Materialsammlung - Formale Methoden: OCL und vdm++

Prof. Dr. Hans-Jürgen Buhl

Wintersemester 2007/2008
Fachgruppe Mathematik und Informatik
FB C
Bergische Universität Wuppertal

Praktische Informatik
PIBUW - WS0708
Oktober 2007
3. Auflage, 2007
Praktische Informatik 02

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

CM.510 /
CM.943 /
CM.944

Formale Methoden Buhl
4 V Mo 12 - 14 D 13.08
Do 12 - 14 D 13.08
Einordnung: Master IT: MIT03; Master Mathematik; Diplom Mathematik; Nebenfächer und Studienschwerpunkte Informatik anderer Studiengänge

Inhalt: Formale Spezifikation von Softwaremodulen kann zu einer enormen Qualitätssteigerung in der Softwareentwicklung führen. Sie ermöglicht mit Hilfe von (mathematischen) Modellen, die (gewünschten) Eigenschaften von Programmen exakt zu definieren und überprüfbar zu machen.

(siehe auch Modulbeschreibung des entsprechenden Studiengangs)

CM.511 /
CM.945 /
CM.946

Übungen zu Formale Methoden Buhl
Ü Mi 16 - 18 D 13.08

Vorbemerkungen:

Formale Methoden

Inhalte:

1. Softwaregüte
2. Zusicherungen in Algorithmen:
   Konstruktoren, Modifikatoren,
   Observatoren und Destruktoren;
   Ausnahmebedingungen
3. Methodik *Programming by Contract*:
   Vorbedingungen, Nachbedingungen und Invarianten;
   Softwareanbieter/Softwarenutzer
4. EBNF zur formalen Spezifikation freier
   Eingabesprachen, UML-Klassendiagramme
5. Startwerte, Vererbung von Klasseninvarianten,
   Methodenvor- und -nachbedingungen
6. Formale Spezifikation (z.Zt. in OCL2):
   UML-Klassendiagramme und *Constraints*
   virtuelle Attribute und Methoden,
   redundante Attribte und Methoden;
   Constraints an Attribte und Methoden und
   Assoziationen; Container-Typen; Frame-Regeln
7. Fallstudien von formal spezifizierter
   Software (Algorithmen und Datenstrukturen)
8. Von der formalen Spezifikation zum (interpretierten) Prototyp

Modulziele:

Die Studierenden lernen formale Software-Modelle lesen, verstehen und kritisieren, um formale Methoden als ein Kommunikationsmittel der Teammitglieder eines Software-Entwicklungsteams schätzen zu lernen. Sie entwickeln mit Hilfe der formalen Spezifikation Teilsysteme von realistischen Softwaremodellen selbst.

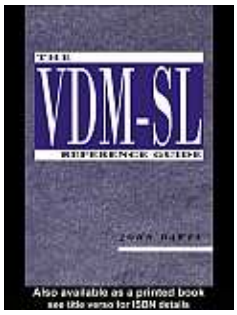
Buchempfehlungen:

Wolfgang Zuser
Software Engineering
Mit UML und dem Unified Process

Sprache: Deutsch
Gebundene Ausgabe - 464 Seiten
Pearson Studium
Erscheinungsdatum: Juni 2004

Auflage: 2., überarb. Aufl.
ISBN: 3827370906

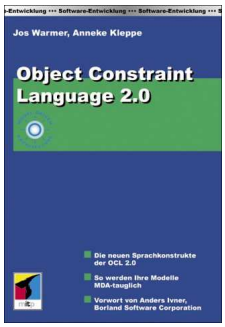
John Dawes
The VDM-SL Reference Guide



Sprache: Englisch
Taschenbuch - 200 Seiten
Routledge,an imprint of Taylor & Francis Books Ltd
Erscheinungsdatum: 12. August 1991

ISBN: 0273031511

Jos Warmer
Object Constraint Language2.0



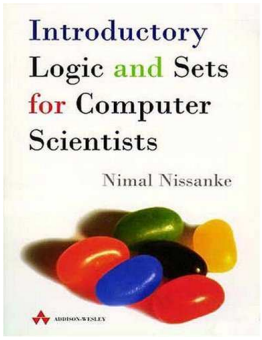
Sprache: Deutsch
Broschiert - 240 Seiten
Mitp-Verlag
Erscheinungsdatum: März 2004

Auflage: 1
ISBN: 3826614453

Nimal Nissanke
Introductory Logic and Sets for Computer Scientists.

Sprache: Englisch Broschiert - 400 Seiten
Addison Wesley
Erscheinungsdatum: Oktober 1998

ISBN: 0201179571



Heide Balzert
UML kompakt

Sprache: Deutsch
Spektrum Akademischer Verlag
Erscheinungsdatum: Juni 2005

ISBN: 3827410541

Harald Störrle
UML 2 für Studenten

Sprache: Deutsch
Pearson Studium
Erscheinungsdatum: Auflage: 1 Mai 2005

ISBN: 3827371430

John Fitzgerald
Validated Designs for Object-oriented Systems

Sprache: Englisch
Springer-Verlag London Ltd
Erscheinungsdatum: 1.Auflage 28. Februar 2005

ISBN: 1852338814

Martin Kreuzer, Stefan Kühling
Logik für Informatiker

Sprache: Deutsch
Pearson Studium
Erscheinungsdatum: März 2006

ISBN: 3827372151

FOLDOC - Free-On-Line-Dictionary-Of-Computing

http://wombat.doc.ic.ac.uk/foldoc/

Eine Suche bei FOLDOC zu **formal methods** und **specification** ergibt folgendes:

Formale Methoden / formal methods
<Mathematik Spezifikation> Mathematisch basierte Technick zur Spezifikation, Entwicklung und Verifikation von Software und Harware Systemen.

Spezifikation / spezifikation
<Jargon> Ein Dokument welches beschreibt, wie ein System arbeiten soll.

- **Formale Methoden**

  sind logikbasierte Technicken für die
  Spezifikation,
  Entwicklung und
  Verifikation
  von SW- und Hardwaresystemen.

- Die **Spezifikation** eines Systems ist ein Dokument, das beschreibt, wie das System arbeiten soll.

- **Beispiele für entsprechende Beschreibungen:**

  a) Eine Funktion kann **implizit** (durch Angabe von Eigenschaften) spezifiziert werden:

  $$\begin{array}{|l} max(s : \mathbb{N}_1\text{-set})m : \mathbb{N}_1 \\ \text{pre card } s \neq 0 \\ \text{post } m \in s \wedge \forall x \in s \cdot m \geq x \end{array}$$

  b) Eine Funktion kann **explizit** (durch Angabe eines Algorithmus) spezifiziert werden:

  $$\begin{array}{|l} max : \mathbb{N}_1\text{-set} \rightarrow \mathbb{N}_1 \\ max(s)\underline{\triangle} \\ \quad (\text{dcl } maxBisher : \mathbb{N}_1 := getFirstElement(s); \\ \quad \text{while } existsNextElement(s) \\ \quad \text{do let } n = getNextElement(s) \text{ in} \\ \quad\quad \text{if } n > maxBisher \text{ then } maxBisher := n; \\ \\ \quad \text{return } maxBisher \\ \quad) \\ \text{pre card } s \neq 0 \end{array}$$

- **Vor-/Nachbedingungen**

```
subSequence(lower:Integer, upper:Integer):Sequence(T)
pre: 1 <= lower
pre: lower <= upper
pre: upper <= self→size()
post: result→size() = upper − lower + 1
post: Sequence{lower..upper}→forAll( index |
            result→at(index − lower + 1) =
                        self→at(index))
```

- **Beispiel zu Spezifikationsmängeln:**

  Euro-Panne bei der Deutschen Bank 24 (Update)

  Geldautomaten der Deutschen Bank 24 müssen sich wohl an den Euro erst noch gewöhnen. Wer Anfang Januar Euro-Beträge von Geldautomaten dieser Bank bezogen hat, durfte sich am heutigen Freitag wundern, dass ihm die Bank das 1,95-fache vom Konto abgebucht hat. Offensichtlich haben die Bank-Computer an Stelle der maßgeblichen Euro-Summe irrtümlich mit dem Zahlenwert des umgerechneten DM-Betrags gerechnet.

  Verunsicherte Kunden erfuhren zunächst nur, dass sogar die Angestellten der Bank dem Problem zum Opfer gefallen sind. Mit der Hoffnung auf hilfreichere Informationen mussten sie sich jedoch vorerst gedulden. Erst gegen elf Uhr konnten die Ansprechpartner an der Telefonhotline für etwas Beruhigung sorgen: "Das Problem ist bekannt, die falschen Buchungen werden automatisch zurückgezogen und korrigiert".

  Inzwischen fand die Bank heraus, dass bei einem nächtlichen Datenverarbeitungslauf einige Tausend der insgesamt etwa 1,5 Millionen angefallenenen Kontobewegungen durch einen Programmfehler falsch bearbeitet worden sind. Theoretisch hätten zwar auch herkömmliche Barabhebungen am Bankschalter betroffen sein können, doch zufällig drehte es sich bei den fehlerhaften Buchungen tatsächlich nur um Abhebungen von Geldautomaten, hieß es bei der Deutschen Bank 24. Das erklärt auch, warum bei anderen Banken, die gebührenfreies Abheben von denselben Geldautomaten wie die Deutsche Bank 24 ermöglichen, keine vergleichbaren Fehler aufgetreten sind.

  Markus Block, Sprecher der Deutschen Bank 24, erklärte gegenüber heise online, alle falschen Buchungen würden bis zum Samstag korrigiert sein, sodass kein Kunde finanzielle Nachteile zu erwarten habe. (hps/c't)

  Link zu diesem Artikel bei heise-online:
  http://www.heise.de/newsticker/meldung/23747

- **Ein Ausweg: Die Benutzung von mit Einheiten versehenen Zahlenwerten, am Beispiel der Datei DM_Euro.cc**

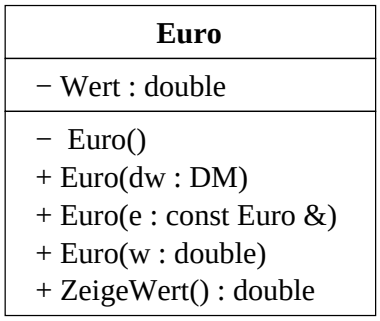


Abbildung 0.1: Die Klasse Euro

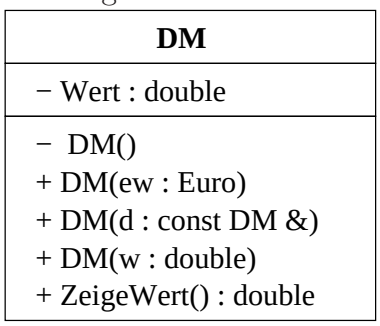


Abbildung 0.2: Die Klasse DM

Durch die jeweilig vorgesehenen Konstruktoren aus der anderen Währung wird eine automatisch Typkonversion ermöglicht.

```
...
class DM;

class Euro
{
private:
  double Wert;
public:
  Euro() : Wert(0.0) {};
  Euro(double w) : Wert(w) {};
  Euro(const Euro &e) : Wert(e.Wert) {};
  Euro(DM dw);
  double ZeigeWert() const { return Wert; };
};
```

```
class DM
{
private:
  double Wert;
public:
  DM() : Wert(0.0) {};
  DM(double w) : Wert(w) {};
  DM(const DM &d) : Wert(d.Wert) {};
  DM(Euro ew) : Wert(ew.ZeigeWert() * 1.95583) {};
  double ZeigeWert() const { return Wert; };
};


Euro::Euro(DM dw)
{
  Wert = dw.ZeigeWert() / 1.95583;
}

void DruckeEuroBetrag(const Euro &e)
{
  cout << "Geldbetrag: " << setiosflags(ios::fixed) << setprecision(2)
       << e.ZeigeWert() << " Euro" << endl;
}

int main()
{
  Euro b1(12.3);
  Euro b2(14.12);
  DM b3(1.23);
  Euro b4;
  Euro b5(b1);

  DruckeEuroBetrag(b1);
  DruckeEuroBetrag(b2);
  DruckeEuroBetrag(b3);
  DruckeEuroBetrag(b4);
  DruckeEuroBetrag(b5);

  return 0;
```

Neuere Programmiersprachen (z.B. Fortress) enthalten als Sprachkonzept Einheiten und Maße(Seite 17), damit Kommunikationsprobleme wie bei der Marssonde hoffentlich nicht mehr auftreten werden.

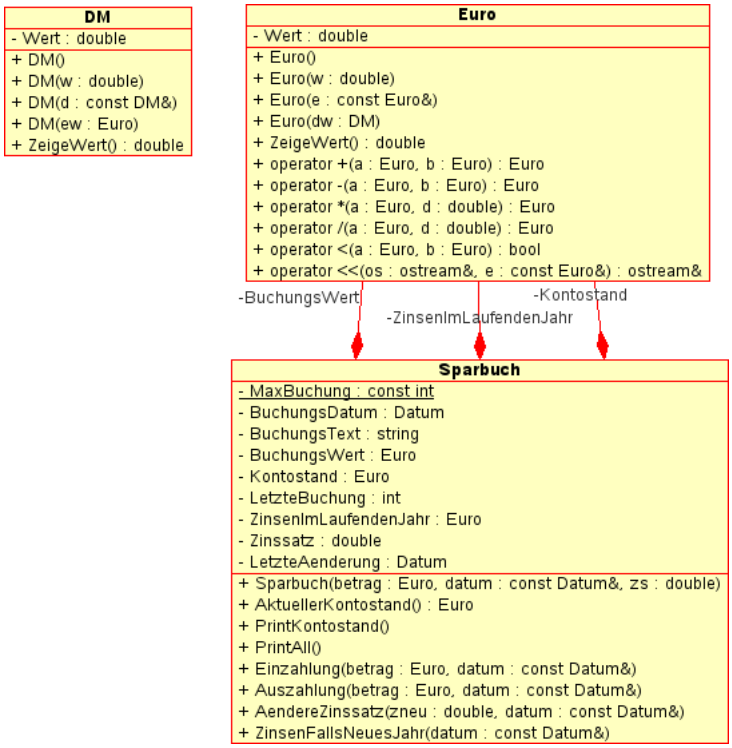


Abbildung 0.3: Ein währungsbewußtes Sparbuch

- **Weitere Beispiele zu Spezifikationsmängeln:**

  - **PC-Problem lässt Walmart-Kunden in den USA dreifach zahlen**

    Ein Computer-Problem hat dazu geführt, dass 800.000 Karten-Transaktionen bei Walmart-Filialen in den ganzen USA doppelt oder dreifach verbucht wurden. Aufgetreten sei der Fehler beim Transaktions-Dienstleister First Data. US-Medien zitieren die First-Data-Sprecherin Staci Busby: "Die mehrfachen Mastercard- und Visa-Buchungen haben wir wieder zurückgenommen, vor Dienstag sind diese aber nicht ausgeführt. Jeder, der am 31. März bei Walmart eingekauft hat, sollte seine Abrechnung noch einmal überprüfen."

    Zu Details des Problems könne sie nichts sagen; klar sei jedoch, dass nur Walmart-Kunden davon berührt seien. Betroffene Kunden würden von First Data kontaktiert, versprach die Firmensprecherin, zudem sei eine kostenlose Info-Hotline geschaltet. (tol/c't)

    Link zu diesem Artikel bei heise-online:
    http://www.heise.de/newsticker/meldung/46278

  - **US-Bezahlsystem mit öffentlichen Kreditkartendaten**

    Durch einen primitiven Fehler auf den Webseiten des amerikanischen Bezahl-Dienstleisters PaySystems waren tausende von Kundendatensätzen einschließ-

lich Kreditkartendaten zugänglich. Jeder PaySystems-Kunde konnte dabei die Daten anderer Kunden einsehen und sogar ändern.

PaySystems bietet an, Bezahlvorgänge zu widerrufen. Dabei wird diesem Vorgang eine Transaktionsnummer zugewiesen, die beim Aufruf der zugehörigen Informationen als Parameter in der URL auftauchte. Durch Ändern dieses Parameters konnte man beliebige Transaktionen anderer Kunden abrufen und anschließend über eine zweite URL auch deren Adresse und Kreditkartendaten.

Besonders erschreckend war auch die Art und Weise, wie die Firma auf die Sicherheitslücke reagiert hat. Ein c't-Leser entdeckte das Problem zufällig und unterrichtete PaySystems unverzüglich. Als nach einer Woche nichts passierte, wendete er sich an heise Security. Auf unsere Nachfragen antwortete PaySystems prompt, dass man den Hinweis zur Kenntnis genommen habe und an der Beseitigung des Problems noch arbeite. Auf weitere Nachfragen, warum man die Seiten nicht unverzüglich gesperrt habe, kam keine Antwort mehr. Mittlerweile ist diese Lücke zwar geschlossen, aber die Daten standen – selbst nachdem PaySystems über das Problem informiert war – noch mindestens eine Woche ungeschützt im Netz.

Das Ausmaß des Problems lässt sich nur schwer abschätzen. Aber die Tatsache, dass die Transaktionsnummern sequenziell vergeben wurden und mehrere Stichproben sofort zum Erfolg führten, lässt darauf schließen, dass hunderttausende solcher Transaktionen zugänglich waren. Über welchen Zeitraum die Daten so offen im Netz standen, können wir nicht beurteilen. Nachdem PaySystems unsere diesbezüglichen Nachfragen ignoriert hat, rechnen wir nicht damit, dass der Dienstleister seine Kunden auf die mögliche Gefährdung der Kreditkartendaten hinweist. (ju/c't)

Link zu diesem Artikel bei heise-online:
http://www.heise.de/newsticker/meldung/45566

- **Report: Wurm Lovsan nicht Schuld an Blackout 2003**

  Eine amerikanisch-kanadische Untersuchungskommission der Energieaufsichtsbehörde (FERC) ist zu dem Ergebnis gekommen, dass der Wurm Lovsan/MSBlaster nicht der Verursacher des gigantischen Stromausfalls im Nordosten der USA im vergangenen Jahr war. Beim Blackout 2003 waren 50 Millionen Amerikaner zeitweise ohne Strom. Da zeitgleich der Wurm im Internet die Runde machte und Millionen von Windows-Rechnern infizierte oder lahmlegte, lag der Schluss nahe, Lovsan könne zum Ausfall beigetragen haben. Immerhin greifen Energieerzeuger schon seit längerem auf Windows für ihre Managementsysteme zurück. Anzeige

  Im Februar dieses Jahres wurde aber bekannt, dass ein Softwarefehler eines Unix-Systems zur Überwachung und Steuerung von Stromnetzen beim Erzeuger FirstEnergy den Ausfall begünstigte. Durch den Fehler wurden Alarme und Meldungen nicht mehr an das Kontrollpersonal weitergeleitet. Damit war es nicht mehr möglich, Gegenmaßnahmen zu ergreifen: Der Ausfall einer Versorgungsleitung führte zum Zusammenbruch des gesamten Stromverbundes.

  Der Fehler des Managementsystems sei aber laut Untersuchungsbericht weder auf Cyberattacken durch Al-Quaida noch durch Würmer oder Viren zurückzuführen. Grundlage der Ermittlungen waren Befragungen von Mitarbeitern, Telefonmitschnitte und Berichte von Behörden und Geheimdiensten. Allerdings habe man nicht die Logdateien von Netzwerkgeräten, Firewalls und Intrusion-Detection-Systemen ausgewertet, die eventuell tiefergehende Hinweise gegeben hätten.

  Link zu diesem Artikel bei heise-online:
  http://www.heise.de/newsticker/meldung/46328

- **Software-Fehler verursachte US-Stromausfall 2003**

  Acht Staaten im Nordosten der USA und Teile Kanadas blieben im August des vergangenen Jahres für fünf Tage ohne Strom. Insgesamt waren 50 Millionen Menschen betroffen. Schuld am Blackout war nach Angaben von SecurityFocus ein Softwarefehler des Managementsystems zur Überwachung und Steuerung von Stromnetzen beim Erzeuger FirstEnergy.

  Das betroffene System XA/21 stammt von General Electric und ist bei Erzeugern weit verbreitet. Der Fehler wurde nach einem mehrwöchigen intensiven Code-Audit gefunden und soll bisher nur beim großen Blackout aufgetreten sein. Nach Angaben des Sprechers von FirstEnergy löste eine besondere Kombination von Ereignissen und Alarmen den Fehler aus, woraufhin das System seinen Dienst einstellte. Der kurz darauf einspringende Backup-Server versagte ebenfalls, da er mit der Zahl der bereits aufgelaufenen, aber nicht verarbeiteten Meldungen überfordert war.

In der Folge nahm das System auflaufende Alarme nicht mehr entgegen und meldete sie nicht an das Bedienpersonal weiter. Hinzu kam, dass den Betreibern nicht einmal auffiel, dass ihr System bereits versagt hatte. Eine Stunde lang soll die Kontrollstation veraltete Daten angezeigt haben. Bei auftretenden Störungen blieb zwangsläufig die Reaktion aus.

Normalerweise koppelt ein Stromerzeuger sein Netz bei größeren Ausfällen von den anderen Stromnetzen ab, um weitere Schäden durch Überlast zu vermeiden. Somit bleibt ein Problem regional begrenzt. Da die Alarme aber nicht registiert wurden, reagierten die Operatoren nicht.

FirstEnergy will nun seine XA/21-Systeme gegen die Produkte eines Wettbewerbers austauschen. Das North American Electric Reliability Council (NERC) hat eine Richtlinie herausgegeben, in der Maßnahmen beschrieben sind, Vorfälle wie am 14. August zu vermeiden. Unter anderem wird darin FirstEnergy aufgefordert, bis zum Austausch ihrer System alle notwendigen Patches für XA/21 zu installieren.

Da sich der Zeitpunkt des Blackouts und der Ausbruch des Wurms Lovsan/Blaster überschnitten, gab es Vermutungen, der Wurm könnte den Ausfall verursacht haben. Auch warnte das CERT/CC Anfang August davor, dass Lovsan Unix-Systeme mit Distributed Computing Environment (DCE) angreift und zum Absturz bringen kann. XA/21 ist ein EMS/SCADA-System (Supervisory Control and Data Acquisition), das auf Unix mit X-Windows basiert. Sicherheitslücken gibt es hier reichlich. Somit kann zukünftig nicht ausgeschlossen werden, dass Würmer, die den Weg in ein Kontrollzentrum gefunden haben, solche Systeme beeinflussen können.

Link zu diesem Artikel bei heise-online:
http://www.heise.de/newsticker/meldung/44621

- **US-Sicherheitsexperten fordern bessere Ausbildung für Softwareentwickler**

  Die National Cyber Security Partnership (NCSP) fordert in ihrem aktuellem "Security Across the Software Development Life Cycle"eine bessere Ausbildung der Entwickler. Der Bericht befasst sich insbesondere mit dem Lebenzyklus von Software. Sicherheit müsse sich über die gesamte Lebensspanne eines Software-Produktes erstrecken. Jeder Abschnitt der Spanne, angefangen vom Design und Spezifikation, über die Implementierung und Tests bis hin zum Patch-Management soll unter den Gesichtspunkten der IT-Sicherheit bearbeitet werden.

  Die Arbeitsgruppe hat zur Definition entsprechender Empfehlungen vier Untergruppen gebildet, die sich mit Schulung von Entwicklern und Anwendern, Softwareprozessen und Patchen beschäftigen. Die vierte Gruppe – Incentive Subgroup – will ein Programm erarbeiten, um Herstellern das Entwickeln von sicherer Software schmackhaft zu machen. Dazu sollen Preisverleihungen und

Zertifizierungen gehören. Daneben stellt man auch die Idee vor, die Sicherheit einzelner Softwaremodule als Messlatte für die weitere Karriere der jeweiligen Entwickler heranzuziehen.

Die vergangenes Jahr gegründete Arbeitsgruppe hat sich die Verbesserung der Cyber Security der US-amerikanischen Informationsinfrastruktur zum Ziel gesetzt. Mitglieder sind diverse Sicherheitsexperten aus Forschung, Lehre und Industrie, sogar Vertreter der National Security Agency finden sich in der Gruppe. Die Vorsitzenden der Gruppe sind Ron Moritz von Computer Associates und Scott Charney von Microsoft. Ähnliche Ziele wie die NCSP verfolgen die Cyber Security Industry Alliance (CSIA) und der Global Council of CSOs

Link zu diesem Artikel bei heise-online:
http://www.heise.de/newsticker/meldung/46241

- **Softwarefehler plagt Mercedes-Diesel**

  Software-Bugs plagen die User nicht etwa nur, wenn sie vor dem Computer am Schreibtisch sitzen oder mit Mobilrechnern unterwegs sind. Internet-Zugang, Navigationsrechner oder multimediale Konsolen lassen das Auto zum IT-Problemfeld werden – darüber hinaus aber kämpfen Automobil-Elektroniker mittlerweile mit immer komplexeren computergestützten Steuerungssystemen und deren Software und damit auch mit den Bugs dieser Software. Jüngstes Beispiel: Wegen eines Softwarefehlers ruft DaimlerChrysler rund 10.000 Transporter der Mercedes-Benz-Modelle Vito und Viano mit Dieselmotoren zurück. In Deutschland sollen rund 3.000 Fahrzeuge betroffen sein.

  Ursache des Rückrufs ist ein Bug in der Software, mit der die Dieselsteuergeräte ausgerüstet sind. Sie aktivieren in Situationen, in denen dies eigentlich nicht vorkommen sollte, die Kraftstoffabschaltung, wodurch der Motor ausgehe. Betroffen seien Fahrzeuge mit Dieselmotoren, die zwischen November 2003 und April 2004 hergestellt wurden. Die Kunden würden durch die Servicestellen von Mercedes-Benz direkt angeschrieben, erklärte der Konzern; die Fahrzeuge erhielten eine fehlerbereinigten Software.

  Link zu diesem Artikel bei heise-online:
  http://www.heise.de/newsticker/meldung/48403

1 Einleitung

1.1 Grundlagen von UML-Klassendiagrammen

1.1.1 Klassen und Objekte

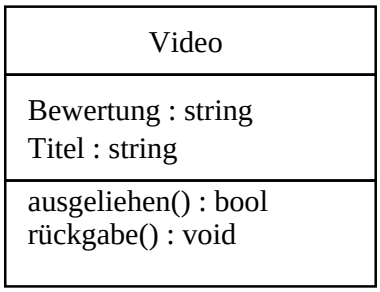


Abbildung 1.1: Eine Klasse

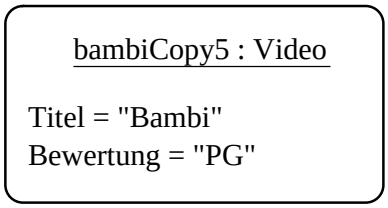


Abbildung 1.2: Ein Objekt dieser Klasse(Instanz)

1.1.2 Klassenbeschreibungsergänzungen

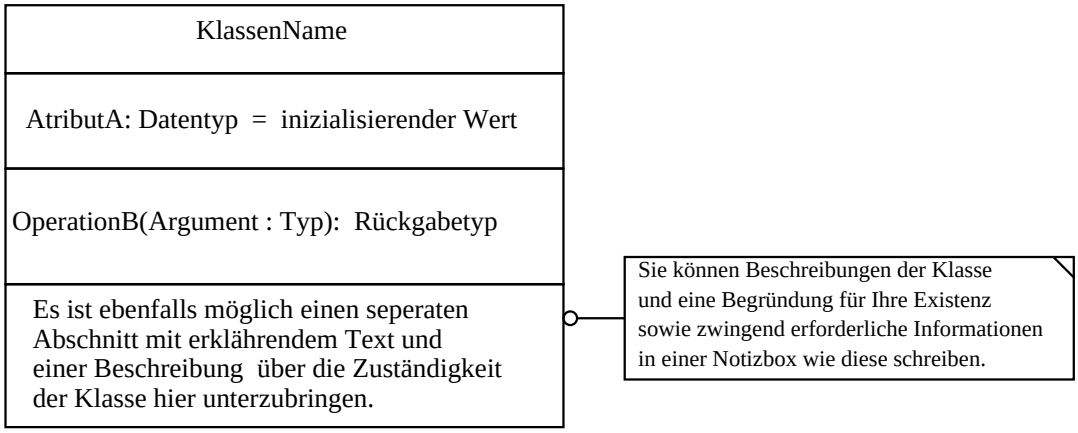


Abbildung 1.3: Beschreibung einer Klasse

Klassenname

Normale Schrift = konkrete Klasse
Italic Schrift oder << *abstrakt* >> = abstrakte Klasse
(Italic-Schriftsätze sind nicht Bildschirmfreundlich - benutzen Sie die Stereotyp-Notation)

Klassen- oder Instanzenattribute / -methoden

Normale Schrift = Instanzenobjekt
Unterstrichen **oder** $ =Klassenobjekt
($ ist kein UML-Standard)
Für abstrakte Methoden benutzen Sie = 0
(=0 ist kein UML-Standard)

Attribut- und Methodensichtbarkeit

+ public (öffentliche Sichtbarkeit)
- private (private Sichtbarkeit)
protected (geschützte Sichtbarkeit)

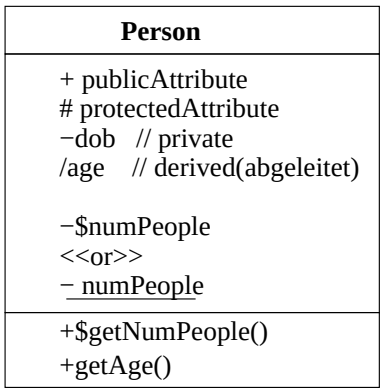


Abbildung 1.4: Eine Klasse: Person

- Das Attribut **age** ist abgeleitet.

- Die Anzahl der Instanzen der Klasse **Person** (**numPeople**) ist ein Attribut der Klasse **Person** selbst und nicht der Instanzen der Klasse. Dies wird als statisches Klassen-Attribut (class static member variable) bezeichnet. Es wirkt wie eine globale Variable der Klasse. Manchmal wird als alternative Schreibweise für Klassenattribute und deren Verhalten das $ Zeichen verwendet.

1.1.3 Links und Assoziationen

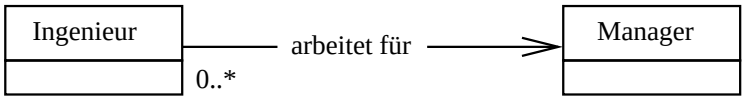


Abbildung 1.5: Assoziationen verbinden Klassenexemplare

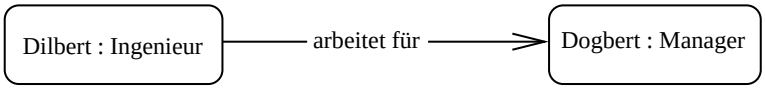


Abbildung 1.6: Assoziationen verbinden Klassenexamplare

1.1.4 Rollen- und Assoziationsnamen

Rollennamen

Benannte Instanzen einer Klasse die an das anderen Ende der Assoziation geschrieben werden, gewöhnlich ein Substantiv.

Assoziationsnamen

Benennen die Assoziation selbst, viele erfordern einen Pfeil, der die Richtung der Assoziation anzeigt, gewöhnlich Verben oder Verbschlagworte.

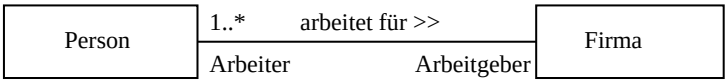


Abbildung 1.7: Rollen in Klassen

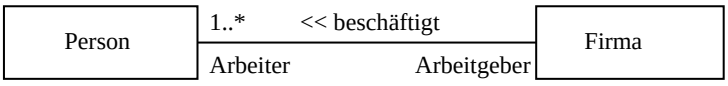


Abbildung 1.8: Rollen in Klassen (Fortsetzung)

1.1.5 Multiplizitäten (Kardinalitäten)

- Multiplizitäten beschreiben die Anzahl der Instanzen am Assoziationsende.

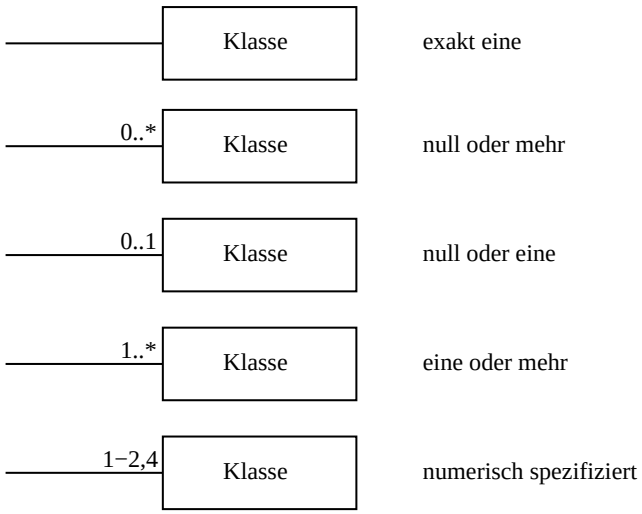


Abbildung 1.9: Multiplizität / Kardinalität

Anmerkung: n und * können anstelle von 0..* verwendet werden.

1.1.6 Stereotype

Stereotyp
Eine konventionelle Kategorisierung für modellierte Entitäten.

- Sie werden oft bei Klassen, Assoziationen und Methoden angewendet.
- Sie bieten einen Weg, UML zu erweitern; sie dienen zur Definition eigener, für spezielle Probleme modellierter Elemente.
- Einige Stereotypen werden von CASE-Werkzeugen (CASE tool generator) erkannt

Es gibt zwei Wege, Stereotypen darzustellen:

- Benutzen Sie normale UML-Elemente, mit dem Stereotypnamen zwischen << und >>.
- Benutzen Sie eigene Icons.

Beispiele:

```
<< abstrakt >>, << interface >>, << exception >>,
<< instantiates >>, << subsystem >>, << extends >>,
<< instance of >>, << friend >>,
<< constructor >>, << thread >>, << uses >>,
<< global >>, << create >>, << invent your own >>,
<< basic observator >>, << derived observator >>,
<< modificator >>, << iterator >>, ...
```

1.1.7 Markierte Werte / Tagged Values

- Tagged Values sind ein weiterer Mechanismuss, UML zu erweitern: Er erlaubt es, dem Modell neue benannte Eigenschaftswerte hinzuzufügen (Name = Wert).

Gebräuchliche Beispiele für **tagged values** sind:

- {Autor = (Dave,Ron)}
- {Versionsnummer = 3}
- {Ort = d:\Location\uml\examples}
- {Ort = Node: mittlere Schicht}

1.1.8 Generalisierung, Spezialisierung und Vererbung

Beispiele:

- **Arbeitnehmer** generalisiert **Manager** und **Ingenieur**.
- **Ingenieur** spezialisiert **Arbeitnehmer**.
- **Manager** ist eine **Art/Sorte** von **Arbeitnehmer**.
- **Manager** und **Ingenieur** erben die Schnittstellen von **Arbeitnehmer** und auch einige Implementierungseinzelheiten.

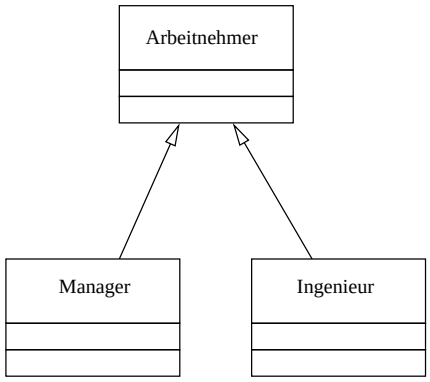


Abbildung 1.10: Generalisierung, Spezialisierung und Vererbung

1.2 Abstrakte Klassen

- Eine Generalisierung ohne vollständige Implementierungsspezifikation.
- Sie wird in UML mit dem Stereotyp `<< abstract >>` angezeigt.
- In C++ werden alle **pure virtual** Methoden = 0 deklariert.
- In Java wird sie mit dem Schlüsselwort "abstract"gekennzeichnet
- Ein **Interface** ist wie eine abstrakte Klasse, aber ohne jede Implementierung.

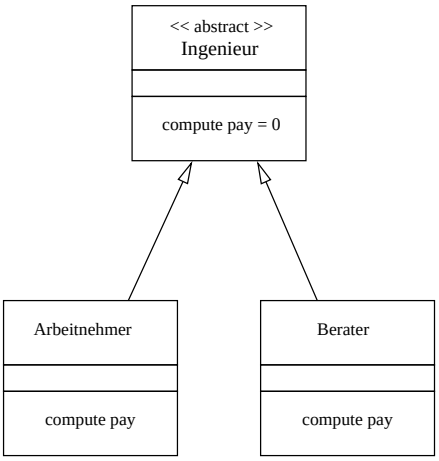


Abbildung 1.11: Abstrakte Klassen

1.3 Komposition / Aggregation

Das Rautenzeichen wird für verschiedene Eigenschaften / Konzepte eingesetzt:

- Teil- / Vollbeziehung (am häufigsten verwendet)
- Hat - ein
- Hat - ein / eine Sammlung - von
- Ist zusammengesetzt - aus

Beachten Sie, wie die Zeit die Kardinalitäten beeinflussen kann: Ein Auto kann viele Fahrer haben, aber zu einem bestimmten Zeitpunkt, kann es nur einer fahren. (Dies wird häufig mittels des Stereotyps << history >> unmissverständlich hervorgehoben.)

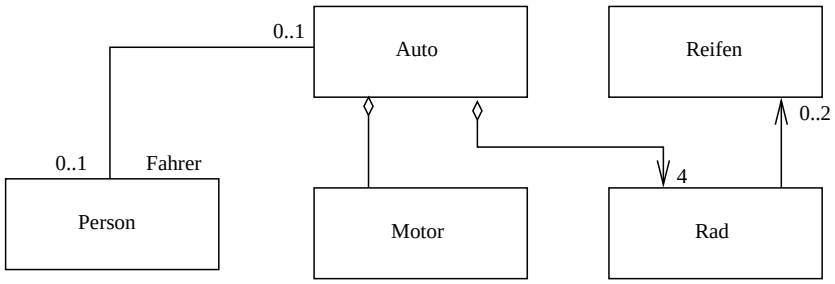


Abbildung 1.12: Komposition / Aggregation

Komposition:

- UML benutzt ein gefülltes Rautensymbol für eine **Komposition**.
- Das leere Rautensymbol beschreibt eine **Aggregation**.
- Eine **Komposition** ist eine stärkere Assoziation als eine **Aggregation**. Der Unterschied besteht darin, dass bei einer **Komposition**, ein Teil nie mehr als ein Ganzes hat und dass ein Teil und ein Ganzes immer einen gemeinsamen Lebenszyklus haben.
- Im folgenden Beispiel sind **Zeilen** ein fester und permanenter Bestandteil des **Layouts**, aber die Anzahl der Zeichen in jeder Zeile verändert sich zur Lebenszeit des Layout-Exemplars.

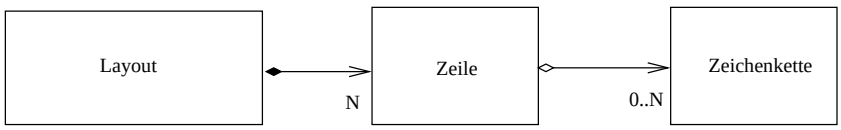
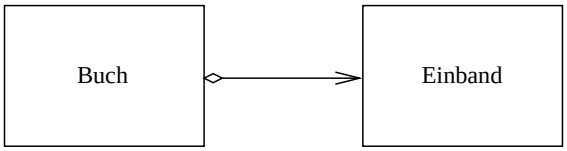


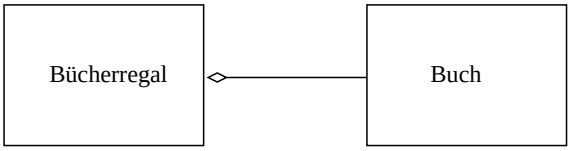
Abbildung 1.13: Komposition zwischen Layout und Zeile

- Das Objekt **Zeile** ist ein Teil vom Objekt **Layout**, so dass Zeilen erzeugt werden, wenn ein Layout erzeugt wird und Zeilen zerstört werden, wenn ein Layout zerstört wird. **Zeile** hat keine selbstständige Existenz.
- Beispiel: Ein Buch besteht aus Seiten (pages) und einem Einband (cover).



Aggregation:

- Instanzen der Klasse B existieren unabhängig von Objekt A, aber Objekt A hat Kentniss von seinen Instanzen der Klasse B (für Sammlungen (Collections) benutzt und um schwache Teil/Ganzes Beziehungen zu beschreiben).
- Beispiel: Ein Bücherregal enthält eine Sammlung von Büchern.



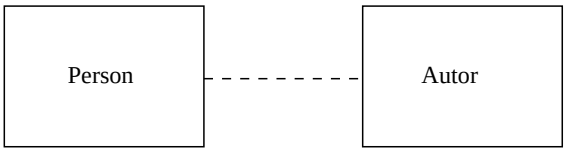
Assoziation:

- Ein Objekt der Klasse A hält eine halb-permanente Referenz zu einem Objekt der Klasse B ohne jede einschränkende Semantik.
- Beispiel: Bücher haben einen Autor

Dependancy:

- Instanzen der Klasse A haben vorübergehende Beziehungen mit Instanzen der Klasse B

- Beispiel: Eine Person liest ein Buch, dann gibt sie es einem Freund.



1.4 Qualifizierte Assoziationen/Qualified Associations

- Sie werden benutzt, damit Instanzen einer Klasse, die in einer "ein zu viele"-Beziehung zu einer anderen Klasse B stehen, über einen eindeutigen Identifizierer schnell auf die Instanzen von B zugreifen zu können.
- Qualifizierte Assoziationen sind für gewöhnlich mit einer Art "Wörterbuch" ausgestattet (auch als assoziative Felder bekannt), etwa ein **Hash Table** oder einer **Map**.
- Warum ist eine qualifizierte Assoziation ein besseres Modell?

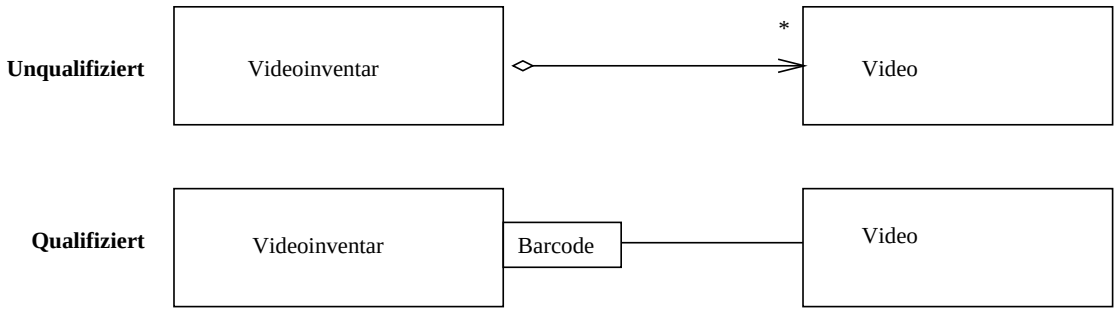


Abbildung 1.14: Qualifizierte Assoziation

1.5 Assoziations-Klassen

Attribute hängen manchal von zwei Objekten ab. Wenn ein solches Attribut viel komplexer ist als ein skalarer Wert, sollte es als eine eigene Klasse modelliert werden.

- Im folgenden Beispiel sind die Arbeitsvertragsdaten Attribute für die **"arbeitet für"**-Assoziation.

- **Anmerkung:** Die Semantik der Assoziationsklasse (so wie sie modelliert wurde) zeigt an, dass für jedes Personen/Firma-Paar, exakt ein Arbeitsvertrag existiert. Somit beschreibt dieses Modell, dass eine Person nicht zu zwei unterschiedlichen Zeiten für dieselbe Firma arbeiten kann.

- **Anmerkung:** Der Stereotyp `<< history >>` erklährt den Zeitaspekt der Beziehung: Er besagt, dass eine Person über die Zeit für viele Firmen arbeiten kann, aber zu einer bestimmten Zeit immer nur für keine (0) oder eine (1) Firma arbeitet.

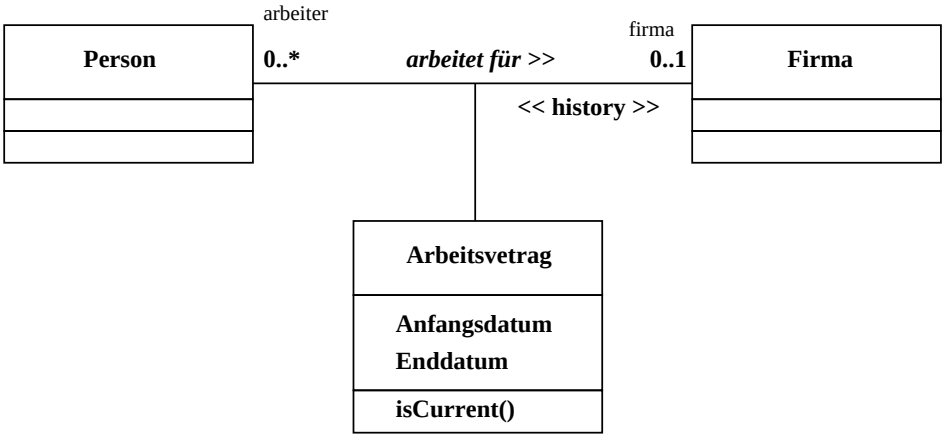


Abbildung 1.15: Assoziationsklasse

- Die Semantik der Assoziatiationsklasse entspricht dem relationalen Datenbankdesign.

- In der Implementierung kann eine Person eine Reihe von Beschäftigungen aufnehmen. Jedes Beschäftigungsverhältniss kennt eine Person und eine Firma.

- Beachten Sie im folgenden Implementierungsvorschlag die Änderung in der assoziierten Kardinalität und die Tatsache, dass die "Arbeiter"-Beziehung nun abgeleitet ist (angezeigt durch "/"):

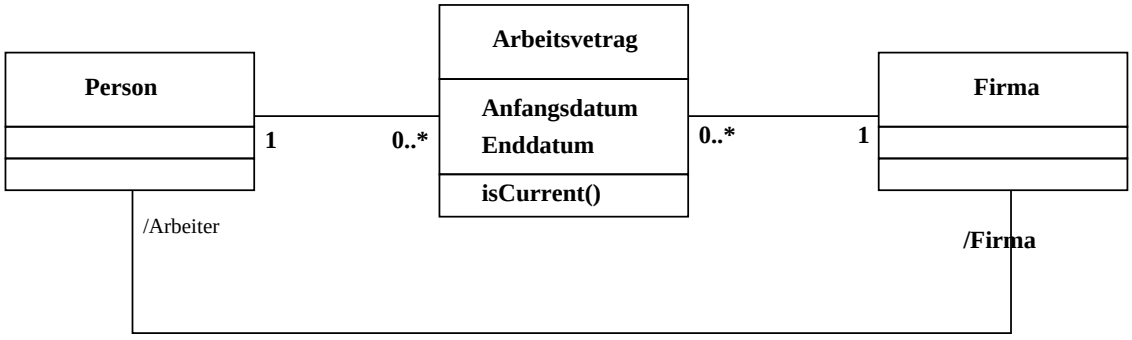


Abbildung 1.16: Assoziationsklasse (Fortseztung)

Standard für IT-Sicherheit

2 OCL

2.1 Spezifikation einfacher Klassen

VALUE , KEY

mydictionary

- keys: vector<KEY> ∗
- values: vector<VALUES> ∗
- count: unsigned int

<<basic queries>>

+ get_count(): unsigned int
+ has(k: const KEY &) : bool
+ value_for (k: const KEY &): VALUE

+ << constructor >> mydictionary()

+ << constructor >> mydictionary(
s: const mydictionary<KEY,VALUE>&)

+ ∽mydictionary():void

/* disable assignment operator */
- operator= (s: const mydictionary<KEY,VALLUE> &): mydictionary&

<< derived queries >>

/* ... */

<< modificators >>

+ put(k: const KEY &, v: const VALUE &): void

+ remove(k: const KEY &): void

Der OCL2-Contract:

```
package mydictionary

context mydictionary
inv: count >= 0
inv: keys->size() = values->size()
inv: keys->size() = count

/* basic observators */

context mydictionary::get_count() : Integer
body: count

context mydictionary::has(k : KEY) : Boolean
post consistentWithCount: get_count()=0 implies not result
/* post: get_count() = KEY->forAll(k | has(k))->size() */

context mydictionary::value_for(k: KEY): VALUE
pre: has(k)

/* constructor */

context mydictionary::mydictionary()
post: get_count() = 0

context mydictionary::mydictionary(s : mydictionary)
post: s.get_count() = self.get_count()
/* post: KEY->forAll(k | s.has(k) implies s.value_for(k) = self.value_for(k)) */

/* modificator */

context mydictionary::put(k: KEY, v: VALUE)
pre: not has(k)
post: has(k)
post: get_count() = get_count@pre() + 1
post: value_for(k) = v
/* post: KEY->forAll(kl | has@pre(kl) implies value_for@pre(kl) = value_for(kl))

context mydictionary::remove(k: KEY)
pre: has(k)
post: not has(k)
post: get_count() = get_count@pre() - 1
```

```
/* post: KEY->forAll(kl | has@pre(kl) implies (kl = k or
                                          value_for@pre(kl) = value_for(kl))) */

endpackage -- mydictionary
```

Prinzipien:

1. Observatoren (und nur diese) haben einen Ergebniswert; sie ändern den Objektinhalt nicht!
   Modifikatoren haben **keinen** Ergebniswert.

2. Unterscheide: "grundlegende Observatoren" von

3. "abgeleiteten Observatoren". Jeder abgeleitete Observator hat eine Nachbedingung, die auf die grundlegenden Observatoren zurückgreift.

4. Für jeden Konstruktor/Modifikator schreibe eine Nachbedingung, die die Werte aller brundlegenden Observatoren am Ende einer Methode festlegt.

5. Für jeden Observator und jeden Konstruktor/Modifikator schreibe notwendige Vorbedingungen.

6. Schreibe für jede Klasse eine Invariante, die die sich nicht ändernden Merkmale der Objekte beschreibt (also **gültige** und **ungültige** Objekte unterscheidet).

7. Die grundlegenden Observatoren sind ein minimaler Methodensatz, der dazu dient den Zustand eines Exemplars vollständig zu charakterisieren. Sie haben außer Konsistenzbeziehungen zu anderen Methoden **keine** Nachbedingungen.

	VBen		NBen			
Methoden	an Attribute	an Parameter	an Attribute	an Parameter	an Result	Bemerkungen
basic queries (const)	-	ja	-	-	als Konsistenzbeziehungen	-
derived queries (const)	-	ja	-	ja	ja	-
Konstruktoren	-	ja	durch basis queries	ja	-	erzeugt Attribute, die Invariante erfüllen
Modifikatoren	durch basis queries	ja	durch basis queries	-	-	überprüft Invarianten
Invarianten als methoden	-	-	-	-	durch basis queries	-

Tabelle 2.1: Tabelle: Programming by Contrakt

2.1.1 Ein einfacher Java-Contract

Ein Beispiel: Die Klasse java.awt.Color

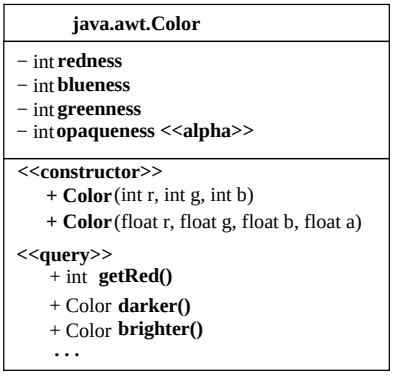


Abbildung 2.1: Die Standard-Farbklasse: java.awt.Color

Was sagt Ihnen dieses Klassendiagramm? Was sagt es nicht?

2.1.1.1 Klassencontract: java.awt.Color

Invarianten: (Für jedes Farbobjekt, c)
0<=redness(c)<=255 **and** 0<=greenness(c)<=255 **and**
0<=blueness(c)<=255 **and** 0<=opaqueness(c)<=255

Konstruktor-Methoden

Listing 2.1: Konstruktor-Methoden

```
public Color(int r, int g, int b)
   pre: 0<=r<=255 and 0<=g<=255 and 0<=b<=255
                  --(verwirft illegale Argumente)
  modifies: redness, greenness, blueness, opaqueness
  post: redness==r and greenness==g and blueness==b and
     opaqueness==255

public Color(float r, float g, float b, float a)
  pre: 0.0<= r <=1.0 and 0.0<= g <=1.0 and 0.0<= b <=1.0
     and 0.0<= a <=1.0
                  --(verwirft illegale Argumente)
  post: redness==r*255 and greenness==g*255 and blueness==b
     *255 and opaqueness==a*255
```

Query Methoden und Modifikatoren

Listing 2.2: Query Methoden

```
public int getRed()
 post: result == redness

public Color darker()
 post: result.redness == redness*0.7
   and result.greenness == greenness*0.7
   and result.blueness == blueness*0.7
   and result.opaqueness == 255

public Color brighter()
 post: (redness/ 0.7)>255 implies result.redness == 255
   and (redness/ 0.7)<=255 implies result.redness/ 0.7
   and (greenness/ 0.7)>255 implies result.greenness == 255
   and (greenness/ 0.7)<=255 implies result.greenness/ 0.7
   and (blueness/ 0.7)>255 implies result.blueness == 255
   and (blueness/ 0.7)<=255 implies result.blueness/ 0.7
   and result.opaqueness == 255
...
```

Bemerkung: Im Sinne des "Programming by Contract" sollte der wesentliche Teil der Spezifikation "völlig" implementierungsunabhängig durchgeführt werden.
Das heißt:

- kein Zugriff auf private Attribute bzw. Methoden
- keine Vorwegnahme von zu benutzten Algorithmen

Alle Vor- und Nachbedingungen sollten mit Hilfe der Basic Queries geschehen.

$\longrightarrow$ get_Attribut()

Bemerkung: Die Spezifikation mit OCL geschieht aber nicht **nur** im Sinne von "Programming by Contract", sondern auch als Hilfe bei der Kodierung und Implementierung von Softwaresystemen. **Hier** sollte natürlich auch auf implementierungsabhängige Einzelheiten Bezug genommen werden können. Z.Bsp. sollten dann die **BasicQueries** selbst mittels Nachbedingungen (zur Nutzung durch das Implementierungsteam, nicht zur Weitergabe an den Client) spezifiziert werden.

2.1.2 Hinweise

1. Füge wo immer nötig implementirungsspezifische Einschränkungen ein (meist in der Form <> 0)

2. Stelle sicher, daß die in den Vorbedingungen benutzten Observatoren "schnell" arbeiten.
   Falls nötig, füge zusätzliche abgeleitete schnell arbeitende Observatoren hinzu (virtuelle, nur zur Spezifikation benötigte Methoden), die in Ihren Nachbedingungen die aufwendigen Observatoren ersetzen.

3. Wenn ein abgeleiteter Observator als Attribut implementiert wird, sollte die Klasseninvariante entsprechend erweitert werden.

4. Um die Neuimplementierung virtueller Methoden zu unterstützen sollte **jede** Nachbedingung einer virtuellen Methode durch Ihre Vorbedingung abgeschirmt sein.

   Beispiel:

a)
```
context mydictionary::remove(k:const KEY &)
pre:  has(k)
post: not has(k)
post: get_count() = get_count@pre() - 1
```

und

b)
```
context mydictionary::remove(k:const KEY&)
pre:  has(k)
post: not has(k)
post: has@pre(k) implies
                        get_count()=get_count@pre() - 1
```

Eine benutzerfreundliche Unterklasse

user_friendly_mydictionary,

die die Vorbedingung von remove abschwächen möchte zu

pre: true

ist für die Variante a) unmöglich, für Variante b) jedoch sehr wohl duch Streichung der Vorbedingung und Ergänzung einer neuen Nachbedingung:

```
context mydictionary::remove(k:const KEY&)
pre:  true
post: not has(k)
post: has@pre(k) implies
                        get_count()=get_count@pre() - 1
post: not has@pre(k) implies
                        get_count()=get_count@pre()
```

5. Explizite Rahmenregeln sollten in einem weiteren Level des Vererbungsdiagramms konzipiert werden. Analoges gilt bei der Unzugänglichmachung virtueller Hilfsobservatoren:

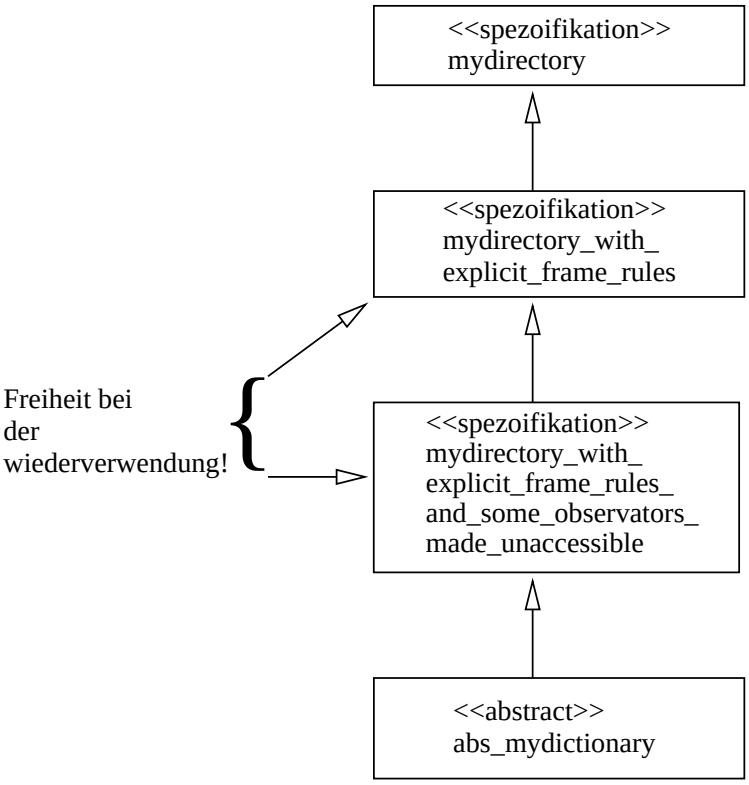


Abbildung 2.2: Vererbung bei der Einführung expliziter Rahmenregeln

2.2 Weitere Hinweise zur Vererbung

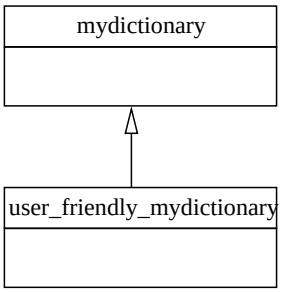
Gemäß Liskovs Ersetzungsprinzip:

is - a

Wenn immer die Instanz einer Klasse erwartet wird, kann ebenfalls die Instanz jeder ihrer Subklassen benutzt werden.

2.2.1 Vor-/Nachbedingungen bei der Vererbung

"Vorbedingungen" in Subklassen können gleich oder schwächer sein als diejenigen in ihrer Elterklasse:



$$VB_{kind} = VB_{Elter} \text{ or } \ldots$$

Abbildung 2.3: mydictionary-
Vererbung

"Nachbedingungen" in Subklassen müssen gleich oder stärker sein als diejenigen in Ihrer Elterklasse:

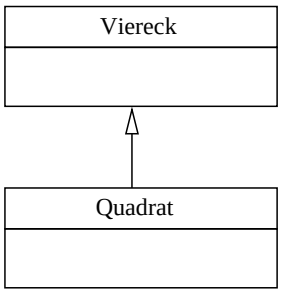
$$NB_{kind} = VB_{Elter} \text{ implies } (NB_{Elter} \text{ and } \ldots)$$

2.2.2 Ivarianten bei der Vererbung

Damit jedes Kind auch ein Elter ist muß gelten:

$$Inv_{Kind} = Inv_{Elter} \text{ and } \ldots$$

Beispiel:



⟵ Viereck **und** alle Seiten gleich lang und alle vier Winkel gleich 90°

Abbildung 2.4: Viereck

2.2.3 Hinweise zu C++

- Konstante Methoden sind (reine) Observatoren
- nichtkonstante Methoden sollten keine Ergebnisse liefern; bei Beendigung sollte die Klasseninvariante erfüllt sein
- Direkter modifizierender Zugriff auf Attribute ist gefährlich (warum?) und sollte deshalb nicht erlaubt sein.

2.3 Subcontracting (is-a-Vererbung)

Ursprüngliche Definition:
invert(epsilon:REAL) is - - Invert martix with precision epsilon
    require epsilon>= 10^(−6)
    ...
    ensure abs ((Current * inverse) - Identy) <= epsilon
end

Redefinition:
invert(epsilon:REAL)is - - Invert matrix with precision epsilon
    require else epsilon>= 10^(−20)
...
    ensure abs ((Current * inverse) - Identy) <= (epsilon/2)
end

aus: http://www.cse.yorku.ca/course_archive/2004-05/F/3311/sectionA/22-InheritDBCgen.pdf

Unternehmer und Subunternehmer:

Ein Vertrag zwischen Kunde und Unternehmer laute:

```
                    Interface Directory
.
.
.
ACTIONS
   Put(IN k:Keys, IN v:Values)
      PRE
         NOT Has(k)
      POST
         Has(k)
         ValueFor(k)= v
         Count = OLD(Count)+1
.
.
.
```

Er kann durch den Unternehmer allein nicht zeitgerecht erfüllt werden. Jedoch bietet ein anderer Unternehmer den folgenden Vertrag an:

```
                    Interface DirectoryB
.
.
.
ACTIONS
   Put(IN k:Keys, IN v:Values)
      PRE
         TRUE
      POST
         Has(k)
         ValueFor(k)= v
         NOT OLD(Has(k)) IMPLIES Count = OLD(Count)+1
         OLD(Has(k))     IMPLIES Cout = OLD(Count)
.
.
.
```

Er kann als Subunternehmer des Unternehmers in die Pflicht genommen werden, da DirectoryB den Vertrag Directory vollständig erfüllt. Der Kunde kann, da an die Vorbedingungen von Directory gebunden, keinen Unterschied zwischen der Benutzung von

Directory und DirectoryB feststellen. Lediglich wenn er sich nicht an die Vorbedingungen des Vertrags Directory hielte, wäre das Verhalten seiner Software anders, aber das darf er ja nicht!
Subcontracting (is-a-Vererbung) verlangt also folgende Beziehungen zwischen den Verträgen der public Methoden einer Kindklasse und denjenigen ihrer Elterklasse:

- Invarianten des Kindes dürfen nicht schwächer sein als diejenigen der Elterklasse,
- Vorbedingungen einer Kindmethode dürfen nicht stärker sein als die Vorbedingungen der Eltermethode,
- Nachbedingungen einer Kindmethode dürfen beim Eintreten der Vorbedingung der Eltermethode nicht schwächer sein als die Nachbedingung der Eltermethode, andernfalls dürfen sie beliebig sein. (Diese Beliebigkeit wird natürlich im allgemeinen vom Subunternehmer dazu genutzt, einen möglichst großen Marktwert für seine Mehrfunktionalität — es gibt mehr Fälle, in denen die Vorbedingung der Kindmethode erfüllt ist, als die Fälle in denen lediglich die Vorbedingung der Eltermethode erfüllt sind — zu erreichen.)

Kurz:

$$Invariante_{Kindklasse} = Invarinte_{Elterklasse} \wedge ...$$
$$Vorbedingung_{Kindmethode} = Vorbedingung_{Eltermethode} \vee ...$$
$$Nachbedingung_{Kindmethode} = \begin{cases} Nachbedingung_{Eltermethode} \wedge ... & \text{, falls } Vorbedingung_{Eltermethode} \\ \text{beliebig} & \text{, sonst} \end{cases}$$

Diese formalen Subcontracting-Regeln können in der Anwendung häufig logisch zusammengefaßt werden. Im obigen Beispiel also:

```
-------------------------------- Directory
Pre-Put_Directory(k,v)  = NOT Has(k)

Post-Put-Directory(k,v) = Has(k) AND ValueFor(k) = v AND
                          Count = OLD(Count)+1
```

liefert nach den obigen Regeln:

```
-------------------------------- DirectoryB
Pre-Put_DirectoryB(k,v)  = TRUE

Post-Put_DirectoryB(k,v) =
  (Pre-Put_Directory(k,v) IMPLIES Post-Put-Directory(k,v))
 AND
  ((NOT Pre-Put_Directory(k,v) AND Pre-Put_DirectoryB(k,v)) IMPLIES "Beliebiges"
```

```
  = (NOT OLD(Has(k))) IMPLIES (Has(k) AND ValueFor(k) = v AND
                                          Count = OLD(Count)+1))
    AND ((OLD(Has(k)) AND TRUE) IMPLIES "Beliebiges")
```

Nach dem Namen der Methode `Put()` ist "Beliebiges" natürlich einzig sinnvoll durch

```
Has(k) AND Count = OLD(Count) AND ValueFor(k) = ?
```

zu ersetzen. Dabei hat man für den letzten Anteil (Wert beim Schlüssel `k`) noch eine gewisse Entscheidungsfreiheit. Mögliche, sinnvoll erscheinende Spezifikationen:

- Der alte Wert bleibt erhalten.
- Der alte Wert wird überschrieben.
- Der Wert wird als „unbestimmt“ gekennzeichnet, da er in der Historie mit unterschiedlichen Wertbindungen versehen werden sollte.

Entscheidet man sich für die vorraussichtlich marktrelevanteste Spezifikation (Wertüberschreibung), so erhält man nach ein paar Zusammenfassungen:

```
-------------------------------- DirectoryB
Pre-Put_DirectoryB(k,v)  = TRUE
Post-Put_DirectoryB(k,v) =
 (NOT OLD(Has(k))) IMPLIES (Has(k) AND ValueFor(k) = v AND
                                     Count = OLD(Count)+1)) AND
 ((OLD(Has(k)) AND TRUE) IMPLIES (Has(k) AND Count = OLD(Count) AND
                                                 ValueFor(k) = v))
  = Has(k) AND ValueFor(k) = v AND
    (NOT OLD(Has(k)) IMPLIES Count = OLD(Count)+1) AND
    ((OLD(Has(k)) IMPLIES Count = OLD(Count))
```

2.4 Spezifikation von Klasseninterdependenzen

2.4.1 size() aller assoziierten Exemplare

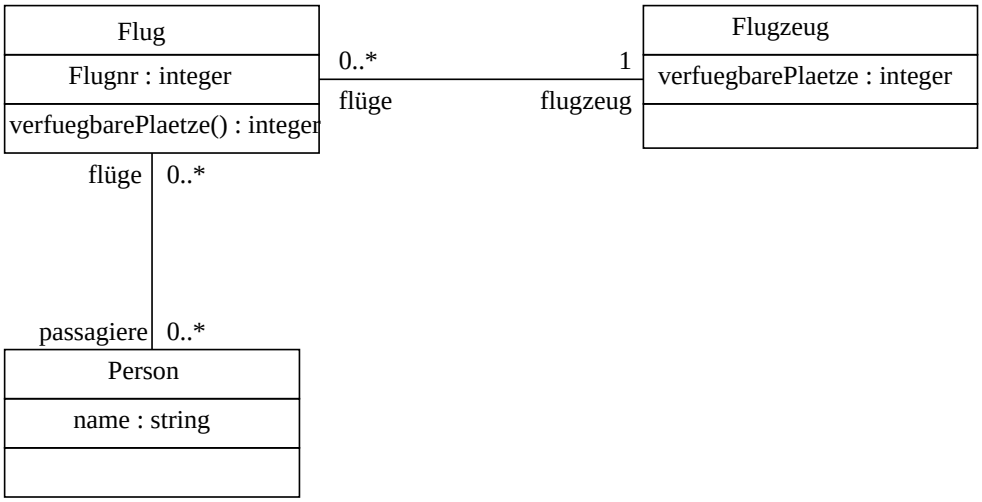


Abbildung 2.5: size() aller assoziierten Exemplare

Z.B. Implementiert durch:

```
class Flug{
Flugzeug* flugzeug;
int      Flugnummer;
vector <Person *>
   passagiere;
int verfuegbarePlaetze();
}
```

```
class Person{
string name;
vector <Flug *> fluege;
}
```

```
class Flugzeug{
vector <Flug *> fluege;
int verfuegbarePlaetze;
}
```

Abbildung 2.6: C++ Implementierung

Momentaner Status etwa:

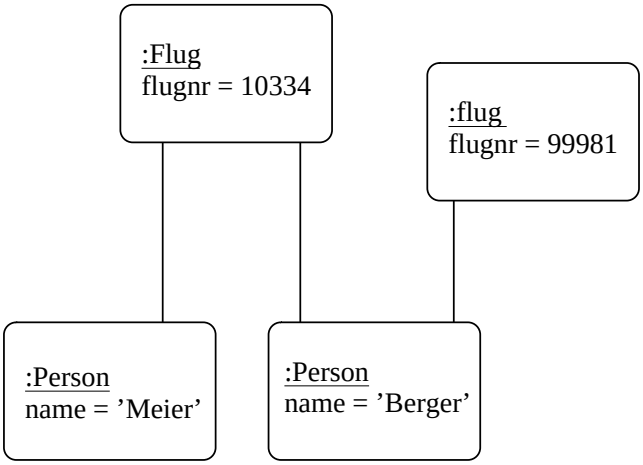


Abbildung 2.7: Zustand eines Objekts

Einschränkungen:

a) **context** Flug
inv: passagiere → size()<= flugzeug.verfuegbarePlaetze

b) context Flug :: verfuegbarePlaetze() : integer
body: flugzeug.verfuegbarePlaetze - passagiere → size()

2.4.2 includes() und forAll()

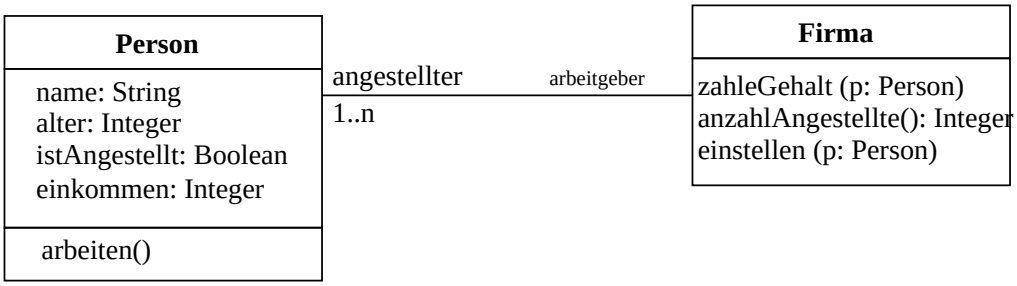


Abbildung 2.8: includes() und forAll()

Zu diesem Klassendiagramm liege der folgende, in OCL formulierte Vertrag vor:

```
context  Person
  inv:   alter > 0

context  Person::arbeiten()
  pre:   istAngestellt

context  Firma::zahltGehalt(p: Peerson)
  pre:   angestellter->includes(p)

context  Firma::anzahlAngestellte() : Integer
  pre:   angestellter->forAll(p | p.arbeitgeber = self)
  post:  result = angestellter->size()

context  Firma::einstellen (p: Person)
  pre:   (p.alter > 13) and not p.istAngestellt
  post:  (angestellter->size()) = angestellter@pre->size() + 1)
          and p.istAngestellt

context  Person
  inv:   if istAngestellt then
               einkommen >= 300
         else
               einkommen < 300
```

or	if -
and	then -
xor	else -
not	endif
=	
<>	
implies	

Tabelle 2.2: Boolsche Operationen in OCL

2.4.3 Assoziationsklassen

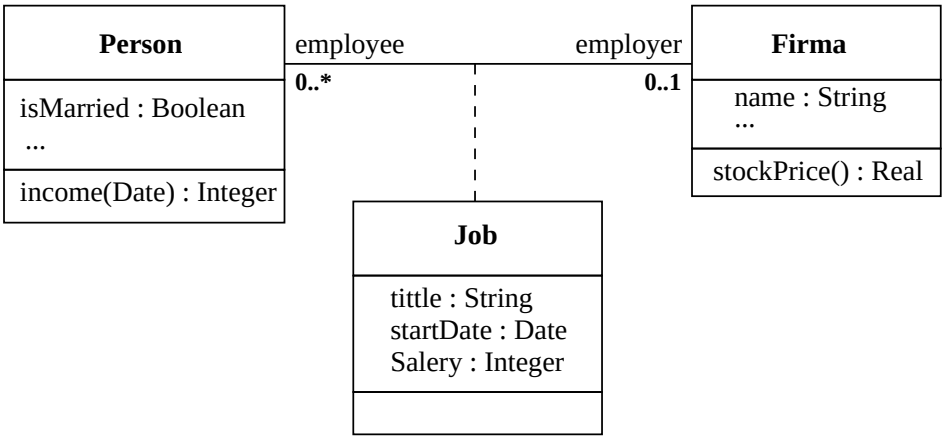


Abbildung 2.9: Includes

ist (automatisch) implementiert als:

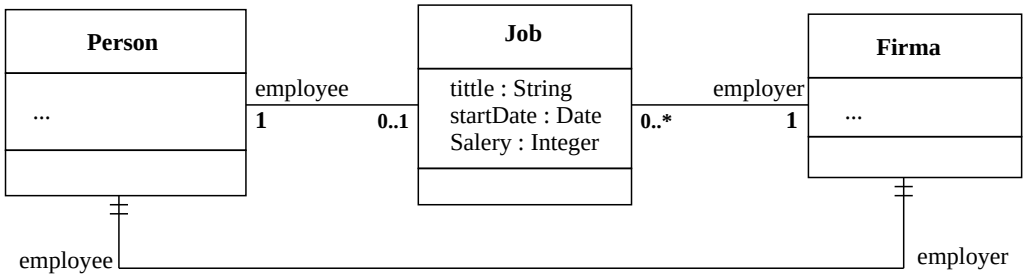


Abbildung 2.10: Includes

Hier gilt: Im **context firma** muß man mittels job.emplyoyer zum Arbeitgeber navigieren und kann ohne Einführung einer redundanten Assoziation nicht unmittelbar von Firma nach employer kommen.
Semantische Gleichheit erhält man nur durch die Einführung von virtuellen Attributen:

context Person
def: employer : Firma = job.employer

sowie

context Firma
def: employee : Set(Person) = job.employee

Die Spezifikation der Redundanzder Operation Person::income() lautet:
context Person :: income (d:Date) : Integer
body: job –> select(d >= startDate) –> sum()

2.4.4 Die Methoden der Collection Set

(im **context** Flugzeug)

fluege→ size()	Integer
fluege→ isEmpty()	Boolean
fluege→ notEmpty()	Boolean
fluege→ includes(Berlin_NewYork) ↑ von Typ Flug	Boolean
fluege→ includesAll(Berlin_NN) ↑ von Typ Set(Flug)	Boolean
fluege→ count(Berlin_NewYork)	⟵ Integer (bei Set immer **1**)
fluege→ excludes (Berlin_NewYork)	Boolean
fluege→ excludesAll (Berlin_NN)	Boolean
fluege.verfuegbarePlaetze→ sum()	Integer, Real, ...
=	Boolean
<>	Boolean
–	Differenzmenge
a → intersection (b)	Durchschnitt
a → union(b)	Vereinigungsmenge
a → symmetricDifference(b)	Menge aller Elemente in A oder B, aber nicht in beiden, mathematisch: $(A \cup B) \setminus (A \cap B)$
flatten()	rekursives Entpacken von ineinander verschachtelten Collections

Tabelle 2.3: Methoden für die Collection Set

Beispiele für Flugnummern:

$\underbrace{fluege}_{set(Flug)} \rightarrow \text{one}\,\underbrace{(FlugNr = 123)}_{vomTypBoolean}$ Boolean

true $\Longleftrightarrow$ Set(Flug) enthält genau einen Flug mit der FlugNr. 113

$\underbrace{fluege}_{set(Flug)} \rightarrow \text{isUnique}\,\underbrace{(FlugNr)}_{vomTypFlug}$ Boolean

true $\Longleftrightarrow$ Jeder Flug in **fluege** hat eine andere FlugNr als jeder andere Flug in **fluege**!