1 UML-Grundlagen

UML steht für „Unified Modelling Language“ (vereinheitlichte Sprache zur Modellbildung). Obwohl „Sprache“ genannt, handelt es sich weniger um eine Sprache, als mehr um einen graphischen Ansatz, der unterstützend bei der Programmentwicklung eingesetzt werden kann.

1.1 objektorientiertes und prozedurales Programmieren

Wer einmal versucht hat, ein objektorientiertes oder über Ereignisse gesteuertes Programm durch ein Fluß- oder Nassi-Shneiderman-Diagramm darzustellen, wird bemerkt haben, daß dies bestenfalls schwierig, meistens jedoch unmöglich ist.

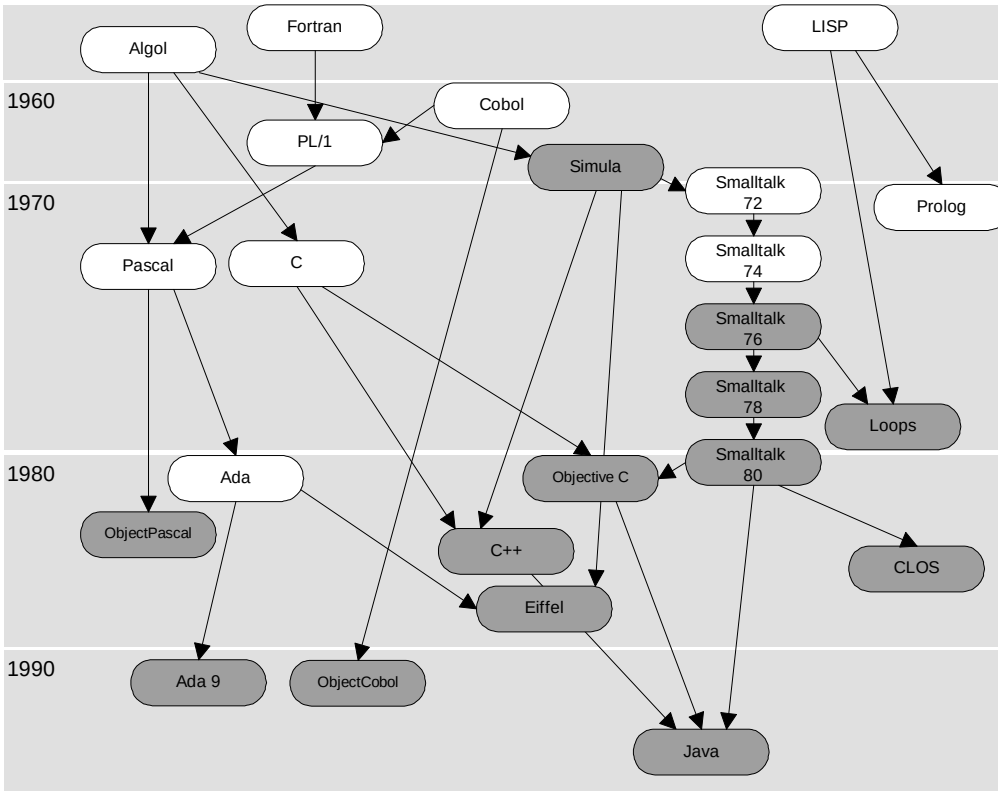

Abbildung 1-1 – Stammbaum der OOP-Sprachen

Ursache dieses Problems ist die Tatsache, daß Flußdiagramme lediglich in der Lage sind strikt sequentielle Abläufe wiederzugeben. Moderne, fenster- und ereignisorientierte Softwaresystem hingegen sind wesentlich komplexer in den resultierenden Möglichkeiten der Interaktion, da der Weg zu einem bestimmten Arbeitsergebnis weniger strikt vorgegeben ist, sondern statt dessen weitgehend intuitiv und nach individueller Präferenz gewählt werden kann.

Beispiel:
Für die Berechnung der Benzinkosten pro km eines Fahrzeugs sind drei Eingaben notwendig:

1. Anzahl der gefahrenen Kilometer seit der letzten Betankung
2. Benzinpreis pro Liter
3. verbrauchtes Benzin in Litern

Fragt man nun, wie in rein sequentiellen Systemen üblich, die drei Werte in genau der oben genannten Reihenfolge ab, so gibt es nur einen Weg zum Ergebnis. Die gewünschte Berechnung kann genau dann angestoßen werden, wenn die dritte Eingabe (verbrauchtes Benzin) eingegeben wurde.
In einem modernen Softwaresystem hingegen ist die Eingabereihenfolge egal. Demnach muß nach jeder Eingabe geprüft werden, ob nun genügend Daten für die Berechnung vorliegen und dann gegebenenfalls die Berechnung angestoßen werden.
Wie leicht zu ersehen ist, wächst die Komplexität der Software sprunghaft an, insbesondere wenn zwischen den einzelnen Feldern Plausibilitäten gelten.
Die beiden folgenden Graphiken stellen den sequentiellen und den über Ereignisse gesteuerten Ablauf gegenüber.

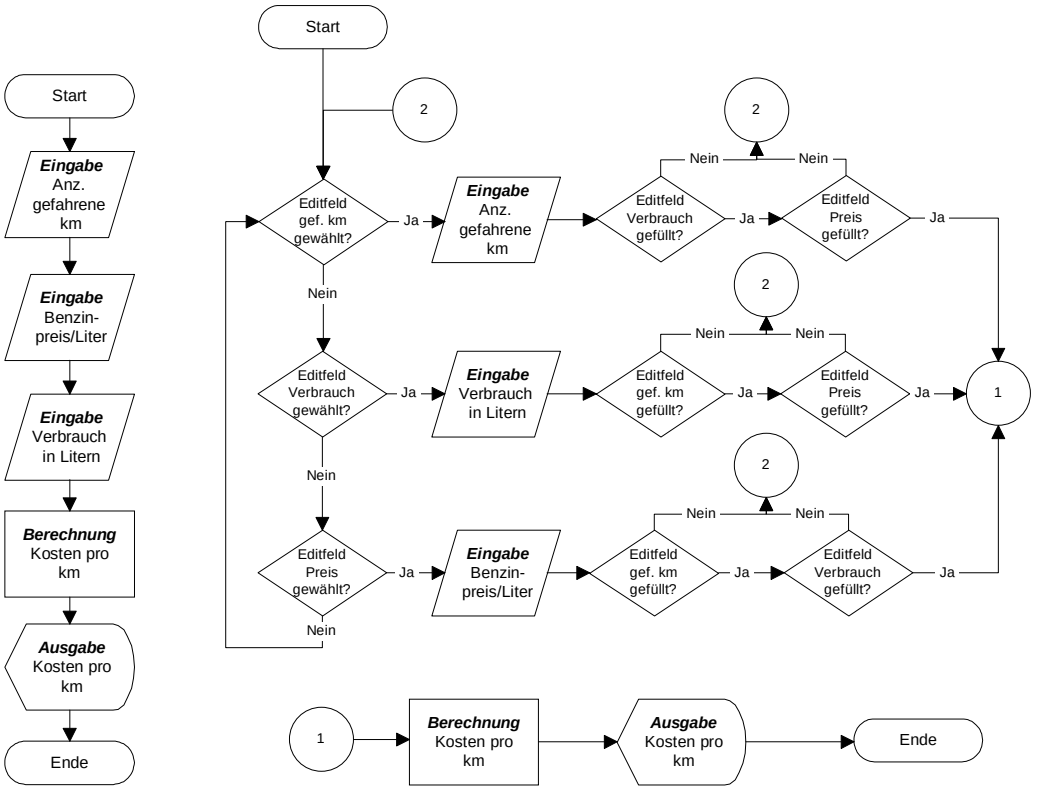

Abbildung 1-2 – Vergleich zwischen sequentieller und ereignisorientierter Programmierung

Wie die Abbildung des OOP[1]-Stammbaums zeigt, setzen alle „neuen“ Programmiersprachen seit den frühen 80‘er Jahren auf den objektorientierten Ansatz Paradigma. Der Wechsel der „Sichtweise“ von prozeduralen zu objektorientierten Vorgehensweisen wird in der Informatik häufig als Paradigmenwechsel[2] bezeichnet.
Der Vergleich der beiden Flußdiagramme hingegen zeigt, daß neue Denk- und Arbeitsweisen immer auch neue Werkzeuge und Darstellungsformen bedingen.
Die UML ist ein Werkzeug zur abstrakten, nicht an eine konkrete Programmiersprache gebundene Darstellung. Mit der UML lassen sich nicht nur Anwendungsprogramme sondern auch Prozessen, Beziehungen, Geschäftsvorfälle und viele andere Dinge darstellen,
Der Gesamtumfang der UML ist wesentlich größer und das Nutzungsspektrum viel breiter, als im folgenden Text dargestellt werden kann.

1.2 Was ist eigentlich Objektorientierung?

Objekte sind vereinfachte Abbildungen von physischen oder begrifflichen Gegenständen unserer Welt. Vereinfacht ausgedrückt ist damit gemeint, daß Software weniger mit Abläufen auf Daten definiert wird, wie in der prozeduralen Programmierung üblich, sondern Gegenstände (wie z.B. Formulare, Aktenordner, Locher, Laufzettel, Merkzettel, Maschinen) softwaretechnisch nachempfunden werden, so daß sie gleiche oder ähnliche Eigenschaften und Bedienungsformen aufweisen wie ihre realen Gegenstücke.
Kernstück der OOP-Sicht ist die sogenannte Datenkapselung (Data-Encapsulation), welche die in der prozeduralen Programmierung übliche Trennung von Daten und Programmfunktionen aufhebt. Daten und Bearbeitung/Veränderung dieser Daten (= Operationen) sind zu Objekten verschmolzen. Dies geht sogar so weit, daß die Datenrepräsentation in einem Objekt nach außen hin nicht mehr sichtbar sind. D.h. ein Objekt „Termin“ bietet zwar die Möglichkeit das Datum über eine Operation zu setze und abzurufen, wie das Datum aber im Terminobjekt gespeichert ist (z.B. als Text oder als Zahlenfeld), ist für den Benutzer des Terminobjektes nicht mehr erkennbar und auch nicht mehr relevant.
Das mag sich im ersten Augenblick ein wenig ungewöhnlich anhören, schließlich sind Programmentwickler es gewohnt die völlige Kontrolle über jedes Bit auszuüben, macht aber, übertragen auf die Alltagswelt (und OOP ist ja eine Annäherung an diese Sichtweise) durchaus Sinn. Als normaler Autofahrer muß ich nur wissen wie man bremst, ob dabei Scheiben- oder Backenbremsen zum Einsatz kommen und wie diese aufgebaut und montiert sind, ist es mir eigentlich egal, Hauptsache sie arbeiten und funktionieren in jedem Auto auf die gleiche Art und Weise.

[1] OOP: **O**bjekt**O**rientierte **P**rogrammierung
[2] Paradigma: eigentlich griech.-lat. für „Beispiel“, am einfachsten zu verstehen wenn man es mit „allgemeingültige oder –geläufige Sichtweise auf etwas“ übesetzt.

Data-Encapsulation:
Das Verbergen von Implementierungsdetails in einem Objekt. Der Objektanwender bekommt nur die funktionale Schnittstelle eines Objektes zur Kenntnis, der interne Aufbau hat ihn nicht zu interessieren.

Der erste Bereich, der massiv objektorientierte Begrifflichkeiten eingesetzt hat, waren die Betriebssysteme. die dort aufgebaute Sichtweise der Mensch-Maschine-Schnittstelle als eine Art von Schreibtischoberfläche[3] (Desktop) hat sich mittlerweile so populär, selbstverständlich und verbreitet, daß wir sie kaum noch wahrnehmen. So werden z.B. Dateien – je nach Inhalt – als Piktogramm in Form einer Textseite oder eines Bildes angezeigt. Verzeichnisse hingegen werden mit Aktenordnern gleichgesetzt.

Die eigentliche Idee dahinter ist weniger neu als sie scheint, schon gegen Ende der 60'er Jahre erschien mit Simula[4] die erste Sprache, die objektorientierte Konzepte beinhaltete, aber erst die graphischen Benutzungsoberflächen konnten der Schreibtischmetapher zum Durchbruch verhelfen.
Mittlerweile hat sich die objektorientierte Sichtweise auch auf nahezu alle anderen Bereiche ausgedehnt und zunehmend auch abstraktere Formen angenommen.
So gibt es unter anderem Berechnungsobjekte, Geschäftsvorfallobjekte, Interfaceobjekte, Utilityobjekte, Zugriffsobjekte und Container[5]-Objekte (Behälter), letzteres sind Objekte, die (wie der Name schon sage) Sammlungen von anderen Objekten aufnehmen können.

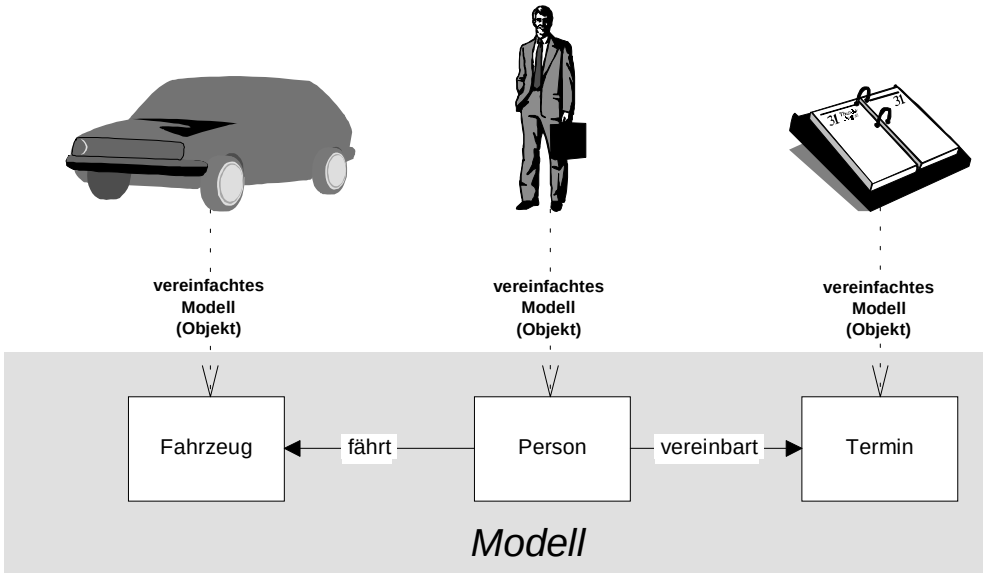

Abbildung 1-3 – Modellbildung

[3] Der Desktop ist wohl der einzige Ort, bei dem der Papierkorb immer auf der Schreibtisch-oberfläche steht...
[4] Akronym: **Simu**lation **la**nguage
[5] Ein Containerobjekt ist wohl der einzige Behälter, der eine theoretisch beliebige Anzahl von Objekten aufnehmen kann, von denen jedes einzelne größer sein kann als der Behälter...

Die Vereinfachung bei der Modellbildung besteht in der Reduktion der vorhandenen Eigenschaften (= Daten) auf die hinreichende Menge, die benötigt wird, um das Ziel zu erreichen.
Die oben abgebildeten, realen Objekte Auto und Person sind ungemein komplizierte Gebilde. Derart komplexe Gebilde datentechnisch vollständig zu beschreiben ist schlicht unmöglich.
Eine Person ist außerdem ein abstraktes Gebilde, d.h. es gibt nicht nur meßbare Attribute (wie z.B. Körpergröße, Gewicht, Anzahl der Finger, Schuhgröße), sondern auch solche, die sich nicht konkret von der physischen Repräsentation einer Person (= Körper) ablesen lassen), die aber zur eindeutigen Identifizierung benötigt werden (Geburtstag, Geburtsort, Name.
Daher muß bei jeder Modellbildung zuerst die Frage gestellt werden, welche Kenngrößen für die zugrundegelegte Problemstellung überhaupt sinnvollerweise benötigt werden.
In den meisten Fällen reichen schon wenige Kenngrößen aus, um das Objekt für die zu lösende Problemstellung hinreichend konkret zu beschreiben.

Angenommen, man möchte das Fahrzeug im Sinne eines Mietwagens modellieren, dann sind sowohl die physischen Attribute des Wagens (Anzahl Türen, Anzahl der Sitze, Sonderausstattung), wie auch die „gesellschaftlichen“ Attribute wichtig (Mietpreis pro Tag, Kategorie).
Das schließlich erzeugte Fahrzeugmodell wird in der UML, genau wie in C++, als Klasse bezeichnet. Die graphische Darstellung erfolgt durch ein Rechteck, welches den Klassennamen enthält:

Abbildung 1-4 – einfache UML-Darstellung einer Klasse

Die Klasse selbst ist noch kein Objekt, sondern nur das Modell eines Objektes, also eine Art Blaupause. D.h. ein Bauplan nach dem Objekte errichtet bzw. erzeugt werden können.

Bezeichnung	Attribut	Wert
Objekt 1	Name	Vespa
	Anzahl Türen	0
	Anzahl Sitze	1
	Sonderausstattung	keine
	Mietpreis pro Tag	35
	Kategorie	V
	Farbe	Blau
	Fahrgestellnummer	V3789249823

Bezeichnung	Attribut	Wert
Objekt 2	Name Anzahl Türen Anzahl Sitze Sonderausstattung Mietpreis pro Tag Kategorie Farbe Fahrgestellnummer	Fiat Panda 2 4 Radio Typ 1 70 A Rot 34F87234987G
Objekt 3	Name Anzahl Türen Anzahl Sitze Sonderausstattung Mietpreis pro Tag Kategorie Farbe Fahrgestellnummer	Fiat Panda 2 4 Radio Typ 1 70 A Grün 34F872231A2S
Objekt 4	Name Anzahl Türen Anzahl Sitze Sonderausstattung Mietpreis pro Tag Kategorie Farbe Fahrgestellnummer	Volvo 940 4 5 Radio Typ 2, Klimaanlage 130 B Silber 213434Vo21643
Objekt 5	Name Anzahl Türen Anzahl Sitze Sonderausstattung Mietpreis pro Tag Kategorie Farbe Fahrgestellnummer	Porsche 911 2 3 Radio Typ 2 250 C Gelb Por43231654344

Ein Objekt entsteht erst dann, wenn gemäß der Bauvorschrift „Klasse“ ein Objekt erzeugt (= Instanziiert) und mit konkreten Daten gefüllt wird (der Begriff „Instanz[6] einer Klasse“ leitet sich vom englischen „Instance of a class“ ab).

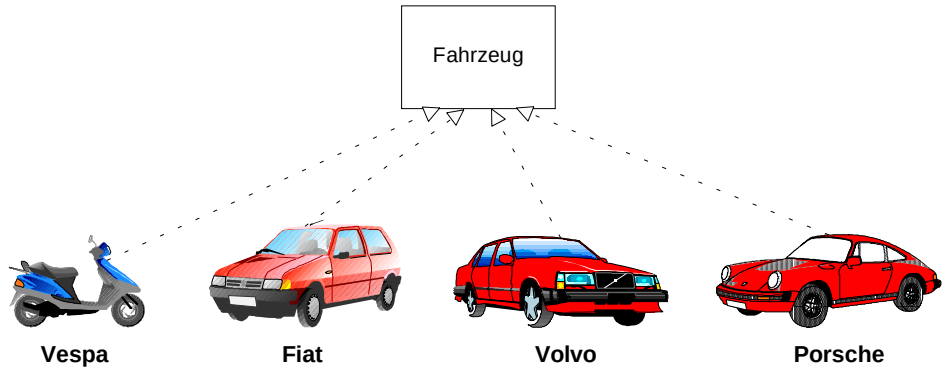

Abbildung 1-5 - Objektableitung

[6] Müßte eigentlich mit „Exemplar einer Klasse“ übersetzt werden. Die Begriffe „Instance“ im Englischen und „Instanz“ im Deutschen sind nicht synonym. Dieser Übersetzungsfehler hat sich jedoch eingebürgert.

Wie man nun erkennen kann, ist Fahrzeug selbst kein Objekt sondern kann als Datentyp aufgefaßt werden. Die oben gezeigte Tabelle definiert also fünf Objekte vom Typ Fahrzeug.

Klasse:
Eine Klasse beschreibt die Struktur und das Verhalten einer Menge von Objekten gleichen Typs.

Objekt:
Ein Objekt ist eine zur Ausführungszeit des Programms erzeugte, speicherplatzverbrauchende (= allozierte) Instanz, die sich gemäß ihrer Typdefinition (= Klasse) verhält.

Dies steht in völliger Übereinstimmung mit dem sprachlichen Gebrauch in der menschlichen Gedankenwelt. Im Straßenverkehr begegnen uns täglich viele Fahrzeuge, aber ein „Fahrzeug an sich“ gibt es nicht.

Die Darstellung eines Objektes erfolgt in der UML, ähnlich wie die einer Klasse, durch ein Rechteck, wobei statt des Klassennamens eine eindeutige, unterstrichene Objektbezeichnung angegeben wird:

Abbildung 1-6 – UML-Darstellung eines Objektes

Obwohl sich in der oben dargestellten Tabelle alle Objekte durch zumindest ein Attribut unterscheiden, muß dies nicht zwangsläufig der Fall sein. Verzichtet man auf die Fahrgestellnummer, so lassen sich leicht Fahrzeuge definieren, die in Puncto Eigenschaften und Ausstattung vollkommen gleich sind. Es ist im Analyse und Design jedoch anzustreben solche Uneindeutigkeiten zu vermeiden, um eindeutige Objektbezeichnungen und –identifizierungen zu ermöglichen. Die meisten Problemlösungen setzen ohnehin eine eindeutige Identifizierbarkeit der Objekte voraus.

Die Instanziierung des Objektes wird mit Hilfe eines gestrichelten, offenen, ggf. beschrifteten Pfeils dargestellt, der vom Objekt auf die Klasse zeigt:

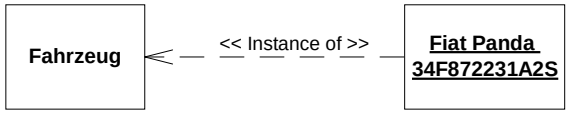

Abbildung 1-7 – UML-Darstellung einer Objekt-Instanziierung

Zu lesen ist dies als „Fiat Panda 34F872231A2S ist ein Exemplar der Klasse Fahrzeug“.

1.3 Was UML ist und was sie nicht ist...

Die UML ist keine Analyse- oder Designmethode, sondern ein graphisches Werkzeug, daß mit den unterschiedlichsten Methoden eingesetzt werden kann.
Eine Methode beschreibt einen Weg von der Analyse eines Problems oder einer Aufgabe, über das Design, bis hin zu ihrer Lösung. Ein methodischer Ansatz gibt bestimmte Schritte vor und verspricht als Ergebnis, bei korrekter Durchführung, eine Lösungsdarstellung, die sich ohne weiteres softwaretechnisch realisieren läßt.
Fast jede der vielen am Markt vertretenen Methoden definiert eigene Diagramme, die den Analyse- bzw. Designprozeß unterstützen sollen. Häufig sind diese Diagramme nicht übergreifend definiert, sondern ihre Ausprägungen ändert sich innerhalb des Analyse- bzw. Design-prozesses von Schritt zu Schritt. Häufig genug werden komplett neue Darstellungsweisen verwendet, anstatt die bereits gewonnenen Ergebnisse weiter zu verwerten.
Zudem werden die meisten Methoden nicht „pur“ eingesetzt, sonder haben hauseigene oder sogar personentypische Spielarten entwickelt, die von der ursprünglich publizierten Form mehr oder weniger stark abweichen.

Die UML versteht sich als Werkzeug, will also die Methoden nicht ersetzen, sondern um eine extrem leistungsfähige graphische Repräsentation der während des Prozesses gewonnenen Ergebnisse bereichern. Die UML bietet fertige Konzepte für die gängigsten und am häufigsten akzeptierten Darstellungsformen. Dies sind logischerweise genau jene Darstellungsformen, die in der Praxis die höchste Bedeutung gewonnen haben, weil sie den Prozeß entscheidend unterstützen.
Darüber hinaus versucht die UML den verschiedenen Prozeßphasen durch unterschiedlich detaillierte Diagramme Rechnung zu tragen, statt unterschiedlich strukturierte Darstellungen zu verwenden. Dazu wird der Prozeß grob in die Phasen Analyse, Design und Implementierung unterteilt, wobei jede dieser Phasen aus mehreren Schritte oder Unterphasen bestehen kann.

Abbildung 1-8 – vereinfachter, linearer Prozeßablauf

Auch bleibt die UML nicht nur – gemäß ihrer Intention als Werkzeug – offen für Erweiterungen und Abwandlungen, sondern bietet dafür Bereiche innerhalb der Diagramme, an denen die Notation erweitert werden kann.

Welche Arten von Diagrammen die UML definiert und wie sich der unterschiedliche Detailgrad in diesen niederschlägt, wird in den nachstehenden Abschnitten ausführlicher behandelt.

2 Entwicklungsphasen und UML

Wie bereits festgestellt, zerfällt eine Programmentwicklung grob gesehen in drei Phasen:

- Analyse – was ist in der Software abzubilden?
- Design – wie ist in der Software abzubilden?
- Implementierung – Erstellung der Software.

Die häufig gestellte Frage, warum man nicht einfach ohne ausgiebige Analyse programmieren soll und die sich dann zwangsläufig ergebenden Probleme löst – es gibt ja ohnehin den unten abgebildeten Entwicklungszyklus, also kann man sich die Zeit sparen – beantwortet der Autor Bernd Oestereich[1] wie folgt:

> „Die Anforderungsanalyse erhält ihre Berechtigung dadurch, daß normalerweise die SoftwareentwicklerInnen keine ExpertInnen in dem Anwendungsfachgebiet sind. Um eine akzeptable und erfolgreiche Software entwickeln zu können, müssen Informationen über das zu lösende Problem gesammelt werden. Erst wenn die zu lösende Aufgabe umfassen beschrieben und evtl. Widersprüche ausgeräumt wurden, kann mit ihrer Lösung begonnen werden. [...] “

Natürlich kann man Software auch ohne ausgiebige Analyse- und Designphase entwickeln. Die Erfahrung zeigt jedoch, daß die dadurch eingesparte Zeit im ***günstigsten*** Fall in der Implementierungsphase benötigt wird, um strukturelle Fehler wie z.B. seltene Ausnahmen, zusätzliche Geschäftsprozesse und andere Dinge zu bereinigen, die in einem Entwurf ad hoc zunächst übersehen werden.

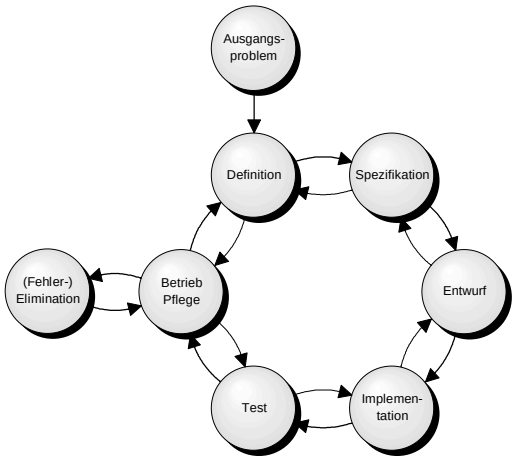

Abbildung 2-1 – klassischer Entwicklungszyklus

Die Betonung liegt hier auf ***günstigsten***, denn ohne Analyse- und Designphase besteht die grundsätzliche Gefahr, daß grobe Entwurfsfehler gemacht werden, die im schlimmsten Fall (worst case)

[1] entnommen: [Oestereich98]

die gesamte Implementierung in Frage stellen. Häufig ist dies z.B. aufgrund falsch eingeschätzter Mengenabschätzungen für bestimmte Geschäftsvorgänge der Fall.

2.1 Analyse

Problemanalysen und -studien sind ein sehr komplexes Thema, welches an dieser Stelle nicht einmal annähernd beschrieben werden kann. Es gibt eine große Anzahl von Methoden, die Wege und Vorschläge unterbreiten, wie eine Problemstellung angegangen werden kann, um zügig und möglichst vollständig zu einer vollständigen Datensammlung über die zu lösende Aufgabe zu kommen.
Zu den gängigen Darstellungsmitteln der Analyse gehören u.a.

- *Fachlexika*
- *Geschäftsprozeßmodelle*
- *Geschäftsklassenmodelle*
- *Anwendungsfallmodelle (= Use-Case-Anwendungen)*
- *Aktivitätenmodelle*
- *CRC-Karten (Class-Response-Collaborators[2])*
- *Mind-Maps*
- *Dialogentwürfe (Bildschirmmasken)*

Die relevante Frage, die von der Analyse zu beantworten ist, lautet ***was*** muß die zu schaffende Software können. Die Frage ***wie*** dieses Ziel zu erreichen ist (und mit welchen Mitteln), stellt sich zu diesem frühen Zeitpunkt noch nicht.
Zu den Fragen, die im Mittelpunkt der Analyse stehen gehören u.a.:

- *Welche Gegenstände (materiell und immateriell) werden zur Bewältigung der Aufgabe verwendet? (Formulare, Geräte, Personen, Systeme etc.)*
- *Welche Eigenschaften bzw. welchen Aufbau besitzen diese Gegenstände? (woraus bestehen diese Dinge, z.B. Subsysteme oder Formularanlagen)*
- *Wie werden diese Gegenstände benutzt? (z.B. eintragen von Daten in ein Formular)*
- *Wer benutzt welche Gegenstände?*
- *Welche Erwartungen haben die zukünftigen Anwender an das System?*
- *Welche Funktionalitäten wären über die reine Abbildung des Arbeitsprozesses hinaus wünschenswert?*
- *Welche Arbeitsanleitungen bzw. Vorgehensweisen/Schemata existieren bereits?*
- *Welche offiziellen und inoffiziellen Absprachen gibt es zwischen den beteiligten Bereichen/Personen?*
- *Wie sind Befugnisse verteilt?*
- *Welche Zwischenergebnisse sind zu produzieren?*

[2] übersetzt: Klassen, Verantwortlichkeiten und Beteiligte – also die klassische Frage wer macht was mit wem?

- *Welche Ausnahmen gibt es von einem Standardvorgehen und wodurch werden diese ausgelöst? (bezogen auf Daten, Zeit und/oder Personen)*
- *Wie verlaufen die Kommunikationswege? (z.B. welche Formulare/Kopien gehen wohin und an wen)*
- *Wie soll auf unerwartete Ergebnisse/Ereignisse reagiert werden?*
- *Welche Abläufe sind aus Sicht der Anwender gut und welche nicht? (es lohnt eventuell schlechte Abläufe neu zu konzipieren)*

Umfassende Darstellungen des Themas Analyse finden sich u.a. in den Abhandlungen von Booch, Coad/Yourdon, Martin, Rambaugh oder Waldén[3]

2.2 Design

In der Designphase werden üblicherweise UML-Klassendiagramme (s.u.) auf geringem Detailniveau aufgebaut und anschließend sukzessiv verfeinert.
Der Vorteil des UML-Einsatzes ist, daß eine Entscheidung über die einzusetzenden Werkzeuge (u.a. die Programmiersprache) im Prinzip noch nicht notwendig ist. Der Einsatz eines Software-Engineering-Tools (CASE[4]) engt den Entscheidungsraum aber meist auf wenige, von diesem Tools unterstützte Sprachen ein.
Am Ende dieser Phase steht ein detailliertes Klassendiagramm, welches in der anschließenden Implementationsphase umgesetzt wird.

2.2.1 Designprobleme

Designprobleme sind die Hauptursache von unzureichenden, in der Praxis enttäuschenden Systemen.
Fast immer sind die Probleme im Design das Ergebnis einer ungenügenden Analyse (wie fast immer in der Informatik ist Zeitmangel ursächlich).
Zumeist handelt es sich um Probleme bei der Einordnung der Klassenstruktur (Vererbungshierarchie) oder dem Aufbau von Beziehungen (z.B. Multiplizität von Beziehungen s.u.).
Als sehr einfaches Beispiel kann die Ableitung von einer Basisklasse „Graphikelement“ auf die Klassen „Ellipse“ und „Kreis“ dienen.
Für die im Beispiel verwendeten Klassen kann es nur drei verschiedene Vererbungsschemata geben:

1. „Kreis“ und „Ellipse“ sind unabhängig von einander, sie basieren auf einer gemeinsamen Basisklasse „Graphikelement“. Diese Lösung ist zwar möglich, widerspricht jedoch dem Gedanken der Wiederverwendung durch Vererbung in der OOP. Das sich „Kreis“ und „Ellipse“ voneinander ableiten lassen dürfte auf der Hand liegen.

[3] [Booch94], [CoadYourdon91a], [Martin92], [Rambaugh93], [Waldén95]
[4] CASE = Computer Aided Software Engineering

2. „Kreis“ ist eine Basisklasse von „Ellipse“. Betrachtet man die notwendigen Datenstrukturen, die aufzubauen sind, so scheint dies die Ableitung der Wahl zu sein. Letztlich läßt sich ein Kreisobjekt einfach durch Mittelpunkt und Radius beschreiben, während ein Ellipsenobjekt zusätzliche Angaben für den zweiten Radius und den Neigungswinkel benötigt:

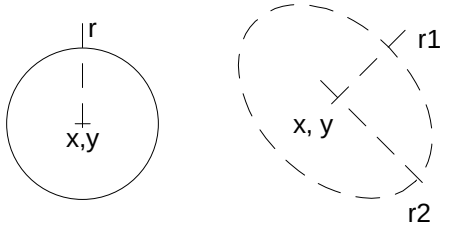

Abbildung 2-2 – Kreis und Ellipse

3. „Ellipse“ ist eine Basisklasse von „Kreis“. Dies entspricht dem mathematischen Denken, den Kreis als Sonderform der Ellipse zu betrachten, in der *zufällig* die beiden Radien gleich sind und der Winkel daher keine Rolle spielt. Obwohl etwas aufwendiger zu implementieren ist dies zugleich auch die korrekte Form für die Vererbung. Im Gegensatz zu (3.) wirft die Ableitungsform (2.) ein Polymorphieproblem auf – nach dieser Ableitung sind alle Ellipsen auch Kreise, was wohl eindeutig auszuschließen ist.

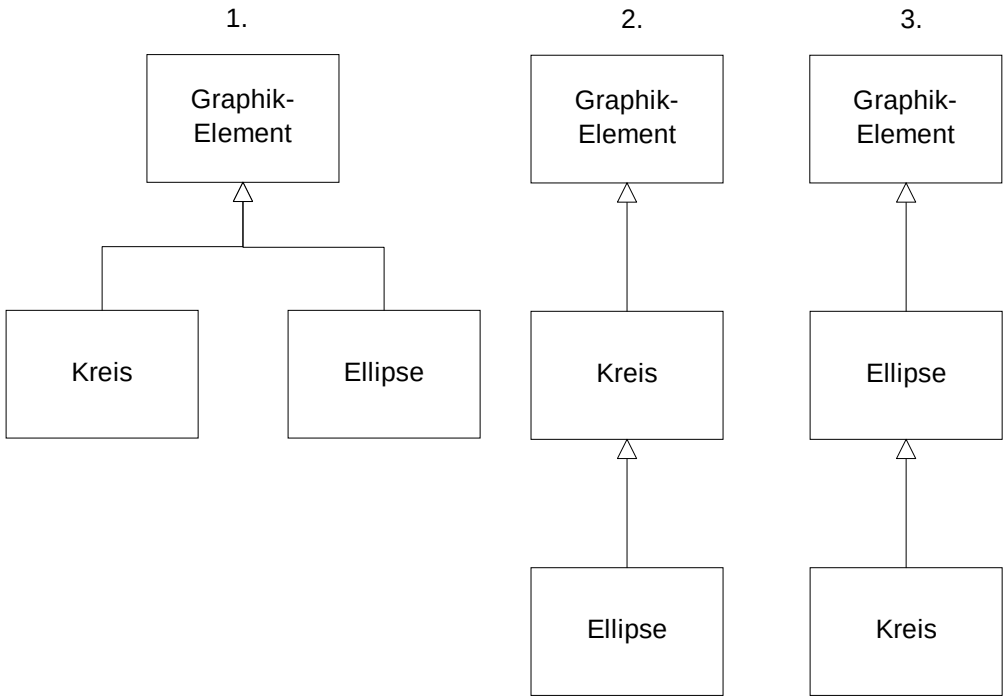

Abbildung 2-3 – Ableitungsmöglichkeiten

2.3 Implementation

Nach Abschluß der Designphase kann mit der Detailimplementation begonnen werden. Ist während der Designphase ein CASE-Tools verwendet worden, so kann üblicherweise zumindest ein Teil des zu implementierenden Systems generiert werden.

Häufig erfolgt die Generierung aus einem System heraus, in dem die Analyse und das Design in graphischer und textueller Form stattgefunden hat. Die graphischen Designteile (UML-Diagramme) lassen sich fast immer direkt in Quellcodes der unterstützten Programmiersprachen überführen, häufig wird auch die gesamte Detailimplementation mit dem CASE-Tool durchgeführt. In diesem Fall wird der gesamte Quelltext des Systems in einer Datenbank gespeichert (Repository) und von dort aus in Quelltexte überführt.

Bei auf Datenbanken basierenden CASE-Tools müssen alle Programmänderungen und Weiterentwicklungen im CASE-Tool erfolgen.

Im Gegensatz dazu stehen auf Quelltexten basierende CASE-Tools, die ohne Datenbanken auskommen. Bei diesen bildet der Quelltext selbst die Grundlage, welche bei jedem Programmaufruf neu analysiert werden muß, wenn sie sich geändert hat (dies erfolgt üblicherweise automatisch). Die auf Quelltexten basierenden CASE-Tools haben den Vorteil auch noch nachträglich und auf bereits bestehenden Systemen eingesetzt werden zu können – sie verführen allerdings auch zur Vernachlässigung der Dokumentationsaspekte von CASE-Tools.

Die textuellen Beschreibungen, Notizen und Anmerkungen der Analyse und Designphase bilden die Dokumentation, die von einem CASE-Tool weitgehend automatisch erzeugt werden kann (sofern der Text vernünftig eingegeben und mit den entsprechenden Klassen-diagrammen verknüpft wurde – zaubern können schließlich auch CASE-Tools nicht). Auf Quelltexten basierende CASE-Tools legen diese Texte als Kommentare in den Quelltexten ab (was äußerst hilfreich ist, wenn man – was natürlich strikt vermieden werden sollte – doch mal per Hand an den Quellcodes ändern muß).

3 UML Diagramme

Das Ergebnis der Analysephase, zumeist Modelle der Geschäftsvorfälle oder Anwendungsfälle, können graphisch umgesetzt werden. Die graphische Repräsentation erleichtert die Umsetzung in Design und Implementierung, da die relevanten Teile leichter zu erkennen und zu gruppieren sind.

3.1 Klassendiagramme

Im Mittelpunkt der UML stehen die Klassendiagramme, die für den Programmentwurf von zentraler Bedeutung sind. Den Klassendiagrammen kommt in der OOP in etwa die gleiche Bedeutung zu, wie in den Flußdiagrammen in der prozeduralen Programmierung.

Klassendiagramme können unterstützend in allen drei Phasen der Prozesses (Analyse, Design und Implementation) eingesetzt werden.

Das folgende Bild zeigt ein Klassendiagramm in voller Ausprägung, wie es in dieser oder leicht reduzierter Form für die Implementierungsphase eingesetzt wird. Während der Design- und Entwurfsphase hingegen werden stark vereinfachte Diagramme verwendet, die unter Umständen nur noch den Klassennamen enthalten.

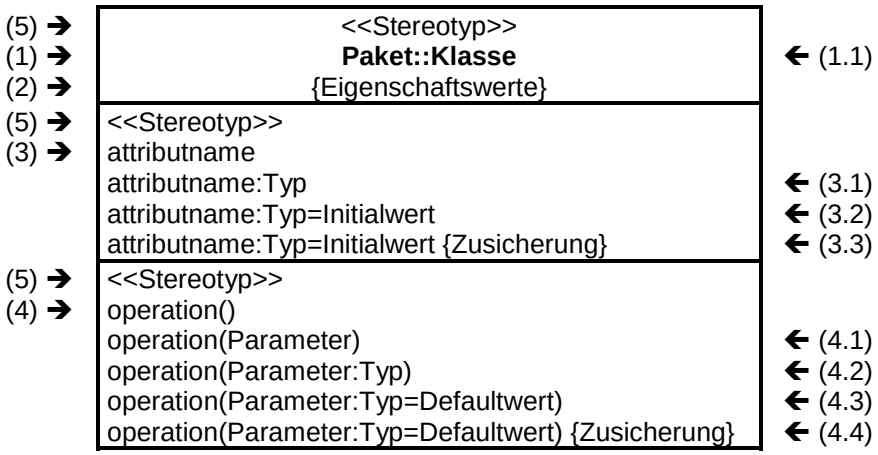

Abbildung 3-1 – Aufbau eines Klassendiagramms

(1) Klasse

Als einzige nicht optionale Angabe steht hier der Name der Klasse an exponierter Stelle.
Ein Klassenname sollte immer gewissen Regeln (Namens-konventionen) gehorchen[1]. Namenskonventionen sollten sicherstellen, daß sich Sinn, Aufgabe und Inhalt der Klasse leicht aus dem Namen erschließen lassen.

[1] Die UML-Notation empfiehlt z.B. alle Attribute und Operationen mit Kleinbuchstaben zu beginnen und nur Klassennamen mit einem großen Buchstaben beginnen zu lassen. Diese im englischen Sprachraum sicherlich einfach anzuwendende Regel wirkt im Deutschen allerdings häufig ein wenig befremdlich und hemmt den gewohnten Lesefluß. Die Übernahme dieser Regel sollte daher wohlüberlegt sein.

Man kann z.B. leicht zwischen Klassen unterscheiden, die reale Objekte repräsentieren und daher mit einem Substantiv benannt werden können (d.h. dem Namen des realen Objektes wie z.B. „`Fahrzeug`“), und solchen Klassen, die Tätigkeiten beschreiben und dabei andere Objekte manipulieren. Für die Benennung letzterer Klassen eignen sich eher Verben oder Substantiv-Verb-Mischungen (z.B. „`FahrzeugNeuErfassen`).

(1.1) Paketname (optional)

Bei der Erstellung großer Softwaresysteme ist eine Zerlegung des Gesamtsystems in mehrere Teilsystem unerläßlich. In der UML werden solche Teile „Pakete“ genannt (andere geläufige Bezeichnungen für Teilsysteme sind „Modul“ oder „Komponente“).
Die Komponenten- oder Paketbildung ist ein abschließender Teil der Designphase und erfordert ein gesundes Maß an Fingerspitzengefühl und Systemverständnis. Ziel der Paketbildung das System so zu zerlegen, daß die einzelnen Komponenten möglichst in sich geschlossene Subsysteme bilden und nur so wenig Berührungspunkte zu anderen Subsystemen aufweisen wie möglich.

(2) Eigenschaftswerte/Zusicherungen (optional)

Eigenschaftswerte bzw. Zusicherungen sind benutzerdefinierte, meist sprachspezifische Hinweise, die dem Klassendiagramm hinzugefügt werden können, um zumeist Implementierungsrelevante Zusatzinformationen hinzuzufügen.
Einer der wichtigsten Eigenschaftswerte ist z.B. der Wert `{abstrakt`[2]`}`, der eine Klasse als virtuelle, nicht-instanziierbare Basisklasse definiert.
Der Eigenschaftswert `{abstrakt}` ist die mit Abstand häufigste Eigenschaft, die schon frühzeitig in UML-Diagrammen dokumentiert werden kann. Daher kann diese Eigenschaft auch dadurch ausgedrückt werden, daß der Klassenname *kursiv* geschrieben wird. Die kursive Notation spart vor allem auf größeren Diagrammen mit einer sehr großen Zahl von Klassen eine Menge Platz. Allerdings eignet sich die kursive Schreibweise nicht für handgezeichnete Diagramme, dort hat sich statt dessen die Kurzbezeichnung `{a}` eingebürgert.
Ebenfalls häufiger zu finden sind Angaben wie `{readonly}` oder `{obsolete}`. Seltener wird der Eigenschaftswert verwendet, um den Autoren oder die Version zu nennen:

```
{Autor=H.Gorski}
{Version=2.1}
```

Die Verwendung mehrerer Eigenschaftswerte ist durch einfache Aufzählung möglich:

```
{abstrakt Version=1.4}
```

[2] Korrekterweise müßte der Eigenschaftswert lauten {abstrakt=true}. Vereinbarungsgemäß können in der UML aber Boolean-Werte entfallen wenn sie *true* sind.

(3) Attributname (optional)

Attribute sind die Variablen, die in einer Klasse verwaltet werden. Die Liste der Attribute wird, zwecks besserer Lesbarkeit von der Klasseninformation durch eine waagerechte Linie getrennt.
Die Sichtbarkeit (`private`, `protected` oder `public`) der Variablen gilt als Attributwert (s.u.), wird aber als Ausnahme vom Standardschema vor dem Attributnamen notiert:

```
-AttributPrivate
#AttributProtected
+AttributPublic
```

Klassenattribute, die in C++ durch das Schlüsselwort static gekennzeichnet sind, werden zusätzlich unterstrichen:

#StaticAttributProtected

(3.1) Attributtyp (optional)

Hier wird, durch einen Doppelpunkt vom Attributnamen getrennt, der Datentyp des Attributs notiert, sofern dieser bekannt ist. Dabei ist es keineswegs notwendig, sich auf die Standarddatentypen der Programmiersprache zu beschränken. Eigene Datentypen, wie im nachstehenden Beispiel, sind natürlich wieder eigene Klassen, die entsprechend im UML-Diagramm aufzunehmen sind:

```
#BLZ : Bankleitzahl
```

(3.2) Attributwerte (optional)

Die Angabe von Attributwerten ist die Darstellung des Initialwertes, die ein Attribut bekommt.

(3.3) Attributzusicherungen (optional)

An dieser Stelle werden bestimmte Eigenschaften des Attributs festgehalten, die für die Implementierung von Bedeutung sind. In C++ handelt es sich z.B. um logische Ausdrücke für den Wertebereich einer Membervariablen, wie z.B.:

```
#Alter : integer {Alter=Heute-Geburtsdatum}
-Farbe : {red, green, blue}
```

Auch als Attributzusicherung gilt die Sichtbarkeit (`private`, `protected` oder `public`) der Variablen. Da diese Angabe jedoch bei praktisch jedem Attribut von Relevanz ist, wird sie in Kurzform als Symbol vor dem Attributnamen notiert.

(4) Operationen (optional)

Operationen sind die Funktionen (Methoden) einer Klasse. Die Liste der Operationen wird, zwecks besserer Lesbarkeit von der Attributliste durch eine waagerechte Linie getrennt. Fehlt die Attributliste, so empfiehlt es sich, die Operationenliste durch zwei waagerechte Linien von der Klasseninformation zu trennen, um diesen Umstand darzustellen.
Die Sichtbarkeit (`private`, `protected` oder `public`) der Methoden wird wie bei den Attributen vor dem Namen notiert:

```
-setzeOperationPrivate()
#setzeOperationProtected()
+holeOperationPublic()
```

Zwischen virtuellen und nicht-virtuellen Methoden wird in der UML nicht unterschieden. Lediglich abstrakte Methoden (pure virtual functions) werden – wie auch die Klassen – durch *kursive* Schreibweise bzw. die Zusicherung `{abstrakt}` ausgewiesen.
Die UML-Namenskonventionen sehen vor, daß alle Operationen mit einem Kleinbuchstaben beginnen. Im Gegensatz zu den Attributen ist dieses auch im Deutschen möglich ohne befremdlich zu wirken, indem man ein Verb der Operation voranstellt. Da eine Operation immer eine Tätigkeit durchführt, läßt sich auch immer ein sinnvolles Verb finden, wie z.B.:

```
+setzeMonat()
+pruefeDatum()
+rechneGesamtbetrag()
+fuelleFahrzeugdaten()
```

(4.1) Operationsparameter (optional)

Bei Bedarf und soweit bekannt, können die Parameter einer Methode mit angegeben werden. Prinzipiell gelten für Parameter die gleichen Möglichkeiten und Darstellungsformen wie für Attribute, lediglich die Attributzusicherungen würden in die Operationszusicherungen (s.u.) einfließen. Somit ist es, analog zu den Attributnamen, nicht nötig, daß die konkreten Datentypen bereits festgelegt werden:

```
+SetzeKoordinaten(x, y)
```

(4.2) Operationsparametertyp (optional)

Hier wird der Datentyp des Parameters, sofern bekannt, durch einen Doppelpunkt vom Parameternamen getrennt notiert. Dabei ist es, wie bei den Attributnamen, keineswegs notwendig, sich auf die Standarddatentypen der Programmiersprache zu beschränken:

```
+berechneGesamtSumme(Summe : Waehrungsbetrag)
+setzeDatum(T : int, M : int, J : int)
#setzeFaktoren(Faktor1 : double, Faktor2)
```

(4.3) Operationsparametertyp mit Defaultwert (optional)

Die Angabe von Defaultwerten (Standardwerten) ist die Darstellung des Initialwertes, den ein Parameter annimmt, wenn er nicht ausdrücklich ein anderer Wert angegeben wird:

```
#setzeFaktor(Faktor : double = 27.3)
```

(4.4) Operationszusicherungen (optional)

An dieser Stelle werden bestimmte Eigenschaften der Funktion festgehalten, die für die Implementierung von Bedeutung sind. In der Regel handelt es sich um logische Ausdrücke (Zusicherungen) für den Wertebereich eines Parameters oder des Rückgabewertes, wie z.B.:

```
+setzeFaktor(Faktor : int) {1 < Faktor < 4}
```

Auch als Zusicherung gilt die Sichtbarkeit (`private`, `protected` oder `public`) der Methoden. Da diese Angabe jedoch bei praktisch jeder Operation von Relevanz ist, wird sie in Kurzform als Symbol vor dem Attributnamen notiert.
Abstrakte Methoden (pure virtual functions) werden – wie auch die Klassen – durch *kursive* Schreibweise bzw. die Zusicherung `{abstrakt}` ausgewiesen:

```
+setzeWert1() {abstrakt}
+setzeWert2(Wert : int = 55) {0 < Wert < 99)
```

(5) Stereotyp (optional)

Stereotypen sollen es einerseits ermöglichen Klassen in bestimmte Kategorien einordnen, in selteneren Fällen dienen sie zur projekt- oder unternehmensspezifischen Erweiterung der UML.
Die UML kennt bereits eine ganze Reihe von vorgefertigten Stereotypen, die man verwenden kann, aber nicht muß:

```
<<model>>          <<view>>         <<controller>>
<<exception>>      <<primitive>>    <<enumeration>>
<<signal>>         <<complete>>     << incomplete>>
<<overlapping>>    <<disjoint>>     <<implement>>
<<include>>        <<extend>>       <<utility>>
```

Ohne Probleme lassen sich weitere Stereotypen hinzufügen, wie z.B.:

```
<<Fachklasse>       <<Hilfsklasse>
<<Zugriffsklasse>>  <<Berechnungsklasse>>
```

Beispiel:

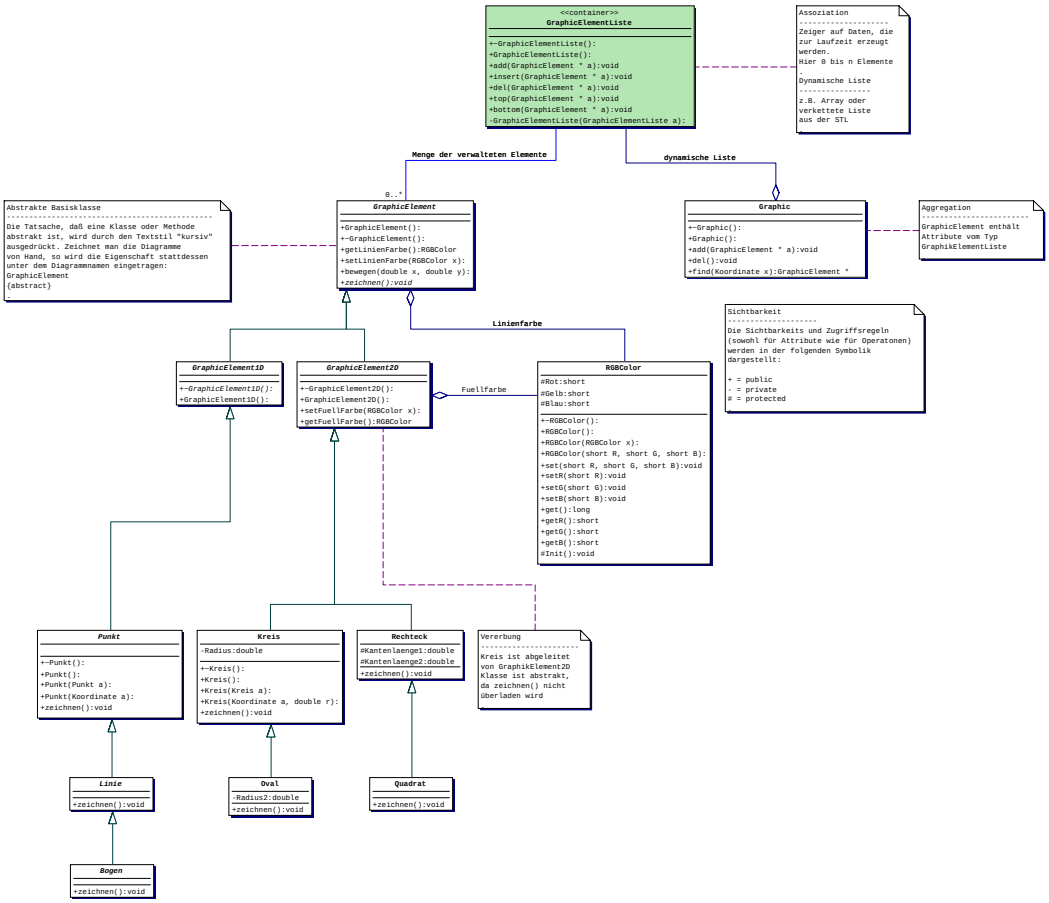

Abbildung 3-2 – Beispiel für ein Klassendiagramm mit Details

Wie bereits erwähnt, werden für die Entwurfs- und Designphase einfachere UML-Klassendiagramme verwendet, da während dieser Phase die Implementierungsdetails wie Zusicherungen meist nicht bekannt bzw. nicht relevant sind.

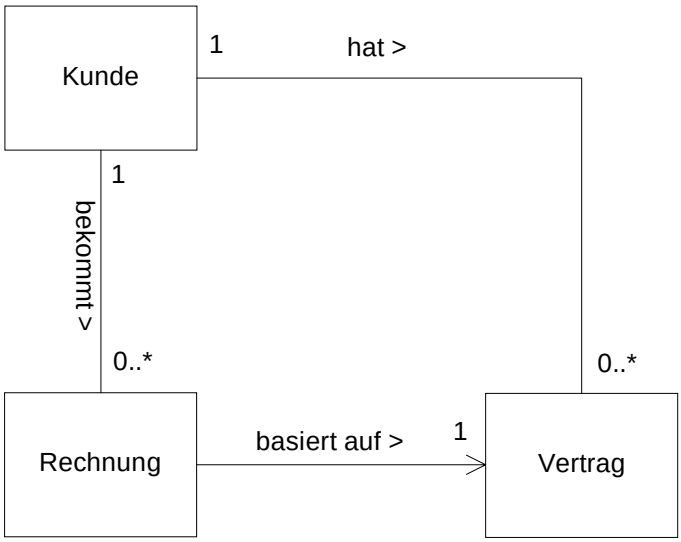

Abbildung 3-3 – Beispiel für ein Klassendiagramm ohne Details

3.2 Objekte

Ein Objekt ist die Instance einer Klasse, d.h. ein Datenspeicher, der gemäß der verwendeten Vorlage (Klasse) aufgebaut wurde.
Im wesentlichen unterscheidet sich die UML-Darstellung eines Objektes nicht von der Darstellung der Klasse, schließlich geht ein Objekt ja aus einer Klasse hervor, ist also zwangsläufig identisch aufgebaut.
Objekte werden in UML-Diagrammen z.B. dann dargestellt, wenn es ein entsprechendes obligatorisches Softwareobjekt im System gibt. Dies könnte z.B. ein Musterkunde sein, der automatisch bei Programmstart angelegt wird und dann den Wünschen des Anwenders entsprechend abgeändert wird. Unterschieden wird ein Objekt von einer Klasse durch die Verwendung des Exemplarnamens anstelle der Paketangabe. Außerdem werden Exemplarname und Klassenname unterstrichen.

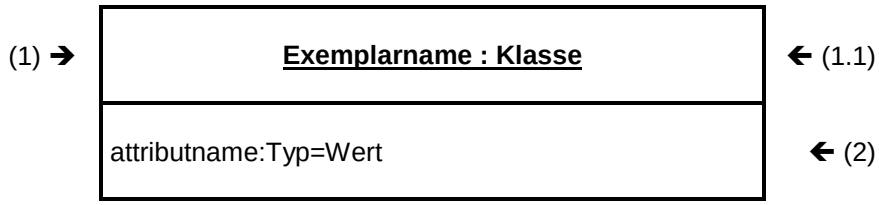

Abbildung 3-4 – Aufbau eines Objektdiagramms

(1) Exemplarname

Der Exemplarname entspricht dem Variablennamen des Objekts, mit dem dieses im Programm verwendet wird und gehorcht damit den Namenskonventionen, die innerhalb eines Projektes oder Unternehmens gelten.
Multiple Objekte (Arrays) werden durch überlagerte Rechtecke dargestellt.

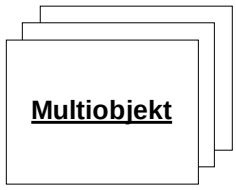

Abbildung 3-5 – Multiobjekt

(1.1) Klassenname (optional)

Der Exemplarname kann entweder durch den Namen der zugrundeliegenden Klasse ergänzt werden (s.o.), oder es ist eine Ableitungsbeziehung durch einen <<Instance of>> Pfeil (gestrichelt) darzustellen. In einem UML-Diagramm darf ein Objekt nie ohne einen Hinweis auf seine Herkunftsklasse notiert werden.

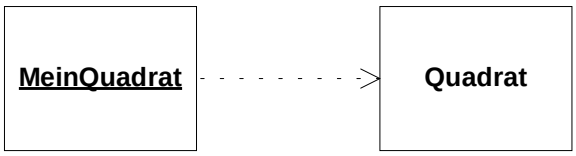

Abbildung 3-6 – Objektableitung ohne Details

(2) Attributwerte (optional)
Sind alle oder ein Teil der Attrubutwerte für ein Objekt bekannt, so können diese im UML-Diagramm vermerkt werden.
Für den oben erwähnten Musterkunden könnten es z.B. der Name „Mustermann“ und Vorname „Max“ sein.

Das folgende Beispiel zeigt den Maximalfall einer Objektdarstellung. Üblicherweise sind die Objektdarstellungen in einem UML-Diagramm weit weniger konkret , weil zu verwendende Standarddaten entweder nicht bekannt sind oder auch gar nicht existieren.

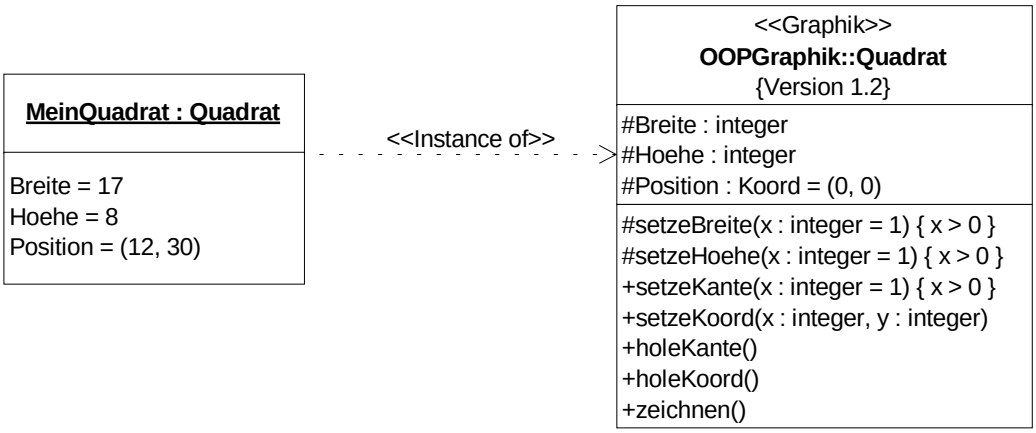

Abbildung 3-7 – Objektableitung mit Details

3.3 Beziehungen

In den oben gezeigten Diagrammen sind nicht nur die bereits eingeführten Klassendiagramme und Objekte zu finden, sondern auch eine Reihe von recht unterschiedlichen Linienverbindungen zwischen den Zeichnungselementen, welche die Beziehungen der Elemente zueinander definieren.

Instanziierung
Unter einer Instanziierung versteht man die Herleitung eines Objektes aus einer Klassenbeschreibung, d.h. die Bildung eines Speicherbereiches, der ein Objekt des angegebenen Typs ausnehmen kann.
In UML-Begriffen bedeutet dies, daß man eine Linie zwischen einem Klassendiagramm und einem Objekt zeichnet und damit festlegt, daß das Objekt entsprechend dieses Klassenvorbilds aufgebaut ist. Definitionsgemäß kann ein Objekt kann immer nur zu genau einem Klassendiagramm eine Instanziierungsbeziehung haben, umgekehrt kann es allerdings an einem Klassendiagramm beliebig viele Instanziierungsbeziehungen geben.

Die Beziehung wird durch einen gestrichelten Pfeil mit einer offenen Pfeilspitze dargestellt. Der Pfeil zeigt vom Objekt zur Klasse, von welches es abgeleitet ist (siehe Abbildung 3-6 – Objektableitung ohne Details).

Vererbung

Vererbungsbeziehungen gehören zu den wichtigsten Darstellungsformen in einem Klassendiagramm. Die Vererbung zeigt, in welcher Form die einzelnen Bausteine (Klassen) aufeinander basieren und wie sie zu größeren Einheiten zusammengesetzt werden können.
Von einer Vererbung wird üblicherweise gesprochen, wenn es eine einfache Ableitung zwischen einer Klasse und ihrer Basisklasse gibt. Man spricht hier auch von einer Spezialisierung der Basisklasse. Umgekehrt wird die Basisklasse auch als Generalisierung der Klasse bezeichnet.
Die Vererbungsbeziehung wird in der UML durch eine durchgezogene Linie mit einer geschlossenen, leeren Pfeilspitze dargestellt. Der Pfeil zeigt immer in Richtung der Basisklasse.

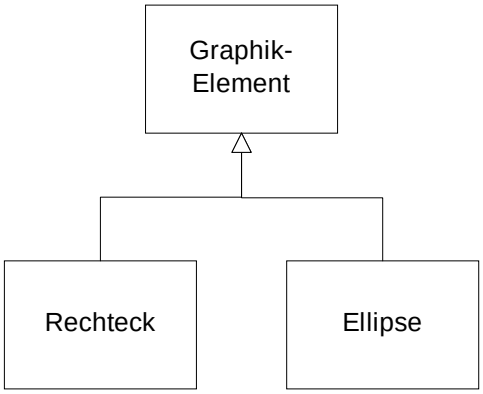

Abbildung 3-8 – Vererbungsbeziehung

Mehrfachvererbung

Hat eine Klasse mehr als eine Basisklasse spricht man von einer Mehrfachvererbung. Die Mehrfachvererbung entspricht in ihrer UML-

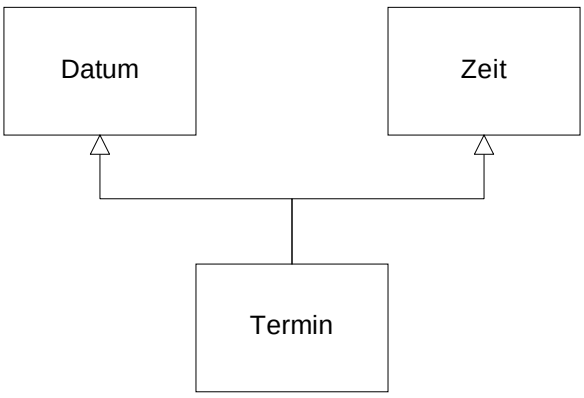

Abbildung 3-9 – Mehrfachvererbung

Darstellung exakt der einfachen Vererbung, nur daß von der abgeleiteten Klasse mehr als ein Beziehungspfeil ausgeht (zu zwei oder mehr Basisklassen).
Das Problem, daß bei der Mehrfachvererbung eventuell ein Basisobjekt aus verschiedenen Vererbungslinien mehr als einmal im abgeleiteten Objekt enthalten sein kann (ambigious inheritance / uneindeutige Vererbung), wird in der UML durch sogenannte Diskriminatoren gelöst.
Ein Diskriminator ist nichts weiter als ein Hinweis in gescheiften Klammern, der am Beziehungspfeil vermerkt wird und genau definiert, ob die Ableitung gemeinsame Basisobjekte mehrfach `{disjunkt}` oder nur einmal `{overlapping}` enthalten soll. Der Overlapping-Diskriminator entspricht der virtuellen Vererbung in C++.
Der Diskriminator wird zwischen den betroffenen Vererbungs-beziehungslinien als gestrichelte, beschriftete Linie ohne Pfeilspitzen eingezeichnet. Ist dies nicht nötig, da sich die Pfeile ohnehin überlappen, wird der Diskriminator ohne zusätzliche Verbindungslinie dargestellt.

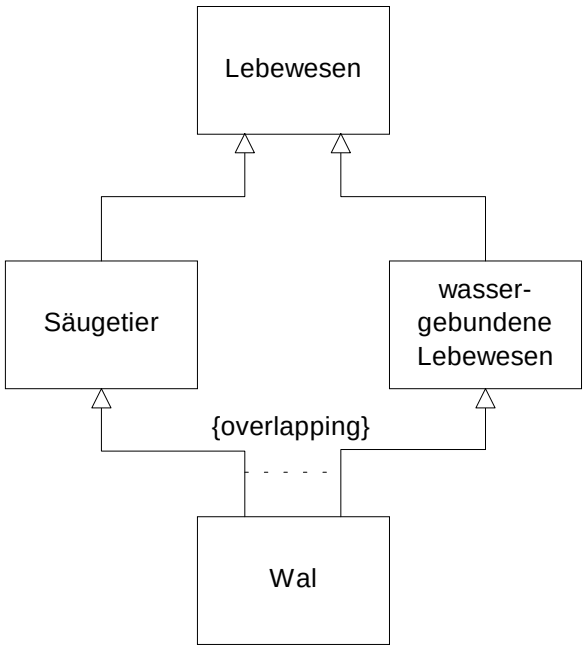

Abbildung 3-10 – Mehrfachvererbung mit Diskriminator

Wie in C++ ist auch in der UML die disjunkte Vererbung der Standardfall und muß nicht gesondert ausgewiesen werden. Da eine uneindeutige Vererbung aber anhand der Vererbungsbeziehungen sehr leicht in einem UML-Diagramm erkannt werden kann, sollte der Diskriminator auf jeden Fall angegeben werden, um deutlich zu kennzeichnen, daß die uneindeutige Vererbung erkannt und bedacht worden ist.
Weitere mögliche Diskriminatoren sind die Angaben `{complete}` und `{incomplete}`, die besagen, daß eine Klasse die letzte Ableitung in der Hierarchie darstellt und keine weiteren Unterklassen mehr gebildet werden. Auf die Angabe dieses Diskriminators wird üblicherweise verzichtet. Hilfreicher ist die Angabe von `{incomplete}`, um zu

kennzeichnen, daß an bestimmten Stellen des Diagramms noch weitere Spezialisierungen vorzunehmen sind.

Assoziation

Assoziationen sind Verbindungen zwischen Klassen, die nicht durch Vererbung gebildet werden. Man bezeichnet Assoziationen auch als Verweise oder Links.
Üblicherweise sind Assoziationen Verbindungen zwischen verschiedenen Klassen, die in C++ über Zeiger (Pointer) gebildet werden, es sind aber auch problemlos rekursive Beziehungen darstellbar.
Assoziationen können den gesamten Programmlauf über bestehen oder auch nur temporär bzw. optional. So kann z.B. zu einer Person eine Adresse bekannt sein, muß aber nicht, so daß man ein Adressobjekt üblicherweise innerhalb eines Personenobjekts als Assoziation speichert.
Die Assoziation wird in der UML durch eine durchgezogene Linie ohne Pfeilspitze dargestellt.
Als zusätzliche Angaben an einer Assoziationslinie können Multiplizität (s.u.), Beziehungsnamen, Beziehungsrichtungen, Rollen, Eigenschaftswerte und Zusicherungen notiert werden.

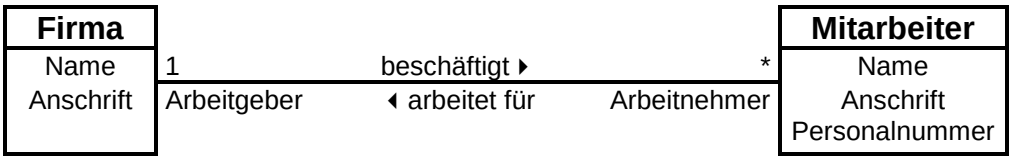

Abbildung 3-11 – Assoziation

Beziehungsnamen: beschäftigt, arbeitet für
Beziehungsrichtungen: ▸, ◂
Rollen: Arbeitgeber, Arbeitnehmer
Multiplizität: 1, *

Multiplizität

Die Multiplizität gibt an, mit wie vielen Objekten eines bestimmten Typs ein Objekt in Beziehung steht. Die Multiplizität wird an der Linie als Zahl angegeben.
Ist die Anzahl variabel, so wird die Anzahl der Beziehungen mittels eines Minimum-/Maximumpaares angegeben, wobei „*“ für eine beliebige, maximale Anzahl von Objekten steht und „0“ für ein Minimum, welches eine Beziehung als optional kennzeichnet. Sind nur bestimmte, konkrete Anzahlen von Beziehungen möglich, so werden diese als Zahlen an der Beziehungslinie aufgereiht.
Da Assoziationen häufig gegenseitig aufgebaut werden, können auf beiden Seiten der Linie Multiplizitätsangaben erfolgen.

Multiplizitätsangaben	
1	genau Eins
0,1	genau Null oder Eins
4,5	Vier oder Fünf
0..9	zwischen Null und Neun

Multiplizitätsangaben	
3..8	zwischen Drei und Acht
0..5,9	zwischen Null und Fünf oder genau Neun
0..*	größer oder gleich Null (Standard wenn Angabe fehlt)
*	größer oder gleich Null (Standard wenn Angabe fehlt)
1..*	größer oder gleich Eins
0..2,5,7..9,12..*	zwischen Null und Zwei oder genau Fünf oder zwischen Sieben und Neun oder größer oder gleich Zwölf.

Aggregation

Unter einer Aggregation versteht man die Zusammensetzung eines Objektes aus mehreren Teilobjekten, also eine „Besteht aus"-Beziehung. Diese wird in C++ durch Variablen und starre oder dynamische Listen gebildet. So ist z.B. eine Zeichenkette eine Aggregation von Zeichen.
Wie bei Assoziationen können auch bei einer Aggregation die Multiplizität, Beziehungsnamen und Beziehungsrichtung angegeben werden. Alledings kann eine Aggregation nur in Ausnahmefällen (dynamische Liste) die Multiplizität „0" annehmen.
Die Aggregation wird in der UML durch eine durchgezogene Linie mit einer leeren Raute als Pfeilspitze dargestellt. Der Pfeil zeigt immer in Richtung des aggregierenden Objekts.

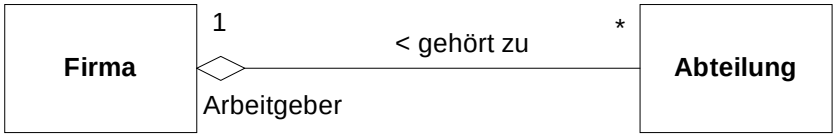

Abbildung 3-12 – Aggregationsbeziehung

Komposition

Eine Komposition ist eine strenge Form der Aggregation, mit allen Eigenschaften einer Aggregation. Allerdings sind die Teile hier vom Ganzen (kompositionsbildende Objekt) existenzabhängig. D.h. wird das kompositionsbildende Objekt gelöscht, so werden auch alle abhängigen Teile mit zerstört.
Die Komposition wird in der UML durch eine durchgezogene Linie mit einer gefüllten Raute als Pfeilspitze dargestellt. Der Pfeil zeigt immer in Richtung des kompositionsbildende Objekts.

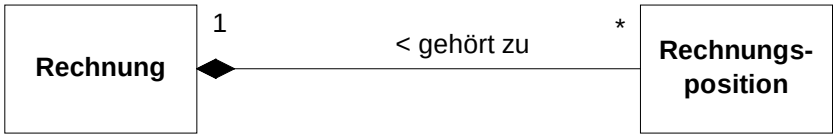

Abbildung 3-13 – Kompositionsbeziehung

Notiz

Eine Notiz ist ein Fließtext, der in UML-Diagrammen an jedem Diagrammteil, gleich ob (Verbindungslinie, Klassendiagramm, Attributeintrag, Pfeilspitze etc.) angebracht werden kann. Zur Darstellung wird ein Kästchen mit einem Eselsohr verwendet.

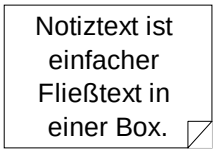

Abbildung 3-14 – Notiz

4 Anhang – Literaturverzeichnis

[Booch94] G. Booch:
Objektorientierte Analyse und Design; Mit praktischen Anwendungsbeispielen
Addision Wesley, Bonn, 1994

[CoadYourdon91a] P. Coad, E.Yourdon:
Object-Oriented Analysis; 2nd Edition
Prentice-Hall, Englewood Cliffs, 1991

[CoadYourdon91b] P. Coad, E.Yourdon:
Object-Oriented Design
Prentice-Hall, Englewood Cliffs, 1991

[Martin92] J. Martin, J. Odell:
Object-Oriented Analysis & Design
Prentice-Hall, Englewood Cliffs, 1992

[Oestereich98] B. Oestereich:
Objektorientierte Softwareentwicklung; 4. aktualisierte Auflage
Oldenbourg, München, 1998

[Rambaugh93] J. Rambaugh, M. Blaha, W. Premerlani, F. Eddi, W. Lorenson:
Objektorientiertes Modellieren und Entwerfen
Hanser, München, 1993

[Waldén95] K. Waldén, J.-M. Nerson:
Seamless Object-Oriented Software Architecture, Analysis and Design of Reliable Systems
Prentice-Hall, London, 1995

Anhang – Abbildungsverzeichnis