Eberhard Karls
Universität
Tübingen

Copyright © 2002 M E Leypold.
Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 published by the Free Software Foundation; with no invariant Sections, with no Front-Cover Text, and no Back-Cover Texts.

Since the license is rather long, I have chosen not to attach the license itself to this document. If this document is distributed (i. e. during a course) with other documents with the same license notice it suffices to distribute just one separate copy of the license. If on the other side this document is distributed alone, I require that a copy of the GNU Free Documentation License, Version 1.1, be attached.

Lösungsskizze 6 : Statische OO-Modelle mit UML

Inhaltsverzeichnis

1 Fingerübungen

Die hier vorgestellten Lösungen für die Fingerübungen sind z.T. ausführlicher als bei der Korrektur und Bewertung der Abgaben im Sommer 2002 erwartet.

1.1 Einleitende Bemerkung

Jede der Fingerübungen hebt auf einen bestimmten Aspekt der statischen Modellierung mit Klassen ab. Ich werde diesen Aspekt jeweils in einem kurzen Kommentar beschreiben.

1.2 Nummernkonto

Nummernkonto	
Nummer : String	{ 1 }
BLZ : String	{ 2 }
Guthaben : Integer	

1: Ein String von genau 10 Zeichen Länge, der nur aus Ziffern besteht.

2: Ein String von genau 8 Zeichen Länge, der nur aus Ziffern besteht.

Daten sind in der objektorientierten Modellierung immer in Objekte eingekapselt. Diese werden als Angehörige von Klassen beschrieben, wobei allen Objekten einer Klasse gemeinsam ist, dass sie Zustände aus derselben Zustandsmenge annehmen können, dass man ihnen Attribute des gleichen Namens und der gleichen Art zuschreiben kann, und auch, dass sie dieselben Botschaften in gleicher Weise verstehen, d. h. dieselben Methoden haben.

Klassendiagramme beschreiben Klassen, also im weiteren Sinn *Arten* von Objekten. Klassendiagramme beschreiben keine Einzelobjekte.

1.3 Nummernkonto mit Überziehungskredit

Nummernkonto	
Nummer : String	{ 1 }
BLZ : String	{ 2 }
Guthaben : Integer	
Kreditlimit: Integer	{ 3 }

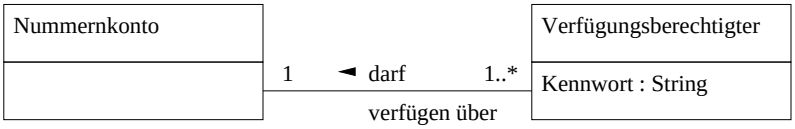
1, 2: Dieselben Beschränkungen wie im letzten Diagramm.

3: Guthaben $\geq$ Kredit.

Es ist notwendig, die erlaubten Zustände für Objekte einer gegebenen Art (Klasse) präzise zu bestimmen. Dazu gehört insbesondere, welche Zustände nicht auftreten dürfen bzw. können. Dies geschieht durch Bedingungen (Constraints), welche den Zustandsbereich einschränken, indem einzelne Attribute nur aus einem bestimmten Bereich kommen können (1, 2), oder die zugelassenen Werte von Attributen von anderen Attributen abhängig machen.

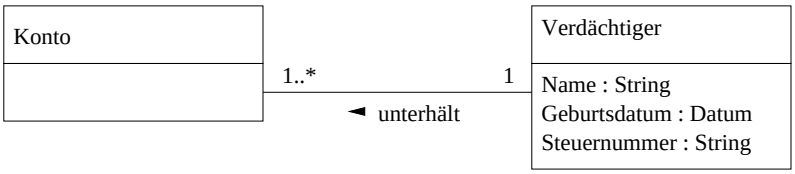
Der Wert einer solchen Beschränkung liegt vor allem darin, dass dadurch vereinbart wird, welche Fälle in der Implementation (von Methoden, welche Objekte dieser Klasse als Parameter nehmen) nicht behandelt werden müssen, da *zugesichert* ist, dass diese Fälle (etwa *Kredit* $\geq$ *Guthaben* gar nicht auftreten.

1.4 Verfügungsberechtigter



Dieses Beispiel illustriert, dass die Struktur des Objektmodells von der problemspezifischen Sicht geprägt wird: Zwar mag in der „Realität“[1] ein Kunde mehrere Kontos haben, diese Möglichkeit lässt sich jedoch aus der Sicht der Bank nicht prüfen, da die Bank Verfügungsberechtigte eben nur über das Kennwort identifizieren (gleichsetzen und unterscheiden) und die Datensätze (Objekte der Klasse *Kunde*) nicht einer „realen“ Person zuordnen kann. Aus diesem Grund kann sich die angesprochene Möglichkeit auch nicht in der Struktur der Software niederschlagen.

1.5 Die Sicht der Steuerfahndung



Dieses Diagramm ist eine weitere Demonstration, wie das Problem das Objektmodell prägt. Hier sind die Kardinalitäten andere, da es vor allem darum geht, alle Konten eines Verdächtigen zu identifizieren.

1.6 Buchungen

1: Für jedes Konto *K:Konto* muss gelten, dass *K.Guthaben* die Summe der Beträge *B.Betrag* aller zugeordneten Buchungen *B:Buchung* ist.

Dieses Beispiel zeigt eine *Aggregation* auch *Part-of-Relation* genannt, die „Zusammensetzung“ eines Objektes (oder eines Teils desselben) aus anderen. Graham (Graham u. a., 2001, S. 258) gibt eine Übersicht über die verschiedenen Interpretationen von Aggregation nach *Odell* (Literaturangabe im vorgenannten Werk), aber es scheint mir, dass Odells Definition in einigen Punkten ins „ontologische Modellieren“ abgleitet. Ich würde eine rein auf datenverarbeitenden Gesichtspunkten basierende Definition bevorzugen: Im vorliegenden Fall wird z. B. das Attribut *K.Guthaben* aus den Attributen *Betrag* der untergeordneten *Buchung*s-Objekte berechnet: Das *Konto*-Objekt ist also nur ein „Frontend“ für die Kollektion von *Buchung*s-Objekten, sein Zustand aus diesen „zusammengesetzt“ bzw. davon determiniert. In solchen Fällen würde ich von einer Aggregation Gebrauch machen wollen, anstelle einer Assoziation: Objekte, deren Zustand durch Assoziationspartner eingeschränkt wird, erscheinen mir als pathologischer Fall, den man vermeiden sollte.

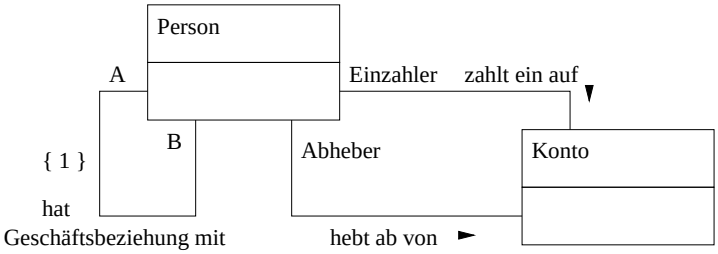
[1] Was immer das nun sein mag. Der Autor sieht sich selbst als radikalen Konstruktivisten, solche Leute müssen Wörter wie „Realität“ immer in Anführungszeichen setzen und können häufig nicht mit anderen Leuten kommunizieren, wenn der der naive Gebrauch von „ist“ (wie in „das *ist* aber GANZ ANDERS“) überhand nimmt. :-)

In der UML werden *Komposition* und *Aggregation* unterschieden. Hierzu möchte ich nochmal Graham (Graham u. a., 2001, S. 258) zitieren:

> Strictly in UML, aggregation and composition are represented by empty and filled diamonds respectively […] and represent programming language level concepts. In UML the empty diamond of aggregation designates that the whole maintains a reference to its part, so that the whole may not have created the part. This is equivalent to a C++ reference or pointer construction. The filled diamond signifies that the whole is responsible for creating its 'parts', which is equivalent to saying in C++ that when the whole class is allocated or declared the constructors of the part classes are called followed by the constructor for the whole (Texel and Williams, 1997). It is clear to me that this has little to do with the analysis of business objects; nor does the definition of composition in terms of ownership and lifetime constraints on instances (Rumbaugh *et al.*, 1999).[2] We continue to use the terms composition and aggregation interchangeably for the common sense notion of assembled components.

Genau so werde ich es im Folgenden halten.

1.7 Konten und Geschäftsbeziehungen



1: Zwei Personen *A:Person* und *B:Person* haben genau dann eine Geschäftsbeziehung miteinander, wenn es ein Konto *K:Konto* gibt, so dass *A* auf *K* einzahlt und *B* von *K* abhebt.

Dieses Beispiel illustriert nocheinmal eindringlich, dass die „Kästen" des Klassendiagrammes *Klassen* notieren, aber keine Objekte, die Assoziationslinien jedoch mögliche Relationen zwischen Objekten aus den beteiligten Klassen bedeuten.

So notiert hier die reflexive Assoziation „hat Geschäftsbeziehung mit", dass eine *Person* mit einer (möglicherweise) anderen *Person* eine Geschäftsbeziehung haben kann. Es wäre jedoch falsch, getrennte Einzahler- und Auszahlerklassen zu modellieren, den eine *Person* kann in beiden Rollen auftreten. Dafür sind dann in UML die Rollennamen da, welche an den Enden der Assoziation notiert werden können.

[2] A statement which the author of *this* paper can only meet with a: Full ACK! I've actually found the book of Graham and his collaborators a continuous source of critical remarks and constructive criticism on the current religion of UML and a fountain of interesting insights throughout. I can only recommend this book (and hardly any other one) to any serious student of OO-technology.

2 Statisches Modell eines Webshop

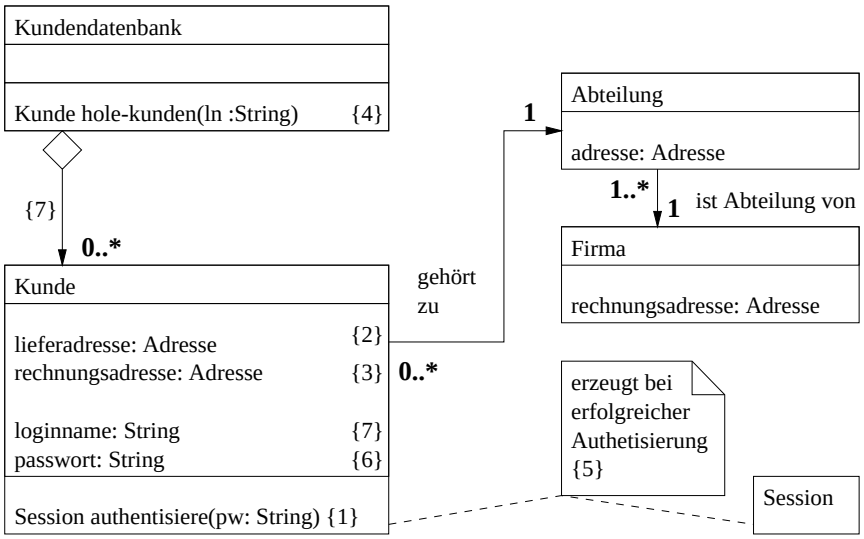
2.1 Einleitende Bemerkung

Das statische Modell des Webshops auf den folgenden Seiten wurde der besseren Übersichtlichkeit halber in vier Diagramme zerlegt, die sich nur in einigen Klassen (*Kunden*, *Warenkorb*, *Bestellung*, *Posten*), meist einer einzigen, überschneiden. Dies illustriert sehr schön, wie Klassendiagramme die Darstellung selektiver Aspekte des Gesamtsystem in verschiedenen Sichten erlauben. Dabei wurde von den Klassen auch jeweils nur die in der jeweiligen Sicht relevante Untermenge von Attributen dargestellt.

Das Gesamtdiagramm kann ohne Probleme zurückgewonnen werden, indem man die Teildiagramme an den gemeinsamen Klassen „übereinanderschiebt“, dabei die Vereinigungsmenge der Attribute der überlappenden Klassen und der Constraints bildet.

Viele datenverarbeitende Prozesse, die im fertigen System ablaufen sollen sind übrigens schon in den durch Constraints notierten statischen Abhängigkeiten festgehalten.

2.2 Kunden und Login



1, 7: Sei *k* :*Kunde*. Die Methode *k.authentisiere(s)* vergleicht den übergebenen *s* :*String* mit *k.passwort*, sind beide gleich, wird eine neue *Session* erzeugt, ansonsten ein Fehler signalisiert.

2: *self.lieferadresse* ist *self.Abteilung.adresse*.

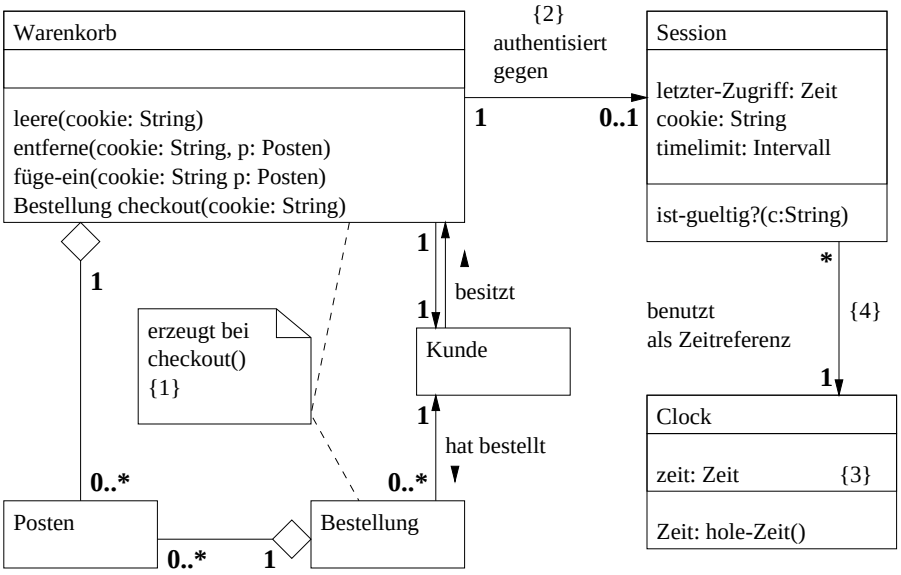
3: *self.rechnungsadresse* ist *self.Abteilung.Firma.rechnungsadresse*.

4: Die Methode *hole-kunden(ln)* sucht einen Kunden anhand des übergebenen Loginnamens *ln* :*String* aus der Datenbank heraus.

6: Darf nicht leer sein.

7: Für alle in der Kundendatenbank enthaltenen Kunden *k* :*Kunde* muss der Loginname eindeutig sein, d. h. sind *k1* :*Kunde* und *k2* :*Kunde* in *db* :*Kundendatenbank* enthalten, so muss $k1.loginname \neq k2.loginname$ sein.

2.3 Warenkorb und Bestellung



1: Sei *w* :*Warenkorb*. Beim Aufruf von *w.checkout(. . .)* wird eine Bestellung erzeugt, die nun die *Posten* aus *w* enthält. Der Warenkorb *w* wird geleert.

2: Alle Methodenaufrufe von *Warenkorb* enthalten einen Parameter *cookie* :*String*. Dieser wird dazu verwendet, um den Identität des Aufrufenden (des – indirekt – aufrufenden Browsers) gegen die offene (assoziierte) *Session* zu authentisieren.

3: Dieses Attribut gibt die aktuelle Zeit wieder, ändert sich also spontan von Aufruf zu Aufruf. Instanzen der Klasse *Clock* kapseln den Zustand der Systemuhr(en).

4: Die Uhr wird von Instanzen *s* :*Session* dazu verwendet, um Timeouts festzustellen: Ist seit dem letzten Zugriff, seit der letzten Authentisierung mehr als *s.timelimit* vergangen, so wird auf *ist-gueltig()* immer *false* geeantwortet, und *letzter-Zugriff* nicht mehr geändert. Das Attribut *s.timelimit* wird bei der Erzeugung von *s* festgelegt.

Bemerkung: Die Aufgabenstellung fordert, nur die statischen Verhältnisse zu modellieren. Wie dort erwähnt, eignen sich Klassendiagramme schlecht um Abläufe, d. h. dynamische Eigenschaften des Systems (oder von Teilsystemen (Ansammlungen von Objekten) zu dokumentieren.

Eine statische Eigenschaft ist eine Aussage, deren Wahrheitsgehalt (zu einem bestimmten Zeitpunkt) anhand eines Schnappschusses des Systemzustandes verifiziert werden werden kann. Tatsächlich lassen sich (korrekte Implementation von Methoden vorausgesetzt) daran auch dynamische Beziehungen zwischen Objekten ableiten. Damit ein Objekt ein anderes *verwenden* also Methoden des anderen Objektes aufrufen kann, muss es Kenntnis von der *Identität des Objektes* haben, sonst kann es keine Botschaften an dieses Objekt schicken. Es muss also eine Objektreferenz auf das andere Objekt halten bzw. dessen "*geheimen und einzig wahren Namen*" kennen. Dies Objektidentität kann das erstere Objekt eigentlich nur in seinem inneren Zustand speichern – und im Prinzip kann diese Tatsache an einem statischen Schnappschuss des Systemzustandes abgelesen werden.

Eine Beispiel ist dazu die unter $\{4\}$ notierte Assoziation "benutzt als Zeitreferenz" zwischen einer *Session* und einer *Clock*. Eine *s* :*Session* muss die Referenz auf die Ihr zugeordnete *c* :*Clock* (die *Identität* der *Clock*) in einem Attribut gespeichert haben. Dieses heisst

normalerweise so, wie die der an der Assoziation notierte Rollenname, per Default wie die am anderen Ende der Assoziation liegende Klasse, hier *Clock*.

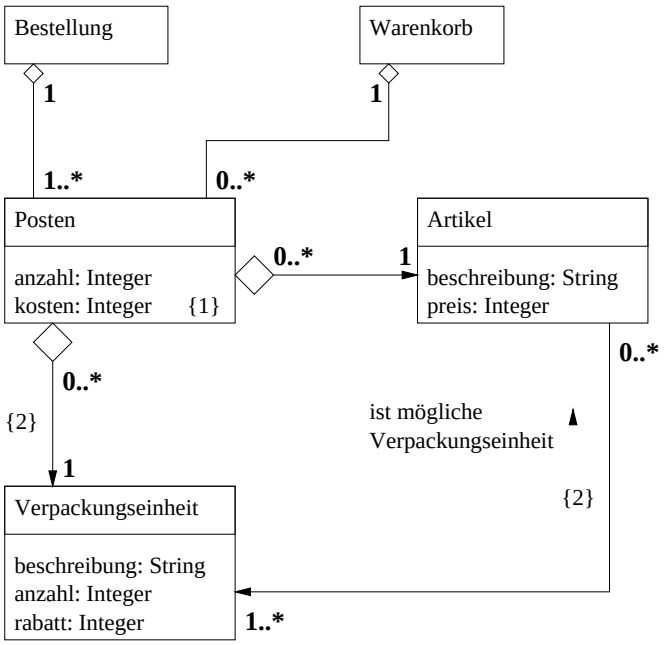
Verwendet also *s* :*Session* die Uhr *c* :*Clock*, so muss *s.Clock* = *c* sein. Diese Assoziation heisst *navigierbar*, da aus dem Zustand von *s* die zugeordnete Uhr *c* ermittelt werden kann.

Wohlgemerkt bedeutet eine Assoziation (zumindest in den ersten Phasen des Entwurfs) nicht immer, dass irgendwo eine Objektidentität gespeichert sein muss. Manchmal sind Assoziationen auch nur einfach als Relationen zu verstehen (und Ausdrücke, die solche Assoziationen enthalten müssen später schrittweise ersetzt werden durch äquivalente (und kompliziertere) Ausdrücke, die navigierbare Assoziationen enthalten, beziehungsweise den Aufruf von Methoden (meist: Methoden ohne Seiteneffekte). Direkte Benutzungsbeziehungen sind aber ohne Navigierbarkeit nicht denkbar und deshalb immer durch statische Inspektion zu entdecken – scheinen also im Klassendiagramm auf.

Von ganz anderer Art ist die Beziehung zwischen *Warenkorb* und *Bestellung*: Ein *Warenkorb* erzeugt zwar eine *Bestellung*, aber es besteht keine Notwendigkeit, dass eines der beiden Objekte Kenntnis davon behält, welche anderen Instanzen es erzeugt hat, bzw. wovon es erzeugt worden ist – und wir setzen das hier auch nicht voraus. Damit ist durch eine statische Inspektion (einen Schnappschuss des Systems) *nicht* feststellbar, dass beispielsweise *w* :*Warenkorb* das Objekt *b* :*Bestellung* erzeugt hat.

Diese Beziehung kann demnach im Klassendiagramm nicht mir Mitteln der Assoziation notiert werden. Stattdessen haben wir zu einer einfachen Notiz (7) gegriffen, die – zusätzlich zu den statischen Verhältnissen – kurz die dynamische Beziehung zwischen *Warenkorb* und *Bestellung* erläutert. Genauere Beschreibungen liefern dynamische Modelle und Diagramme, wie z. B. die Lösungsskizze zu Blatt 7.

2.4 Bestellung, Struktur eines Bestellpostens



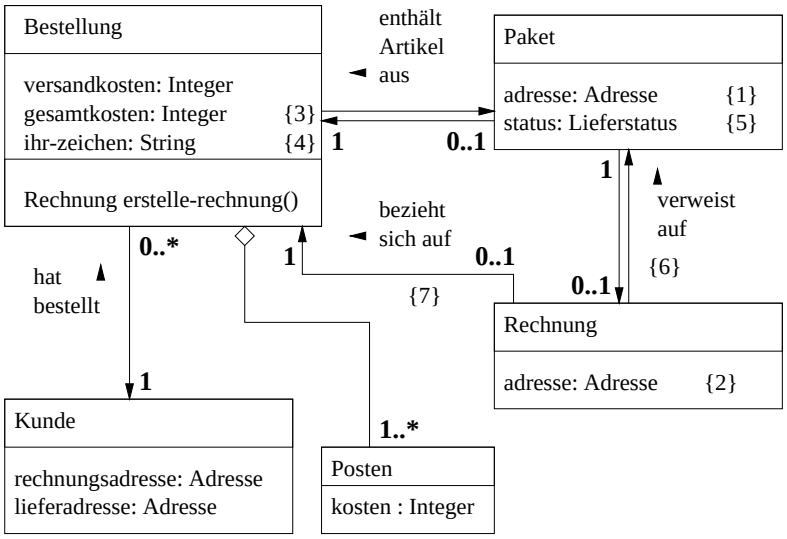
1: Es muss gelten:

$$\begin{aligned} &\textit{self.kosten} = \\ &\quad (\textit{self.anzahl} * \textit{self.Verpackungseinheit.anzahl} * \textit{self.Artikel.preis}) \\ &* \quad (1000\text{-}\textit{self.Verpackungseinheit.rabatt})/1000 \end{aligned}$$

Das bedeutet auch: Der Rabatt wird in Promille angegeben.

2: Für p :*Posten* muss *p.Verpackungseinheit* in *p.Artikel.Verpackungseinheit* sein.

2.5 Bestellung, Auslieferung und Rechnungsstellung



1: *self.adresse* = *Bestellung.Kunde.lieferadresse.*

2: *self.adresse* = *Bestellung.Kunde.rechnungsadresse.*

3: $\textit{self.gesamtkosten} = \textit{self.versandkosten} + \sum_{p \in \textit{self.Posten}} p.\textit{kosten}$.

4: Der Text *ihr-zeichen* muss beim Erzeugen der Bestellung (checkout) generiert werden und auf die Zeit der Bestellung, den Loginnamen des Kunden und den Webshop verweisen. Beispielsweise: “Web-Bestellung mr-cool 2002-09-08 1”. Es ist dieses Zeichen, auf das der Kunde bei telefonischen Reklamationen und Nachforschungen Bezug nehmen wird.

5: $\textit{Lieferstatus} \in \{\textit{IN-AUSLIEFERUNG}, \textit{ZUGESTELLT}\}$.

6: Einem Paket p :*Paket* darf nur dann eine Rechnung zugeordnet sein, wenn gilt:

$$p.\textit{Lieferstatus} = \textit{ZUGESTELLT}$$

7: *Rechnung.Bestellung* = *Rechnung.Paket.Bestellung*

3 Abschließende Notizen zur verwendeten Notation

Die verwendete Notation und Terminologie sind stark von der von Graham u. a. (2001) in *Object-oriented methods: principles & practice* beeinflusst. In fast allen Fällen und soweit sinnvoll wurde (das geht über das in der Aufgabe verlangte hinaus) die Navigationsrichtung eingezeichnet. Die Leserichtung für den beschreibenden Text der Assoziation wurde nur angegeben, wenn er verschieden von der Navigationsrichtung ist.

Die Constraint wurden in natürlicher Sprache und einer improvisierten mathematischen Notation beschrieben, die im Wesentlichen für jedermann sofort verständlich sein sollte. Es handelt sich dabei *nicht* um OCL (Object Constraint Language). *Self* notiert jeweils das Objekt, über das gerade gesprochen wird (also in dessen Kontext der jeweilige Constraint notiert wird). Dieser Gebrauch sollte mit dem in der OCL übereinstimmen.

Literatur

[Graham u. a. 2001] GRAHAM, Ian ; O'CALLAGHAN, Alan J. ; WILLS, Alan C.: *Object-oriented methods: principles & practice*. Erste Auflage. Addison-Wesley, 2001 (The Addison-Wesley object technology series). – ISBN 0-201-61913-X