Studienarbeit

"Versionierung von Materialien im WAM-System"

am Fachbereich Informatik, Arbeitsbereich Softwaretechnik der Universität Hamburg

Betreuer: Prof. Dr. H. Züllighoven

Autoren:
Joachim Käßer, Matrikelnummer: 4174755
Stephan Schulmeister, Matrikelnummer: 4226451

August 2001

Danksagung

Wir danken allen, die uns bei der Erstellung dieser Studienarbeit geholfen habe. Insbesondere unserem Betreuer Prof. Dr. Heinz Züllighoven, sowie unserem Berater Stefan Roock.

Inhaltsverzeichnis

Kapitel 1: Einleitung

1.1. Motivation

Anwendungssoftware durchlebt, wie jede andere Software auch, eine gewisse Entwicklung im Verlaufe ihres Einsatzes. Die Software wird gewartet, fehlerbereinigt und weiterentwickelt. Die Softwareentwicklung folgt einem zyklischen Entwicklungsansatz [Zül1998], in dem das Softwaresystem sich ständig in einer Phase des Überganges von einer Version zu einer neuen Version befindet. Der Zyklus der Softwareentwicklung besteht aus Analyse, Modellierung und Bewertung.
In früheren Jahren gab es verschiedene Beschreibungsmodelle, wie z.B. das Wasserfallmodell oder das Spiralmodell, die dem zyklischen, Anwendungsorientierten Softwareentwicklungsprozeß nicht Rechnung getragen haben. Als Konsequenz davon hat sich das idealisierte evolutionäre Entwicklungsmodell[Zül1998] herausgestellt, welches den zyklischen Ansatz vollständig beschreibt. Wichtiger Punkt hierbei ist unter anderem die permanent mögliche Rückkopplung zwischen Analyse, Modellierung und Bewertung, die eine entsprechende Qualitätssicherung erst möglich macht.
In diesem Entwicklungsprozeß wird einer Problematik häufig nicht genügend Aufmerksamkeit geschenkt. Daten, welche schon während des Entwicklungsprozesses persistent gemacht wurden oder während einer früheren Phase im Produktionsbetrieb anfielen, müssen dem Entwicklungsstand angepaßt werden.
Die Designentscheidung für eine Klasse mit persistenten Objekten, die zu einer gewissen Phase im Entwurfsprozeß vom Anwendungsentwickler getroffen wurde, ist in der Regel nicht endgültig und unumstößlich. So kann die Anwendungssituation es z.B. in einem einfachen Fall erfordern, daß eine Klasse in einer neuen Programmversion ein zusätzliches Attribut erhält oder daß ein Attribut wegfällt. In komplizierteren Fällen ändern sich die Beziehungen von Objektgeflechten dieser Klassen untereinander.

1.2. Problemstellung

All dies wäre kein Problem, wenn die Objekte die Laufzeit eines Programmes nicht überdauern. Die meisten Anwendungen erfordern jedoch eine dauerhafte, persistente Speicherung dieser Objekte. Unabhängig vom Persistenzmedium ergibt sich nun das Problem, daß diese von einer Version eines Programmes in eine neue Version übernommen werden müssen. Definiert die neue Softwareversion für eine bestimmte Klasse ein gegenüber der Vorversion geändertes Schema, dann ist eine explizite Konvertierung der persistenten Objekte erforderlich. Die Konvertierung kann dabei etwa „nach Bedarf“ für einzelne Klassenobjekte geschehen oder es werden alle auf ein-

mal zu einem Zeitpunkt konvertiert.
Komplizierter wird das Problem, wenn unterschiedliche Programmteile auf einem dieser Objekte arbeiten können und jeder Teil eine bestimmte Version des Objektes erwartet bzw. nur mit einer bestimmten Version arbeiten kann.

1.3. Anwendungsfachliche Beispiele

In einer Firma wird eine Office-Management-Software eingesetzt, die unter anderem Kundendaten verwalten kann. Dazu gibt es im Quellcode eine Klasse „Firma“, die mit entsprechenden Attributen ausgestattet ist.

- Integer intID
- String strFirmenbezeichnung
- String strVorname
- String strNachname
- String strTelefon

Im Laufe der Softwareentwicklung stellt sich nun heraus, daß es ungünstig ist, den Ansprechpartner mit in der Klasse Firma als Attribut zu speichern. Die Klasse „Ansprechpartner“ wird eingesetzt, und die Klasse „Firma“ so umgeschrieben, daß die Klasse Ansprechpartner als eigenständige Klasse neben der Klasse „Firma“ existiert. Diese Änderung bedingt natürlich das Ändern bzw. Umschreiben (migrieren) der schon persistent abgelegten Informationen. Die Migration eines solchen Datenbestandes kann, sofern größere Bestände existieren, sehr aufwendig werden und muß zumeist auch am Einsatzort der Software selbst durchgeführt werden, um einen längeren Ausfall der Software zu vermeiden.
Weiterhin müssen die Daten so migriert werden, daß keine Inkonsistenzen in den Daten auftreten und keine Invarianten von Klassen verletzt werden. Eine Klasseninvariante ist eine Zusicherung, die eine generelle Regel für die Konsistenz einer Klasse ausdrückt, welche für alle Exemplare dieser Klasse gelten muß [Mey1997].
Entsprechend einer Invariante kann es in einigen Fällen nötig sein, bestimmte Klassenexemplare zwingend vor anderen zu migrieren. So z.B. wenn eine Klasse Attribute anderer Klassen referenziert bzw. verrechnet, die aber erst nach einer Migration in die neue Version initialisiert sind.

1.4. Zielsetzung

Ziel dieser Studienarbeit ist die Realisierung einer Versionskontrolle und der damit verbundenen Migration von persistenten Objekten.

Unsere Studienarbeit konzentriert sich dabei auf die Entwicklung einer Versionierung im JWAM Rahmenwerk und bedient sich dabei deren Registratur [www.jwam.de]. Wird am Registrator ein Exemplar einer Klasse angefordert, welches nicht der aktuell verwendeten Klassenbeschreibung entspricht, meldet der Registrator einen Versionskonflikt und versucht diesen durch eine automatische Datenmigration zu beseitigen.

Falls Abhängigkeiten zwischen den Klassen der gespeicherten Exemplare bestehen, kann eine automatische Migration evtl. nicht erfolgen. Es besteht die Möglichkeit von Inkonsistenzen bei der Migration. In diesem Fall stellen wir eine Funktion zur Massenkonvertierung zur Verfügung, die es ermöglicht untereinander abhängige Klassenexemplare „en masse“ zu migrieren. Die Massenkonvertierung ermöglicht es, die alle Objekte in einer bestimmten Reihenfolge zu migrieren, und stellt dadurch sicher, daß keine Inkonsistenzen bei der Migration auftreten.

1.5. Gliederung der Studienarbeit

Zunächst erfolgt eine Einführung in den Begriff der Schemaevolution und Versionierung. Die Begriffe Schemaevolution und Versionierung haben in verschiedene Kontexten unterschiedliche Bedeutungen. Wir führen die Begriffe aus der Sichtweise eines Anwendungsentwicklers ein.

Im Anschluß folgt ein Kapitel über die Konzepte in JWAM: Material und Registratur. Dieses Kapitel ermöglicht dem Leser ein Überblick über deren Funktionsweise und Arbeitsweise.

Nachdem die Grundlagen für die Versionierung im Rahmen dieser Arbeit eingeführt wurden, erstellen wir im Anschluß das Konzept der Versionierung. Hier wird evaluiert, welche Funktionalität vorhanden sein muß, um eine Versionierung zu ermöglichen.

Im Anschluß folgt die Implementation der Versionierung. Sie stützt sich auf die Entwicklung eines Prototypen, da eine vollständige Entwicklung den Rahmen dieser Studienarbeit sprengen würde.

Das Konzept und die Implementation der Versionierung sind so angelegt, daß sie im weiteren Entwicklungsprozeß erweitert und ergänzt werden können. Ein Ausblick auf diese Möglichkeiten wird dem Leser im letzten Kapitel unserer Arbeit gegeben.

Kapitel 2: Versionierung und Schemaevolution

2.1. Versionierung

Wann immer ein Begriff in der Softwaretechnik genauer untersucht werden soll, wird es sich lohnen, Analogien zu entsprechenden Begriffen außerhalb der Softwaretechnik zu ziehen. So war es z.B. in einer Arbeit [Hav1999], in der es um die persistente Speicherung von Materialien ging, sinnvoll, die Handlungen eines Registrators in einer Registratur zu betrachten. Mithilfe dieser Untersuchung von Abläufen aus der "realen Welt" konnten wertvollen Rückschlüsse für den Entwurf eines Persistenzsystems gemacht werden .

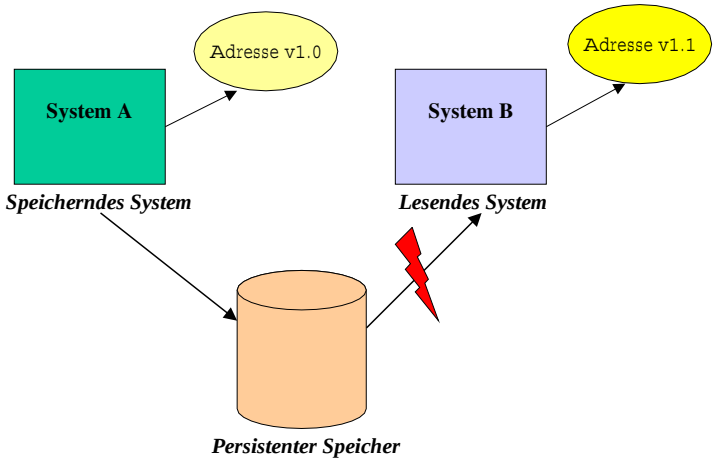
Wie sieht es nun für den Begriff "Versionierung" aus? Auch hier zeigte sich nach genauerer Betrachtung, daß wir die Welt der Registratur nicht zu verlassen brauchen. Wir können uns vorstellen, daß es ebenfalls zu den Aufgaben eines Registrators gehört, den zeitlichen Verlauf, also die Entwicklungshistorie, von Dokumenten zu verwalten. Ausleiher, sprich Personen, die sich Dokumente aus einem Dokumentenregister entleihen, interessieren sich nicht nur für den aktuellen, gegenwärtigen Zustand eines Dokumentes, sondern sie interessieren sich mitunter ebenso für vergangene Zustände des selben Dokumentes. Die Entwicklungsgeschichte eines Dokumentes bzw. Materials verläuft in den seltensten Fällen geradlinig und ist nicht nur durch Informationsgewinn charakterisiert. Vielmehr wird es so sein, daß das Dokument eine Reihe von Überarbeitungen durchläuft, bei denen Teile des Dokumentes wieder verworfen werden. Versionierung und Dokumentenhistorie dienen hier also nicht zuletzt auch zur Vermeidung eines Informationsverlustes, welcher z.B. durch vorschnelles Verwerfen von Teilen des Dokumentes entstehen kann. Die Wahrscheinlichkeit für einen Verlust von Information kann vom Registrator über die Granularität der Versionierung gesteuert werden. Je mehr Versionen eines Dokumentes im Entwicklungsverlauf festgehalten werden, d.h. je feiner die Granularität der Versionierung ist, desto niedriger ist auch die Gefahr des Informationsverlustes.
Der Gegenstand der Versionierung, hier als "Dokument" oder "Material" bezeichnet, ist bewußt abstrakt gehalten. Hinter "Dokument" kann sich z.B. ein Buch verbergen, eine Anwendungssoftware, ein Objekt aus einem elektronischen Hypertextsystem, eine Klasse in einem Framework oder ein Datensatz aus einem Datenbanksystem. Für die Versionierung spielt dies zunächst einmal keine Rolle, wobei die Versionierung eines Buches (Ausgabe) sicherlich weniger komplex ist als die Versionierung von Objekten aus einem elektronischen Hypertextsystem, in dem sich Versionen baumartig verzweigen können.

In Datenbanken werden häufig Kopien zur Synchronisation nebenläufiger Zugriffe auf Datensätze eingesetzt (optimistisches Locking). Die Erzeugung dieser Kopien (also einer neuen Version) wird dem Benutzer nicht bekannt gemacht. Ebenso ist die Lebensdauer dieser Version nur temporär, d.h. sie existiert nur so lange, wie sie zur Vermeidung von "Deadlocks" zwischen Schreibern gebraucht wird.
Im Kontext unserer Studienarbeit wollen wir uns nicht weiter mit dieser Form der Versionierung beschäftigen. Wir interessieren uns ausschließlich für dauerhafte (persistente) Versionen, die dem Benutzer (Anwendungsentwickler) bekannt sind.
Dem aufmerksamen Leser wird sicher nicht entgangen sein, daß die Diskussion sich an dieser Stelle bereits von der fachlichen Ebene wegbewegt hin zur technischen Ebene. Dies ist auch nicht unerwünscht, denn das Thema unserer Studienarbeit lautet "Versionierung von Materialien im WAM-System". Das WAM-System, insbesondere seine Implementierung als Framework in Java, wird in Kapitel "Einführung zu JWAM" auf Seite 19 beschrieben.
Das Ziel, welches wir in unserer Studienarbeit verfolgen, besteht nicht nur darin, allgemeine Mechanismen zur Versionierung von Materialien in das JWAM-Framework zu integrieren. Viel mehr ist es das Ziel, diese Mechanismen dazu zu benutzen dem Anwendungsentwickler Werkzeuge an die Hand zu geben, mit denen er persistente Objekte (Materialien) von einer alten in eine neue Version migrieren kann.

2.2. Schemaevolution

Schemaevolution, Versionierung und Migration sind eng miteinander verknüpft. Der Begriff „Schema" hat seinen Ursprung in der Welt der relationalen Datenbanken [Mey1997]. Auch in der objektorientierten (Datenbank-) Welt spricht man vom „Schema", nur ist der Schemabegriff hier nicht so starr wie in relationalen Systemen. Die Schemaevolutionsmechanismen eines modernen objektorientierten Datenbanksystems (OODBS) gehen weit über die Mechanismen hinaus, die relationale Systeme bieten. In einem OODBS wird das Schema häufig durch Objekte repräsentiert. Dabei sind die Repräsentationsobjekte nicht direkt änderbar. Updates auf diese Objekte werden in der Regel über eine spezielle Sprache, die Data-Definition-Language (DDL) durchgeführt. Zu einem Relationenschema können i.d.R. lediglich neue Attribute hinzugefügt sowie komplette Relationen neu erzeugt oder gelöscht werden [Heu1997]. Demgegenüber bieten OODBs Mechanismen an, die dem komplexen Problem der Schemaevolution besser Rechnung tragen. Allerdings erreicht man mit diesen Mechanismen bei weitem noch keine Universallösung, sie erleichtern aber die Weiterentwicklung des Schemas sowie die Migration persistenter Objekte.
Rundensteiner et.al. [Run1994a], [Run1994b] stellen das Problem der Aktualisierung des Datenbankschemas in einer Studie über ein Medizinisches Management System

heraus. Dort hat sich im Verlaufe des Projektes über einen Zeitraum von nur 18 Monaten die Anzahl an Relationen um 139% und die Anzahl der Attribute sogar um 274% erhöht. Eine weitere Untersuchung hat ergeben, das Datenbankmodelle in den seltensten Fällen wirklich stabil sind. Im Schnitt ändern sich mit der Zeit ca. 59% der Attribute in typischen Datenbank-Projekten. Rundensteiner et.al. betonen, daß Schemaevolution ein essentielles Feature von OODBS, wie z.B. CAD/CAM-Anwendungen, VLSI-Entwurf oder Büroinformationssystemen sei. Dies gilt insbesondere für den evolutionären Charakter von objektorientierten Datenbanksystemen, die im Gegensatz zu herkömmlichen hierarchischen und relationalen DB-Systemen als offenes System betrachtet werden können. Das Schema solcher Systeme ist von vornherein darauf ausgelegt, sich weiterzuentwickeln. Rundensteiner et.al. betonen ebenfalls die enge Verzahnung zwischen Schemaevolution und Migration, die zu einer Schemaversionierung (dort basierend auf Datenbank-Views) führen muß.
Meyer [Mey1997] definiert Schemaevolution folgendermaßen: *„Wir sprechen von Schemaevolution, wenn mindestens eine Klasse eines Systems, das persistente Objekte lesen will (Lesendes System), sich von seinem Gegenstück in dem System unterscheidet, in welchem diese Objekte gespeichert wurden (Speicherndes System).“*
Die Konsequenz von Schemaevolution sind Fehler (Versionskonflikte) beim Lesen persistenter Objekte.



Versionskonflikt nach Schemaänderung

2.2.1. Taxonomie der Schemaevolution

Banerjee [Ban1987] war einer der ersten, der eine Taxonomie zur Änderung eines ob-

jektorientierten Schemas vorgeschlagen hat:

1	Änderungen am Inhalt eines Knotens (Klasse)
1.1	Änderungen an einem Attribut
1.1.1	Hinzufügen eines Attributes zu einer Klasse
1.1.2	Entfernen eines Attributes von einer Klasse
1.1.3	Ändern des Namens eines Attributes
1.1.4	Änderung der Vererbungsbeziehung (Basisklasse) eines Attributes
1.1.5	Auflösen eines zusammengesetzten Objektes (Entfernen eines aggregierten Objektes)
1.2	Änderungen an einer Methode
1.2.1	Hinzufügen einer Methode zu einer Klasse
1.2.2	Entfernen einer Methode von einer Klasse
1.2.3	Änderung des Namens einer Methode
1.2.4	Änderung des Methodenrumpfes
1.2.5	Ändern der Vererbungsbeziehung (Basisklasse) einer Methode
1.2.6	Änderungen des Wertebereichs einer Methode
1.2.7	Verwerfen des Wertebereichs einer Methode

2	Änderungen einer Kante (Vererbungsbeziehung)
2.1	Klasse S wird eine Superklasse der Klasse C
2.2	Klasse S wird aus der Liste der Superklassen von C entfernt
2.3	Änderung der Reihenfolge der Superklassen von C
3	Änderung eines Knotens (Klasse)
3.1	Hinzufügen einer Klasse
3.2	Entfernen einer Klasse
3.3	Ändern des Namens einer Klasse

Die am Markt verfügbaren kommerziellen OODBS implementieren lediglich eine Untermenge dieser Taxonomie für ihre Schemaevolutionsmechanismen [Run1994a] mit Ausnahme der Systeme ORION / ITASCA.
Dabei sind die Mechanismen dieser OODBS keineswegs einheitlich und in der Leistung sehr unterschiedlich. Heuer [Heu1997] klassifiziert Schemaevolutionsmechanismen nach folgenden Gesichtspunkten:

- Welche Änderungen aus der oben genannten Taxonomie sind erlaubt und welche nicht?
- Können Änderungen zur Laufzeit durchgeführt werden oder muß die Anwendung nach einer Schemänderung neu übersetzt werden?

- Werden persistente Objekte des alten Schemas automatisch geändert?
- Werden Programme und Methoden, welche das alte Schema benutzen, automatisch geändert?

Um die letzten beiden Punkte zu realisieren, schlägt Heuer vier Möglichkeiten vor:

1. Schemaänderungen sind nur dann erlaubt, wenn noch keine persistenten Objekte des Schemas vorliegen.
2. Die Objektmenge wird nach einer Schemaänderung sofort aktualisiert.
3. Die Objektmenge wird erst dann aktualisiert, wenn wieder auf die Menge zugegriffen wird (verzögerte Aktualisierung).
4. Das alte Schema und die alte Objektmenge bleiben erhalten und werden bei Bedarf auf das neue Schema abgebildet. Dies entspricht einer Versionsführung.

Meyer [Mey1997] bezeichnet die in den ersten beiden Punkten genannten Ansätze als naiv. Der Punkt 1 ist eigentlich nur dann geeignet, wenn es sich um ein Neusystem handelt, zu dem noch keine persistenten Objekte existieren. Dies trifft auf die meisten Systeme nicht zu. In diesen Systemen werden bereits Objekte existieren, die migriert werden müssen. Meyer spricht von Schema Revolution statt Evolution, wenn diese Objekte aufgegeben werden müssten. Tatsächlich bieten die meisten Systeme so wenig Unterstützung für Schemaevolution, daß sie in diese Kategorie fallen. In solchen Systemen muß für eine korrekte Objektmigration ein neuer Typ eingefügt werden, in den dann die Daten aus dem alten Typ kopiert werden.
Die Lösung, die im zweiten Punkt genannt wurde, hält Meyer ebenfalls nicht für praktikabel. Eine sofortige Aktualisierung der gesamten Objektmenge ist gleichbedeutend mit einer Massenkonvertierung von alten Objekten, bei der ein Migrationspfad vom alten Format zum neuen bereitgestellt werden muß. Dies ist sicher nicht geeignet für Systeme mit sehr großem persistentem Speicher und für Systeme, die permanent verfügbar sein müssen (z.B. Mehrbenutzersysteme in Banken und Versicherungen). Diese Lösung bezeichnet Meyer als „en masse“ – Konvertierung. Wir werden diese Konvertierungsart im weiteren Verlauf unserer Arbeit als „at once“ – Konvertierung bezeichnen.
Meyer fordert eine Lösung, Objekte „on the fly“ zu migrieren, also zu dem Zeitpunkt, zu dem auf die Objekte zugegriffen wird. Dies entspricht dem dritten Punkt. Hier koexistieren also Objekte des alten und des neuen Schemas im Persistenzraum. Die Version des Objektes muß in einem solchen System besonders kenntlich gemacht werden, um Versionskonflikte aufzudecken. Diese Aktualisierungform wird in der Literatur häufig auch als „lazy update“ oder als „lazy evaluation on schema changes“ [Run1994a] bezeichnet. Wir werden diese Form als Migration „on demand“ bezeich-

nen.
Die vierte Lösung ist die komplexeste. Man bezeichnet diese Lösung auch als „Schemamapping“. Diese Lösung beruht, ähnlich wie die Lösung im dritten Punkt, auf einer verzögerten Aktualisierung der Objekte. Hier ist eine Zuordnung des aktuellen Schemas zu allen vorangehenden Schemata des Typs erforderlich. Es müssen alle Schemata paarweise miteinander verknüpft werden. Die Benutzer sehen dasselbe Objekt durch verschiedene Versionen des Schemas. Diese Lösung ist so kompliziert und wahrscheinlich auch ineffizient, daß sie bisher noch in keinem uns bekannten System implementiert wurde.

2.2.2. Klassifikation von Schemaänderungen

Alle Schemaänderungen lassen sich in die weiter oben eingeführte Taxonomie einordnen. Dabei müssen wir für jeden Punkt der Taxonomie überlegen, wie schwerwiegend die entsprechende Schemaänderung ist. Es genügt hier eine Zweiteilung in einfache und komplexe Schemaänderungen vorzunehmen. Der Grad der Schwergewichtigkeit läßt sich danach bemessen, wie aufwendig eine spätere, durch diese Schemaänderung veranlaßte, Migration der Objekte ist. Die Taxonomie reicht für eine Bemessung des Grades alleine nicht aus. Wichtiger ist es, zu berücksichtigen, ob das zu migrierende Objekt Teil eines Objektgeflechtes ist oder nicht.

Einfache Schemaänderungen

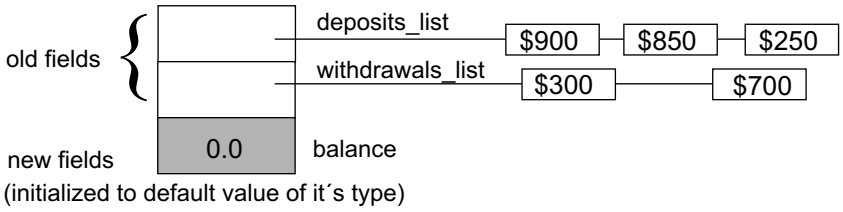
Wir bezeichnen Schemaänderungen immer dann als einfach, wenn sie Klassen betreffen, die nicht in einem komplexen Objektgeflecht auftreten. Das bedeutet, daß Änderungen keine Änderungen anderer Klassen voraussetzen bzw. nach sich ziehen. In der Praxis ist dies ein sehr seltener Fall.

Hinzufügen, Entfernen und Ändern von Attributen

Attribute können nur dann sorglos entfernt werden, wenn feststeht, daß keine Klasse existiert, die dieses Attribut direkt, oder über eine Zugriffsmethode referenziert. Es ist dabei egal, ob es sich bei der referenzierenden Klasse um eine Klasse handelt, die die Schnittstelle der referenzierten Klasse benutzt (Klientenklasse) oder ob sie die referenzierte Klasse vollständig aggregiert (z.B. als “inner class”). Die Migration erweist sich in diesem Fall als trivial.

Schnell möchte man die Meinung vertreten, auch beim Hinzufügen neuer Attribute handele es sich um einen Trivialfall. Dies ist aber nicht so. Meyer [Mey1997] führt hierfür ein Beispiel an: Eine Klasse “Konto” besitzt die Attribute “Liste der Einzahlun-

gen" und "Liste der Abhebungen". Der Kontostand dieses Kontos wird ermittelt, indem die Summe der Beträge der Abhebungen von der Summe der Beträge der Einzahlungen abgezogen wird. Die Implementation der Klasse "Konto" soll nun verbessert werden. Die neue Version der Klasse "Konto" erhält zusätzlich ein Attribut "Kontostand". Abfragen des Kontostandes können damit effizienter durchgeführt werden, da das Attribut nach jeder Transaktion auf dem Konto aktualisiert wird. Es stellt sich die Frage, wie das neue Attribut bei einer Migration von Objekten der alten Klasse "Konto" initialisiert wird. Der intuitive Ansatz, alle Attribute mit "0" zu initialisieren ist falsch. Dies ist leicht einzusehen, da ein Nullwert das Objekt in einen inkonsistenten Zustand versetzt. In diesem Fall korrespondiert der Wert in "Kontostand" nicht mehr mit dem Wert, der aus den Listen für Einzahlungen und Abhebungen errechnet werden kann.
Nachfolgende Abbildung zeigt ein Konto mit einer Liste von Einzahlungen und einer Liste von Abhebungen, welches mit einem inkonsistenten Wert für den Kontostand initialisiert wurde.



Meyer bietet eine Lösung an, die auf einem automatischen Fehlerbehandlungsmechanismus beruht. In diesem Beispiel ist dies eine Routine, welche eine vererbte Fehlerbehandlungsroutine überschreibt. Die überschriebene Methode könnte z.B. lauten:

```
void correct_mismatch()
{
    balance := deposits_list.total() - withdrawals_list.total();
};
```

Diese Codezeile stellt eine fachlich korrekte Initialisierung des Attributs "balance" dar. In dem Model nach Meyer ist es falsch, die Fehlerbehandlungsroutine nicht zu überschreiben. Dadurch wird sichergestellt, daß der Programmierer der neuen Version eine korrekte Initialisierung bereitstellt. In vielen Fällen reicht die Initialisierung mit einem "Null"-Wert aus. Es ist aber trotzdem nicht sinnvoll, diese Initialisierung in der Basisklasse durchzuführen, denn dadurch würden viele fehlerhafte Initialisierungen unentdeckt bleiben. Im Kapitel über die Implementierung des Versionierungsframe-

works werden wir eine andere Lösung vorschlagen, die nicht auf materialeigenen Fehlerbehandlungsroutinen basiert. Hier geschieht die Fehlererkennung in einer eigenen Instanz, der Erkennungsinstanz für Versionskonflikte. Die Bekanntmachung und Beseitigung des Versionskonfliktes wird von anderen Instanzen übernommen, die wir in dem Kapitel ebenfalls noch genauer beschreiben werden.

Unter Attributänderungen verstehen wir Änderungen des Namens oder des Typs des Attributes. Beide Fälle können dadurch behandelt werden, indem die Änderungsoperation in eine Lösch- und eine Hinzufügeoperation aufgespalten wird. Bei Typeänderungen muß zusätzlich eine Transformation auf den neuen Typ durchgeführt werden. Namensänderungen können einfach durch eine generische Kopiermethode abgehandelt werden.

Hinzufügen, Entfernen und Ändern von Methoden zu Klassen

Relationale Datenbanksysteme bieten häufig sogenannte “Stored Procedures” an. Dies sind Routinen, die zusammen mit den Daten in der Datenbank gespeichert werden und einen optimierten Zugriff auf diese Daten ermöglichen. Mit dem Einsatz von “Stored Procedures” kann der Datenbankbenutzer somit im Einzelfall einen Performancegewinn erzielen gegenüber Methodenaufrufen, die auf der Clientseite ablaufen.
Die Methoden von Klassen aus objektorientierten Systemen werden auf Persistenzseite in der Regel nicht mit gespeichert. Somit sollte daß Hinzufügen, Entfernen und Ändern von Methoden für das Migrationsproblem nicht ausschlaggebend sein. Leider ist das Problem doch nicht ganz so einfach beschaffen. Das Verändern von Methoden kann sehr wohl Migrationsprobleme erzeugen. Dies geschieht immer dann, wenn die Methodenänderung zu einer Änderung des Typs der Klasse führt. Ein Beispiel soll dies verdeutlichen:
Eine Klasse “KFZ” besitzt als Attribut eine Angabe der gesetzlich vorgeschriebenen Abgaswerte. Das Attribut “Abgaswert” ist als einfacher Integerwert implementiert. Die Klasse “KFZ” hat zusätzlich eine Methode “setzeAbgaswert(int abgaswert)”, die diesen Wert schreibt. Innerhalb des Methodenrumpfes der Klasse wird eine Überprüfung des Wertebereiches durchgeführt (Vorbedingung). Die Vorbedingung stellt sicher, daß sich der Wert innerhalb einer gültigen Wertemenge befindet, die durch eine Abgasnorm definiert ist. So liegt der Wertebereich z.B. zwischen 100 und 500. Nach Anpassung der Abgasnorm auf den Bereich 100 - 300 muß diese Klasse umgeschrieben werden. Die Vorbedingung innerhalb der neuen Version der Klasse “KFZ” ist jetzt auf den geänderten Wertebereich angepaßt. Der Typ des zugehörigen Attributes ändert sich nicht. Es ergibt sich jetzt ein Problem bei der Migration alter Objekte, deren Abgaswerte außerhalb dieses Bereichs liegen. Eine Migration der Objekte ist erforderlich. Ob die Abgaswerte alter KFZ tatsächlich änderbar sind, soll hier nicht von Bedeutung

sein. Ausschlaggebend ist lediglich, daß die Objekte durch die Anpassung des Wertebereichs der Methode migriert werden müssen. Das Beispiel soll verdeutlichen, daß auch Änderungen von Klassen, in denen nur Methoden und keine Attribute geändert werden, eine Migration erfordern können.
Methodenänderungen bzw. das Hinzufügen oder Entfernen von Methoden sind also immer dann ausschlaggebend für die Migration, wenn die Änderung einen Einfluß auf die Invariante der Klasse hat.

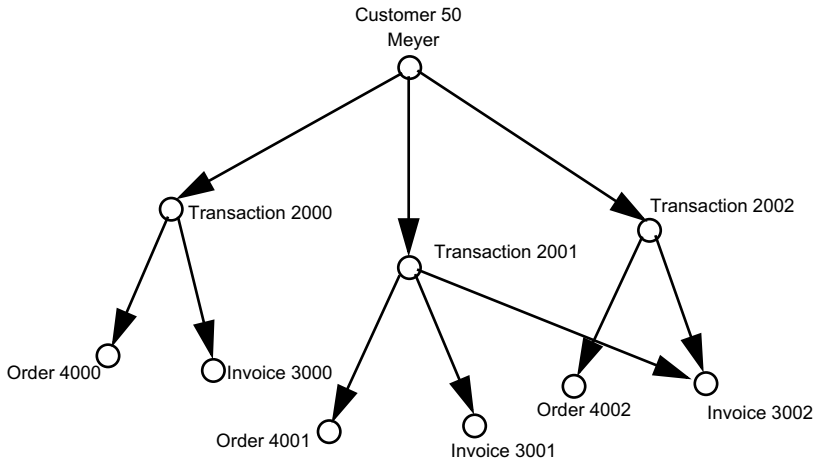
Komplexe Schemaänderungen

Werden Klassen von anderen Klassen referenziert, so spricht man von Klassen- bzw. von Objektgeflechten. Eine Änderung an einer zu einem Geflecht gehörigen Klasse kann so das Ändern von weiteren Klassen in diesem Geflecht voraussetzen. Wir werden dies weiter unten an einem Beispiel demonstrieren.

Die Reihenfolge der Änderungen, bzw. der Datenmigration der Objekte kann dann nicht willkürlich geschehen, da so Inkonsistenzen auftreten können und ein Datenverlust nicht auszuschließen ist.

Um Datenbestände eines ganzen Objektgeflechtes zu migrieren, muß das Geflecht zunächst analysiert werden. Hierbei gilt es, die entsprechenden Abhängigkeiten der Objekte untereinander zu ermitteln. Es muß ein Migrationsalgorithmus entwickelt werden, der sicherstellt, daß eine Migration sequentiell erfolgen kann ohne daß es zu einem Datenverlust kommt.
Die folgende Graphik zeigt ein Objektgeflecht, in denen die Objekte Kunde, Auftrag und Rechnung miteinander in Beziehung stehen. Es handelt sich hierbei um einen azyklisch gerichteten Graphen, in dem Rechnungen ihre zugehörigen Aufträge und die Aufträge ihre Kunden referenzieren. Der Graph enthält eine Masche, da die Rechnung “invoice 3004” zwei verschiedene Aufträge referenziert.

Beispiel für ein Objektgeflecht



Zur Anschaulichkeit haben wir jedes Objekt mit einem Schlüsselwert versehen, um zu unterstreichen, daß es sich um Objekte mit eigenständiger Identität handelt. Wie die Referenzen technisch realisiert werden, ist hier zunächst nicht von Bedeutung.
Es stellt sich die Frage, ob die Existenz der Masche ein Problem für die Migration darstellt. Einen Datenverlust durch eine falsche Migrationsreihenfolge können wir vermeiden, indem wir z.B. die Referenzen fachlich implementieren. Eine fachliche Referenzierung könnte z.B. über die Kontonummer eines Objektes “Konto” hergestellt werden. Eine Migration wird in diesem Fall die Werte einer fachlichen Referenz nicht verändern, so daß auch ein bereits migriertes Objekt noch genauso referenziert werden kann, wie vor der Migration. Die Migrationsreihenfolge spielt aber dann eine Rolle, wenn auf Attribute Bezug genommen wird, die in einer neuen Version eines Objektes nicht mehr vorhanden sind oder verändert wurden. Die Reihenfolge wird somit durch fachliche Inhalte der Attribute bestimmt und nicht durch die Schlüsselreferenzen. Interessant hierbei ist, daß die fachliche Bezugnahme auf Attribute häufig entgegen der technischen Referenzierung durch die Schlüssel verläuft. In unserem Beispiel betrifft dies die Klassen “Transaction” und “Order”, bzw. “Invoice“. Der technische Schlüssel auf die “Transaction” wird in “Order“ bzw. “Invoice” gespeichert. Fachlich nimmt aber das “Transaction”-Objekt Bezug auf das Attribute der Klassen“Invoice” bzw. “Order“, um den noch ausstehenden Restbetrag des Auftrages zu berechnen (Siehe “Beispiele” auf Seite 52.).

Die Auflösung von Objektgeflechten zur Ermittlung von Abhängigkeiten funktioniert nur, wenn keine zyklischen, technischen Abhängigkeiten zwischen den Objekten bestehen. Bestehen solche Abhängigkeiten, dann kann das Geflechtschema nicht ohne weiteres in eine Baumstruktur überführt werden.

Wir betrachten den Fall von Objektzyklen hier nicht weiter. In vielen Fällen kann durch ein Redesign ein solcher Zyklus vermieden werden. Sollte ein Zyklus doch einmal unvermeidbar sein, z.B. im Fall von bidirektionalen Assoziationen zwischen Klassen, die fachlich begründet sind, kann der fachliche Zyklus evtl. durch Verschieben von Attributen in eine andere Klasse aufgebrochen werden. Ist auch dies nicht möglich, dann muß dafür Sorge getragen werden, daß die beteiligten Attribute in einer unteilbaren Operation migriert werden.

2.3. Versionskonflikte

Ein System, welches ein vernünftiges Maß an Mechanismen zur Schemaevolution unterstüzt, wird mindestens die Punkte 2 und 3, also „en masse“ – Konvertierung und „on the fly“ – Konvertierung realisieren. Das Versionierungssubframework, welches wir in Form eines Protoypen als Erweiterung des JWAM Frameworks realisiert haben, unterstützt diese Migrationsformen ebenfalls.
Folgen wir der Argumentation von Meyer weiter, dann ergeben sich bei der Migrationsform „on the fly“ drei wichtige Punkte:

1. Erkennung von Versionskonflikten
2. Benachrichtigung (des lesenden Systems)
3. Korrektur des Versionskonfliktes

Meyer betont, es sei nicht so wichtig, die Frage zu diskutieren, ob das lesende System die Berechtigung habe, die gespeicherten Objekte zu aktualisieren. Gehen wir davon aus, daß mehrere Systeme existieren, die auf den selben Datenbestand zugreifen. Ein einzelnes System wird die Berechtigung, die Objekte zu aktualisieren, in der Regel nicht besitzen. Es geht vielmehr darum, daß die Objekte, die von dem System eingelesen werden, konsistent mit ihrer eigenen Klassenbeschreibung sind.

Meyer beschreibt die drei Aufgaben folgendermaßen:

- Erkennung ist die Aufgabe, Versionskonflikte zum Zeitpunkt des Einlesens von obsoleten Objekten abzufangen.

- Benachrichtigung ist die Aufgabe, das lesende System auf den Versionskonflikt aufmerksam zu machen, damit nicht ein Objekt mit einem inkonsistentem Zustand weiterbearbeitet wird.
- Korrektur ist die Aufgabe des lesenden Systems, das Objekt transient wieder in einen konsistenten Zustand zu bringen und es zu einem korrekten Exemplar der neuen Version seiner eigenen Klasse zu machen. Die Aktualisierung der persistenten Daten ist ein anderes Problem.

2.3.1. Nominale und Strukturelle Erkennung

Die Erkennung von Versionskonflikten kann auf zwei verschiedene Arten geschehen: nominal oder strukturell [Mey1997]. Wir sprechen von nominaler Erkennung, wenn jede Version einer Klasse durch einen Versionsnamen identifiziert werden kann. Dieser Ansatz erfordert einen besonderen Registrationsmechanismus (ähnlich der Registratur im JWAM Framework), der wiederum in zwei Ausprägungen vorliegen kann:

- Ein Konfigurations-Managementsystem registriert Klassen und liefert den Versionsnamen. Alternativ kann der neue Versionsname manuell zugeordnet werden.
- Automatische Systeme, die zufällige Nummern vergeben (z.B. Microsofts OLE2 [Mey1997] oder Systeme, die dynamische IP-Adressen zuteilen), welche die Version der Klasse identifizieren. Die Gefahr der Vergabe von zwei gleichen Nummern ist dabei vernachlässigbar klein, wenn der Wertebereich der Nummern ausreichend groß gewählt ist.

Laut Meyer erfordern beide Varianten eine zentrale Registratur. Dies ist auch der Grund, warum der nominale Ansatz am besten zum JWAM-Framework paßt. Denn auch dort gibt es eine zentrale Registratur und einen Registrator, der für die Verwaltung des Persistenzraumes zuständig ist.
Der strukturelle Ansatz beruht auf der Einführung einer Klassenbeschreibung (class descriptor). Die Klassenbeschreibung wird mit jedem Objekt zusammen gespeichert (sinnvollerweise nur einmal für jede Version einer Klasse) und ermöglicht die einfache Aufdeckung von Versionskonflikten. Beim Auslesen aus dem Speicher wird dabei die Klassenbeschreibung jedes Objektes mit der neuen Klassenbeschreibung verglichen. Werden dabei Unterschiede festgestellt, dann handelt es sich um einen Versionskonflikt.
Meyer beschreibt vier Ansätze:

1. Nur der Klassenname wird verwendet.

2. Der gesamte Klassentext wird verwendet.
3. Der Klassenname und die Liste aller Attribute (inklusive Name und Typ der Attribute) werden verwendet.
4. Der Klassenname, die Attributliste und die Invariante der gesamten Klasse werden verwendet.

Punkt 1 ist sicher nicht geeignet, um Versionskonflikte zuverlässig aufzudecken. Zwei Klassen, die den gleichen Namen haben, müssen noch lange nicht die gleichen Attribute haben. Somit werden in dieser Variante viele Versionskonflikte unentdeckt bleiben, solange man nicht die Versionsnummer in den Klassennamen aufnimmt.
In Punkt 2 wird der gesamte Klassentext zur Bildung der Klassenbeschreibung herangezogen. Im Gegensatz zu Punkt 1 ist dies übertrieben, da in diesem Fall zu häufig Versionskonflikte gemeldet werden. Einige Änderungen am Klassentext sind unproblematisch, z.B. solche Änderungen, die eine Methode hinzufügen, aber die Liste der Attribute und sämtliche Invarianten unangetastet lassen.
Punkt 3 ist ein realistischer Ansatz und reicht in den meisten Fällen für eine zuverlässige Konflikterkennung aus, da es sehr unwahrscheinlich ist, daß zwei verschiedene Klassen denselben Namen und die gleichen Attribute besitzen.
Ist dennoch eine höhere Zuverlässigkeit gefordert, dann kann wie in Punkt 4 zusätzlich noch die Klasseninvariante in die Klassenbeschreibungen einfließen. Änderungen an Methoden, die die Semantik der Klasse nicht ändern, ändern auch die Klasseninvariante nicht und führen somit auch nicht zu einem Versionskonflikt. Im Umkehrschluß gilt dann aber auch, daß Methodenänderungen, die eine Änderung der Invariante nach sich ziehen, ebenso zuverlässig zu einer Erkennung des Versionskonfliktes führen.

Kapitel 3: Einführung zu JWAM

Das Kürzel JWAM steht für Java-Werkzeug-Automat-Material. JWAM ist ein an der Universität Hamburg entwickeltes Rahmenwerk für Applikationsentwickler, welches versucht, die realen Gegebenheiten einer Arbeitsumgebung auf Klassenstrukturen abzubilden [ObjSpec1999].
JWAM ermöglicht dadurch sehr effiziente Programmierung von Anwendungen für klassische Arbeitsumgebungen und für Webanwendungen.
Dinge werden in JWAM zu Automaten, Materialien und Werkzeugen klassifiziert. Ein Werkzeug ist immer ein „Tool“, um ein oder mehrere Materialien zu bearbeiten. Diese Materialien können persistent gemacht werden.

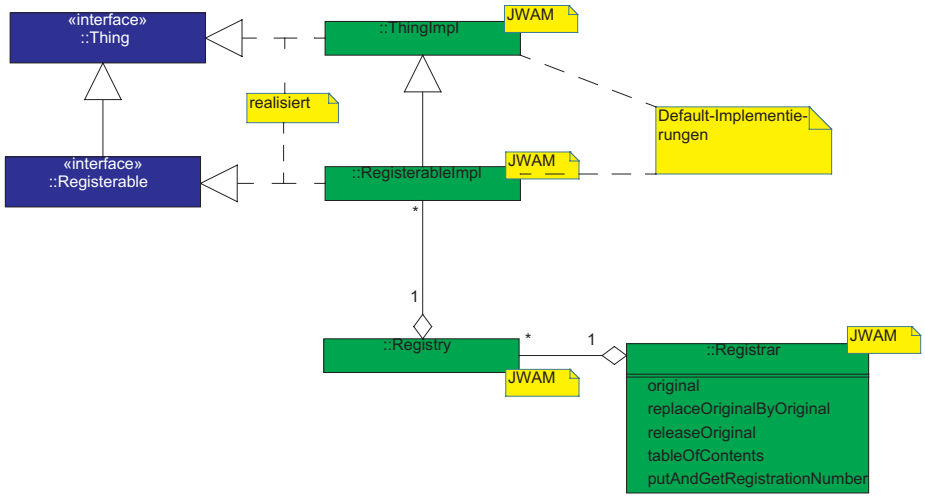
Wir führen ein Beispiel an, um diesen Sachverhalt zu verdeutlichen:
Es existiert eine Arbeitsumgebung in einem Büro. Typischerweise beinhaltet diese Umgebung eine ganze Menge Formulare auf dem Schreibtisch, die für verschiedene Zwecke benötigt werden. Unter anderem z.B. ein Laufzettel, der als Protokoll eines Projektes eingesetzt wird. Dieser Laufzettel wird im Verlauf des Projektes herumgereicht und von verschiedenen Personen auf unterschiedliche Art und Weise bearbeitet. Der Projektleiter trägt z.B. die täglichen Projektstadien handschriftlich ein, während ein Anwendungsentwickler nur sein Kürzel unter die Änderungsliste setzt, um anzuzeigen, daß er die Änderungen durchgeführt hat.
In JWAM sind diese Vorgänge so modelliert, daß der Laufzettel als ein Material angesehen wird, welches mit verschiedenen Werkzeugen bearbeitet wird. Die Metapher des Werkzeugs wird im Falle von JWAM auf Softwarewerkzeuge übertragen.
Verschiedene Werkzeuge können so auf dasselbe Material zugreifen bzw. können so dasselbe Material bearbeiten, sofern das Werkzeug zu dem Material „paßt“.
Das bedeutet natürlich, daß sofern sich die Materialklasse im Laufe der Entwicklung ändert, die Werkzeuge entsprechend angepaßt werden müssen. Weiter muß sichergestellt sein, daß auch alle zu dem Material „gehörenden“ Werkzeuge auf dieselbe Material-Version angepaßt sind.

3.1. Grundlagen der JWAM Registratur

Im JWAM-Framework wurde eine Registratur von Materialien implementiert, welche sich an den Umgangsformen einer Registratur der realen Arbeitswelt orientiert [Hav1999]. Wie in der realen Arbeitswelt gibt es auch im JWAM-Framework einen Registrator, der sich um die Belange der Registrierung von Arbeitsmaterialien kümmert. Im Framework sind es Anwendungen, welche Zugriff auf persistente Materialien

nehmen möchten und keine Sachbearbeiter, die Dokumente einsehen wollen. Dennoch ähnelt sich die Situation der realen und der softwaretechnischen Welt, da in beiden Fällen ein konkurrierender Zugriff auf die Materialien bzw. Dokumente besteht. Der Registrator ist die zentrale Instanz, die diesen konkurrierenden Zugriff mit geeigneten Mitteln abhandelt.

Die Registratur im JWAM-Framework ist ein fachlicher Behälter, der registrierbare Materialien über Registrierungsnummern verwaltet. Über die Registratur können Originale oder Kopien ausgeliehen werden.
Der größte Vorteil an der Registratur ist, daß sie mit verschiedenen Persistenzmedien zusammenarbeiten kann (z.B. OODBS, RDBS, Filesystem). Sie bietet eine Abstraktionsschicht zum Persistenzmedium, so daß sich der Anwendungsentwickler nicht mit Einzelheiten des jeweiligen Persistenzmediums befassen muß.



Klassendiagramm JWAM-Registratur

Die Registratur besteht genaugenommen aus zwei Teilen. In JWAM wird als Registratur die Instanz bezeichnet, welche die eigentliche Speicherung vornimmt, also auf das Persistenzmedium zurückgreift. Sie ist durch das Programm nicht direkt erreichbar, sondern nur über einen sogenannten Registrator. Der Registrator ist eine der Registratur vorgeschaltete Instanz, die die Schnittstelle zur Registratur bildet. Anwendungen (Klienten) welche Zugriff auf persistente Materialien nehmen wollen, wenden sich an den Registrator.
Sofern der Einsatz in einer verteilten Umgebung geschieht, ist auf dem Client ein Registrator-Proxy implementiert, der die Schnittstelle zur Registratur bildet.
Ein Proxy ist ein Stellvertreterobjekt, welches ein Objekt repräsentiert, das sich in ei-

nem anderen Adressraum befindet [Gam1996]. Der Proxy kapselt dabei die Kommunikation über eine Middleware, meistens CORBA oder RMI.

Der programmtechnische Ablauf sieht z.B. wie folgt aus:
Zunächst wird ein neu erzeugtes Material beim Registrator angemeldet und erhält somit eine Registriernummer. Das Material muß eine von „RegisterableImpl“ abgeleitete Klasse sein. Nur diese verfügt über die zur Registrierung notwendigen Attribute („Registerable“ ist ein Interface aus dem JWAM Framework, welches von jedem registrierbaren Material implementiert werden muß). Die Registrierungsnummer wird dabei vom Registrator an die Anwendung durchgereicht.
Dieses Material kann, wie in einer Registratur der realen Arbeitswelt, zur Bearbeitung ausgeliehen werden. Ausleihen darf allerdings nur ein Client zur Zeit. Hat also ein Client das Material entliehen, so besteht für andere Clients nur noch die Möglichkeit eine Kopie des Materials zu entleihen. Diese Kopie kann dann zwar verwendet werden, darf aber nicht als Original in die Registratur zurückgeschrieben werden. Über entliehene Materialien gibt der Registrator Auskunft in Form von Fehlkarten (missing-cards). Neben diesem Original-Kopie Modell gibt es noch weitere Kooperationsmodelle, wie z.B. das Kopie-Kopie Modell.

Züllighoven [Zül1998] beschreibt ein für das JWAM-Modell gültiges Konzept von Original und Kopie:

> *Materialien, besonders Schriftstücke oder Dokumente, werden häufig in Arbeitsumgebungen als Originale und Kopien zur Verfügung gestellt. Dabei soll die alltagssprachliche Bedeutung der Begriffe aufrecht erhalten werden. In diesem Sinne sind Kopien, zur besseren Unterscheidung auch Arbeitskopien genannt, Materialien aus eigenem Recht, die wie Fotokopien weiter bearbeitet werden können. Zwischen Kopie und Original besteht keine technische Verbindung, so daß Änderungen an dem einen keine Auswirkungen auf das andere haben.*
>
> *Von Arbeitskopien sind technische Kopien zu unterscheiden. Diese können bei der Realisierung des Systems aus softwaretechnischen Gründen auch von fachlichen Originalen erzeugt werden, etwa um Zugriffszeiten zu minimieren.*

Züllighoven unterscheidet zwischen einer fachlichen und einer technischen Sicht von Original und Kopie. Fachlich müssen Kopien wie “Materialien aus eigenem Recht” behandelt werden. Das bedeutet, daß diese Kopie, unabhängig von ihrem Ursprung, weiterbearbeitet werden kann. Es gibt keinen technischen Mechanismus, der dies verhindert. Konfliktsituationen können nur anwendungsfachlich aufgelöst werden. In einem solche Konfliktfall muß im fachlichen Kontext klargestellt werden, ob es sich bei einem Material um ein Original oder eine Kopie handelt. Niemals ist aber die Eigen-

schaft “Original” oder “Kopie” ein inhärentes Attribut des Materials.
Materialien, welche der JWAM-Registrator als Kopie an den Klienten ausgegeben hat, können im oben beschriebenen Sinne verändert werden, dürfen aber nicht mehr als Original in die Registratur zurückgestellt werden. Die Kopie kann nur dann als Original in die Registratur zurückgestellt werden, wenn sie eine neue Registrierungsnummer erhält. Damit wird die Kopie auch anwendungsfachlich zu einem neuen Original, die keine Verbindung mehr zu ihrer ursprünglichen Quelle aufweist.

3.2. Klassen der JWAM-Registratur

Die Implementation der Registratur in JWAM befindet sich in dem Package „de.jwamx.handling.registry“. Im nachfolgenden betrachten wir nur diejenigen Klassen der Registratur, die im Rahmen dieser Studienarbeit für die Versionierung von Bedeutung sind.

Das Interface„Registerable“

Dieses Interface muß von allen Materialien implementiert werden, die persistent gemacht werden sollen.

Die Klasse „Registrator“

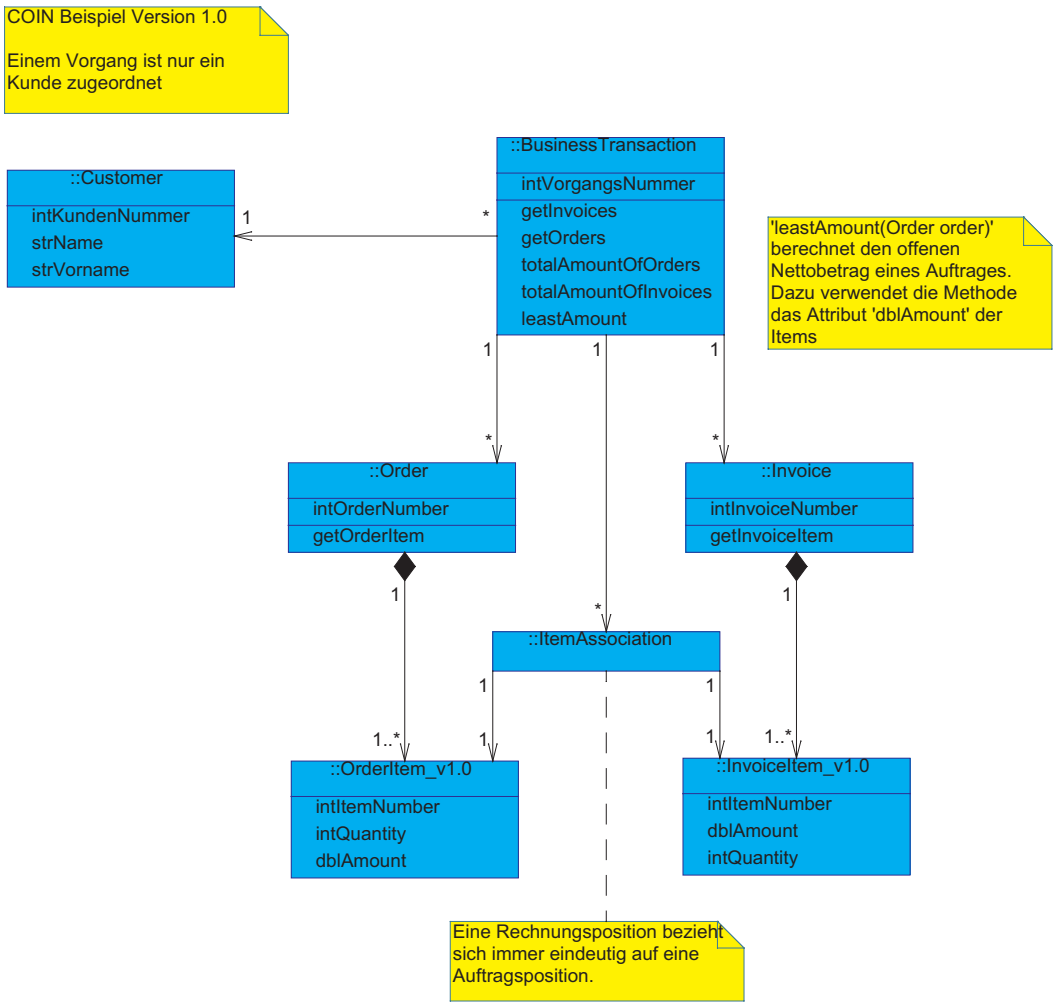
Über den Registrator werden Materialien registriert und können wieder abgefragt werden. Er ist die zentrale Instanz der Materialverwaltung.

3.3. Komplexe Materialien in der JWAM Registratur

Dieser Abschnitt faßt im wesentlichen die Ergebnisse der JWAM Architekturgruppe der Universität Hamburg vom 22.11.99 zusammen.

Unter Objektgeflechten verstehen wir komplexe Materialien, deren Bestandteile (Shared Objects) auch einzeln aus der Registratur angefordert und unabhängig vom Rest bearbeitet werden können. Die Bestandteile eines Objektgeflechtes haben aus diesem Grunde jeweils eigene Registriernummern sind aber über fachliche Querbezüge referenziert. In vielen Fällen sind fachliche Materialien vorstellbar, die nicht nur durch eine einzelne Klasse implementiert sind, sondern durch ein Objektgeflecht realisiert werden und in bestimmten Anwendungsfällen nur im ganzen bearbeitet werden dürfen. Im unten gezeigten Fall ist es nicht sinnvoll, nur eine Rechnungsposition für sich zu bearbeiten, ohne eine möglicherweise bestehende Verbindung zu einer Auftragspo-

sition zu berücksichtigen. Das komplexe Material kann durch eine solche Operation in einen inkonsistenten Zustand geraten.



Objektgeflecht Kunde - Auftrag - Rechnung

Die Migration dieses Objektgeflechtes wirft Probleme auf, die auch schon bei der unabhängigen Bearbeitung der einzelnen Objekte des Geflechtes durch verschiedene Werkzeuge auftreten. Die fachlichen Querbezüge müssen bei der Migration erhalten bleiben. Es kann z.B. ein Problem auftreten, wenn durch ein Auflistungswerkzeug eine Auftragsposition gelöscht wird, welche noch durch eine Verbindung (ItemAssociation) mit einer Rechnungsposistion referenziert wird. Dann entsteht eine Inkonsistenz des komplexen Materials.

Für dieses Problem gibt es verschiedene Lösungsansätze.

Referenzzählung

Es könnte ein Referenzzähler implementiert werden. So ein Referenzzähler ist in jedem Objekt des komplexen Materials enthalten. Wird das Objekt durch ein anderes refernziert, dann wird der Zähler erhöht. Wenn die Referenz entfernt wird, wird der Zähler wieder heruntergesetzt. Steht der Zähler auf „0“ darf das entsprechende Objekt einzeln bearbeitet oder gelöscht werden.
Diese Lösung ist sehr eingeschränkt, da Sie keine detaillierte Informationen über die refernzierenden Objekte liefert.

Nur "Hülle" komplexer Materialien ausgeben

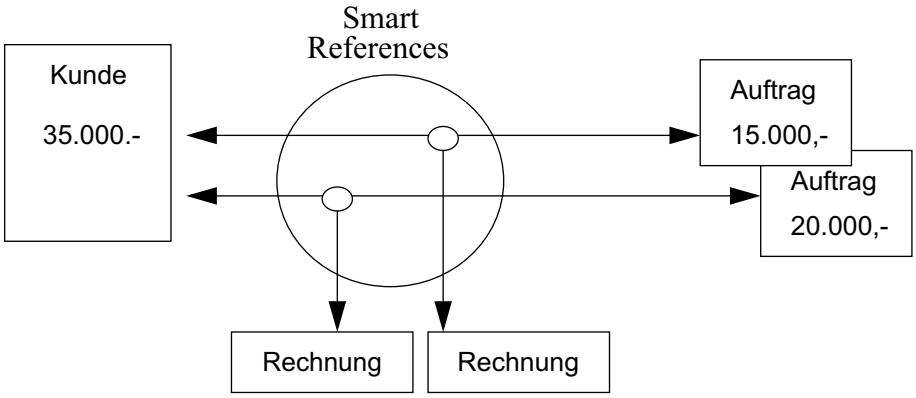
Der zweite Ansatz geht in die entgegengesetzte Richtung und verhindert gänzlich das Ausleihen einzelner referenzierter Objekte. Stattdessen wird immer das gesamte Meta-Objekt entliehen. Diese Lösung hat insbesondere den Nachteil, daß sie bei großen Objektstrukturen langsam und speicherintensiv wird. Zudem erhöht sich die Gefahr, daß es zu Zuständen kommen kann, in denen das Ausleihen nicht mehr möglich ist. In diesem Fall können andere Clients, die Teile dieses Objektgeflechtes bearbeiten wollen, solange nicht mit Ihrer Arbeit fortfahren, bis das Objektgeflecht als Ganzes wieder in die Registratur zurückgestellt wird.

Spezielle Registraturen

Die bisher diskutierten Registraturen besitzen keine Kenntnis über den Inhalt und die fachlichen Zusammenhänge der Materialien, die sie verwalten. Um die fachliche Konsistenz von komplexen Materialien zusichern zu können, wäre es denkbar, Registraturen um fachliches Wissen zu erweitern. Mit Hilfe dieses fachlichen Wissens sind diese speziellen Registraturen dann in der Lage, die fachlichen Bezüge zwischen einzelnen Komponenten eines komplexen Materials aufrecht zu erhalten. Alle Operationen, die diese Konsistenz zerstören könnten, würden dann von der Registratur abgewiesen.
Diese Registraturen haben den Nachteil, daß sie nicht mehr universell einsetzbar sind, weil sie für jeden einzelnen Fall angepaßt werden müssen.

Intelligente Referenzen

Der letzte Lösungsansatz geht auf die Referenzen der Objekte ein und ermöglicht dadurch differenziertes Ausleihen einzelner Materialien aus der Registratur.
Durch ein spezielles Referenzhandling lassen sich die obigen beschriebenen Probleme elegant verhindern. Dazu werden sogenannte „Smart References“ oder auch intelligente Referenzen eingeführt, welche die Referenzen zwischen den Objekten aufnehmen. Sie stellen das Bindeglied zwischen den einzelnen Objekten des Objektgeflechtes dar.



Smart References

Eine „Smart Reference“ verwaltet Wissen über Referenzen. Zum Beispiel erlaubt oder verweigert sie das Löschen von Objekten. Soll z.B. ein Auftrag gelöscht werden, ermittelt der Registrator die zum Auftrag gehörigen “Smart References”. Sind keine „Smart References“ für ein Objekt vorhanden, so kann es sofort gelöscht werden. Im anderen Fall wird der Löschwunsch an das Smart-Reference-Objekt weitergeleitet. Dieses entscheidet, ob dem Löschwunsch entsprochen werden kann. Sollte ein Smart-Reference-Objekt eine negative Rückmeldung geben, darf nicht gelöscht werden.

Fazit für die Versionierung

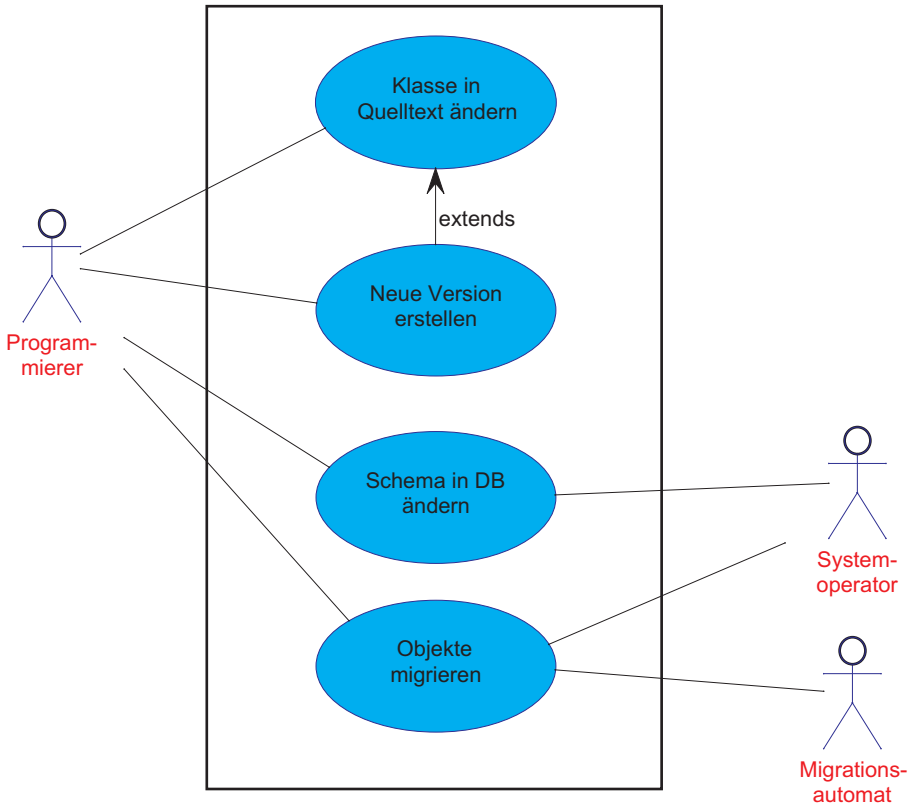
“Smart References” eignen sich gut für die Migration “on-the-fly”, da sie das Ausleihen auch von Teilen eines komplexen Materials unterstützen. Für die Migration “at-once” sind sie daher eher hinderlich. Für eine komplette Migration aller in der Registratur enthaltenen Materialien sollten möglichst alle Teile des komplexen Materials verfügbar sein.

Für die Migration von Objektgeflechten bietet sich so das erste Konzept, die äußere Hülle an. Da in der JWAM Registratur in der Regel keine großen Objektgeflechte zu erwarten sind (sie implementiert noch keine „Smart References“), haben wir zur Demonstration der Versionierung ein kleines Objektgeflecht entsprechend dem Auftrag-Kunde-Rechnung Beispiel gewählt.

Kapitel 4: Konzept des Versionierungssubframeworks

4.1. Anforderungen an die Versionierung

Um ein Konzept für die Versionierung zu entwickeln, ist es nötig sich den Aufgabenbereich der beteiligten Akteure anzusehen, um die Anforderungen an die Versionierung zu ermitteln.
Im unteren Bild ist dazu das UseCase-Diagramm dargestellt, welches deutlich macht, auf welche Weise Programmierer, Systemoperator und Migrationsautomat interagieren. Der Migrationsautomat stellt eine selbständig agierende Einheit dar, weswegen wir ihn entgegen UseCase üblichen Konventionen in das Diagramm aufgenommen haben.



UseCase - Diagramm

Klasse im Quelltext ändern:
Der Programmierer ändert den Quelltext seiner Klasse. Solange die Klasse sich noch im Zustand des Überganges von einer Version zur nächsten befindet, d.h. die Klasse noch keinen Meilenstein im Bearbeitungszustand erreicht hat, ordnet der Programmierer ihr noch keine neue Versionsnummer zu.

Neue Version erstellen
Der Programmierer erstellt eine neue Version einer Klasse, die sich wesentlich von ihrer Vorgängerversion unterscheidet. Dazu wählt der Programmierer eine neue Versionsnummer aus.

Schema in DB ändern
Der Systemoperator pflegt die Datenbank und ist zuständig für die Anpassungen des Datenbankschemas. Das Datenbankschema wird entsprechend den Anforderungen auf der Clientseite geändert.

Objekte migrieren
Der Systemoperator initiiert das Migrieren von Objekten. Der Migrationsautomat kann Objekte auch selbständig migrieren, wenn ein Versionskonflikt auftritt (on-the-fly Migration).

Schemaänderungen bestimmter Art sind nicht berechenbar, bzw. folgen keinem mathematisch begründetem Muster, sondern beruhen auf konzeptionellen Änderungen der Programmierer.
Für eine automatische Migration von Objekten werden Beschreibungen benötigt, die in geeigneter Weise den Migrationsvorgang spezifizieren. Die Konsistenz der Objekte und Ihrer Beziehungen untereinander muß dabei erhalten bleiben. Die zur Migration benötigten Beschreibungen können nicht automatisch erzeugt werden, sondern müssen von Anwendungsentwicklern verfaßt werden.

Der Programmierer ist also gezwungen, die Migrationsanweisungen für seine Klassen selbst zu definieren. Er muß dafür Sorge tragen, seine Beschreibungen vollständig zu formulieren, und Beschreibungen für jede geänderte Klasse einzubetten.
Die Versionskonflikterkennung und Migration der Objekte sollte weitgehend transparent durchgeführt werden.
Hierzu bedarf es neben der Migrationsbeschreibung weiterer Komponenten für die Er-

kennung des Konfliktes, die Delegierung der Migration, die Ausführung der Migration und das Zurückschreiben bzw. Reregistrieren der migrierten Objekte.

4.2. Gegenstand der Versionierung

Gegenstand der Versionierung ist die Materialklasse.

Klassenversionierung

Bei dieser Form der Versionierung werden einzelne Klassen unabhängig von einander und von der Zugehörigkeit zu einem Package mit Versionsnummern versehen.
Da Klassen zumeist in Packages gruppiert werden, stellt sich die Frage, ob eine Versionierung von ganzen Packages nicht sinnvoller ist als die Versionierung einzelner Klassen.

Packageversionierung

In prozeduralen Programmiersprachen ist das Modul das fundamentale Konzept zur Modularisierung von Software. In objektorientierten Sprachen ist dieses Konzept durch das Konzept der Klasse abgelöst worden. Im Vergleich zum Modul stellt die Klasse jedoch eine sehr viel feingranularere Einheit dar. In großen Softwaresystemen ist es sinnvoll, Klassen zu einer höheren Ordnungseinheit zu gruppieren. So gibt es z.B. in der objektorientierten Programmiersprache Java die Möglichkeit, fachlich oder technisch zusammengehörige Klassen zu Packages zusammenzustellen. Das Java Development Kit von Sun ist selbst in Packages dieser Art organisiert. Packages helfen dabei, große Softwaresysteme sinnvoll in Subsysteme zu gliedern. Ab der Version 1.2 des JDKs wird mit der Klasse “Package” eine programmiersprachliche Unterstützung dieses Konzeptes angeboten.
Eine Alternative zur direkten Versionierung von Klassen stellt die Versionierung von Packages dar. Die Versionsnummer wird dabei mit in den Packagenamen aufgenommen, bzw. an das Ende des Packagenamens angehängt, z.B. „de.jwamalpha.handling.versioning.example.coin_v1000“. In diesem Kapitel wollen wir die Vor- und Nachteile untersuchen, die eine Packageversionierung mit sich bringt. Wir vergleichen im Anschluß daran die Packageversionierung mit der Versionierung von Klassen.

Vor- und Nachteile von Packageversionierung

Vorteile:

- Alle Klassen, welche zu einer Packageversion gehören, können zusammenarbeiten.

- Der Klassenname ändert sich nicht mit der Version des Packages, da die Versionsbezeichnung nicht am Klassennamen hängt. Somit können Klientenklassen die Materialklasse immer unter dem selben Namen referenzieren.
- Die Migration von Objekten, deren Spezifikation sich nicht geändert hat, ist sehr einfach, da sich der Name der zugehörigen Klasse nicht geändert hat.

Nachteile:
- Auch Klassen, die sich mit der neuen Packageversion nicht geändert haben, erhalten implizit eine neue Versionsbezeichnung.
- Bei der Migration ergibt sich das Problem, daß die beteiligten Versionen der Klassen mit ihrem vollständig qualifizierten Namen angegeben werden müssen, da sich der Klassenname von einer Version zur nächsten nicht geändert hat.

Vor- und Nachteile von Klassenversionierung

Vorteile:
- Nur Klassen, die sich geändert haben, erhalten eine neue Versionsnummer.
- Es müssen nur Objekte migriert werden, deren Klassenbeschreibung geändert wurde.

Nachteile:
- Es muß paarweise spezifiziert werden, welche Klassenversionen zueinander kompatibel sind.
  Beispiel: Zwei Klassen A und B stehen in einer Benutztbeziehung zueinander. Version 1.0 der Klasse A arbeitet mit Version 1.1 der Klasse B zusammen aber nicht Version 1.0 der Klasse B mit Version 1.1 der Klasse A. Die Kompatibilität der Klassen A und B zueinander ist nicht anhand ihrer Versionsbezeichnung erkennbar. Die Kompatibilität muß also explizit beschrieben werden.
- Tragen Klassen ihre Versionsnummer im Namen, dann muß dies bei der Referenzierung in Klientenklassen berücksichtigt werden. Dies ist zwar eingeschränkt auch bei der Packageversionierung notwendig, ein Problem ist dies allerdings nur, wenn eine Klasse vollständig mit ihrem Packagenamen qualifiziert werden muß.

Bewertung der Ansätze zur Versionierung

Wir vertreten die Meinung, daß die Packageversionierung für eine Versionierung von großen Softwaresystemen besser geeignet ist. Die “Java Product Versioning Specification” vom 30. November 1998 spezifiziert das Package als eine in sich abgeschlossene Einheit, die in ihrer Form verpackt, geprüft, aktualisiert und vertrieben werden kann.
Seit der Version 1.2 des JDKs ist eine rudimentäre Unterstützung der Versionierung in der Klasse “java.lang.Package” implementiert. Wir sprechen von “rudimentärer” Un-

terstützung, da diese Klasse nur einfache Versionsinformationen anbietet.

Folgende Informationen sind enthalten:
- Versionierung der “Virtual Machine”
- Versionierungsinformationen über die Java-Laufzeitumgebung
- Package-Versionierung
- Spezifikation des JAR-Formates (Manifest)

Für uns sind nur die Informationen über die Packageversionierung interessant. Diese sind im einzelnen:
- Implementation-Title
- Implementation-Version
- Implementation-Vendor
- Specification-Title
- Specification-Version
- Specification-Vendor

Diese Informationen liegen alle in Form von Strings vor. Die Semantik von Versionsnummern ist nicht eindeutig festgelegt. Sie muß in jedem Softwareprojekt neu definiert werden. Die Packageversionierung des JDK unterstützt auch nur Evolutionsformen von Klassen, die abwärtskompatibel sind. Für die Klasse “Package” sind zwei Versionen einer Klasse kompatibel, wenn die vorliegende Versionsnummer größer oder gleich der Versionsnummer des verglichenen Objektes ist. Diese Annahme impliziert eine Reihe von Beschränkungen bei der Entwicklung neuer Klassenversionen. Praktisch werden nur Änderungen unterstützt, die nicht das Interface und damit die Semantik der Klasse verändern.
Die Klasse “Package” bietet keine weitere Unterstützung für den Migrationsprozess an. Sie ist aber ein guter Ausgangspunkt für die Anbindung eines Versionierungsframeworks.

Wir konzentrieren uns im Rahmen dieser Studienarbeit auf das Versionieren von Pakkages.

4.3. Materialien als versionierbare persistente Objekte

Da das Material gespeichert (persistent gemacht) wird, ergibt sich die Frage nach der Version des gespeicherten Objektes. Wie in der Einführung schon erwähnt, kann sich die Anwendung im Laufe der Zeit ändern (sog. Updates erhalten). Die Anwendung ist darauf angewiesen, daß die persistenten Objekte zu ihrer Beschreibung passen.
Ein fehlerhafter Zugriff auf ein Material aufgrund falscher Objektbeschreibung kann

u.U. schwerwiegende Folgen haben. Ist die Struktur eine andere als angenommen, könnte der Fall eintreten, daß zwar ein Lesen der Daten zu keinem Fehler, ein weiteres Verwenden der Daten aber zu falschen Ergebnissen führt.
Angenommen, es gäbe z.B. eine Klasse „Kostenobjekt“, mit folgenden Attributen:

- Integer intID
- String strZweck
- Float flNettoBetrag
- Float flMwSt

Das Attribut „flMwSt“ wird durch das Programm berechnet und mit den übrigen Attributen des Objektes persistent gemacht.

In der nachfolgenden Version wird die Klasse nun, da dies programmiertechnisch erheblich sauberer ist, wie folgt umgeschrieben:

- Integer intID
- String strZweck
- Float flMwSt
- Float flNettoBetrag
- Float CalculateMwSt(Nettobetrag*flMwSt/100)

Die Klassenbeschreibung ist in Hinblick auf ihre Attribute, vollkommen gleich geblieben. Das Attribut „flMwSt“ erhält durch die hinzugekommende Methode aber eine völlig neue Bedeutung. Es steht jetzt nicht mehr für den MwSt-Betrag an sich, sondern enthält den MwSt-Satz. Dieser wird nun statt der MwSt gespeichert und die Mehrwertsteuer wird aus ihm und dem Nettobetrag berechnet.
Sollte in diesem Fall eine neue Programmversion alte Objekte laden, und hierbei keine logische Überprüfung der Daten durchführen, kämen völlig falsche Mehrwertsteuerbeträge zusammen.

Um die Objekte mit den Klassenbeschreibungen abgleichen zu können muß eine Versionsnummer für Materialien eingeführt werden. Nur so kann ein Fall, wie vorher beschrieben, verhindert werden.
Anhand dieser Versionsnummer können die Werkzeuge dann erkennen, um welche Version es sich handelt, diese mit ihrer Beschreibung des Objektes abgleichen und im Bedarfsfall eine Migration vornehmen.

Dieses Konzept birgt das Problem, daß jedes Werkzeug für die Überprüfung und Migration seiner Materialien selbst verantwortlich ist, und es so bei der Benutzung und

Migration derselben Objekte zu Überschneidungen kommen kann.
Aus diesem Grund sollte die Überprüfung an zentraler Stelle stattfinden, so daß die Anfrage von Programmen am Persistenzmedium kontrolliert werden kann.

In JWAM gibt es ein Konzept zur Verwaltung von Materialien, welches sich um die Speicherung der Materialien kümmert. Die Verwaltung und Speicherung der Materialien geschieht hiermit völlig transparent gegenüber dem Werkzeug und Material. Weder das Werkzeug noch das Material müssen sich um die Persistenzoperationen kümmern. Die Implementation dieser Materialverwaltung ist in JWAM als Registratur eingebettet und wird im nachfolgenden detaillierter behandelt.
Die Erkennung, Delegation und Auflösung von Versionskonflikten sollte ebenfalls möglichst unabhängig von Werkzeug und Material erfolgen. Aus diesem Grunde ist die Registratur eine geeignete Instanz für die Versionsüberprüfung.

4.4. Die Erkennungsinstanz

Wie wir im Kapitel über JWAM gezeigt haben, werden sämtliche persistenten Materialien über die Registratur verwaltet. Dadurch bietet sie im Rahmen des Versionierungssubframeworks die zur Erkennung am besten geeignete Komponente.
Im folgenden werden wir nur noch auf den Registrator eingehen, da er die Instanz ist, über die die Anwendung die Materialien registriert.
Solange die Entwickler dieser Regel folgen, und den JWAM-Registrator zur Verwaltung ihrer Materialien einsetzen, bietet er die beste Möglichkeit zur Erkennung eines Versionskonfliktes, da er primären Zugriff auf persistente Materialien besitzt.
Im zweiten Schritt kann auch er die Delegation des Migrationsvorganges übernehmen, so das die Migration stattfinden kann, noch bevor die Anwendung eine obsolete Version des angeforderten Materials erhält.

Der Registrator muß zur Erkennung von Versionskonflikten und zur Delegation der Konfliktbehebung angepaßt bzw. erweitert werden.
Meyer spricht eigentlich von den drei Punkten “Erkennung”, “Benachrichtigung” und “Korrektur” von Versionskonflikten. Da der Registrator eine zentrale Komponente in unserem System ist, und gleichzeitig die Aufgaben des “Lesenden” und “Schreibenden” Systems (Meyer) warnimmt, entfällt dieser Punkt. Statt dessen betrachten wir die Aufgabe der “Delegation” der Konfliktbehebung an eine andere Komponente.

Tabelle 1: CRC Card „Erkennungsinstanz“

Responsebility	Collaboration
Die Erkennungsinstanz kann anhand der Versionsinformation bzw. der Versionsnummer eines Materials einen Versionskonflikt erkennen. Delegiert einen Migrationsauftrag an die Migrationsinstanz bei der Migration “on demand”.	Migrationsinstanz, Versionsinformation Migrationsauftrag Migrationsauftrags-Sammlung

Das Interface „Registerable“ ist für die Versionierung von Bedeutung, weil es von jedem Material zur Registrierung implementiert wird. Die Standardimplementierung („RegisterableImpl“) ist ein guter Ansatzpunkt, um Versionierungsinformationen zu plazieren.

Da eine Versionsüberprüfung nur beim Ausleihen eines Materials erforderlich ist, sind nur die beiden nachfolgend aufgeführten Methoden dieses Vorganges von Bedeutung:

- original(dvUserIdentificator user, dvRegistrationNumber regNo)
  Sofern verfügbar, gibt die Registratur das Original heraus.
- copy(dvUserIdentificator user, dvRegistrationNumber regNo)
  Die Registratur gibt eine Kopie heraus.

Zur Rückführung bzw. erneuten Registrierung des migrierten Materials benötigen wir die Methode „replaceOriginalByOriginal“, die das „alte“ Material mit der migrierten Version überschreibt. Die Methode “replaceOriginalByCopy” kommt nicht zum Einsatz, da wir zuerst ein neues Original einer neuen Klasse bauen und damit das Original der alten Klasse ersetzen. Im Anschluß wird das Material mit der Methode “releaseOriginal” wieder freigegeben.

4.5. Die Migrationsinstanz

Der Registrator wird die Migration an eine Migrationsinstanz delegieren. Diese Instanz ist dafür verantwortlich die Migration der Objekte so durchzuführen, das keine Inkonsistenzen auftreten und die referenzielle Integrität erhalten bleibt. Das bedeutet z.B., daß keine “hängende Referenzen” entstehen, wie in dem Fall, daß ein Objekt gelöscht wird, welches noch von einem anderen Objekt referenziert wird.

Um die referenzielle Integrität zu erhalten, bedient sich die Migrationsinstanz der im letzten Abschnitt angesprochenen Migrationsanweisungen des Programmierers. Diese

könnten die Migration der einzelnen Klassen selbst durchführen.
Die Migrationsinstanz ist dann für die Reihenfolge der Migration zuständig, da diese für die Integrität der Daten bei Objektgeflechten bzw. Abhängigkeiten der Klassen untereinander entscheidend ist. Sie hat dementsprechend die Kontrolle über die einzelnen Objektmigrationen, welche durch die Migrationsanweisungen durchgeführt werden.

Im folgenden ist die CRC Karte der Migrationsinstanz dargestellt.

Tabelle 2: CRC Card „Migrationsinstanz“

Responsebility	Collaboration
Migriert Objekte in das Schema einer neuen Version. Unterstützt zwei unterschiedliche Migrationsansätze: a) Migration “on demand” b) Migration “at once” Verwaltet Migrationsaufträge (Scheduling) und registriert Migrationsbeschreibungen.	Erkennungsinstanz, Versionsbeschreibung Migrationsauftrag

4.6. Die Versionsbeschreibung und die Versionsnummer

Damit eine Versionserkennung stattfinden kann, müssen die persistent gemachten Objekte eine Versionsbeschreibung mit zugehöriger Versionsnummer erhalten.

Tabelle 3: CRC Card „Versionsinformation“

Responsibility	Collaboration
Sichert Aspekt der Versionierbarkeit zu. Liefert Versionsnummer der zugehörigen Materialien um es der Erkennungsinstanz zu ermöglichen einen Versionskonflikt zu erkennen.	Material Versionsnummer Erkennungsinstanz Migrationsinstanz

Damit die Erkennungsinstanz in der Lage ist, Versionskonflikte erkennen zu können, muß sie auf die Versionsinformationen Zugriff haben. Dazu nutzt sie insbesondere die Versionsnummer, welche aus diesem Grund innerhalb der Versionsinformation eine besondere Bedeutung einnimmt. Die Migrationsinstanz erzeugt eine neue Version des Materials und muß daher ebenfalls auf die Versionsinformation zugreifen können.

Tabelle 4: CRC Card „Versionsnummer“

Responsebility	Collaboration
Stellt Wertebereich für strukturierte Nummerierung zur Verfügung. Versionsnummern sind untereinander vergleichbar. Gibt Auskunft über einzelne Ebenen der Nummernstruktur.	

4.7. Die Migrationsbeschreibung

Die Migrationsbeschreibung beinhaltet Änderungsinformationen über die zugehörigen Klassen und kann die Migration eines Objektes ihrer zugehörigen Klasse vornehmen. Die Migrationsanweisung besitzt Wissen über das der Entwicklung zugrundeliegende Schema (Siehe “Schemaevolution” auf Seite 6.). Aus diesem Grunde wird Sie vom Programmierer selbst geschrieben.

Tabelle 5: CRC Card „Migrationsbeschreibung“

Responsebility	Collaboration
Enthält Handlungsanweisungen, durch die Objekte eines älteren Schemas in die neue Version überführt werden können und sichert dadurch die Konsistenz der persistenten Materialien. Stellt generische Funktionalität für die Migration von Materialien zur Verfügung. Kann für spezifische Materialien erweitert werden.	Migrationsinstanz Material

Die Migrationsinstanz migriert die Attribute des Materials in der Reihenfolge, wie sie in der Migrationsbeschreibung aufgeführt sind.

4.8. Der Migrationsauftrag

Tabelle 6: CRC Card „Migrationsauftrag“

Responsebility	Collaboration
Auftrag an die Migrationsinstanz zur Migration von Materialien. Legt den Zeitpunkt der Migration fest (Scheduling).	Migrationsinstanz Migrationsbeschreibung

Die Migrationsinstanz kann neben einer Migrationsbeschreibung auch einen Migrati-

onsauftrag erhalten. Damit wird die Migration eines Materials ab einem festgelegtem Zeitpunkt ermöglicht.

4.9. Die Migrationsauftrags-Sammlung

Tabelle 7: CRC Card „Migrationsauftrags-Sammlung“

Responsebility	Collaboration
Faßt mehrere Migrationsaufträge zu einer Sammlung zusammen. Stellt sicher, daß die Migrationsaufträge in einer bestimmten Reihenfolge in der Migrationsinstanz ausgeführt werden.	Migrationsinstanz Migrationsbeschreibung

Die Migrationsinstanz kann neben einer Migrationsbeschreibung und einem Migrationsauftrag auch eine Migrationssammlung erhalten. Damit wird die Migration verschiedener Materialien ab einem festgelegtem Zeitpunkt ermöglicht.

4.10. Möglicher Ablauf eines Versionierungsvorganges in JWAM

Im folgenden skizzieren wir einen Versionierungsvorgang, wie er mit Hilfe des Registrators ablaufen könnte.

1. Der Client fordert persistentes Material (Original oder Kopie) beim Registrator über die zugehörige Registrierungsnummer an. Der Client gibt bekannt, welche Version er benötigt. Die Versionsnummer erhält der Client aus dem Packagenamen des angeforderten Materials.
3. Der Lademechanismus läuft wie bisher ab, d.h. das alte Objekt wird in den transienten Speicher geladen (Vorbedingung: das alte Classfile ist verfügbar). Die Versionsnummer ist im Packagenamen des Materials codiert.
4. Der Registrar vergleicht die Versionen:
   a) Die Versionen sind gleich: Übergabe wie bisher
   b) Die Versionen sind unterschiedlich:
   b1) Quellversion < Targetversion -> Migrationsbeschreibung anfordern
   b2) Quellversion > Targetversion -> Exception
   b3) ConversionSpec nicht verfügbar -> Exception
5. Migration des Objektes anhand der Information aus ConversionSpec.
   Aufruf „Migrate“ am OMA mit Angabe der Registrierungsnummer.
6. Gibt es Abhängigkeiten zu anderen Klassen?
   Sofern Abhängigkeiten zu anderen Objekten bestehen, scheidet eine Mi-

gration „en demand“ aus und es muß mit Hilfe einer Liste von Migrationsaufträgen eine Migration „at once“ durchgeführt werden (Abbruch dieses Vorgangs).

7. Migration des Objektes durch ConversionSpec (evtl. MigrationException)
8. ConversionSpec legt Objekt mit neuer Klassenbeschreibung an.
9. Neues Objekt wird unter vorhandener Registrierungsnummer in der Registratur registriert, das alte Objekt wird gelöscht.

Kapitel 5: Implementation des Versionierungssub-frameworks

Aus dem vorgelegten Konzept ergibt sich die Notwendigkeit ein kleines Rahmenwerk (engl. Framework) zu implementieren, welches die Aufgaben der Versionierung übernimmt. Wir werden dieses als „Versionierungssubframework“ bezeichnen, oder auch in Kurzform als „VSFW“.

5.1. Einschränkungen der Implementation

Wir können in dieser Studienarbeit kein vollständiges Rahmenwerk zur Versionierung realisieren, da dieses den Rahmen dieser Studienarbeit sprengen würde.
Konzeptionell ist das Versionierungsframework so angelegt, daß es erweitert bzw. vervollständigt werden kann. Wir haben zunächst einen Prototypen implementiert, an dem die Funktionalität des VSFW (Erkennung, Delegation, Migration) demonstriert werden kann.

Dieser Prototyp verfügt über die Fähigkeit, einzelne Objekte zu migrieren, d.h. sie von der Registratur zu bekommen, sie dann zu migrieren und sie wieder in die Registratur zu stellen. Da der Vorgang später zum einen transparent ablaufen soll (Erkennung und anschließende Migration), zum anderen aber auch manuell (Migration en-masse), haben wir entsprechende Werkzeuge und Materialklassen implementiert, mit denen der verantwortliche Systemadministrator die Migration steuern kann.

Die größte Einschränkung ist zunächst, daß das VSFW keine Vollständigkeits- oder Plausibilitätsprüfung durchführt. Eine erfolglose Konvertierung bzw. fehlerhafte Feststellung von Versionierung wird lediglich durch eine Exception angezeigt und ist irreversibel (kein Rollback). Dieser Punkt bietet sich für eine weiterführende Behandlung in einer anderen Arbeit an.

5.2. Änderungen an der JWAM Registratur

Der Registrator aus dem JWAM-Framework ist die geeignete Instanz zur Erkennung von Versionskonflikten und zur Delegation der Migration an die Migrationsinstanz.
Um die Erkennung eines Versionskonflikts und die Beauftragung an eine Migrationsinstanz zu implementieren, ist es nötig Änderungen bzw. Erweiterungen an der JWAM Registratur vorzunehmen.

Registrator - „Original“-Methode

Die Zugriffsmethoden auf registrierte Materialien müssen geändert werden. Dies betrifft die beiden Methoden „Original“ und „Copy“ des JWAM-Registrators.
Die Methoden werden jeweils durch eine mit Versionierungsinformationen bestückte Methode überladen. Dabei wird zum Aufruf das Objekt „Versionable“, welches zu jedem versionierten Objekt existiert, als zusätzlicher Parameter angegeben, aus dem der Registrator die angeforderte Version auslesen kann. Bei dem “Versionable”-Objekt handelt es sich um ein statisches Objekt, da es für alle Objekte einer bestimmten Materialklasse gültig ist. In Java ist es nicht möglich, statische Objekte bzw. Member persistent zu machen. Solche Objekte müssen daher im transienten Speicher neu erstellt werden (d.h. die Versionsnummer aus dem Packagenamen wird in dieses Objekt eingelesen). Diese Aufgabe wird vom Client übernommen, da nur er sein zugehöriges Package kennt.

Desweiteren muß der Registrator über die Fähigkeit verfügen, die Version eines persistenten Objektes und die der angeforderten Klasse zu vergleichen.
Diese zusätzliche Funktionalität haben wir in die von uns überlagerte Methode „Original“ integriert.

```
public Registerable original (dvUserIdentificator user, dvRegistrationNumber regNo, Versionable
vTarget) throws ThingNotRegisteredException, MissingOriginalException, RemoteException, Mi-
grationVersionException, OmaNotRegisteredException
{
    Contract.require(vTarget != null, this, "vTarget != null");
    Registerable sourceReg = original(user, regNo);
    Versionable vSource = new Versionable();
    vSource.setConversionSpec(vTarget.getConversionSpec());
    ((RegisterableImpl)sourceReg).setVersionable(vSource);
    dvVersionNumber sourceVersion = vSource.getVersion(sourceReg);
    dvVersionNumber targetVersion = vTarget.getVersion();
    if (sourceVersion.equals(targetVersion))
        {
            return sourceReg;
        }
    else if (sourceVersion.isPredecessor(targetVersion))
    {
        if (Environment.instance().workspace().has("OMA"))
        {
            ObjectMigrationAutomatonImpl oma =
            (ObjectMigrationAutomatonImpl)Environment.instance().workspace().thing
            ByName("OMA");
            Registerable targetReg = oma.migrate(sourceReg, false);
            try
            {
                replaceOriginalByOriginal(targetReg);
            }
            catch (ThingIsNotOriginalException e)
            {
```

```
                System.out.println("ThingIsNotOriginalException");
            }
            catch (ThingNotRegisteredException e)
            {
                System.out.println("ThingNotRegisteredException");
            }
            return targetReg;
        }
        else // oma not registered
        {
            throw new OmaNotRegisteredException("OMA not registered at workspace!");
        }
    }
    else if (sourceVersion.isSuccessor(targetVersion))
    {
        throw new MigrationVersionException(sourceReg.id(), sourceReg.registrationNumber(),
        sourceVersion, "migration to lower version not possible!");
    }
    return sourceReg;
}
```

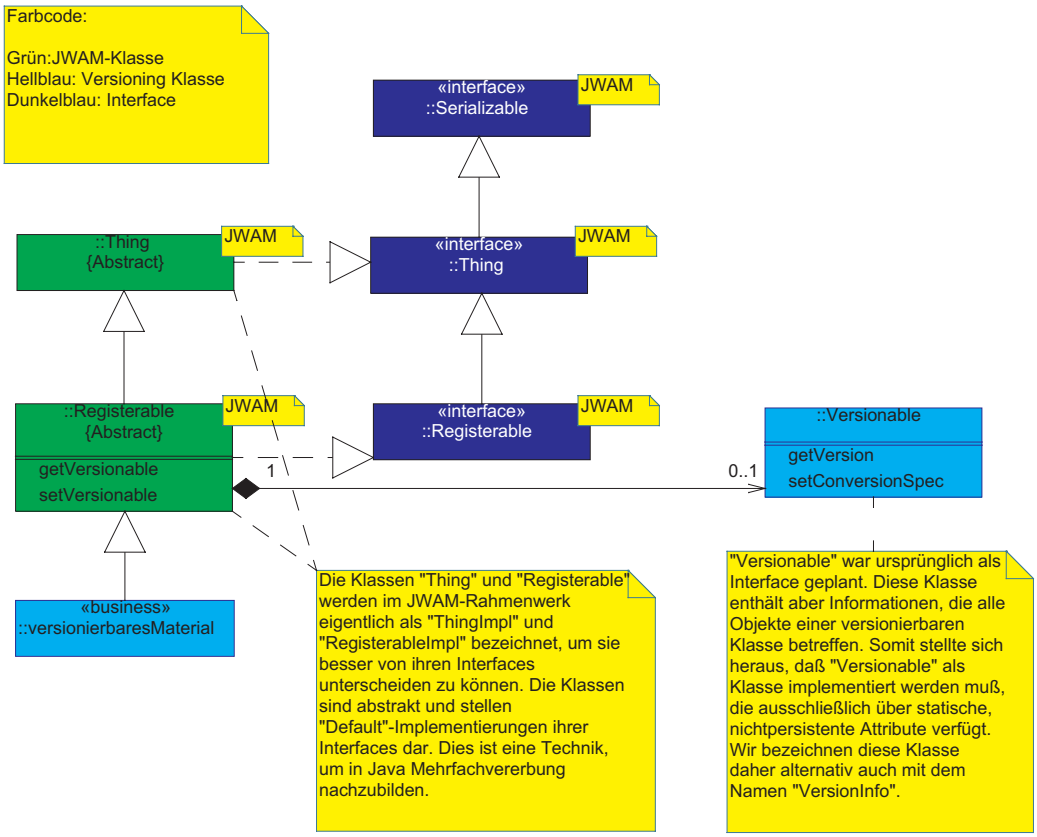
Nach der Initialisierung einiger für diese Methode wichtiger Variablen, wird überprüft, ob die Version des angeforderten Objektes mit der Version des im Persistenzraumes liegenden Objektes übereinstimmt. Ist dies nicht der Fall, wird der Objektmigrations-automat mit der Migration beauftragt. Ist die Migration abgeschlossen, versucht der Registrator das erfolgreich migrierte Objekt wieder in die Registratur zurückzustellen.

5.3. Einführung und Implementation des Versionable Objektes

Zur Identifikation der Version existiert eine Versionsbeschreibung, die wir als „versionable“ implementiert haben. Das Versionable-Objekt (siehe CRC Karte “Versionsinformation”), welches die zur Versionierung notwendigen Informationen für die Materialklasse beinhaltet ist über eine Aggregation an die Klasse „RegisterableImpl“ angefügt. Da jede zu registrierende Klasse von „Registerable“ abgeleitet wird, verfügt somit auch jedes registrierbare Objekt über ein „Versionable“.
Da die Information, die das Versionable-Objekt trägt für jede Materialklasse nur einmal benötigt wird, ist die Anbindung an die Materialklasse als statisches Attribut über statische Zugriffsmethoden gelöst.

```
public static void setVersionable (Versionable versionable)
 {
    _versionable = versionable;
 }

public static Versionable getVersionable ()
 {
    return _versionable;
```

```
}
```



Klassendiagramm des „Versionables“

Das Versionable-Objekt enthält folgende Getter- und Setterfunktionen der Versionsnummer.

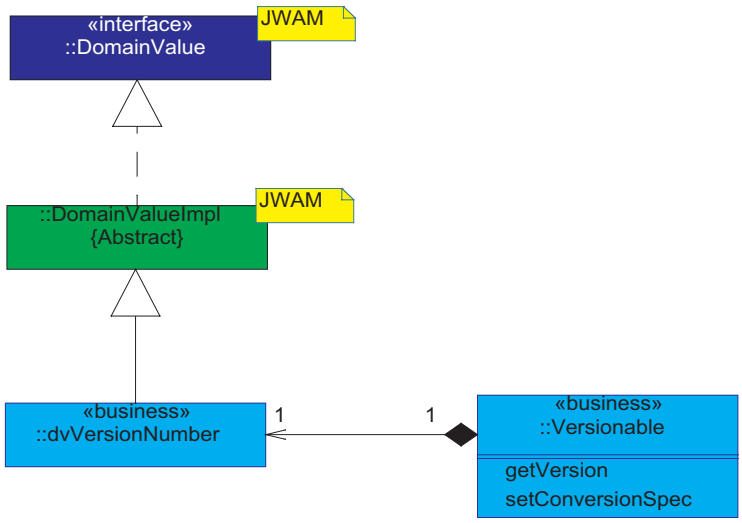
```
public dvVersionNumber getVersion(Registerable reg)
{
    try
    {
        retrieveVerNrFromPackageName(reg.getClass());
    }
        catch (MissingVersionNumberException e)
    {
        System.out.println(e.getMessage());
    }
    return _dvVersionNr;
}

public dvVersionNumber getVersion()
{
    Contract.require(_dvVersionNr != null, this, "_dvVersionNr != null");
    return _dvVersionNr;
```

```
 }

protected dvVersionNumber setVersion(dvVersionNumber version)
{
    _dvVersionNr = version;
    return _dvVersionNr;
}
```

Die Versionsnummer ist als DomainValue (Fachwert in JWAM) realisiert.
In JWAM werden Fachwerte als Klassen implementiert, die eine Wertesemantik besitzen. In vielen Fällen ersetzen Fachwerte einfache skalare Datentypen (z.B. Integer oder Float), da sie anwendungsfachliche Operationen bereitstellen (siehe [Mül1999]). Für den Fachwert "Versionsnummer" können so geeignete Methoden zum Vergleich implementiert werden.

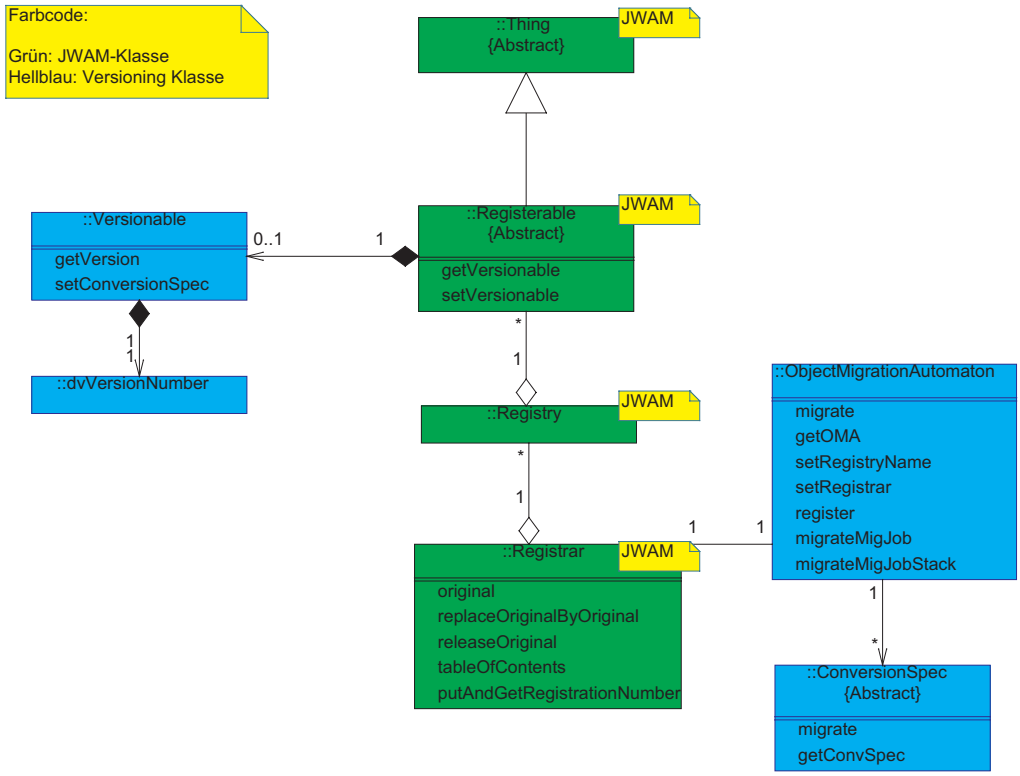
Das Versionable beinhaltet weiter die Information, welche ConversionSpec-Klasse zu dem ihm zugehörigen Objekt zuzuordnen ist.

```
public String getConversionSpec()
{
    return _strConversionSpec;
}

public void setConversionSpec(String strConversionSpec)
{
    _strConversionSpec = strConversionSpec;
}
```



Klassendiagramm des Fachwertes „dvVersionNumber"

Wir stellen hier zunächst das Klassendiagramm der oben aufgeführten Klassen dar, um im Anschluß auf die Implementation, ihre Besonderheiten und die damit verbundenen Probleme einzugehen.



Klassendiagramm der Versionierungssubframeworks

5.3.1. Versionable - Aggregation an Registerable

Da das Versionable-Objekt an „Registerable“ aggregiert ist, benötigen wir am Registerable-Interface entsprechende Methoden, die Zugriff auf das Versionable-Objekt ermöglichen. Diese haben wir folgendermaßen implementiert:

```
public static void setVersionable (Versionable versionable)
{
    _versionable = versionable;
}

public static Versionable getVersionable ()
{
    return _versionable;
```

```
}
```

5.4. Implementation der ConversionSpec

Die Migrationsanweisung stellt die Komponente für die Durchführung der Migration von Exemplaren einer bestimmten Klasse dar. Wir haben diese als abstrakte Klasse implementiert und diese „ConversionSpec“ genannt. Der Programmierer muß von ihr ableiten, um so die zu einer seiner Materialklasse zugehörige Migrationsbeschreibung zu erzeugen.

```
public abstract class ConversionSpecImpl extends ThingImpl implements ConversionSpec
```

Die Klasse „ConversionSpec“ besitzt vorgegebene Methoden (insbesondere Getter- und Setterfunktionen), die der Entwickler für seine spezifischen ConversionSpec Klassen benutzen oder auch überschreiben kann. Eine Methode zum Kopieren von Werten des alten Objektes in Attribute des neuen Objektes ist bereits vollständig implementiert.
Die „ConversionSpec“ muß zur Migration des Objektes eine programmiersprachliche Hürde meistern. In Java werden Attribute einer Klasse üblicherweise mit dem Zugriffschlüssel „private“ gekennzeichnet.
Das sind Attribute, welche nur durch ihre eigene Klasse gesetzt bzw. gelesen werden können. Die Attribute sind also vollständig gekapselt. Sofern man ein Attribut als „private“ kennzeichnet, will man keinen Zugriff auf das Attribut von außerhalb der Klasse. Um aber die Objekte vollständig zu migrieren muß die ConversionSpec auch diese Attribute lesen und schreiben können.
In JWAM selbst findet sich hierzu eine geeignete Lösung.
Mit Hilfe des Interfaces „Inspectable“ können private Attribute von Objekten gelesen und geschrieben werden.
Das Interface „Inspectable“ verwendet Core-Reflections, um private Attribute von Klassen lesen und schreiben zu können. Die Core-Reflection ist die Implementation im JDK, welche sich im Package „java.lang.reflect“ befindet.
Ob das Lesen des Wertes des alten Objektes und das Schreiben des Wertes in das Attribut des neuen Objektes möglich ist, wird in der if-Abfrage innerhalb der „copyAttribute“ Methode abgefragt, welche nachfolgend dargestellt ist.

```
public void copyAttribute(Registerable regSource, Registerable regTarget, String strSourceField,
String strTargetField) throws MigrationCopyFieldException
{
    Contract.require(regSource != null && regTarget != null, this,
    "regSource != null && regTarget != null");
```

```
        Contract.require(strSourceField != null && strTargetField != null, this,
        "strSourceField != null && strTargetField != null");

        if(regTarget.hasAttribute(strTargetField) && regTarget.canWriteAttribute(strTargetField)
        && regSource.hasAttribute(strSourceField) && regSource.canReadAttribute(strSourceField))
        {
            regTarget.writeAttributeValue(strTargetField, regSource.attributeValue(strSourceField));
        }
    }
```

Die Klasse “ConversionSpec” besitzt ein Attribut “dependencies”, das eine Abhängigkeit zu einer anderen “ConversionSpec” anzeigt. Dieses Attribut kann vom Objektmigrationsautomaten ausgewertet werden.

5.4.1. Das Singleton-Problem

Die Klasse „ConversionSpecImpl“ enthält die Migrationsbeschreibung für Objekte einer bestimmten Klasse. Das bedeutet, daß für alle Objekte einer bestimmten Materialklasse nur ein „ConversionSpec“- Exemplar existieren muß, um die Migration aller Objekte durchführen zu können.
Eine herkömmliche Klasse hat die Eigenschaft, daß man von ihr beliebig viele Exemplare erzeugen kann. Diese Eigenschaft ist bei den „ConversionSpec“-Klassen aber nicht erwünscht. Es sollte nur ein Exemplar erzeugt werden, welches vom ObjectMigrationAutomaton genutzt wird.
Die Lösung dieses Problems bietet ein Entwurfsmuster (siehe [Gam1996]), das sogenannte Singleton-Muster.
Dieses Entwurfsmuster stellt sicher, daß es von der Klasse nur ein einziges Exemplar (das Singleton) erzeugt wird.

Der Zugriff bzw. die dynamische Erzeugung des Objektes wird nicht direkt vorgenommen, sondern geschieht über eine vorgeschaltete Instanz, eine Registratur für „ConversionSpecs“. Diese Registratur ist nicht mit der JWAM-Registratur aus dem Package „de.jwamx.handling.registry“ zu verwechseln. Die hier genannte Registratur verwaltet nicht Materialien, sondern Migrationsbeschreibungen („ConversionSpecs“).

Diese Registratur sorgt dafür, daß wann immer ein Exemplar einer bestimmten „ConversionSpec“ benötigt wird, das Exemplar nur dann erzeugt wird, wenn noch keines vorhanden ist. Ansonsten wird immer eine Referenz auf das eine bereits bestehende Exemplar (eben das Singleton) zurückgegeben. Dies kann z.B. durch eine statische Member-Variable „_exemplar“ erreicht werden, die ein Singleton-Exemplar hält. Sofern noch keines vorhanden war, wird es mit „New Singleton“ initialisiert, bzw. die Referenz auf das Singleton zurückgeliefert.

Wir benötigen zugleich mehrere Singletons, da wir für jede versionierte Klasse ein „ConversionSpec“-Exemplar bereitstellen müssen. Jedes „ConversionSpec“-Exemplar wird daher am „ObjectMigrationAutomaton“ registriert.

Wir haben diese Registratur, im folgenden „ConversionSpecRegistry“ genannt und im “ObjectMigrationAutomaton” implementiert, da er auch diejenige Komponente darstellt, die mit den „ConversionSpec“-Exemplaren interagiert.

Die Methoden der Registratur für „ConversionSpec“, sind folgendermaßen implementiert:

„register“ ist die Methode, um ein „ConversionSpec“ zu registrieren. Die Methode benutzt als Hilfsmittel eine HashMap, um darin die „ConversionSpecs“ aufzunehmen. Die “register”-Methode bekommt dazu als Parameter den Namen der “ConversionSpec” sowie eine Referenz auf die “ConversionSpec”.
Die „register“-Methode überprüft zunächst, ob das „ConversionSpec“-Exemplar bereits registriert ist. Falls nicht, wird das Exemplar in die HashMap (Singleton-Map) eingetragen.

```
public static void register(String strConversionSpec, ConversionSpecImpl ConversionSpec)
{
        Contract.require(strConversionSpec != null, "strConversionSpec != null");
        Contract.require(ConversionSpec != null, "ConversionSpec != null");
        Contract.require(!_singletonMap.containsKey(strConversionSpec),
            "!_singletonMap.containsKey(strConversionSpec)");
        Contract.require(!_singletonMap.containsValue(ConversionSpec),
            "!_singletonMap.containsValue(ConversionSpec)");

        _singletonMap.put(strConversionSpec, ConversionSpec);
        _ConversionSpec = ConversionSpec;

        Contract.ensure(!_singletonMap.isEmpty(), "!_singletonMap.isEmpty()");
        Contract.ensure(_singletonMap.containsKey(strConversionSpec),
            "_singletonMap.containsKey(strConversionSpec)");
        Contract.ensure(_singletonMap.containsValue(ConversionSpec),
            "_singletonMap.containsValue(ConversionSpec)");
}
```

Um eine Referenz auf ein „ConversionSpec“-Objekt zu bekommen, kann auf die Methode „retrieveConversionSpec()“ zugegriffen werden.

```
protected void retrieveConversionSpec()
{
    _strConversionSpec = ((RegisterableImpl)_registerable).getVersionable().getConversion-
Spec();
     search(_strConversionSpec);
```

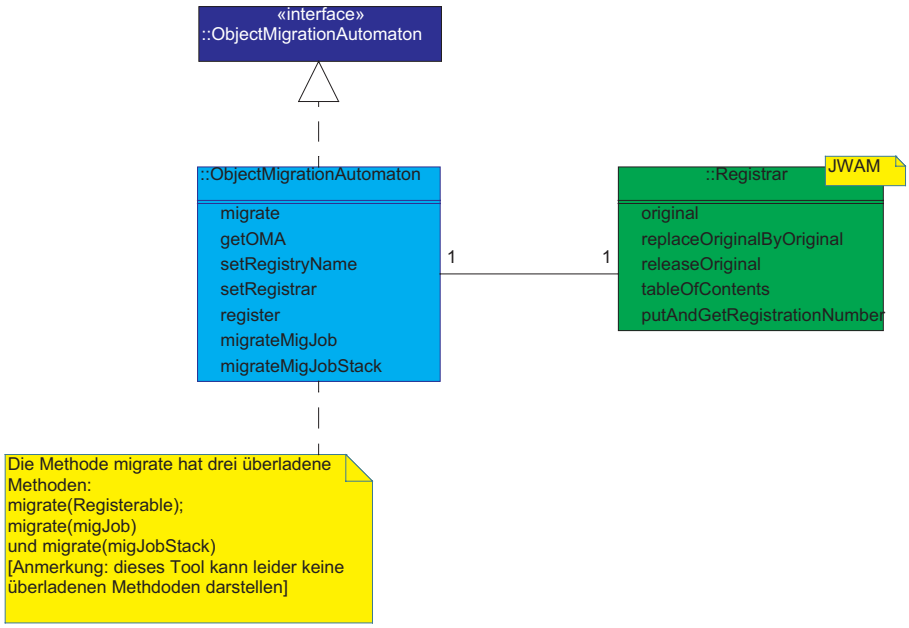
```
}
```

Um gegebenenfalls eine Überprüfung durchzuführen, ob ein „ConversionSpec“ schon registriert ist, kann von außen auf die unten dargestellte, öffentliche Methode „isRegistered(strConversionSpec)“ zugegriffen werden.

```
public static boolean isRegistered(String strConversionSpec)
{
    Contract.requireNotNull(strConversionSpec, "strConversionSpec != null");
    return _singletonMap.containsKey(strConversionSpec);
}
```

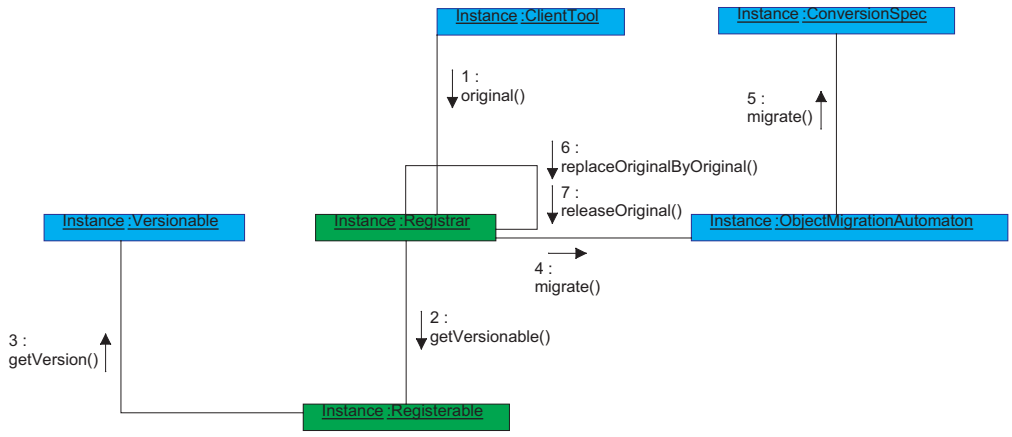
5.5. Implementation des Object Migration Automaton

Die Verwaltung des Migrationsvorganges, sowie der Migrationsbeschreibungen übernimmt eine Instanz, die wir als „ObjectMigrationAutomaton“, kurz „OMA“ implementiert haben. Diese Komponente haben wir als Automat realisiert, weil sie auch in der Lage sein muß, selbständig zu agieren. D.h. Aufträge an den Automaten zur Migration müssen vom Automaten zu einem bestimmten Zeitpunkt angestoßen werden können. Dies ist z.B. sinnvoll bei der Migration „at once“, die möglichst zu Zeiten zu denen das System nicht stark frequentiert wird durchgeführt, werden soll. Der OMA stellt eines der Kernstücke der Implementation des VSFW dar. Er ist für den korrekten Ablauf der Migrationen von Objekten verantwortlich.



Klassendiagramm des ObjectMigrationAutomaton

Im folgenden Collaboration-Diagramm kann man gut erkennen, das strukturell zwischen dem „Versionable“ und „ConversionSpec“ Objekt keine direkte Bindung, aber eine konzeptionelle Beziehung besteht. Weiter läßt sich erkennen, daß der OMA immer zwischen dem Registrator und dem ConversionSpec steht, da er letztendlich die Kontrolle über die Objektmigration behält.



Collaboration Diagramm des Versionierungssubframeworks

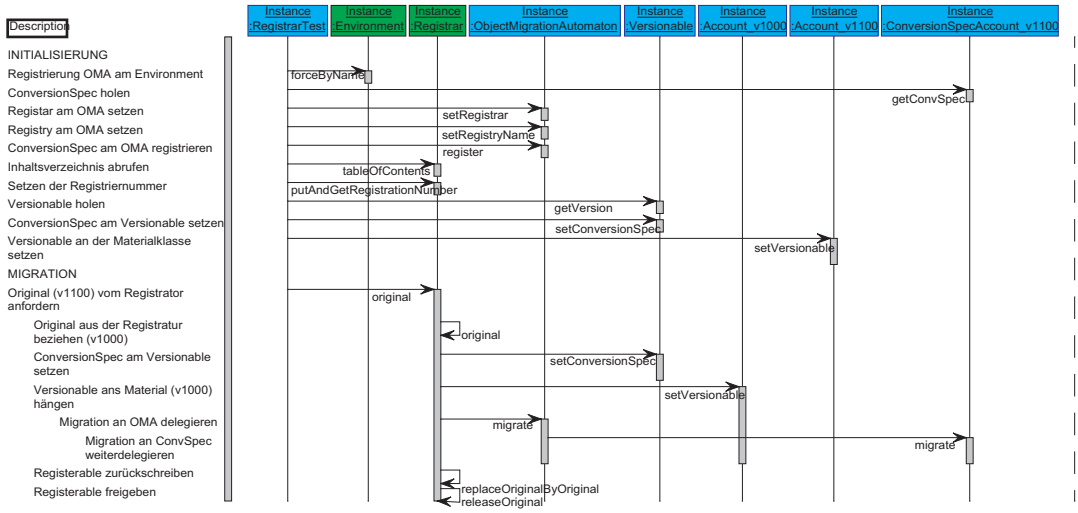
1. Der Client fordert ein Original vom Registrator an.
2. Der Registrator erhält das Versionable-Objekt vom registrierbaren Material.
3. Der Registrator bezieht die Versionsnummer vom Versionable-Objekt und stellt ggf. einen Versionskonflikt fest.
4. Der Registrator delegiert die Migration an den Objektmigrationsautomaten.
5. Der Objektmigrationsautomat führt die Migration mit Hilfe der ConversionSpec durch und gibt das migrierte Objekt an den Registrator zurück.
6. Der Registrator ersetzt das alte Objekt durch die neue Version, welche somit zum Original wird.
7. Der Registrator gibt das neue Original frei. Damit kann dieses Objekt wieder ausgeliehen werden.

Migration „on demand“

Der OMA muß die Fähigkeit besitzen, von der Registratur ein „Registerable“ entgegenzunehmen und von der Migrationsbeschreibung („ConversionSpec“) des Objektes ein Exemplar zu erzeugen. Hierbei ergibt sich ein Problem, welches wir weiter oben als „Das Singleton-Problem“ behandeln.

Gehen wir hier zunächst davon aus, daß der OMA von dieser Migrationsbeschreibung („ConversionSpec“) ein Exemplar (das Singleton) anlegt, bzw. eine Referenz besitzt. Sofern er einen Migrationsauftrag erhält, fragt er dann die Information aus dem Attribut „dependencies“ ab.
Sofern die Migrationsbeschreibung Abhängigkeiten zu anderen Klassen bzw. deren Objekten beschreibt, ist der OMA (im Prototypen) gezwungen die Migration abzubrechen, da er logische Verknüpfungen nicht generisch auflösen und bearbeiten kann.
Sollten keine Abhängigkeiten beschrieben sein, fährt der OMA mit der Migration des Objektes fort. Objektgeflechte werden nur bei der Migration „at once“ von unserem Prototypen berücksichtigt. Objektgeflechte müssen bei einer Migration nach dem “on-the-fly”-Prinzip gesondert behandelt werden (Siehe “Komplexe Materialien in der JWAM Registratur” auf Seite 22.). Für eine korrekte Implementierung bei der „on demand“-Migration müßte ein generisches Schlüsselkonzept (wie z.B. bei den “Smart-References”; Siehe “Komplexe Materialien in der JWAM Registratur” auf Seite 22.) erstellt werden. Mit Hilfe dieses Schlüsselkonzeptes könnte der OMA Abhängigkeiten in Objektgeflechten erkennen und auflösen.
Die Migration des einzelnen Exemplars übernimmt dann die „ConversionSpec“ selbst. Hierzu übergibt der OMA das Objekt an die zugehörige „ConversionSpec“ und erhält eine migrierte Version zurück. Dieses Objekt kann dann unter der alten Registrie-

rungsnummer in die Registratur des JWAM zurückgeschrieben werden.



Sequenzdiagramm einfacher Migrationsvorgang

Migration „at once“

Die Migration eines einzelnen Exemplares „on demand“ ist für die meisten Exemplare, eben diejenigen, welche Abhängigkeiten besitzen, nicht ausreichend. Da wir das Auflösen von Abhängigkeiten nicht generisch durchführen können, haben wir eine Alternative für die Migration von Exemplaren mit Abhängigkeiten implementiert.

Dieses beinhaltet die Implementation der Klassen „MigrationJob“ (Migrationsauftrag) und „MigrationJobStack“ (Migrationsauftrags-Sammlung).
Die Klasse „MigrationJob“ enthält den Auftrag zur Migration von Objekten einer bestimmten Klasse. Der Typ der zugehörigen Klasse kann entweder gleich im Konstruktor als Parameter angegeben, oder über eine gesonderte Methode gesetzt werden. Im folgenden zeigen wir beide Möglichkeiten.

```
public MigrationJob (Class type)
{
    Contract.requireNotNull(type,"type != null");
    _classType = type;
    retrieveTocRegisterables();
};

public void setType (Class type)
{
    Contract.requireNotNull(type,"type != null");
    _classType = type;
    retrieveTocRegisterables();
```

```
};
```

„RetrieveTocRegisterables“ ist eine geschützte Hilfsfunktion, die am Registrar die Liste der zu diesem Typ gehörigen Objekte als Inhaltsverzeichnis abruft.

```
protected void retrieveTocRegisterables()
{
    if (_registrar == null)
    {
        getRegistrar();
    }
    try
    {
        _dvTocRegisterables = _registrar.tableOfContents(_classType);
    }
    catch (RemoteException e)
    {
        System.out.println("retrieveTocRegisterables: RemoteException");
    }
};
```

Um nicht einzelne „MigrationJobs“ manuell aufrufen zu müssen, haben wir eine zweite Klasse implementiert, welche eine Liste mit „MigrationJobs“ enthält.
Die Liste ist als Stack implementiert, weswegen wir die Klasse „MigrationJobStack“ genannt haben. Denkbar wäre auch eine Implementation als Queue.

Wir haben für jeden der Migrationswege entsprechende Beispiele programmiert, die diese Eigenschaften demonstrieren. Die zugehörigen Packages befinden sich im Verzeichnis „de.jwamalpha.handling.versioning.example“.

Die Packages für die einfache Migration „on demand“ heißen:

- „de.jwamalpha.handling.versioning.example.v_1000“
- „de.jwamalpha.handling.versioning.example.v_1100“

Bei diesen Packages handelt es sich um eine Erweiterung des Account-Beispiels aus dem JWAM-Rahmenwerk. Hier wird ein Konto-Werkzeug bereitgestellt, mit dem auf fachlichen Konto-Materialien ein- und ausgezahlt werden kann. Die Versionsnummern der Packages enthalten zwei verschiedene Entwicklungsstände des Account-Beispieles. Im Beispiel mit der Versionsnummer 1100 zeigen wir, wie ein Versionskonflikt behandelt wird. Im Package v_1000 befindet sich eine Hilfsklasse, die eine Anzahl von Konten der Version 1000 erzeugt. Diese Konten werden dann im Package v_1100 vom Registrator und dem Objektmigrationsautomaten migriert, sobald ein Versionskonflikt festgestellt wird.

Die Packages für die Migration der Objektgeflechte „at once“ heißen:

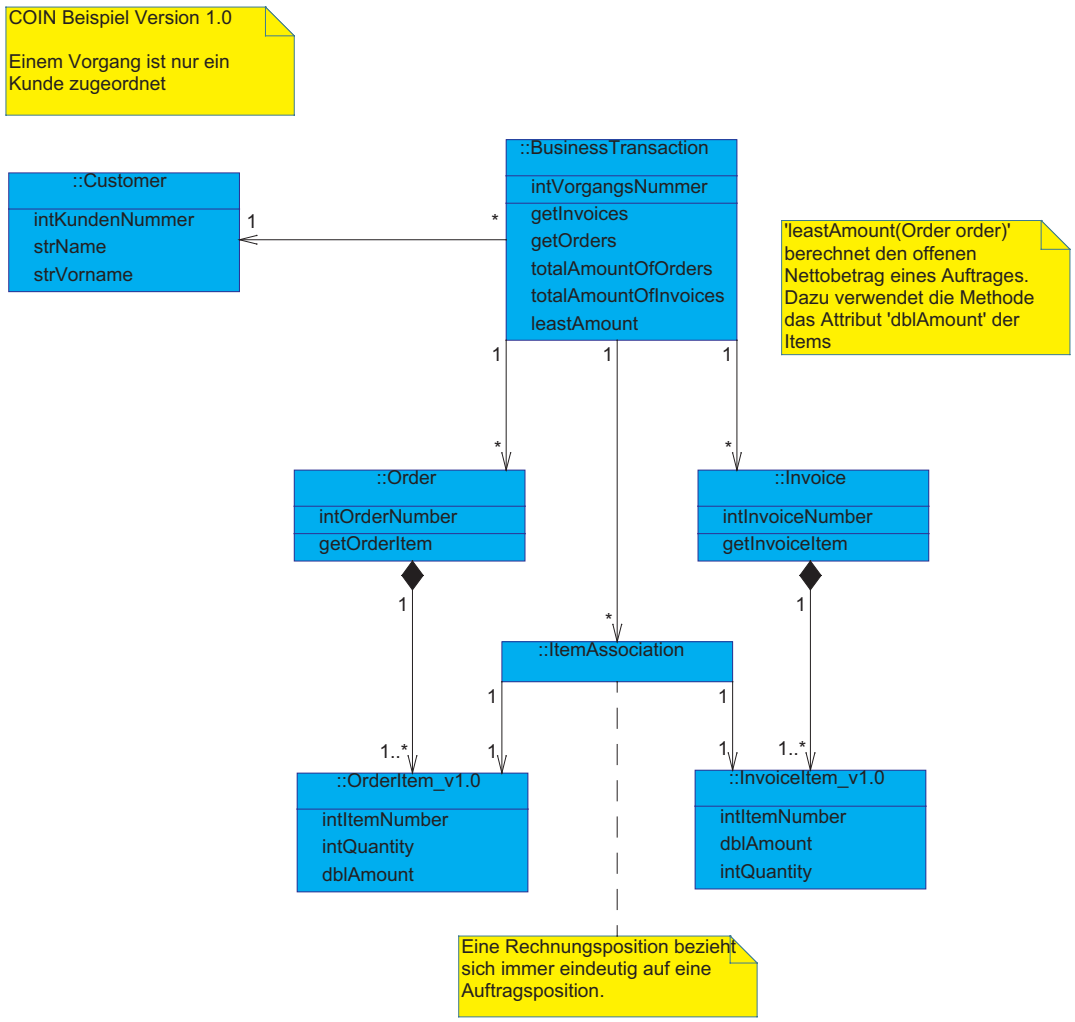
- „de.jwamalpha.handling.versioning.example.coin_v1000“
- „de.jwamalpha.handling.versioning.example.coin_v1200“

Mit diesen beiden Packages demonstrieren wir die Migration “at-once”. Das Beispiel zeigt die Behandlung von Objektgeflechten durch eine Massenmigration.

COIN (Customer-Order-INvoice)

Wir greifen hier das oben erwähnte Beispiel „Kunde-Auftrag-Rechnung“ auf, und haben dieses folgendermaßen implementiert.
Ein Vorgang (“BusinessTransaction”) kann mehrere Aufträge von einem Kunden zusammenfassen. Zu jedem Auftrag können eine oder mehrere Rechnungen existieren. Die Aufträge (“Orders”) und Rechnungen (“Invoices”) enthalten mehrere Positionen (“Items”) die über ein Beziehungsobjekt (“ItemAssociation”) verknüpft werden.

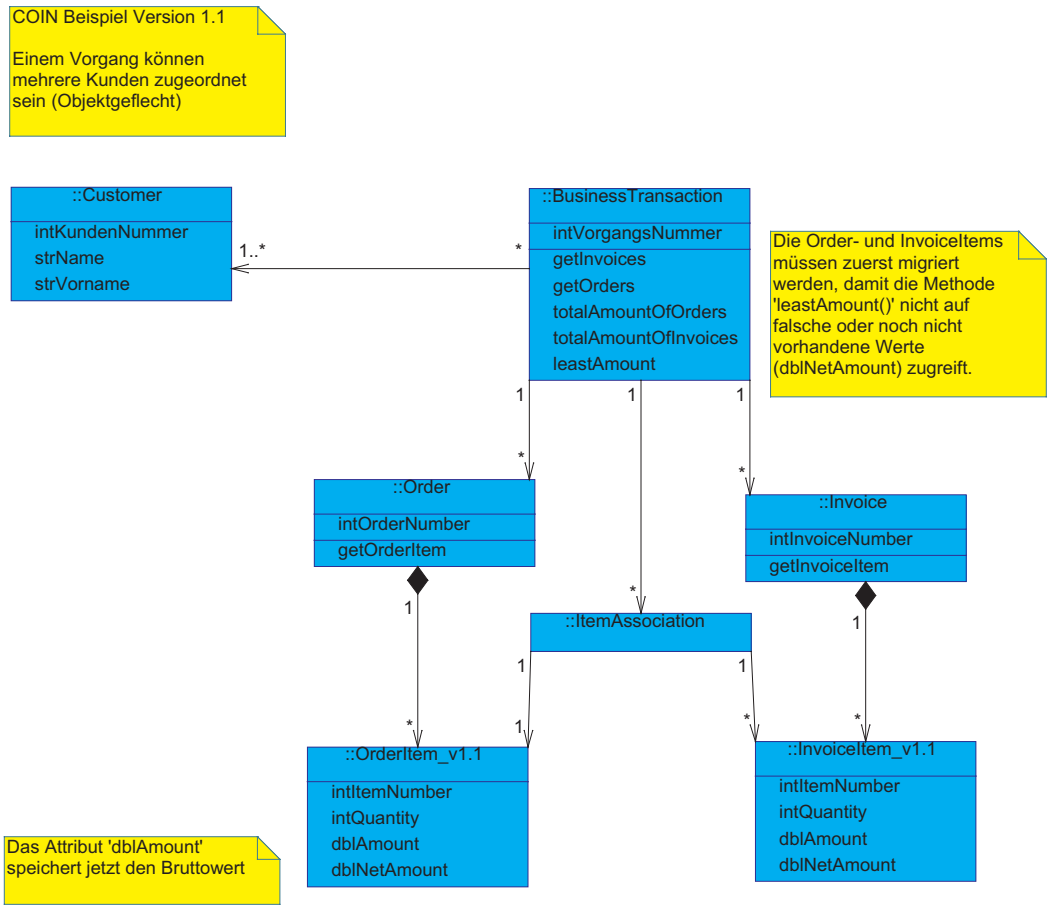
In der ersten Version des Beispiels besitzt die “BusinessTransaction” die Methode “leastAmount”, die den offenen Nettobetrag eines Auftrages berechnet. Dazu benutzt sie das Attribut “dblAmount” aus den Items.



COIN-Beispiel - Version 1.0

In der neuen Version (1.1) wurden die Klassen wie folgt geändert:
Die Items haben zusätzlich das Attribut “dblNetAmount” bekommen, welches den Nettobetrag speichert. Das Attribut “dblAmount” speichert nun im Gegensatz zur Vorversion den Bruttobetrag. Im Falle einer Migration müssen zuerst die “Item”-Objekte migriert werden, um zu verhindern, daß die Methode “leastAmount” der BusinessTransaction auf falsche oder nicht vorhandene Werte zugreift.

.



COIN-Beispiel - Version 1.1

Anmerkung: die Versionsnummern hinter den Item-Klassen dienen nur dazu, die Versionen im Modellierungswerkzeug zu differenzieren. Versionsnummern hängen sonst am Packagenamen.

Der OMA ist also gezwungen - sofern ein BusinessTransaction-Objekt migriert werden soll - das Objektgeflecht aufzulösen und die Item-Objekte als Erstes zu migrieren. Die Migration erfolgt entgegen der Richtung der Referenzierung wie im Bild angezeigt. Erst im Anschluß können die BusinessTransaction Objekte migriert werden und deren „totalAmount“ dabei entsprechend angepaßt werden.

Zu diesem Zweck müßte der OMA eine sequentielle Liste aufbauen, die er im Anschluß in umgekehrter Reihenfolge durchläuft. Das würde er nur dann tun, wenn er Objektgeflechte automatisch behandeln könnte. Im Prototypen wird dies jedoch von außen durch „MigrationJobStack“ gesteuert und funktioniert nur bei der Migration „at

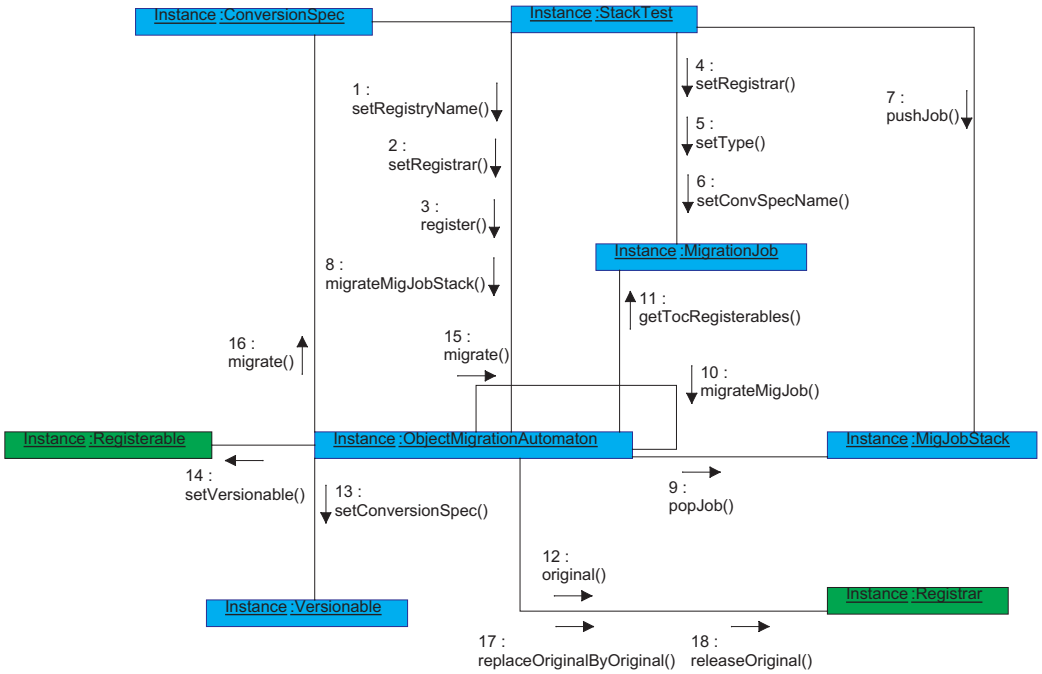
once“.

Aufbau des „MigrationStack“

Objektgeflechte kann der OMA nicht vollständig selbst auflösen. Die schon vorher festgestellt, bedarf es hier einer geeigneten Beschreibung des Geflechtes durch den Programmierer. Diese Information ist in der „ConversionSpec“ enthalten.

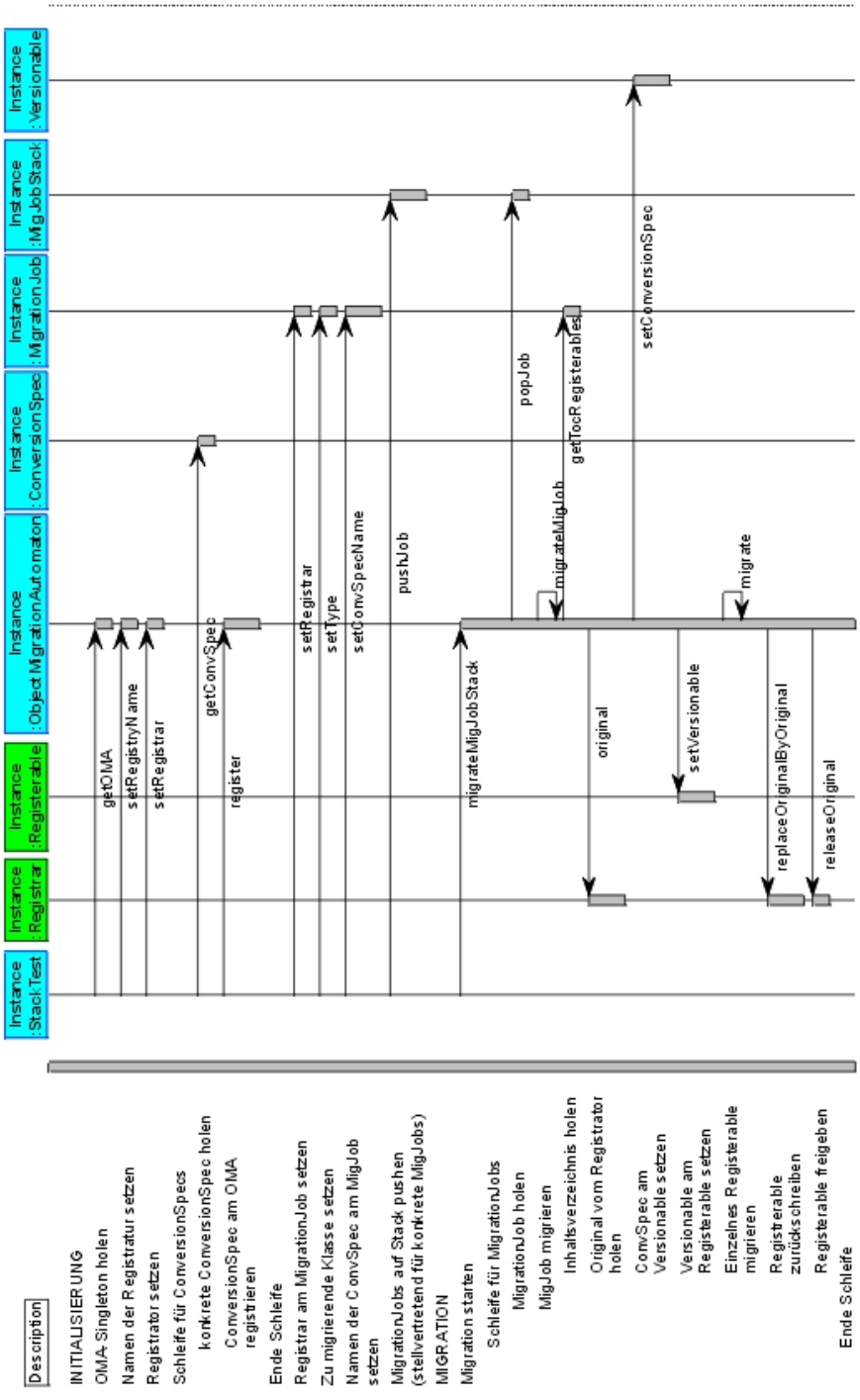
In der „ConversionSpec“ des Customers befindet sich ein entsprechender Eintrag auf die Orders-Objekte, bzw. auf die „CustomerRefNumber“ der Order-Objekte. Der OMA liest diese Referenz aus und sucht in der Registratur nach den entsprechenden Objekten, deren Registrierungsnummer im Anschluß in den „MigrationStack“ eingetragen werden.

Bei Lesen des „ConversionSpecs“ der Klasse BusinessTransaction, wird der OMA auf die Abhängigkeit zu den Item-Objekten stoßen. Dieser Beschreibung entsprechend sucht der OMA die zugehörigen Objekte und stellt auch diese in den Stack. Alternativ kann man auch über die Implementierung als „Queue“ nachdenken.
Das nachfolgende Collaboration-Diagram gibt einen Überblick über den Migrationsprozeß im COIN-Beispiel.



Collaboration Diagramm für den Versionierungsablauf

Das Sequenzdiagramm beschreibt detailliert den zeitlichen Ablauf der Methodenaufrufe während der Migration des Objektgeflechtes.



Sequenzdiagramm COIN-Beispiel

Weitere Materialien und Werkzeuge

Um das VSFW zu testen und die Funktionalität zu demonstrieren haben wir verschiedene Werkzeuge und zugehörige Materialien implementiert.

Account Generator

Der Account Generator ist ein kleines Werkzeug ohne GUI, welches eine vorgegebene Anzahl von Accounts generiert und bei der Registratur registriert. Es bedient sich hierzu der Klasse Account aus dem gleichnamigen JWAM Beispiel im Package „de.jwam.examples“.

ConversionSpec Tool

Das ConversionSpec Tool soll dem Anwendungsentwickler helfen, eine neue Version seiner Entwicklung mit ConversionSpec´s anzureichern. Hierzu wählt der Entwickler in dem Tool das zu versionierende Package aus. Im Anschluß kann er dann diejenigen Klassen auswählen, die sich geändert haben. Das Tool generiert entsprechende „ConversionSpec“-Vorlagen, die der Entwickler dann weiter anpassen kann.

Migration Tool

Das Migration Tool kann dem Client bei einer Massenmigration helfen. Der Client hat die Möglichkeit, über das Tool eine bestimmte Klasse von Materialien auszuwählen und diese „at once“ zu konvertieren.

Die beiden letztgenannten Werkzeuge sind nur ansatzweise implementiert.

Kapitel 6: Fazit und Ausblick

Wir haben uns in dieser Studienarbeit damit befaßt, die Grundlagen für ein Versionierungsframework zu erstellen und in einem Prototypen zu implementieren. Als Basis für unsere Arbeit diente dazu das JWAM Rahmenwerk, welches am Arbeitsbereich Softwaretechnik des Fachbereich Informatik der Universität Hamburg entwickelt wird. Eine Anforderung an das Versionierungsframework war die Integration in das bereits bestehende JWAM Rahmenwerk.

Zu Beginn der Arbeit haben wir uns mit den Grundlagen von Versionierung und Schemaevolution befaßt und die Zusammenhänge zwischen diesen beiden Begriffen herausgestellt. Als ein Ergebnis dieses Abschnittes haben wir die Begriffe “Erkennung”, “Benachrichtigung” und “Behebung” von Versionskonflikten herausgearbeitet.

Im nachfolgenden Abschnitt haben wir eine kurze Einführung in das JWAM-Rahmenwerk gegeben. Wir haben dann das Rahmenwerk auf passende Komponenten untersucht, welche die Aufgaben “Erkennung”, “Benachrichtigung” bzw. “Delegation” und “Behebung” übernehmen könnten. Für die Aufgaben, die nicht bereits durch das JWAM Rahmenwerk abgedeckt werden konnten, haben wir ein Konzept für ein Versionierungssubframework erstellt.

Das Konzept haben wir in einem weiteren Abschnitt soweit verfeinert, daß es durch eine überschaubare Anzahl von Klassen implementiert werden konnte.

Nachfolgend haben wir die Anwendung des Versionierungssubframeworks anhand von einem einfachen und einem komplexem Beispiel gezeigt, in dem auch Objektgeflechte behandelt werden. Die Beispiele zeigen, daß nicht alles generisch behandelt werden kann, sondern der Programmierer auch immer gefordert ist, eigenen Code hinzuzufügen.

Die Entwicklung dieses Versionierungsframeworks hat gezeigt, wie schwierig eine generische Lösung zu realisieren ist bzw. daß eine vollständig generische Lösung nicht möglich ist.

Der vorliegende Prototyp beinhaltet eine gewisse Grundfunktionalität, ist aber für eine marktreife Version noch stark zu erweitern.
Insbesondere müssen mehr generische Funktionen zur Migration in der Klasse „ConversionSpecImpl“ implementiert werden. Die Signaturen hierfür sind zu einem großen Teil schon vorbereitet.

Ein weiterer offener Punkt ist das Fehlen eines Werkzeugs zur Analyse von Abhängigkeiten zwischen Materialklassen, die Auswirkungen auf die Migration haben.

Eine Weiterentwicklung des Prototypen zu einer gebrauchsfähigeren Variante stellt bei dem vorliegenden Konzept keine technischen Probleme.

Literaturverzeichnis

[AtH1996] Attia, Karin H., Studienarbeit „Persistenz in objektorientierten Anwnedungssystemen: Über die Abbildung von Objektstrukturen auf relationale Datenbanken“, FB Informatik, Universität Hamburg, Oktober 1996

[BaH1996] Balzert, Heide: „Methoden der objektorientierten Systemanalyse“, Spektrum Verlag 2. Aufl.1996

[Bal1996] Balzert, Helmut: „Lehrbuch der Softwaretechnik“, Spektrum Verlag 1996

[BrG1996] Brammer, Ulrich und Göhmann, Markus „Ein Anforderungskatalog an die Verwaltung von Materialien in Einzelplatz-Arbeitsumgebungen aus fachlicher und technischer Sicht“, FB Informatik, Universität Hamburg, Juli1996

[Ec12000] Eckel, Bruce: „Thinking in Patterns with Java“, Revision 0.3, Internet-PDF http://www.bruceeckel.com/

[Ec22000] Eckel, Bruce: „Thinking in Java 2nd Edition“, Release 09, Prentice-Hall 2000, http://www.bruceeckel.com/

[Fow1999] Fowler, Martin: „Refactoring“, 3. Ausgabe, Addison Wesley 1999

[Gam1996] Gamma, Erich - Helm, Richard - Johnson, Ralph - Vlissides, John: "Entwurfsmuster", 1.Auflage, Addison Wesley 1996

[Hav1999] Havenstein, Andreas: „Unterstützung kooperativer Arbeit durch eine Software-Registratur“, Studienarbeit am FB Informatik, Arbeitsbereich Softwaretechnik, Universität Hamburg, 1999

[Heu1997] Heuer, Andreas: „Objektorientierte Datenbanken“, Addison-Wesley 2. Aufl. Bonn 1997

[Krü1999] Krüger, Guido: „Go To Java 2“, Addison-Wesley 1999

[Mül1999] Müller, Klaus: „Konzeption und Umsetzung eines Fachwertkonzeptes“, FB Informatik, Arbeitsbereich Softwaretechnik, Universität Hamburg, 1999

[Mey1997] Meyer, Bertrand: „Object Orientated Software Construction“, 2nd Edition, PrenticeHall 1997

[NGU1988] Nguyen, Gia-toan und Rieu, Dominique: "Schema Evolution in Object-Oriented Database Systems", Rapports de Recherche, Grenoble 1988

[Ram1998] Ramme, Kay: „Dynamisches Rebinden in kooperierenden peristenten Objektsystemen“, Diplomarbeit 1998, Universität Hamburg, Fachbereich Informatik

[Run1994a] Lei Zhou, Elke A. Rundensteiner, and Kang G. Shin: "Schema Evolution for Real-Time Object-Oriented Databases", CSE-TR-199-94, The University of Michigan, 1994

[Run1994b] Ra, Young-Gook and Rundensteiner, Elke A.: "A Transparent Object-Oriented Schema Change Approach Using View Evolution", CSE-TR-211-94, The University of Michigan, 1994

[Ska1987] Skarra, Andrea H., Zdonik, Stanlay B.:“Type Evolution in an Object-orientated Database“, erschienen in “Research Directions in Object-orientated Programming“, MIT-Press 1987, pp1-15

[Zel1996] Zeller, Andreas: „Configuration Management with Version Sets - A unified software versioning model and its application.“, Dissertation. Universität Braunschweig 1996.

[Zel1997] Zeller, Andreas: „Versioning Software Systems through Concept Descriptions“, Informatik-Bericht 97-01, techn. Universität Braunschweig 1997

[Zül1998] Züllighoven, Heinz: „Das objektorientierte Konstruktionshandbuch“, 1. Aufl., dpunkt Verlag, 1998

Glossar

Entwicklungsmodell, evolutionäres	Zyklische Abfolge von Analyse, Modellierung und Bewertung. Es gibt keine feste Reihenfolge von Entwicklungsaktivitäten. Alle Dokumenttypen sind im gesamten Projektverlauf bearbeitbar.
Erkennung, nominal	Jeder Version einer Klasse ist ein Versionsname zugeordnet. Es gibt einen Registrierungsmechanismus für Versionsnamen.
Erkennung, strukturell	Mit jeder Klassenversion wird eine Klassenbeschreibung gespeichert, die von der Struktur der Klasse abgeleitet wird.
Hypertextsystem	Retrievalsystem mit inhaltlichen Querverweisen (Links), die für den Anwender verborgen sind; Navigationsgestaltung. In einem Hypertextsystem hat der Benutzer anstelle einer sequentiellen Suche die Möglichkeit, sich relativ frei zwischen verschiedenen verwandten Themen zu bewegen.
Konfigurationsmanagementsystem	System zur Verwaltung von Softwarekomponenten im Software-Engineering.
Migration	Konvertierung vorhandener Objekte in das Schema einer neuen Version.
Migration, at-once	Alle Objekte werden auf einem Mal migriert.
Migration, en-masse	(siehe Migration, at-once)
Migration, on-demand	(siehe Migration, on-the-fly)
Migration, on-the-fly	Einzelne Objekte werden nach bedarf migriert.
Migrationsautomat	Ein Automat, welcher die Datenmigration ausführt.
OODBS	Objektorientiertes Datenbanksystem.
Persistenzmedium	Medium (z.B. Dateisystem, relationale Datenbank, objektorientierte Datenbank) daß Objekte dauerhaft speichern kann.
Prototyp	Ablauffähiges Modell ausgewählter Aspekte des zukünftigen Anwendungssystems. P. sind eine Diskussions- und Entscheidungsbasis für Entwickler und Anwender. Sie dienen ihnen zum Experimentieren und zum Sammeln von Erfahrungen. P. helfen so, relevante Spezifikations- oder Entwicklungsprobleme zu klären.
RDBS	Relationales Datenbanksystem.
Registrator	Eine Instanz in JWAM, die Zugriff auf die Registratur ermöglicht.
Registratur	Eine Verwaltungs- und Speicherungsinstanz für Materialien im JWAM-Rahmenwerk.
Schema	Begriff aus der Welt der relationalen Datenbanken. Beschreibt die Architektur einer Datenbank. In einem objektorientierten Kontext beschreibt das S. ebenfalls die Typen, die durch die Klassen definiert sind.
Schemaänderung	Änderung am Schema einer relationalen Datenbank oder eines objektorientierten Systems. S. führen häufig zu Versionskonflikten.
Schemaevolution	Weiterentwicklung des Schemas einer relationalen Datenbank oder eines objektorientierten Systems (siehe auch 'Schemaänderung').
Schlüsselreferenz	Zeiger auf eine Tabelle in einem RDBS (technischer Schlüssel) oder fachliche Referenz in einem OODBS.

Singleton	Entwufsmuster, welches sicherstellt, daß es nur ein einziges Exemplar einer Klasse gibt.
Version	Gefestigter Zustand einer Klasse zu einem bestimmten Zeitpunkt im Entwicklungsverlauf.
Version, Source	Version einer Klasse, deren persistente Objekte die Quelle des Migrationsprozesses bilden.
Version, Target	Version einer Klasse, in die der Migrationsprozess neue persistente Objekte abbildet.
Versionenhistorie	Folge von Versionen eines Objektes, die durch Schemaevolution auseinander hervorgegangen sind.
Versionenidentifikation	Problem der Bestimmung der Version eines versionierten Objektes
Versionskonflikt	Konflikt, der auftritt, wenn das Schema persistenter Objekte nicht mehr zum Klassenschema des lesenden Systems paßt.
Versionskonflikt, Benachrichtigung	Benachrichtigung von interessierten Adressaten über das Entstehen von Versionen, die Änderung von Versionen oder die Ausführung von bestimmten Operationen auf Versionen.
Versionskonflikt, Erkennung	Aufdeckung von Versionskonflikten explizit durch Vergleich von Versionsnummern (nominal) oder implizit durch Vergleich des Schemas (strukturell).
Versionskonflikt, Korrektur	Auflösung von Versionskonflikten durch Objektmigration. Die Automatische Migration wird ergänzt durch Null-Initialisierungen von neuen Attributen, oder durch Initialisierungen, die explizit durch den ändernden Programmierer in Form einer Richtlinie vorgegeben werden. Richtlinien sind wichtig um die Verletzung von Invarianten zu vermeiden.
Versionskonflikt, Delegation	Beauftragung einer dritten Instanz zur Korrektur des Versionskonfliktes.
Versionskontrolle	System zur Erkennung und Behebung von Versionskonflikten.
Versionsnummernmodell	Modell für die Struktur der Versionsnummer.