnelle Verarbeitung von Massendaten ist somit erfüllt.

7.3.1 Test der vom Prototyp erzeugten Dateien

Um die Korrektheit zu überprüfen, wurde die Ausgabedatei mittels des Editors analysiert. Dabei stimmte die Transaktionssumme und die Anzahl der darin enthaltenen Zahlungen mit der vorher verarbeiteten Zahlungsdatei überein. Wurde beispielsweise die Datei `average10_10.xml` verarbeitet, dann befanden sich 70 Transaktionen in der Datei für angereicherte Zahlungen (angereicherte und bereits mit Mandat vorhandene Zahlungen), und 30 Transaktionen in der Datei für ausgesteuerte Zahlungen. Die *CtrlSum* stimmte ebenfalls mit der Summe der Einzeltransaktionen überein. Außerdem wurden Zahlungen, deren Mandat zwar in der Datenbank vorlag, jedoch schon älter als 3 Jahre waren, automatisch in die Datei für ausgesteuerte Zahlungen eingefügt.

7.3.2 Test der Transaktionssicherheit

Die Abbildung 7.1 zeigt den Verarbeitungsstand in der Tabelle P_TRANSACTION. In Zeile 1 mit der FID=1 ist die Datei `average10_1000.xml` angegeben und zeigt mit dem Flag IMPORTCOMPL_STATE (Spalte 3) das sie bereits komplett importiert, jedoch nur bis zum 51. *PmtInf*-Block angereichert wurde. In der Spalte ENRICHCOMPL_STATE (Spalte 6) ist deshalb der Wert *0* gesetzt und in der Spalte ENRICH_COUNT (Spalte 7) mit dem Wert *51* initialisiert. Die zweite Zeile mit der FID=2 zeigt die Datei `average10_10000.xml` und stellt dar, dass bisher 1130 *PmtInf*-Blöcke importiert wurden. Eine fertige Verarbeitung ist in Zeile 3 für die Datei *average10_100.xml* abgebildet.

FID	FILENAME	IMPORT...	GRPHDRCOMPL...	PMTINF_COUNT	ENRICH...	ENRICH_COUNT	EXPORT...	EXPORTCOMPL_NOK...
1	average10_1000.xml	1	1	0	0	51	0	0
2	average10_10000....	0	1	1130	0	0	0	0
3	average10_100.xml	1	1	0	1	0	1	1

Abbildung 7.1: Darstellung des Verarbeitungsstandes in der Tabelle P_Transaction

Nach einer erneuten Ausführung dieser Dateien haben alle Statusflags den Wert *1* und die Zählvariablen PMTINF_COUNT und ENRICH_COUNT sind auf den Wert *0* gesetzt. Dies zeigt, dass die Transaktionssicherheit ordnungsgemäß funktioniert und ist in Abbildung 7.2 dargestellt.

FID	FILENAME	IMPORT...	GRPHDRCOMPL...	PMTINF_COUNT	ENRICH...	ENRICH_COUNT	EXPORT...	EXPORTCOMPL_NOK...
1	average10_1000.xml	1	1	0	1	0	1	1
2	average10_10000....	1	1	0	1	0	1	1
3	average10_100.xml	1	1	0	1	0	1	1

Abbildung 7.2: Darstellung des Verarbeitungsstandes in der Tabelle P_Transaction

Anzahl der Zahlungen	Datei-größe [MB]	Import [s]	Enrich-ment [s]	Export OK [s]	Export NotOK [s]	Gesamt-zeit [s]
100	0,14	1,1	0,4	0,1	0,1	1,8
500	0,70	1,1	1,7	0,1	0,1	3,0
1000	1,40	1,6	2,8	0,1	0,1	4,6
5000	7,0	5,1	12,6	0,4	0,1	18,2
10 000	14,0	8,9	22,2	0,8	0,1	31,9
50 000	70,2	47,6	159,1	3,4	0,1	210,3
100 000	140,3	77,2	227,3	24,1	0,1	328,7
500 000	701,7	300,9	911,8	81,7	0,5	1294,8
1 000 000	1403,3	693,0	1896,2	231,3	0,6	2821,0

Tabelle 7.1: Messergebnisse für angereicherte Zahlungen - `enrich10.xml`

Anzahl der Zahlungen	Datei-größe [MB]	Import [s]	Enrich-ment [s]	Export OK [s]	Export NotOK [s]	Gesamt-zeit [s]
100	0,14	1,0	0,3	0,1	0,2	1,5
500	0,70	1,3	1,0	0,1	0,2	2,5
1000	1,40	1,6	2,1	0,1	0,2	4,1
5000	7,0	4,7	9,9	0,1	0,4	15,0
10 000	14,0	9,7	12,9	0,1	0,7	23,3
50 000	70,1	53,8	60,4	0,1	3,0	117,1
100 000	140,2	74,6	118,5	0,1	6,1	199,1
500 000	700,8	305,0	652,0	0,2	106,3	1063,6
1 000 000	1401,5	535,6	1338,6	0,5	187,5	2062,0

Tabelle 7.2: Messergebnisse für ausgesteuerte Zahlungen - `disqualify10.xml`

Anzahl der Zahlungen	Dateigröße [MB]	Import [s]	Enrichment [s]	Export OK [s]	Export NotOK [s]	Gesamtzeit [s]
100	0,13	1,0	0,4	0,1	0,1	1,6
500	0,70	1,1	1,2	0,1	0,2	2,5
1000	1,34	1,4	1,9	0,1	0,2	3,6
5000	6,7	2,7	9,3	0,4	0,4	12,8
10 000	13,4	6,7	34,0	0,6	0,5	41,7
50 000	67,1	24,0	74,1	2,3	1,2	101,5
100 000	134,2	47,6	150,8	4,5	2,5	205,4
500 000	670,9	198,8	715,9	80,2	45,0	1039,8
1 000 000	1341,7	287,6	1481,2	161,9	83,5	2014,1

Tabelle 7.3: Messergebnisse für gemischte Zahlungen - `average10.xml`

8 Zusammenfassung

8.1 Fazit

Diese Masterarbeit befasste sich mit der Analyse eines neuen Zahlungsinstruments in Europa namens SEPA Direct Debit. SEPA verfolgt das Ziel, dass europaweit bargeldlose Zahlungen über Ländergrenzen hinweg genauso unkompliziert und sicher werden, wie es diese bisher innerhalb der Nationalstaaten sind. Zu Beginn der Arbeit ist für das Verständnis ein Überblick über SEPA gegeben sowie die Besonderheiten und Hintergründe erläutert. So stellt beispielsweise die Ablösung der Kontonummer durch die international gültige IBAN eine wesentliche Neuerung dar.

Darauf aufbauend wurde das deutsche Lastschriftverfahren mit dem neu definierten Verfahren SEPA Direct Debit verglichen. Hierbei stellte sich insbesondere die unkomplizierte Abwicklung von Lastschriften innerhalb Deutschlands heraus. So existieren für das deutsche Lastschriftverfahren zwei Varianten, das Einzugsermächtigungs- und das Abbuchungsauftragsverfahren. Die Analyse der SEPA-Lastschrift zeigte einen vollkommen neuen Prozessablauf mit einem viel höheren Verwaltungsaufwand. Für die SEPA-Lastschrift wurden vom EPC sowohl vorhandene Ansätze aufgegriffen als auch vollkommen neue Verfahren entwickelt. Beispielsweise erfolgt die eindeutige Identifizierung des Creditors anhand einer UCI. Weiterhin bietet ein spezielles B2B-Verfahren, welches ausschließlich für Geschäftskunden vorgesehen ist, eine wesentliche Zeitersparnis im Geschäftsbereich. Dieses B2B-Verfahren ist dabei an das bisherige Abbuchungsauftragsverfahren angelehnt, speziell was den Widerruf von autorisierten Lastschriften betrifft. Eine grundlegende Neuerung stellt die Form der Autorisierung von Lastschriften durch ein Mandat dar. Um den Nachweis erbringen zu können, dass getätige Lastschriften gültig sind, wurde der Creditor vom EPC verpflichet diese Mandate zu archivieren. Für diese Archivierung ist, insbesondere bei einem hohen Aufkommen von Lastschriften, ein IT-gestütztes System unerlässlich.

An dieser Stelle setzt nun ein IT-System in Form des Mandatsmanagements an. Die Anforderungen an ein solches System wurden in Kapitel 4 untersucht, spezifiziert und anschließend in einem Architekturentwurf dargelegt. Dabei konnten drei Aufgabenbereiche identifiziert werden: das Verwalten von Mandaten, das Anreichern von Lastschriften mit den zugehörigen Mandaten und das Prüfen von Mandaten. Im Anreicherungsprozess muss der Creditor an jede Lastschrift das dazugehörige Mandat anhängen. Damit dieser Vorgang spezifiziert werden konnte, war zunächst eine Analyse des neuen Zahlungsablaufs notwendig. Hierbei wurde festgestellt, dass jede SEPA-Lastschrift ein dazugehöriges Mandat enthalten muss. Der Creditor ist der Initiator der Lastschrift und deshalb vom Standard her für die Verwaltung der Mandate festgelegt. Aus diesem

Grund ist er auch für die Anreicherung der Lastschrift mit den zugehörigen Mandaten zuständig. Die Untersuchung des Zahlungswegs brachte zwei Zahlungsformate hervor, das PAIN-Format für die Kunde-Bank-Schnittstelle und das PACS-Format im Interbankenbereich.

Für den Creditor ist dabei ausschließlich das PAIN-Format von Bedeutung und wurde deshalb im Abschnitt 5.3 genauer untersucht. Die Zahlungsdateien sind im XML-Format aufgebaut und enthalten bereits definierte leere Felder für die Mandatsdaten. Um diese Informationen an die gewünschte Stelle einfügen zu können, war eine umfassende Untersuchung zur Verarbeitung dieser Daten notwendig. Für diesen Zweck bietet die Programmiersprache Java bereits verfügbare Technologien zur XML-Verarbeitung (Parser) an. Deshalb folgte eine Untersuchung von verschiedenen XML-Parser-Konzepten, um den Vorgaben einer schnellen und ressourcenschonenden Verarbeitung zu entsprechen. Hierbei zeigten streambasierte Parser einen deutlichen Vorteil gegenüber modellbasierten Parsern. Den besten Kompromiss aus Schnelligkeit und Speicherverbrauch stellt die Firma Codehaus mit dem streambasierten Parser Woodstox zur Verfügung, weshalb dieser Parser auch für die weitere Implementierung festgelegt wurde. Anschließend müssen die geparsten Daten noch persistiert werden. Bezugnehmend auf vorausgegangene Erkenntnisse und eigene Untersuchungen, stellte sich die Speicherung großer Datenmengen in Form von Blobs als beste Möglichkeit hinsichtlich Geschwindigkeit und Speicherverbrauch heraus. Danach wurde der Ablauf für den Anreicherungsprozess genauer definiert und ist in drei Module (Import, Enrichment und Export) untergliedert worden.

Im Kapitel 6 wurde für jedes Modul ein Datenmodell entworfen und der Ablauf festgelegt. Anschließend folgte die prototypische Implementierung jedes dieser Module. Dabei stellte sich zunächst das Problem heraus, dass das Mandat nicht ohne eine weitere Verarbeitung generiert und eingefügt werden konnte. So findet man beispielsweise die zur Identifizierung benötigte UCI an zwei verschiedenen Stellen vor, je nachdem ob es sich beim Creditor um eine Privatperson oder um eine geschäftliche Organisation handelt. Des Weiteren muss das Mandat oberhalb der UCI im XML-Dokument eingefügt werden. Da jedoch der Woodstox-Parser streambasiert ist und sich seine Verarbeitungsposition nicht merken kann, musste die Navigation innerhalb des XML-Dokumentes selbst implementiert werden. Für diesen Zweck wurde u. a. auf Flags zurückgegriffen, was im Kapitel 6 unter Abschnitt 6.1.3.1 genauer erläutert ist.

Außerdem mussten die großen Zahlungsdateien in kleinere Teile gesplittet werden, jedoch ohne dabei ihre Integrität zu zerstören. Um dieses garantieren zu können, wurden die großen Datenmengen gesplittet und als einzelne Teile in Form von Blobs anschließend in eine Datenbank persistiert.

Im nächsten Schritt wurden diese Blobs mit den zugehörigen Mandaten angereichert. Für diesen Zweck mussten die Mandate zunächst auf ihre Existenz und Gültigkeit geprüft werden. Liegt für die Zahlung kein Mandat vor, so musste diese Zahlung von den anderen Zahlungen abgesplittet und eigens gespeichert werden. Eine Zahlung stellt jedoch nur einen kleinen Teil aus dem XML-Dokument dar, weswegen eine weitere Möglichkeit zur Verknüpfung dieser Daten gefunden werden musste. Dies ist im Abschnitt 6.2.3.3 erklärt. Nach der Anreicherung liegen sowohl die

angereicherten als auch die ausgesteuerten Zahlungen in einer eigenen Tabelle als Blobs vor. Um daraus wieder eine gültige Zahlungsdatei zu erzeugen, müssen die verarbeiteten Daten als eine neue Datei exportiert werden. Für diesen Zweck ist eine Anpassung des Headers der Zahlungsdatei notwendig, da darin die Anzahl aller in der Zahlungsdatei enthaltenen Transaktionen sowie der Gesamtbetrag aller Transaktionen stehen.

Im Kapitel 7 wurde der entstandene Prototyp auf seine ordnungsgemäße Funktionalität untersucht. Dabei konnte auch die Transaktionssicherheit sichergestellt und nachgewiesen werden. Die Erfüllung der Vorgabe einer schnellen Verarbeitung lässt sich bei der Anreicherung von 1 000 000 Zahlungen erkennen. Eine solche Datei hat ungefähr eine Größe von einem Gigabyte und konnte auf dem Testsystem in 45 Minuten verarbeitet werden. Mit dieser Arbeit konnte ein voll funktionsfähiger Prototyp geschaffen werden, der speziell den Anreicherungsprozess von SEPA Direct Debits umsetzt.

8.2 Ausblick

Der produktive Einsatz für das SEPA-Lastschriftverfahren ist vom EPC für November 2009 vorgesehen. Mit diesem entwickelten Prototyp konnte ein Teil des sehr komplexen Mandatsmanagements entworfen werden. Für eine finale Lösung ist die Einbeziehung eines konkreten Kunden unerlässlich, da viele Details sehr kundenspezifisch sind. Bisher existiert auch noch kein SEPA-fähiges Zahlungssystem mit Mandatsmanagement, was einen Vergleich zu anderen Alternativen damit nicht zulässt.

Für einen produktiven Einsatz ist die Anbindung an eine Logging-Funktionalität unerlässlich. Ebenso gibt es interessante Ansätze, die den Umgang mit XML und Java weiter erleichtern könnten. So stellt beispielsweise das Serialisierungsframework Castor [Exo09] eine gute Alternative zum Zugriff und der Weiterverarbeitung von XML-Daten zur Verfügung. Inwieweit es jedoch den Anforderungen an einen geringen Ressourcenverbrauch genügt, müsste weitergehend untersucht und getestet werden.

Eine weitere Komponente in der zukünftigen Nutzung der SEPA-Lastschrift gibt das EPC selbst vor, in dem es zum Zeitpunkt des Abschlusses dieser Arbeit die Verwendung eines elektronisch übermittelten Mandats (beispielsweise via eines Internetportals bei der Bank) spezifiziert. Diese Möglichkeit stellt ebenfalls eine neue Anforderung an ein Mandatsmanagement, welches jedoch in dieser Arbeit noch nicht mit berücksichtigt werden konnte.

A Anhang

A.1 Ablauf von PACS-Nachrichten zwischen den Banken und dem Clearer

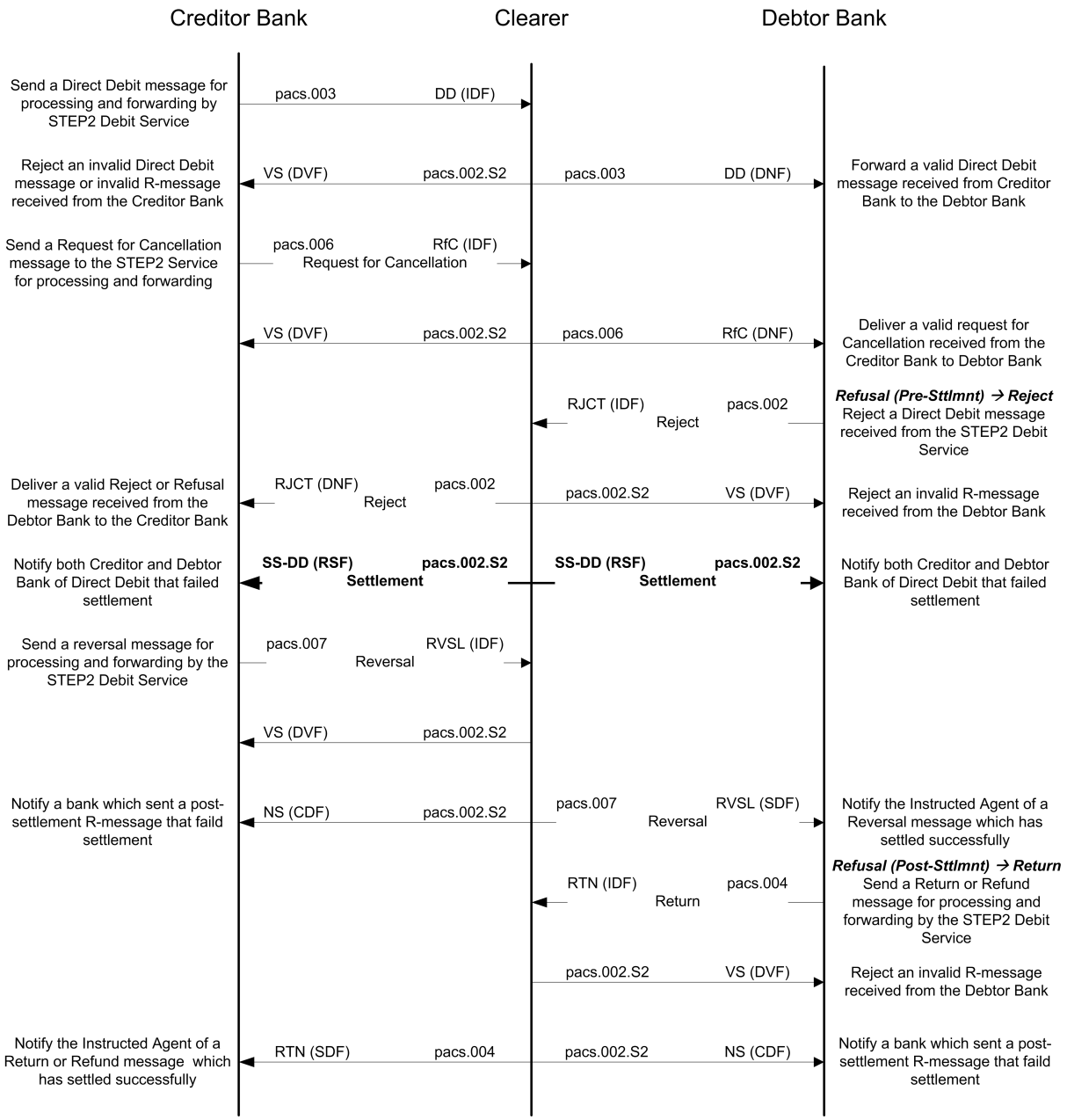


Abbildung A.1: Schematischer Ablauf von PACS-Zahlungsnachrichten

A.2 Test des JAXB-Parsers

Für den Test des JAXB-Parsers muss zunächst die JAXB-Klasse eingebunden werden. Getestet wurde mit JAXB in der Version 2.0.5, welches unter `https://jaxb.dev.java.net/` heruntergeladen kann. Weiterhin bietet JAXB ein Kommandozeilenprogramm an, welches aus einem gegebenen XML-Schema die passenden Java-Klassen für die Bearbeitung des XML-Dokumentes erzeugt.

Folgender Befehl wird für die Generierung der Java-Klassen aus dem gegebenen Schema `pain.008.001.01.xsd` verwendet.

```
xjc -p de.htwm.mhauste.jaxb.pain -d src pain.008.001.01.xsd
```

Mit den daraufhin erzeugten Java-Klassen ist die einfache Verarbeitung des XML-Dokumentes möglich. Mit Hilfe dieser Java-Klassen ist die Struktur des XML-Dokumentes nachgebildet und ermöglicht so eine recht einfache Navigation innerhalb eines XML-Dokumentes.

Transaktionszahl	Größe des Java-Heapspace [MB]	Benötigte Zeit zum Parsen [s]
500	Standard	0,7
1000	Standard	0,8
5000	Standard	1,3
10 000	Standard	1,8
50 000	Standard	6,0
100 000	512	11,6
500 000	512	78,1
1 000 000	1024	Nicht möglich

Tabelle A.1: Zeitmessung für das Parsen mit JAXB

Um die Geschwindigkeit des JAXB-Parsers zu testen, wurden einige Elemente aus einer Beispielzahlungsdatei geparst in Abhängigkeit der darin enthaltenen Transaktionen. Das Listing «» zeigt, wie JAXB zum parsen genutzt wird und die Tabelle A.1 stellt die Ergebnisse dar.

Für diese Messungen ist jedoch eine Anpassung des Java-Heapspace notwendig, welchen Java normalerweise selbst verwaltet. Dieser ist üblicherweise mit 1 MB standardmäßig voreingestellt und wird durch zunehmenden Ressourcenanstieg schrittweise erhöht. Mit dem Parameter *-Xmx512M* kann der Java-Heapspace manuell auf 512 MB festgelegt werden. Für den Test mit dem JAXB-Parser war eine Erhöhung bereits ab 100 000 Transaktionen notwendig. Bei mehr als 1 000 000 Zahlungen brach JAXB, trotz festgelegtem Heapspace auf 1 GB, dennoch mit einem Fehler das Testprogramm ab. Diese Fehlermeldung ist im Listing A.2 kurz dargestellt.

```java
public void parseWithJAXB(InputStream is) {
  try {
    /* Create a JAXBContext capable of handling classes generated into */
    JAXBContext jc = JAXBContext.newInstance( "de.htwm.mhauste.jaxb.pain" );
    /* Create an Unmarshaller */
    Unmarshaller u = jc.createUnmarshaller();
    Document document = (Document)((JAXBElement) u.unmarshal(is)).getValue();
    /* Display something */
    System.out.println( "Some content inside PmtInf: " );
    /* Get Content */
    Pain00800101 pain = document.getPain00800101();
    GroupHeader24 grphdr = pain.getGrpHdr();
    grphdr.getCreDtTm();
    for (int i=0; i< pain.getPmtInf().size(); i++) {
      pain.getPmtInf().get(i).getPmtMtd()); //DD
      for (int j=0; j < pain.getPmtInf().get(0).getDrctDbtTxInf().size(); j++) {
        pain.getPmtInf().get(0).getDrctDbtTxInf().get(j).getDrctDbtTx().getCdtrSchmeId().getId().getPrvtId
            ().getOthrId().getId().toString();   // ID
      }
    }
  } catch( JAXBException je ) {
    je.printStackTrace();
  }
}
```

Listing A.1: Beispielmethode zum Parsen mittels JAXB

```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
  at java.util.Arrays.copyOfRange(Unknown Source)
  at java.lang.String.<init>(Unknown Source)
  at java.lang.StringBuilder.toString(Unknown Source)
  at com.sun.xml.internal.bind.v2.model.impl.RuntimeBuiltinLeafInfoImpl$1.parse(Unknown Source)
  at com.sun.xml.internal.bind.v2.model.impl.RuntimeBuiltinLeafInfoImpl$1.parse(Unknown Source)
  at com.sun.xml.internal.bind.v2.runtime.reflect.TransducedAccessor$CompositeTransducedAccessorImpl.
      parse(Unknown Source)
  at com.sun.xml.internal.bind.v2.runtime.unmarshaller.LeafPropertyLoader.text(Unknown Source)
  at com.sun.xml.internal.bind.v2.runtime.unmarshaller.UnmarshallingContext.text(Unknown Source)
  at com.sun.xml.internal.bind.v2.runtime.unmarshaller.SAXConnector.processText(Unknown Source)
  at com.sun.xml.internal.bind.v2.runtime.unmarshaller.SAXConnector.endElement(Unknown Source)
  at com.sun.org.apache.xerces.internal.parsers.AbstractSAXParser.endElement(Unknown Source)
  at com.sun.org.apache.xerces.internal.impl.XMLDocumentFragmentScannerImpl.scanEndElement(Unknown
      Source)
   ...
  at com.sun.xml.internal.bind.v2.runtime.unmarshaller.UnmarshallerImpl.unmarshal(Unknown Source)
  at javax.xml.bind.helpers.AbstractUnmarshallerImpl.unmarshal(Unknown Source)
  at javax.xml.bind.helpers.AbstractUnmarshallerImpl.unmarshal(Unknown Source)
  at de.htwm.mhauste.jaxb.ParseWithJAXB.parseWithJAXB(ParseWithJAXB.java:51)
  at de.htwm.mhauste.jaxb.ParseWithJAXB.setJAXB(ParseWithJAXB.java:28)
  at de.htwm.mhauste.jaxb.TestJAXB.main(TestJAXB.java:24)
```

Listing A.2: Fehlermeldung ab 1 000 000 Transaktionen beim Parsen mit JAXB

A.3 Aufbau der Tabelle P_MANDATE

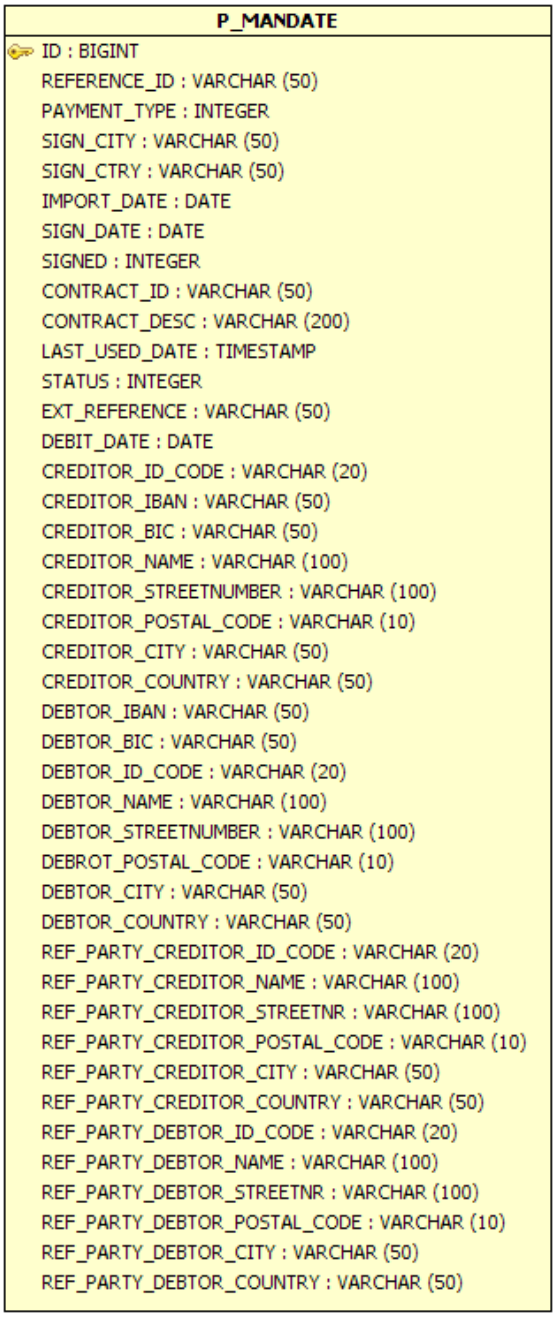


Abbildung A.2: Aufbau der Tabelle P_MANDATE des Prototyps

A.4 Klassendiagramm des Testprogramms

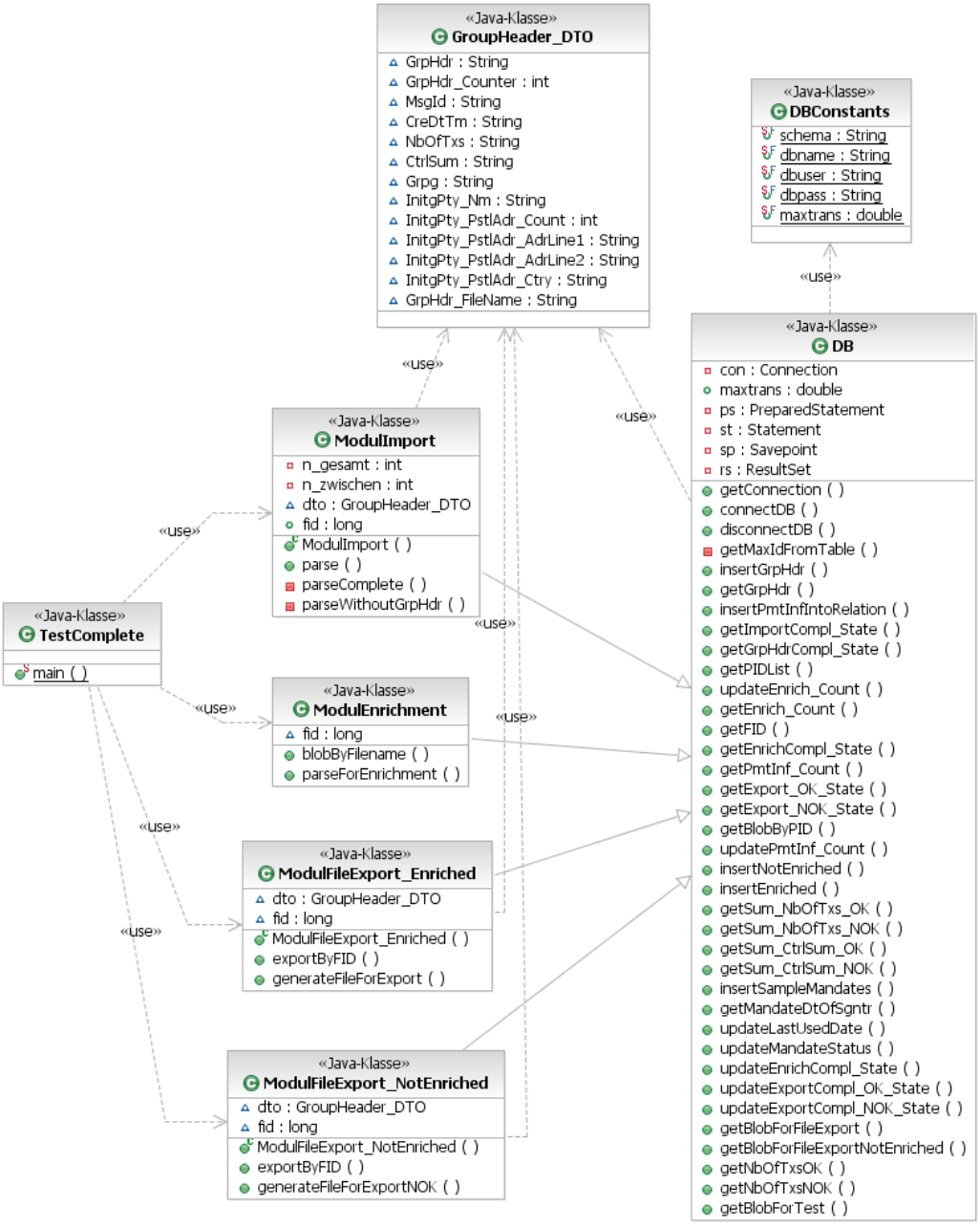


Abbildung A.3: Klassendiagramm für den Test des Prototypen

A.5 Verknüpfung zwischen Attributen aus dem Standard und den Tags der PAIN.008-Zahlungsnachricht

Feld im Regelwerk	Beschreibung des Feldes	Tagname	Pfadangabe
AT-01	Unique Mandate reference	MndtId	PmtInf/DrctDbtTxInf/ DrctDbtTx/MndtRltInf/
AT-02	Identifier of the Creditor	CdtrSchmeId	PmtInf/DrctDbtTxInf/ DrctDbtTx/
AT-03	Name of the Creditor	Nm	PmtInf/Cdtr/
AT-04	IBAN account number of Creditor	CdtrAcct	PmtInf/
AT-05	Address of the Creditor	PstlAdr	PmtInf/Cdtr/
AT-06	Amount of the Collection in euro	InstdAmt	PmtInf/DrctDbtTxInf/
AT-07	IBAN account number of Debtor	DbtrAcct	PmtInf/DrctDbtTxInf
AT-08	Identifier of the underlying contract		
AT-09	Address of Debtor	PstlAdr	PmtInf/DrctDbtTxInf/ Dbtr/
AT-10	The Creditor's reference of Collection		
AT-11	Due Date of Collection	ReqdColltnDt	PmtInf/
AT-12	BIC code of Creditor-Bank	BIC	PmtInf/CdtrAgt/ FinInstnId
AT-13	BIC code of Debtor-Bank	DbtrAgt	PmtInf/DrctDbtTxInf/
AT-14	Name of Debtor	Nm	PmtInf/DrctDbtTxInf/ Dbtr/
AT-15	Name of Debtor Reference Party	UltmtDbtr	PmtInf/DrctDbtTxInf/
AT-16	Placeholder for electronic signature	ElctrncSgntr	PmtInf/DrctDbtTxInf/ DrctDbtTx/MndtRltInf/

Tabelle A.2: Verknüpfung zwischen Attributen aus dem Standard und den Tags der PAIN-Nachricht

Feld im Regelwerk	Beschreibung des Feldes	Tagname	Pfadangabe
AT-18	Identifier of the original Creditor who issued the Mandate	OrgnlCdtrSchmeId	PmtInf/DrctDbtTxInf/ DrctDbtTx/MndtRltInf/ AmdmntInfDtls
AT-19	Unique Mandate reference as given by original Creditor who issued the Mandate	OrgnlMndtId	PmtInf/DrctDbtTxInf/ DrctDbtTx/MndtRltInf/ AmdmntInfDtls
AT-20	Identification code of SEPA Direct Debit Mandate		
AT-21	Transaction Type	SeqTp	PmtInf/PmtTpInf/
AT-22	Remittance Information from Creditor to Debtor		
AT-24	Reason for amendment of the Mandate	AmdmntInfDtls	PmtInf/DrctDbtTxInf/ DrctDbtTx/MndtRltInf/
AT-25	Date of Signing	DtOfSgntr	PmtInf/DrctDbtTxInf/ DrctDbtTx/MndtRltInf/
AT-27	Debtor identification code	Id	PmtInf/DrctDbtTxInf/ DbtrAcct/

Tabelle A.3: Verknüpfung zwischen Attributen aus dem Standard und den Tags der PAIN-Nachricht

B Thesen

1. Mit der Einführung von SEPA-Zahlungsinstrumenten werden bargeldlose Zahlungen innerhalb Europas verbessert.
2. Das neue SEPA Direct Debit-Verfahren ist wesentlich komplexer als das deutsche Lastschriftverfahren und benötigt deshalb ein unterstützendes Mandatsmanagement für den Zahlungsverkehr.
3. Mit dem Einsatz eines Mandatsmanagements wird eine neue Qualifikation in der Abwicklung von Lastschriften erreicht, da bisherige Zahlungssysteme nicht für den Umgang mit Mandaten konzipiert sind.
4. Insbesondere der neu definierte Prozess des Anreicherns innerhalb des Mandatsmanagements stellt hohe Anforderungen von schneller und zugleich ressourcenschonender Verarbeitung an das System.
5. Die Nutzung von streambasierten Parsern ermöglicht insbesondere im Prozess des Anreicherns von sehr großen Datenmengen eine wesentlich schnellere und speichereffiziente Verarbeitung als dies mit modellorientierten Parsern möglich ist.
6. Durch die Speicherung von XML-basierten Zahlungsdateien in Form von Blobs im Anreicherungsprozess wird eine wesentlich höhere Geschwindigkeit erzielt, als dies mit strukturierter Speicherung möglich ist.
7. Eine fehlerfreie Verarbeitung der Zahlungsinformationen nach einem Abbruch oder Systemfehler wird durch Statusflags in der Datenbank garantiert.
8. Durch eine sorgfältige Auswahl der zu nutzenden Javatypen für die Implementierung, beispielsweise die Nutzung der Klasse StringBuffer, ist eine große Einsparung an Verarbeitungszeit möglich.
9. Die Testergebnisse der Implementierung in dieser Arbeit belegen, dass mit Java eine hohe Verarbeitungsgeschwindigkeit in Bezug auf den Anreicherungsprozess bei gleichzeitig akzeptablen Ressourcenverbrauch erzielt werden kann.
10. Die Nutzung eines ConnectionPools kann kritische Fehler durch zuviele gleichzeitig geöffnete Datenbankverbindungen verhindern.

Literaturverzeichnis

[Bar08] Bartsch, Christian: *Zahlungsverkehrsfragen.de*. `http://www.zahlungsverkehrsfragen.de/lastschrift.html`, verfügbar am 19.05.2008.

[Bun06] Bundesverband deutscher Banken: *ISO 20022 im Überblick*, Dezember 2006.

[Bun08] Bundesverband deutscher Banken: *SEPA 2008 - Einheitliche Zahlungsinstrumente für Europa*, Dezember 2008.

[Cod09] Codehaus Foundation: *Woodstox - High-performance XML processr*. `http://woodstox.codehaus.org/`, verfügbar am 12.01.2009.

[Deu08a] Deutsche Bundesbank: *Die Vision eines einheitlichen Zahlungsverkehrsraums (SEPA)*, 2008.

[Deu08b] Deutsche Bundesbank: *SEPA - Der einheitliche Euro-Zahlungsverkehrsraum*. `http://www.bundesbank.de/zahlungsverkehr/zahlungsverkehr_sepa.php`, verfügbar am 04.06.2008.

[Deu08c] Deutsche Bundesbank: *SEPA - Gläubiger-Identifikationsnummer*. `http://www.deutschebundesbank.de/zahlungsverkehr/zahlungsverkehr_sepa_identifikation.php`, verfügbar am 04.06.2008.

[Dyn08] Dynamicdrive: *Lastschrift - Lastschriftverfahren*. `http://www.finanz-lexikon.de/lastschrift%20-%20lastschriftverfahren_174.html`, verfügbar am 19.05.2008.

[EBA08a] EBA Clearing: *STEP2 Multi Purpose Direct Debits - Core Service and B2B-Service - Functional Description*, Oktober 2008.

[EBA08b] EBA Clearing: *STEP2 Multi Purpose Direct Debits - Core Service and B2B-Service - Interface Specifications*, Oktober 2008.

[Eur08a] European Payments Council: *SEPA Business to Business Direct Debit Scheme Rulebook v1.1*, Juni 2008.

[Eur08b] European Payments Council: *SEPA Core Direct Debit Scheme Rulebook v3.1*, Juni 2008.

[Eur09] European Payments Council: *SEPA Core Direct Debit Scheme Customer-To-Bank Implementation Guidelines v3.2*, Januar 2009.

[Exo09] ExoLab Group, Intalio Inc., and Contributors: *The Castor Project - An Open Source data binding framework for Java*. `http://www.castor.org/`, verfügbar am 28.03.2009.

[Göß04] Gössmann, Wolfgang; Weber, Beatrixe: *Recht des Zahlungsverkehrs*. Erich Schmidt Verlag, 4. Auflage, 2004.

[IBI08] IBI Research an der Universität Regensburg: *Der E-Commerce Leitfaden*. `http://www.ecommerce-leitfaden.de/keine-chance-ohne-risiko.html`, verfügbar am 25.07.2008.

[Int09] International Organization for Standardization: *ISO 20022 Universal financial industry message scheme - Catalogue of ISO 20022 messages*. `http://www.iso20022.org/catalogue_of_unifi_messages.page`, verfügbar am 12.02.2009.

[jav09a] java.sun.com: *Sun Developer Network - Tutorials and Code Camps - Connection Pooling*. `http://java.sun.com/developer/onlineTraining/Programming/JDCBook/conpool.html`, verfügbar am 12.03.2009.

[jav09b] java.sun.com: *Sun Developer Network - Metro Web Services Reference*. `http://java.sun.com/webservices/reference`, verfügbar am 23.02.2009.

[Lei08] LeinertConsult: *Lastschrift Zahlung im Internet-Shop*. `http://www.leinert.com/lastschrift/index.html`, verfügbar am 19.05.2008.

[Mar02] Maruyama, Hiroshi; Tamura, Kent; Uramoto Naohiko; ..: *XML and Java - Developing Web Applications*. Addison-Wesley, Second Edition, 2002.

[Mit08] Mittelstand.de: *Deutschland ist Lastschrift-Europameister*. `http://www.mittelstanddirekt.de/c116/m187/um255/d4156/default.html?aktseite=2`, verfügbar am 22.09.2008.

[Nie06] Niedermeier, Stefan; Scholz, Michael: *Java und XML - Grundlagen, Einsatz, Referenz*. Galileo Computing, 1. Auflage, 2006.

[SN08] SEPA-News: *SEPA - News*. `http://www.sepa-news.de/`, verfügbar am 27.05.2008.

[sof09] software-engineering - www.software-kompetenz.de: *XML-Parser*. `http://www.software-kompetenz.de/?12798`, verfügbar am 16.02.2009.

[Ste09] Steiner, David: *Ressourcen-Optimierung von speicher- und kommunikationsintensiven Java-Anwendungen*. Diplomarbeit, Hochschule MIttweida (FH), Januar 2009.

[VNR08] VNR Verlag für die Deutsche Wirtschaft AG: *Das Lastschriftverfahren - bequem, aber manchmal riskant*. `http://www.vnr.de/vnr/nonprofit/verein/praxistipp_15010.html`, verfügbar am 19.05.2008.

[Wes08] WestLB AG: *Infopaket Formulare*. http://www.westlb.de/cms/sitecontent/westlb/payments/en/sepa_/infopaket_formate.standard.gid-N2FkNDZmMzU4OWFmYTIyMWM3N2Q2N2Q0YmU1NmI0OGU_.html, verfügbar am 13.11.2008.

[Zah08] Zahlungsverkehrsfragen.de: *Lastschriftinkasso*. http://www.zahlungsverkehrsfragen.de/lsinkasso.html, verfügbar am 19.05.2008.

[Zen08] Zentraler Kreditausschuss: *Zahlungsverkehr - SEPA*. http://www.zka-online.de/zka/zahlungsverkehr.html, verfügbar am 13.11.2008.

[Zim08] Zimmer, Frank: *Vorlesung Web-Programmierung II*, Januar 2008.

Selbstständigkeitserklärung

Ich erkläre, dass ich die vorliegende Arbeit selbstständig und nur unter der Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Mirko Haustein Chemnitz, den 29.04.2009