

ABAP für Fortgeschrittene

Teil 1: ABAP Objects

Copyright

- Das vorliegende Skriptum baut zu großen Teilen auf den Publikationen zum TAW11- und TAW12-Kurs – das Copyright dieser Teile liegt bei der SAP AG.
- Die in diesem Kurs verwendeten Abbildungen wurden – falls nicht anders gekennzeichnet – in Anlehnung zum TAW11- und TAW12-Kurs erstellt. Das Copyright dieser Teile liegt bei der SAP AG.
- Für alle Screenshots im Skriptum, auch wenn diese nur verkürzt oder auszugsweise gezeigt werden, gilt der Hinweis: Copyright SAP AG
- Die Weitergabe und Vervielfältigung dieser Publikation oder von Teilen daraus sind, zu welchem Zweck und in welcher Form auch immer, ohne die schriftliche Genehmigung von Prof. Dr. Heimo H. Adelsberger, Dipl.-Wirt.-Inf. Pouyan Khatami und Dipl.-Wirt.-Inf. Taymaz Khatami nicht gestattet..

Inhaltsverzeichnis

COPYRIGHT	2
INHALTSVERZEICHNIS	3
ABBILDUNGSVERZEICHNIS	5
3 EINFÜHRUNG IN DIE OBJEKTORIENTIERTE PROGRAMMIERUNG MIT ABAP	8
3.1 HINTERGRUND	8
3.2 WARUM OBJEKTORIENTIERUNG?	10
3.3 OBJEKTE UND KLASSEN.....	16
3.4 OBJEKTORIENTIERUNG IN ABAP	17
3.5 UNIFIED MODELING LANGUAGE.....	20
3.5.1 Das UML-Klassendiagramm.....	20
3.5.1.1 Assoziationsbeziehungen.....	21
3.5.1.2 Generalisierungs- / Spezialisierungsbeziehungen.....	23
3.5.2 Das UML-Objektdiagramm	24
3.5.3 Das UML-Sequenzdiagramm.....	26
3.5.4 Kontrollfragen	28
3.5.5 Antworten.....	28
3.6 GRUNDLEGENDE OBJEKTORIENTIERTE SYNTAX IN ABAP	29
3.6.1 Definition von Attributen	30
3.6.2 Referenzvariablen.....	34
3.6.3 Zugriff auf Attribute.....	39
3.6.4 Definition und Implementierung von Methoden.....	40
3.6.5 Zusätzliche Eigenschaften von Komponenten in UML	44
3.6.6 Methodenaufrufe	46
3.6.7 Konstruktoren.....	50
3.6.8 Hinweise zur Namensgebung	52
3.6.9 Kontrollfragen	53
3.6.10 Lösungen	54
3.7 PRAXIS: EIN OBJEKTORIENTIERTES PROGRAMM	54
3.7.1 Anlegen von Paket und Transportauftrag.....	54
3.7.2 Erstellen des Programms	57
3.7.3 Die Klasse lcl_employee	60
3.7.4 Die Klasse lcl_department	64
4 SPEZIELLE OBJEKTORIENTIERTE KONZEPTE IN ABAP	66
4.7 VERERBUNG	66
4.7.1 Casting und Polymorphie	73
4.7.2 Fazit: Vererbung.....	80
4.7.3 Praxis: Übung zu Vererbung, Casting und Polymorphie	81
4.8 ABSTRAKTE KLASSEN UND INTERFACES	84
4.8.1 Abstrakte Klassen.....	84
4.8.2 Praxis: Übung zu abstrakten Klassen	86

4.8.3	Interfaces	87
4.8.4	Praxis: Übung zu Interfaces	99
4.8.4.1	Erzeugen des Interfaces	99
4.8.4.2	Das Interface in Klassen implementieren	99
4.8.4.3	Erstellen des Hauptprogramms	100
4.8.4.4	Verwenden von Aliasen	101
4.8.5	Fazit: Abstrakte Klassen und Interfaces	103
4.9	EREIGNISSE	103
4.9.1	Praxis: Übung zu Ereignissen	108
4.9.2	Kontrollfragen	110
4.9.3	Antworten	111
4.10	KAPITELABSCHLUSS	112

Abbildungsverzeichnis

Abbildung 1: ABAP in der Entwicklungsgeschichte von Programmiersprachen	9
Abbildung 2: Funktionen und Daten im Prozeduralen Modell	10
Abbildung 3: Aufbau eines prozeduralen ABAP-Programms	11
Abbildung 4: Trennung des Speichers	12
Abbildung 5: Repräsentation als Funktionsgruppe	13
Abbildung 6: Verwendung der Funktionsgruppe	13
Abbildung 7: Repräsentation mehrerer Instanzen?	14
Abbildung 8: Hauptspeichersicht bei Kapselung	15
Abbildung 9: Perspektiven auf ein Gebäude	17
Abbildung 10: Client und Server	18
Abbildung 11: Iterativer Software-Entwicklungsprozess	19
Abbildung 12: Eine Klasse in UML	20
Abbildung 13: Beispiel für eine Assoziationsbeziehung	21
Abbildung 14: Mehrfache Assoziation	22
Abbildung 15: Rekursive Assoziation	22
Abbildung 16: Beispiel für eine Assoziationsklasse	23
Abbildung 17: Beispiel für eine Aggregation	23
Abbildung 18: Beispiel für eine Komposition	23
Abbildung 19: Beispiel mit zwei Notationsarten	24
Abbildung 20: Beispiel für ein Objektdiagramm	25
Abbildung 21: Unzulässiges Objektdiagramm	26
Abbildung 22: Aufbau von Sequenzdiagrammen	27
Abbildung 23: Delegationsbeispiel als Sequenzdiagramm	27
Abbildung 24: Definitions- und Implementierungsteil einer Klasse	29
Abbildung 25: Eine Klasse für Mitarbeiter	30
Abbildung 26: Syntaxfehler im objektorientierten Kontext: SAP-System-Screenshot	31
Abbildung 27: Beispiele für Attributdefinitionen: SAP-System-Screenshot	31
Abbildung 28: Sichtbarkeitsbereiche von Attributen	32
Abbildung 29: Öffentliche und private Attribute	33
Abbildung 30: Instanz- und statische Attribute	34
Abbildung 31: Statische Attribute zur Laufzeit	34
Abbildung 32: Definition von Referenzvariablen	35
Abbildung 33: Erzeugung neuer Objekte	36
Abbildung 34: Mehrere Referenzen auf ein Objekt	37
Abbildung 35: Beispiel zum Garbage Collector	38
Abbildung 36: Interne Tabellen aus Referenzvariablen	39
Abbildung 37: Zugriff auf öffentliche Attribute	40
Abbildung 38: Methodendefinition und –implementierung	41
Abbildung 39: Sichtbarkeit von Methoden	41
Abbildung 40: Zugriff auf eine private Methode	42
Abbildung 41: Statische Methoden und Instanzmethoden	43
Abbildung 42: Verwendung der Selbstreferenz	44
Abbildung 43: Erweiterte Angaben in UML	45
Abbildung 44: Parameter und Rückgabetyp in UML	46
Abbildung 45: Aufrufsyntax von Instanzmethoden	47
Abbildung 46: Aufrufsyntax statischer Methoden	48
Abbildung 47: Beispiel für einen funktionalen Methodenaufruf	49
Abbildung 48: Kurzform der Parameterliste	49
Abbildung 49: Syntax für den Instanzkonstruktor	50
Abbildung 50: Beispiel für einen Instanzkonstruktor	51
Abbildung 51: Beispiel für einen statischen Konstruktor	52

Abbildung 52: Favoriten anlegen: SAP-System-Screenshot.....	54
Abbildung 53: Eingabe des Transaktionscodes: SAP-System-Screenshot	54
Abbildung 54: Object Navigator im Favoritenmenü: SAP-System-Screenshot.....	55
Abbildung 55: Technische Namewn aktivieren: SAP-System-Screenshot	55
Abbildung 56: Eingabe des Paketnamens (Abbildung ist ein Beispiel - Sie verwenden Ihre eigene Nummer) : SAP-System-Screenshot	55
Abbildung 57: Nachfrage des SAP-Systems: SAP-System-Screenshot.....	56
Abbildung 58: Angabe einer Kurzbeschreibung: SAP-System-Screenshot	56
Abbildung 59: Abfrage des Transportauftrags: SAP-System-Screenshot.....	56
Abbildung 60: Transportauftrag anlegen: SAP-System-Screenshot	57
Abbildung 61: Anlegen des Programms: SAP-System-Screenshot	57
Abbildung 62: Eingabe des Entwicklerschlüssels: SAP-System-Screenshot.....	58
Abbildung 63: Eigenschaften des Programms: SAP-System-Screenshot	59
Abbildung 64: Anlegen des Includes: SAP-System-Screenshot	59
Abbildung 65: Meldung des SAP-Systems: SAP-System-Screenshot	60
Abbildung 66: Eine Klasse für Mitarbeiter	61
Abbildung 67: Referenzvariable im Debugger: SAP-System-Screenshot	63
Abbildung 68: Attribute im Debugger: SAP-System-Screenshot	63
Abbildung 69: Ausgabe des Programms: SAP-System-Screenshot	64
Abbildung 70: Abteilung und Mitarbeiter	64
Abbildung 71: Ausgabe des Programms mit Abteilungsklasse: SAP-System-Screenshot	65
Abbildung 72: Beispiel zum Vererbungskonzept.....	66
Abbildung 73: Beispiel für die Definition einer Vererbung.....	67
Abbildung 74: Vererbung mit Redefinition einer Methode	68
Abbildung 75: Beispiel zur Redefinition von Methoden	69
Abbildung 76: Zugriff auf Oberklassenkomponenten.....	69
Abbildung 77: Überschreiben eines Konstruktors	70
Abbildung 78: Vererbungshierarchie mit 3 Klassen	71
Abbildung 79: Beispiel mit Protected-Attributen	72
Abbildung 80: Ein Objekt und die Adressierbarkeit seiner Komponenten	72
Abbildung 81: Beispiel für einen Up-Cast	74
Abbildung 82: Beispieldaten	75
Abbildung 83: Implementierung einer Methode im Szenario	75
Abbildung 84: Methodenaufruf im Szenario.....	76
Abbildung 85: Generische Aufrufe ohne objektorientiertes ABAP	77
Abbildung 86: Down-Cast	78
Abbildung 87: Methode, für die ein Down-Cast benötigt wird.....	79
Abbildung 88: Implementierung des Down-Casts	80
Abbildung 89: Fragwürdige Vererbungsbeziehungen.....	81
Abbildung 90: Anlegen des Programms: SAP-System-Screenshot	81
Abbildung 91: Vererbungsbeziehung für die Übung	82
Abbildung 92: Nicht erreichbare Anweisung: SAP-System-Screenshot.....	83
Abbildung 93: Ausgabe des Programms: SAP-System-Screenshot.....	83
Abbildung 94: Ausgabe des durchschnittlichen Provisionssatzes: SAP-System-Screenshot.....	84
Abbildung 95: Abstrakte Klasse in UML.....	84
Abbildung 96: Mehrere abstrakte Klassen	85
Abbildung 97: Fehler bei Instanziierungsversuch: SAP-System-Screenshot.....	86
Abbildung 98: Fehlende Implementierung einer abstrakten Methode: SAP-System-Screenshot	87
Abbildung 99: Erweiterte Ausgabe des Programms: SAP-System-Screenshot	87
Abbildung 100: Beispiel für in ABAP unzulässige Mehrfachvererbung	88
Abbildung 101: Protokollfestlegung durch den Verwender.....	89
Abbildung 102: Modellierung mit Interface	90
Abbildung 103: Syntaxbeispiel zu Interfacedefinition und -verwendung	91
Abbildung 104: Vereinfachung durch Aliasdefinition	92

Abbildung 105: Verwendung über Interfaces typisierter Referenzvariablen	93
Abbildung 106: Komponentenzugriff über Interfaces	94
Abbildung 107: Down-Cast mit Interfaces	95
Abbildung 108: Down-Cast mit Fehlerbehandlung	95
Abbildung 109: Szenario mit 2 Interfaces.....	95
Abbildung 110: Zuweisungen zwischen Referenzvariablen unterschiedlicher Interfaces	96
Abbildung 111: Fehlermeldung bei falscher Zuweisung: SAP-System-Screenshot	96
Abbildung 112: Interface-Hierarchie	97
Abbildung 113: Zugriff auf Komponenten von Komponenten-Interfaces	98
Abbildung 114: Unzulässige Definitionen	98
Abbildung 115: Ausgabe des Programms: SAP-System-Screenshot / Montage.....	101
Figure 116: Syntaxfehler: SAP-System-Screenshot.....	102
Abbildung 117: Ereignisbehandlung in UML.....	104
Abbildung 118: Ein Ereignis, viele Behandler.....	104
Abbildung 119: Syntaxelemente zum Auslösen und Behandeln von Ereignissen	105
Abbildung 120: Auslösseite des Szenarios	106
Abbildung 121: Behandlerdefinition im Szenario.....	107
Abbildung 122: Ausgabe des Programms: SAP-System-Screenshot.....	109

3 Einführung in die objektorientierte Programmierung mit ABAP

3.1 Hintergrund

Wenn Sie den Kurs „Einführung in ABAP“ erfolgreich absolviert haben, sind Sie bereits in einem Ausblick einige Schritte mit der Objektorientierung gegangen, so dass Sie vielleicht bereits eine Vorstellung davon haben was sie erwartet. In diesem Kurs wird das Thema nun detailliert behandelt. Objektorientierte Vorkenntnisse sind für diesen Kurs jedoch nicht erforderlich. Der Kurs kann keinen Universitätskurs in objektorientierter Theorie ersetzen, liefert jedoch die erforderlichen theoretischen Grundlagen, die erforderlich sind, um die Objektorientierung selbstständig in ABAP anwenden zu können.

Es ist nicht die Absicht des Kurses, eine normative Aussage über die objektorientierte Programmierung zu treffen. Es gibt eine Reihe von Vorteilen, die mit der Objektorientierung verbunden werden und die Ihnen auch im Laufe des Kapitels nähergebracht werden, eine konkrete Bewertung hängt jedoch immer von den Umständen des jeweiligen Einsatzumfelds ab (z. B. vorhandener Code, Kenntnisse der Mitarbeiter usw.). Sie sollten jedoch wissen, dass neue Entwicklungen von SAP inzwischen grundsätzlich objektorientiert umgesetzt werden.

Bevor der Einstieg in die objektorientierte Programmierung erfolgt, soll dieser Kurs Ihnen zunächst einen Einblick in die objektorientierte Sichtweise auf die Welt vermitteln. So lernen Sie zu verstehen, wie Sachverhalte objektorientiert Modelliert werden, so dass Sie schließlich in einem objektorientierten Programm als Code umgesetzt werden können.

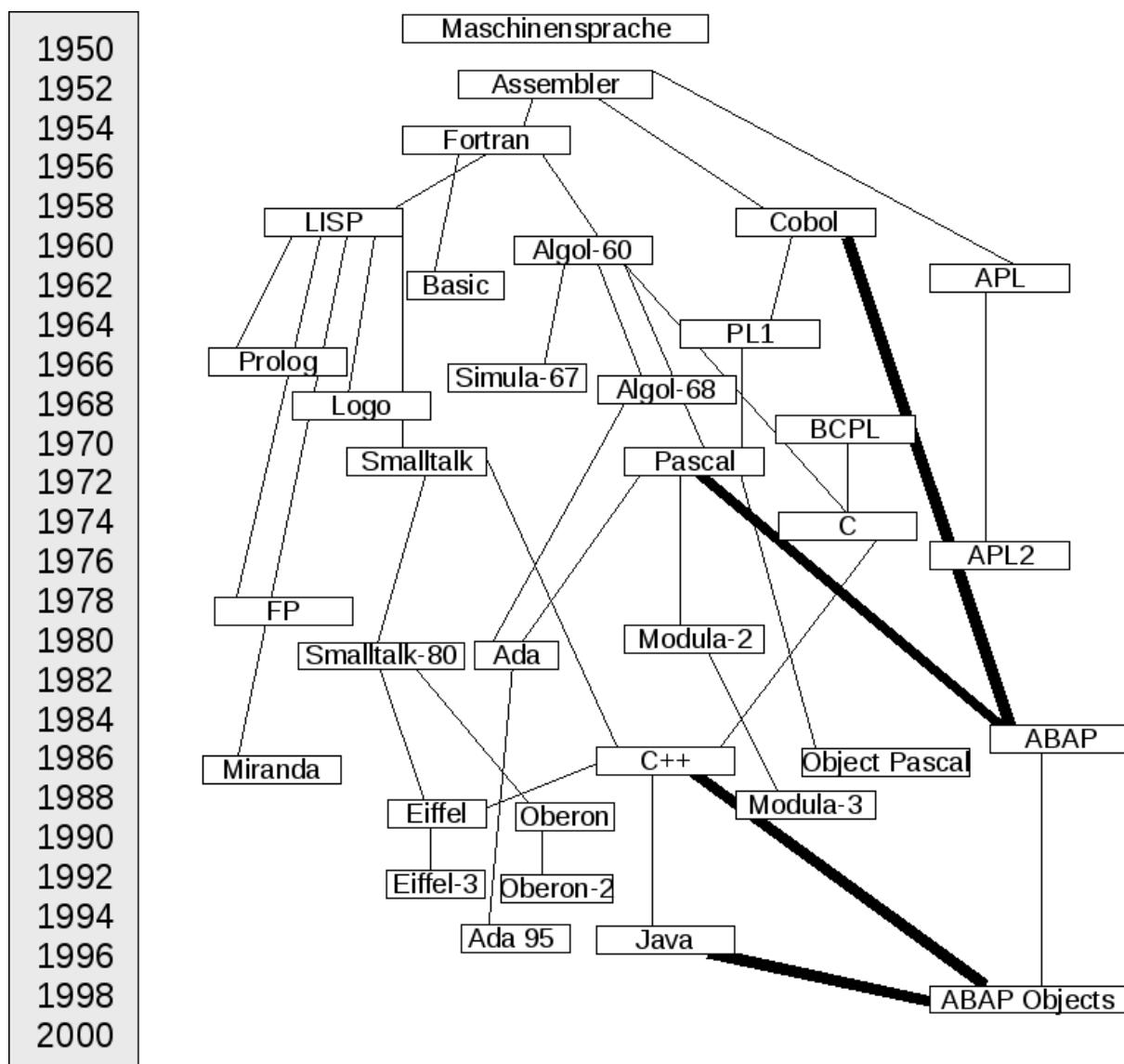


Abbildung 1: ABAP in der Entwicklungsgeschichte von Programmiersprachen

Bei ABAP Objects handelt es sich um die objektorientierte Weiterentwicklung der Programmiersprache ABAP. Wie Sie an der obigen Abbildung erkennen können, wurden für ABAP Objects als Erweiterung von ABAP Konzepte der Programmiersprachen C++ und Java übernommen. Auch die Einflüsse, die bei der Entwicklung des klassischen ABAP (zum Hinweis, dass es sich um eine „4GL“-Sprache handelt, auch als „ABAP/4“ bezeichnet) eine Rolle spielten, können Sie der Grafik entnehmen, wobei gesagt werden kann, dass ABAP relativ selbstständig entstand.

Sicherlich sind Ihnen in Ihrem Studium bereits ähnliche Darstellungen der Entwicklungsgeschichte von Programmiersprachen begegnet. Je nach Autor mögen die Zusammenhänge etwas unterschiedlich dargestellt oder andere Sprachen berücksichtigt sein; die hier verwendete Grafik ist also nur eine mögliche Darstellung von vielen und soll Ihnen helfen, ABAP Objects in den Kontext bekannter anderer Sprachen einordnen zu können.

Über die Anlehnung an Java und C++ hinaus besitzt ABAP Objects zum Teil auch Konzepte, die sich in diesen Sprachen so nicht wiederfinden, die Sie aber in diesem Kurs kennen lernen. Umgekehrt wurden natürlich nicht alle Konzepte aus diesen Sprachen übernommen

– verzichtet wurde auf solche, die sich aus Sicht der Entwickler von ABAP Objects nicht bewährt hatten. Wenn Sie bereits über Erfahrungen mit C++ oder Java verfügen (oder auch mit einer anderen objektorientierten Programmiersprache), kann dies von Vorteil für diesen Kurs sein. Es ist aber keine Voraussetzung.

Wenn Sie sich die Abbildung der Entwicklungsgeschichte anschauen und Sie vielleicht einige der anderen dargestellten Sprachen kennen, wird Ihnen Auffallen dass es bereits früh objektorientierte Sprachen gab, so etwa Simula 67. Die Programmiersprachen, die für große betriebswirtschaftliche Anwendungen eingesetzt wurden, waren jedoch zunächst keine objektorientierten Sprachen. Viel früher setzte sich das prozedurale Paradigma durch. Dies spiegelt sich auch in der Entwicklung von ABAP wieder, das erst mit der Einführung von ABAP Objects objektorientiert wurde, obwohl es bereits seit längerer Zeit andere objektorientierte Sprachen gab.

3.2 Warum Objektorientierung?

Zunächst stellt sich natürgemäß, vor allem für Programmierer die stark in der prozeduralen Welt verwurzelt sind, die Frage, warum überhaupt die Objektorientierung in ABAP eingeführt wurde. Hierzu werden prozedurale und objektorientierte Konzepte gegenübergestellt und die Vorteile, die mit der Objektorientierung verbunden werden, erläutert.

Typischerweise werden in der prozeduralen Programmierung Daten und Funktionen voneinander getrennt. Dies geschieht etwa durch die Definition von globalen Datenfeldern (Variablen) und Unterprogrammen, die die Logik der Funktionen enthalten.

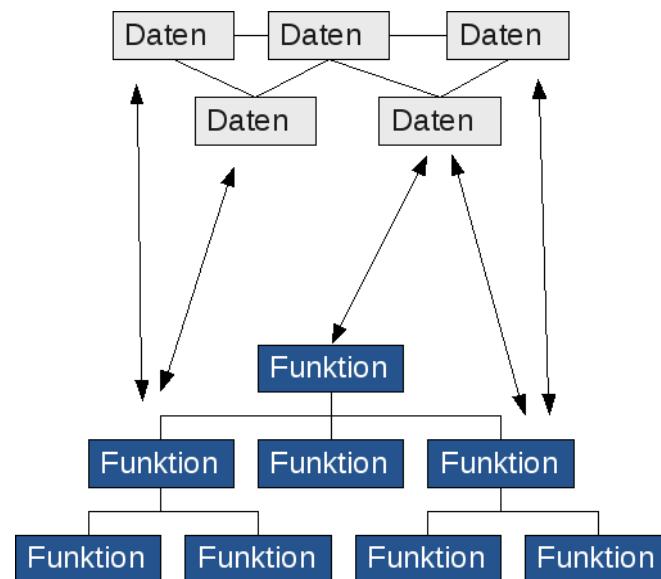


Abbildung 2: Funktionen und Daten im Prozeduralen Modell

So ist es grundsätzlich möglich, dass ein Unterprogramm auf eine beliebige Variable zugreift, ohne dass eine Kontrolle erfolgt. Der Nachteil ist hierbei darin zu sehen, dass unerwünschte Zugriffe nicht ausgeschlossen werden können, sondern die Disziplin und stete Kontrolle durch den Entwickler erfordern.

Ein typisches prozedurales Programm könnte einen Aufbau haben, wie er in der folgenden Abbildung skizziert wird:

REPORT ...	
*-----	
TYPES: ...	Typdefinitionen
DATA: ...	Datendeklarationen
PERFORM form1 ...	Hauptprogramm
CALL FUNCTION 'FB1'	Aufruf von Unterprogrammen
...	
CALL FUNCTION 'FB2'	Aufruf von Funktionsbausteinen
...	
*-----	
FORM f1 ...	Definition der Unterprogramme
...	
ENDFORM.	

Abbildung 3: Aufbau eines prozeduralen ABAP-Programms

Eine global definierte Variable könnte in diesem Fall vom Unterprogramm f1 gelesen und manipuliert werden (wie z. B. im Grundlagenkurs im ersten Programm zur Mehrwertsteuerberechnung). Durch ein Fehlverhalten dieses Unterprogramms könnte so die Konsistenz der Daten des Hauptprogramms verletzt werden, so dass die Datenfelder sich nicht in einem vorgesehenen Zustand befinden.

Die Möglichkeiten der Kapselung sind in diesem Fall also relativ schwach. Es ist nicht möglich zu verhindern, dass Unterprogramme auf eine bestimmte Variable des Hauptprogramms zugreifen.

Etwas fortschrittlichere Kapselungsmöglichkeiten sind im klassischen ABAP durch Funktionsgruppen und Funktionen gegeben.

Die Funktionsgruppe wird bei Verwendung eines Funktionsbausteins der Gruppe pro Programmausführung einmal in den internen Modus geladen und bleibt dort bis zur Beendigung des Hauptprogramms aktiv. Die Speicherbereiche von Hauptprogramm und Funktionsgruppen werden dabei aber nicht geteilt. Der Datenaustausch erfolgt nur über die Schnittstelle der Funktionsbausteine. Es ist also insb. nicht möglich, aus dem Funktionsbaustein auf die Daten des Hauptprogramms über die dort definierten Namen zuzugreifen.

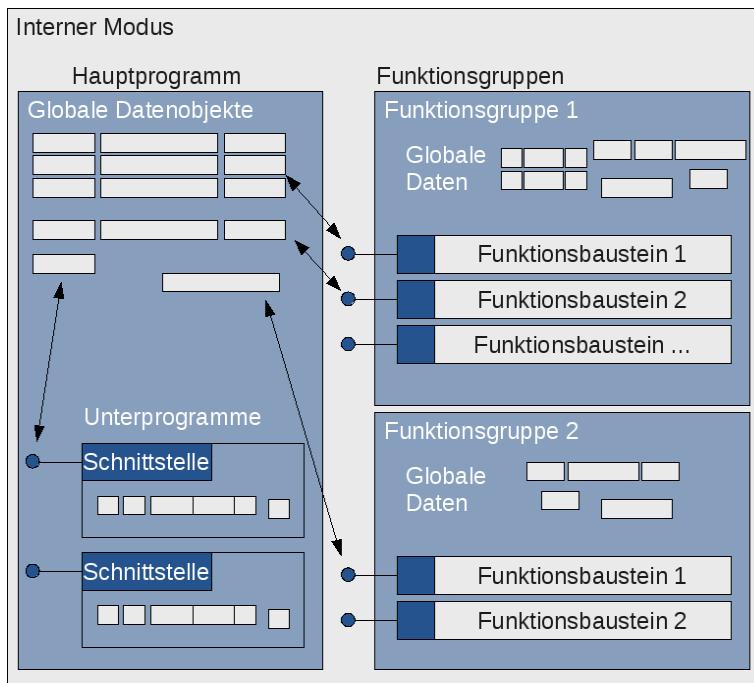


Abbildung 4: Trennung des Speichers

Möglich ist aber ein Zugriff der Funktionsbausteine auf globale Daten der Funktionsgruppe. Hier spielt es keine Rolle, ob die Bezeichner bei diesen Daten mit denen des Hauptprogramms übereinstimmen. Das Hauptprogramm erreicht die dort definierten Daten, der Funktionsbaustein die der Funktionsgruppe.

Durch diese Trennung und den nur über die Schnittstelle der Funktionsbausteine stattfindenden Datenaustausch wird eine Kapselung von Daten und Funktionalität erreicht. So muss weder der Verwender (hier das Hauptprogramm) wissen, wie die Implementierung im Funktionsbaustein genau aussieht (diese kann also unabhängig von der Verwendung ausgetauscht werden), noch muss der Funktionsbaustein wissen, wie z. B. Felder des Hauptprogramms benannt sind.

Eine solche Form der Datenkapselung, die eines der Merkmale objektorientierter Programmierung darstellt, ist also auch schon mit Mitteln von klassischem ABAP umsetzbar. Dies bildete die Grundlage für die Entwicklung von BAPIs (Business Application Programming Interfaces), die Funktionsbausteine sind, und Business Objects, die Funktionsgruppen sind.

Schaut man sich nun ein konkretes Beispiel an, stößt man jedoch schnell an die Grenzen dieser Repräsentation und es lässt sich zeigen, wie die Objektorientierung hierbei für Abhilfe sorgen kann.

Die Objektorientierung wird häufig als eine besonders natürliche Art angesehen, Dinge wahrzunehmen. Gegenstände der realen Welt können durch Klassen und Objekte im Programm direkt abgebildet werden. Wichtig ist hierbei, dass es sich nicht um eine reine Abbildung von Datenstrukturen handelt. Wenn Sie bisher viel mit strukturierter Programmierung und Datenbanken gearbeitet haben, besteht eine gewisse Gefahr, das ebenfalls abgebildete Verhalten zu vernachlässigen. Zur Veranschaulichung soll im

Folgendes ein konkretes Beispiel herangezogen werden: Abgebildet werden soll hier ein Mitarbeiter eines Unternehmens.

Ein Mitarbeiter könnte Eigenschaften wie einen Namen oder ein Gehalt haben. Eine Abbildung über eine Funktionsgruppe könnte wie hier skizziert aussehen:

```
FUNCTION-POOL staff.

* income ist hier eine globale
* Variable der Funktionsgruppe
DATA: income TYPE i.

...
FUNCTION inc_income.
...
    income = income + imp_income.
ENDFUNCTION.

FUNCTION dec_income.
...
    income = income - imp_income.
ENDFUNCTION.

FUNCTION get_income.
    exp_income = income
ENDFUNCTION.
```

Funktionsgruppe mit Funktionen zum Gehalt eines Mitarbeiters

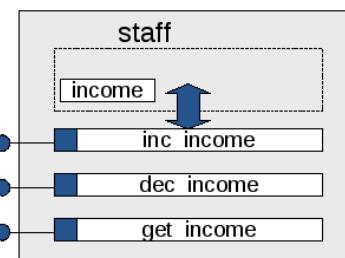


Abbildung 5: Repräsentation als Funktionsgruppe

In dieser Funktionsgruppe wäre das Gehalt (`income`) des Mitarbeiters gekapselt und könnte aus einem Programm nur über die Funktionsbausteine gelesen oder geschrieben werden, die die Funktionsgruppe zur Verfügung stellt:

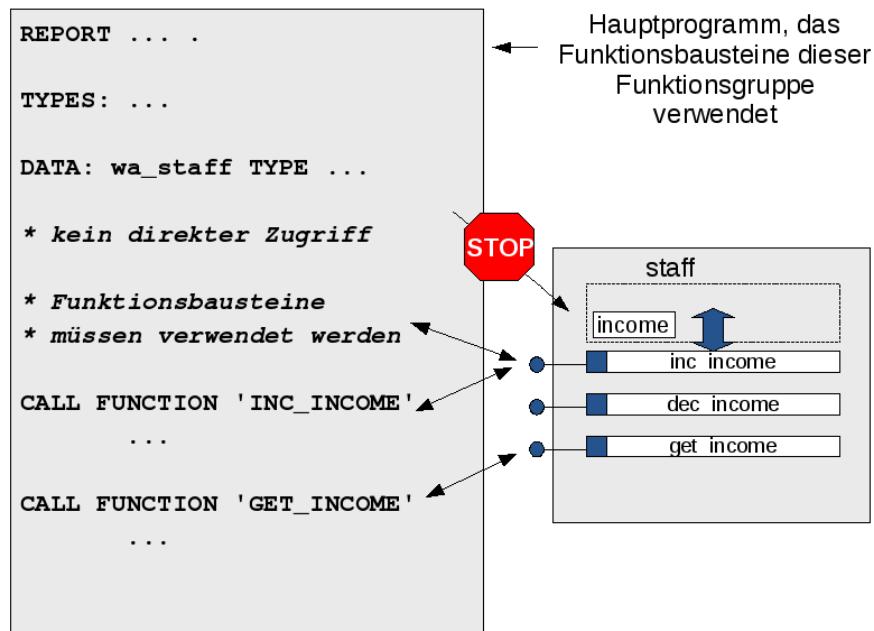


Abbildung 6: Verwendung der Funktionsgruppe

Es stellt sich nun die Frage, wie bei einem solchen Konzept mehrere Mitarbeiter gehandhabt werden sollen. Es können nicht mehrere Instanzen einer Funktionsgruppe gebildet werden. Sicherlich wäre es möglich, in der Funktionsgruppe eine Datenstruktur zu verwenden, die Daten zu mehreren Mitarbeitern enthalten kann. Dadurch würde jedoch die Entsprechung der Funktionsgruppe zu genau einem Mitarbeiter verloren gehen.

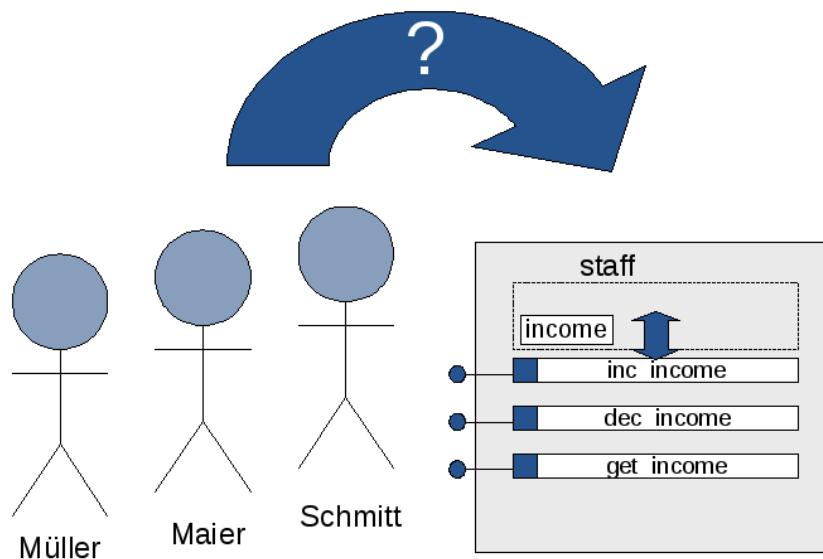


Abbildung 7: Repräsentation mehrerer Instanzen?

In der objektorientierten Programmierung besteht dieses Problem nicht. Hier könnte für jeden Mitarbeiter ein *Objekt* einer *Klasse* instanziert werden.

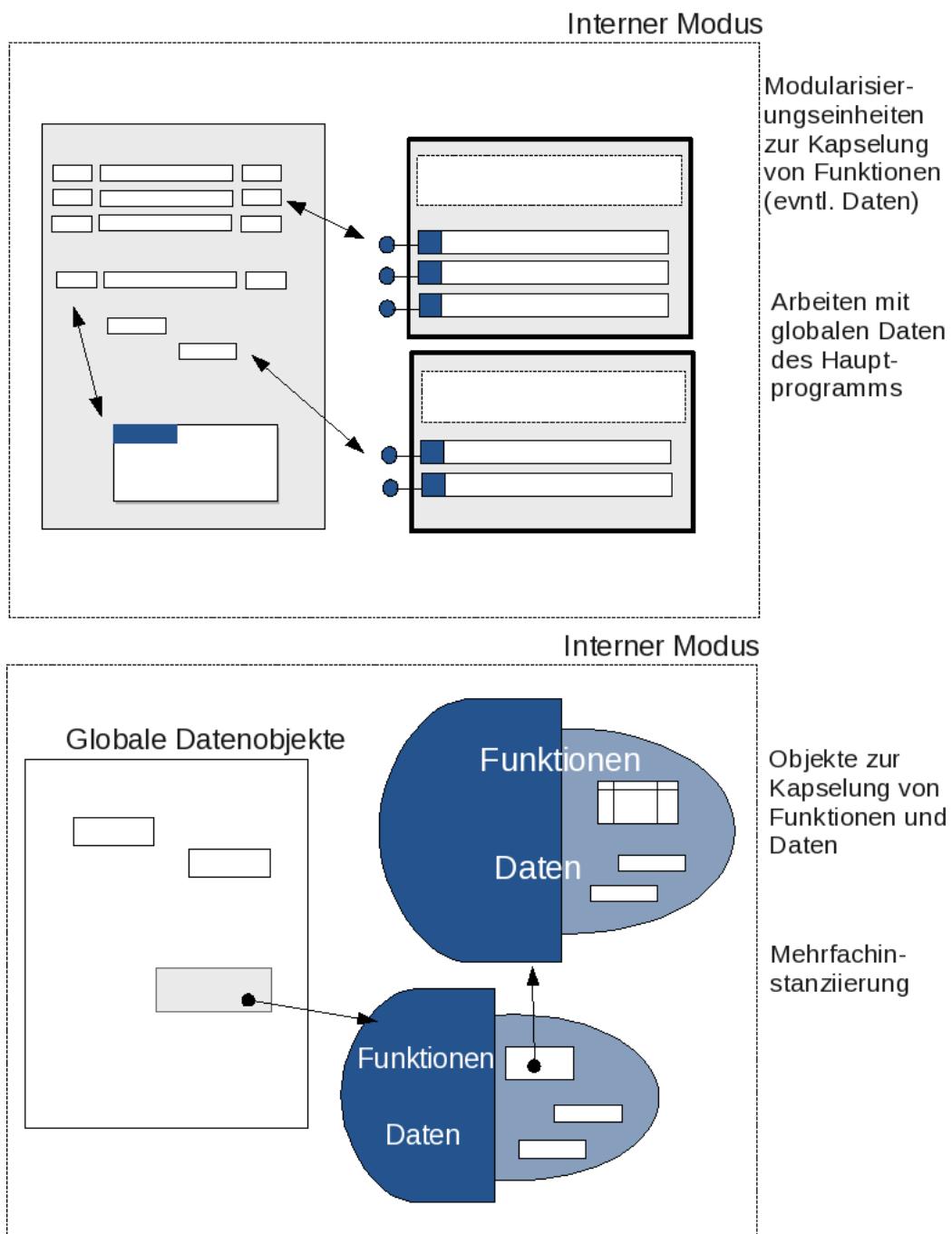


Abbildung 8: Hauptspeichersicht bei Kapselung

Aus Sicht des Hauptspeichers bestehen große Ähnlichkeiten zu Funktionsgruppen. Auch Objekte liegen im selben internen Modus wie das Programm, in dem Sie verwendet werden. Die Daten in den Objekten sind gekapselt und so von denen des umgebenden Programms separiert.

3.3 Objekte und Klassen

Die Begriffe Objekt und Klasse spielen in der objektorientierten Welt eine zentrale Rolle. Die reale Welt wird als eine Ansammlung von Objekten verstanden, etwa der Mitarbeiter Müller, das Gebäude in dem Herr Müller arbeitet, oder das Hemd das Herr Müller heute trägt. Ebenso könnte das Gebäude, in dem Herr Schmitt arbeitet als Objekt verstanden werden.

Hier fällt auf, dass die Objekte „Gebäude, in dem Herr Müller arbeitet“ und „Gebäude, in dem Herr Schmitt arbeitet“ sich sehr ähnlich sind: Sie haben Merkmale wie eine Etagenanzahl, eine Höhe, eine Farbe usw., man kann sie aufschließen, zuschließen usw. Allgemeiner kann man also sagen: Sie haben die gleichen Merkmale (was nicht heißen muss, dass die Merkmalsausprägungen gleich sind) und stellen das gleiche Verhalten dar. Das Objekt „Hemd von Herrn Müller“ hat hingegen ganz andere Merkmale, etwa eine Art des verwendeten Stoffes oder eine Konfektionsgröße, es kann gewaschen oder gebügelt werden.

Die Eigenschaften der ähnlichen Objekte, wie hier das „Gebäude in dem Herr Müller arbeitet“ und das „Gebäude in dem Herr Schmitt arbeitet“ werden in einer *Klasse* zusammengefasst. Gemeint sind also die Merkmale und Verhaltensweisen, die alle Objekte der Klasse aufweisen. In diesem Beispiel könnte es also eine Klasse „Gebäude“ geben, die besagt, dass jedes Gebäude eine Höhe, eine Farbe usw. haben muss, und dass man es auf- und zuschließen können muss. Die Klasse kann damit als eine Art Bauplan oder Schablone verstanden werden. Die Objekte sind dann *Instanzen* der Klasse: Sie haben konkrete Ausprägungen für die Merkmale, so könnte es ein Objekt der Klasse „Gebäude“ geben, das 10 Etagen hat und 30 Meter hoch ist, in dem Herr Müller arbeitet. Jedes Objekt in der objektorientierten Welt kann eindeutig identifiziert werden, es hat eine *Identität*. Diese ist vom *Status*, der den Merkmalsausprägungen entspricht, unbedingt auseinander zu halten: Würde das Gebäude in dem Herr Müller arbeitet umgebaut, so dass es nun eine zusätzliche Etage besitzt und 33 Meter hoch ist, wäre es kein anderes Gebäude, sondern noch immer das Gebäude in dem Herr Müller arbeitet (Spitzfindige werden fragen, ob Herr Müller nicht den Arbeitsplatz wechseln könnte. Dadurch bleibt das Gebäude mit seiner Identität erhalten, es muss nicht zwangsläufig abgerissen werden. Wir meinen dann mit dem Gebäude, in dem Herr Müller arbeitet, ein anderes Objekt. Es ist also darauf zu achten, dass stets klar ist, von welchem Gebäude genau die Rede ist).

Das Gebäude, in dem Herr Schmitt arbeitet, könnte genau so hoch sein und genau so viele Etagen besitzen wie das Gebäude in dem Herr Müller arbeitet. Dennoch würde es sich um zwei unterschiedliche Gebäude handeln, die eindeutig unterschieden werden können. Es handelt sich um zwei unterschiedliche *Instanzen* der Klasse „Gebäude“.

Es stellt sich nun die Frage, welche Merkmale (Attribute) überhaupt in der Klasse abgebildet werden sollen. Hier spielt nun die Sichtweise eine entscheidende Rolle:

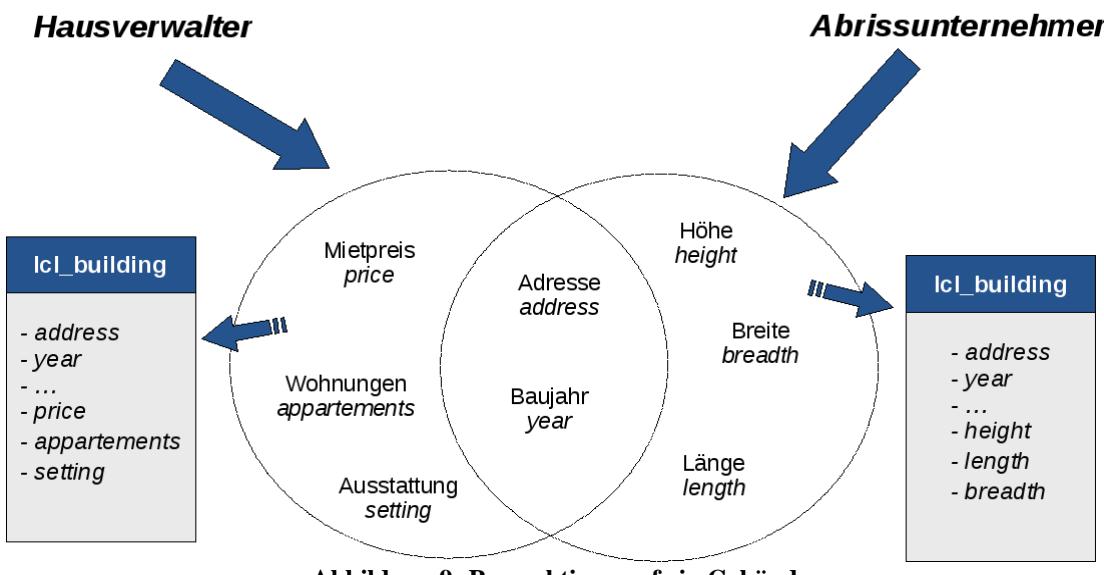


Abbildung 9: Perspektiven auf ein Gebäude

Ein Hausverwalter könnte sich bei einem Gebäude beispielsweise für den Mietpreis, die Adresse, das Baujahr, die Anzahl der Wohnungen und die Ausstattung interessieren. Ein Abrissunternehmer hätte eine ganz andere Sichtweise: Für ihn wären etwa auch Adresse und Baujahr, aber Höhe, Breite und Länge anstatt der anderen Attribute interessant. Daher würde sich die Klasse „Gebäude“ (hier: lcl_building, für local class building; mehr zur Namensgebung später) stark unterscheiden, je nachdem ob gerade eine Software für einen Hausverwalter oder eine Software für einen Abrissunternehmer erstellt werden soll.

Es gilt also sowohl relevante Klassen als auch relevante Attribute und Methoden der Klassen zu identifizieren, die für den jeweiligen Sachverhalt von Bedeutung sind. Es ist stets Abstraktion gefragt, um die relevanten Aspekte auszuwählen.

Objekte werden als Instanzen der Klassen gebildet. Die Klasse definiert die allgemeinen Eigenschaften zur Beschreibung der Objekte der Klasse in Form von Attributen (Zustandsinformationen) und Methoden (Verhaltensweisen). Das Objekt stellt dann ein Konkretes Exemplar dieser Klasse dar: Es hat bestimmte Ausprägungen der Attribute und bildet so einen Ausschnitt der realen Welt ab.

Es ist für das Verständnis der weiteren Kursinhalte unerlässlich, die Begriffe *Klasse* und *Objekt* genau auseinanderhalten zu können.

3.4 Objektorientierung in ABAP

Die Einführung von ABAP Objects wurde auch dazu genutzt, die Sprache von veralteten Sprachelementen zu bereinigen. Dies bedeutet nicht, dass die entsprechenden Sprachelemente nicht mehr funktionieren und existierende Programme umgeschrieben werden müssen, um auf einem ABAP Objects-basierten System lauffähig zu sein. SAP garantiert die Aufwärtskompatibilität aller älteren Sprachelemente. Diese Garantie galt auch schon vor der Einführung von ABAP Objects und hat zu einem hohen Sprachumfang geführt. Statt also Elemente der Sprache, die nicht mehr verwendet sollen, vollständig zu entfernen, dürfen diese veralteten Sprachelemente lediglich nicht innerhalb des objektorientierten Kontexts benutzt werden. Im objektorientierten Kontext finden auch strengere Typprüfungen statt, als dies im prozeduralen Kontext der Fall war.

Benutzt ein Entwickler im objektorientierten Kontext veraltete Sprachelemente, so führt dies zu einem Syntaxfehler, der bei der Prüfung des Programms angezeigt wird und behoben werden muss. Im prozeduralen Kontext können diese hingegen weiter verwendet werden, auch wenn dies nicht empfohlen wird. Die Bereinigung der Sprache hat nicht zuletzt auch das Ziel, Programme einfacher lesbar zu machen. Weiterhin können im prozeduralen Kontext die Anweisungen von ABAP Objects genutzt werden: So können etwa Klassen definiert oder Objekte erzeugt werden. Veraltete Sprachelemente sind in der Schlüsselwortdokumentation als obsolet gekennzeichnet. Dort ist das Verbot im objektorientierten Kontext noch einmal separat vermerkt. Um die Schlüsselwortdokumentation aufzurufen, können Sie in Ihrem Quelltext ein Schlüsselwort auswählen und mit **F1** zur entsprechenden Stelle in der Dokumentation navigieren.

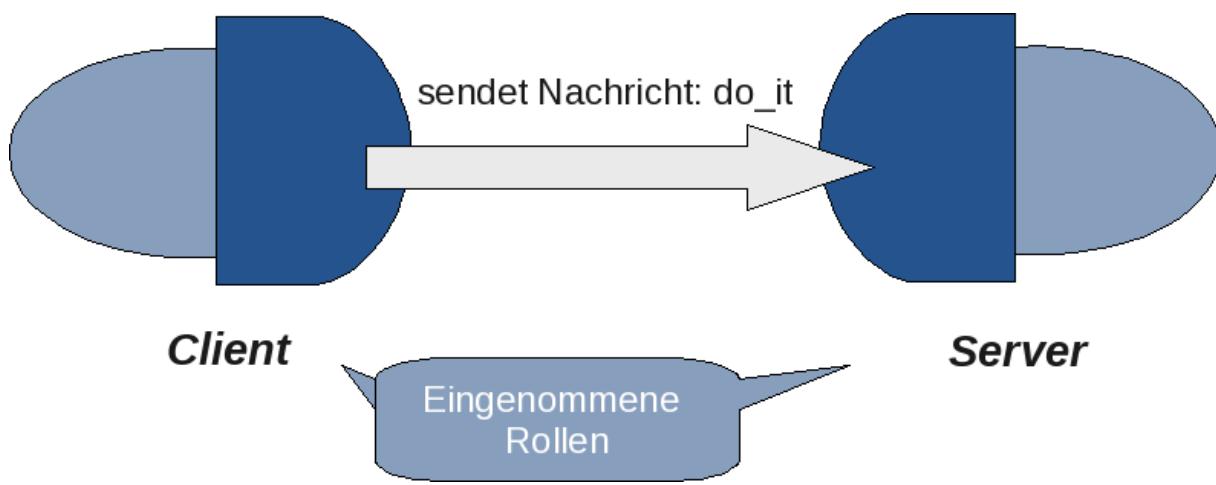


Abbildung 10: Client und Server

Objekte verhalten sich untereinander wie Clients und Server. Unter einem Server versteht man einen Dienstanbieter, unter einem Client den Konsumenten dieses Dienstes. Der Client fordert durch eine Nachricht an den Server ein bestimmtes Verhalten von diesem an. Die Rollen von Client-Objekt und Server-Objekt sind dabei nicht statisch vorgegeben sondern werden von den Objekten eingenommen und können im Verlauf der Kommunikation wechseln.

Für den Austausch von Nachrichten zwischen den Objekten muss es ein klar definiertes Protokoll geben. Dieses Protokoll wird vom Serverobjekt vorgegeben. Der Client orientiert sich an diesem Protokoll.

Die Objektorientierung erlaubt es so, die Aufgaben die durch ein Softwaresystem zu lösen sind, auf mehrere Objekte zu verteilen, die jeweils bestimmte Aufgaben wahrnehmen. Jedes Objekt nimmt die Aufgaben wahr, für die es konzipiert ist, und ruft für alle weiteren Aufgaben die dafür zuständigen Objekte auf. Dieses Konzept wird als **Delegation** bezeichnet. So könnte etwa ein Objekt für die Beschaffung von Daten zuständig sein, während ein anderes Objekt diese ausgibt. Der Vorteil besteht dann etwa darin, dass die Dienste der Objekte wiederverwendet werden können, so kann also das Objekt zur Datenbeschaffung auch ein anderes Objekt für die Ausgabe verwenden, das diese Funktionalität anders realisiert.

Neben dem direkten Nachrichtenaustausch können Objekte auch über **Ereignisse** kommunizieren. Der Unterschied besteht darin, dass zum Entwicklungszeitpunkt nicht bekannt sein muss, ob und von wem eine Reaktion auf das Ereignis erfolgt. Das Protokoll für die Kommunikation wird hier im Gegensatz zum herkömmlichen Nachrichtenaustausch nicht durch den Empfänger der Nachricht, sondern durch den Absender der Nachricht festgelegt.

Jedes Objekt gehört einer Klasse an. Klassen stehen wiederum in Beziehung zueinander. Eine besonders wesentliche Art der Beziehung zwischen Klassen stellt die **Vererbung** dar. Bei dieser Beziehung stellt eine Klasse einen speziellen Fall einer anderen Klasse dar. Dies ermöglicht die Wiederverwendung der Teile der allgemeineren Klasse, die auch auf den spezielleren Fall zutreffen. Mithilfe der Vererbung wird in der Objektorientierung **Polymorphie** umgesetzt. Diese liegt dann vor, wenn Objekte verschiedener Klassen dieselbe Nachricht empfangen können, aber unterschiedlich darauf reagieren. In den Beispielen in diesem Kurs werden Sie sehen, welche Vorteile dadurch erreicht werden.

Zusammenfassend lässt sich sagen, dass die wesentlichen Vorteile der objektorientierten Programmierung im Vergleich zur konventionellen, prozeduralen Programmierung in den folgenden Bereichen liegen:

- Bessere Strukturierung von Software
- Geringerer Wartungsaufwand sowie geringere Fehleranfälligkeit
- Durch direkte Abstraktion der Realwelt Möglichkeit einer besseren Einbindung von Auftraggeber und Anwender in den Entwicklungsprozess, vom Analyseprozess über das Design bis hin zur Wartung
- Einfachere Erweiterung der Software
- Einheitliche Kommunikation über alle Entwicklungsphasen hinweg

Die folgende Abbildung zeigt einen typischen, iterativen Entwicklungsprozess von Software. Dabei kann durchgängig mit den Begriffen der Objektorientierung gearbeitet werden, da diese sowohl in frühen Phasen die Beschreibung des Realweltausschnitts ermöglichen, als auch in späteren Phasen die Implementierung selbst beschreiben können. Gerade hier zeigt sich der Vorteil der direkten Repräsentation von Realweltobjekten im Programm.

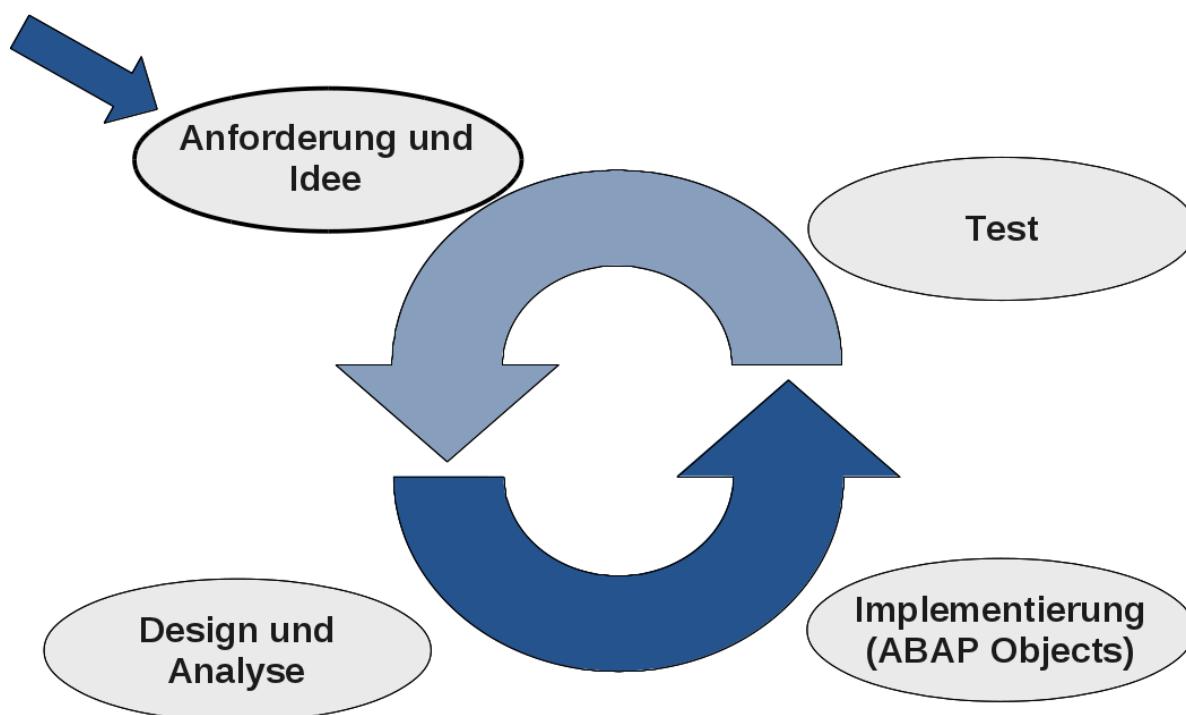


Abbildung 11: Iterativer Software-Entwicklungsprozess

Der Prozess wird in der Regel durch Modelle unterstützt. Hierdurch wird ermöglicht, von Anfang an formal strukturiert vorzugehen. In der objektorientierten Modellierung hat sich die Unified Modelling Language durchgesetzt, die im Folgenden vorgestellt wird.

3.5 Unified Modeling Language

Um objektorientierte Software modellieren zu können, werden in der Regel Diagramme der Unified Modeling Language, kurz UML, verwendet. Sie dienen dazu, Software zu designen, bevor mit der Implementierung von Code begonnen wird, weiterhin dient sie der Visualisierung, Kommunikation, Dokumentation und Spezifikation von Modellen für Softwaresysteme. Bei großen Projekten sind Modelle nicht mehr wegzudenken, aber auch bei kleineren Implementierungen ist es äußerst sinnvoll, sich zunächst anhand eines Modells zu überlegen, wie eine Software aussehen soll.

Die UML ist von der ISO standardisiert und wird von der Object Management Group (OMG) seit 1997 weiterentwickelt. Auf der Webseite der OMG (<http://www.omg.org>) finden Sie weitere Informationen zur UML, darunter auch die Spezifikation.

Innerhalb der UML gibt es eine Vielzahl unterschiedlicher Diagrammtypen. Nicht alle diese Diagrammtypen sind für Sie in diesem Kurs relevant.

Grundsätzlich lassen sich UML-Diagrammtypen in Strukturdiagramme und Verhaltensdiagramme einteilen. Das wichtigste Strukturdiagramm ist das Klassendiagramm, das Ihnen im Folgenden noch näher vorgestellt und öfters begegnen wird.

Verhaltensdiagramme fokussieren auf zeitliche Abfolgen von Verhaltensweisen der Objekte in einer Software.

Auf UML-Komponentendiagramme geht dieser Kurs nicht näher ein. Sie sollten jedoch wissen, dass die Komponenten, deren Organisation und Abhängigkeiten hierin dargestellt werden, den Paketen aus dem SAP-System entsprechen und so direkt umgesetzt werden können.

3.5.1 Das UML-Klassendiagramm

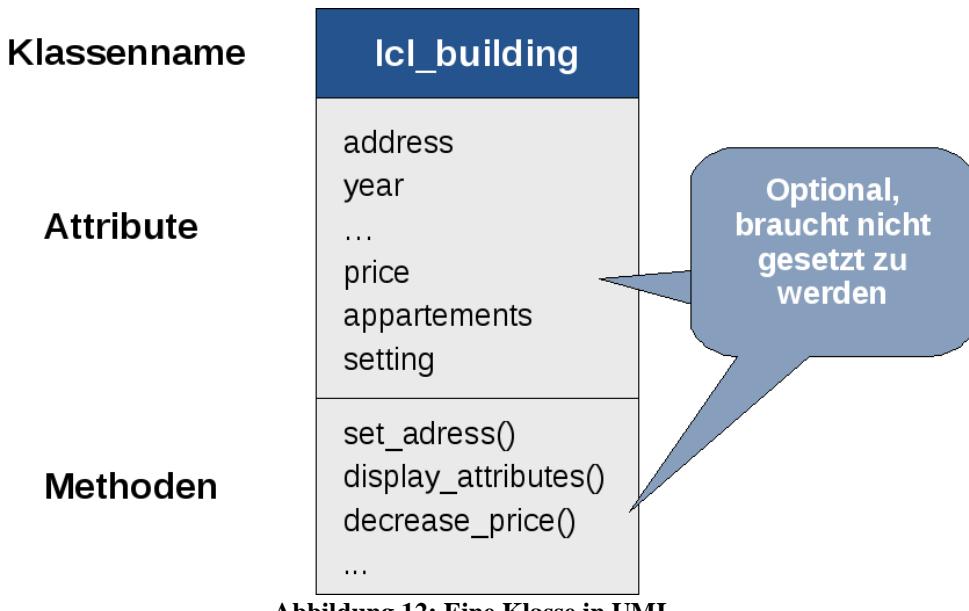


Abbildung 12: Eine Klasse in UML

Das UML-Klassendiagramm stellt Klassen und ihre statischen Beziehungen untereinander dar. Die obige Abbildung zeigt ein Beispiel für eine Klasse aus einem Klassendiagramm. Beachten Sie dass die Farben nur zur Illustration dienen und nicht Teil der UML-Spezifikation sind. Klassen werden als Rechtecke dargestellt und bestehen aus drei Teilen. Der Oberste Teil stellt den Namen der Klasse dar (zur Namensgebung später mehr). Darunter befinden sich die Attribute der Klasse. Hier können zusätzliche Angaben gemacht werden, die Sie im Kursverlauf kennen lernen. Im unteren Teil werden Die Methoden dargestellt, die als Funktionen das Verhalten repräsentieren. Die beiden unteren Teile sind Optional und müssen nicht verwendet werden, die Angabe eines Namens für die Klasse ist hingegen immer erforderlich.

In einem Klassendiagramm wird dargestellt, welche (statischen) Beziehungen die verschiedenen Klassen eines Klassensystems untereinander besitzen. Hierbei werden zwei Grundlegende Beziehungsarten unterschieden: Assoziationsbeziehungen und Generalisierungs- bzw. Spezialisierungsbeziehungen.

3.5.1.1 Assoziationsbeziehungen

Das folgende Beispiel zeigt eine einfache Assoziationsbeziehung, bei der es um eine Klasse für Kunden und eine Klasse für Reservierungen geht, die die Kunden tätigen:

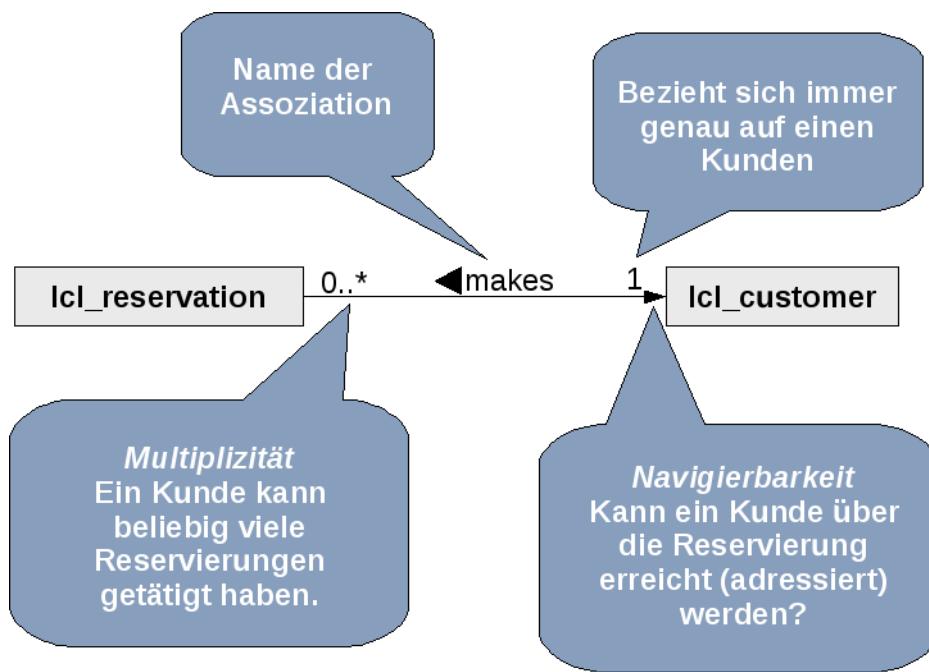


Abbildung 13: Beispiel für eine Assoziationsbeziehung

Zunächst erkennen Sie, dass die Assoziation durch eine durchgezogene Linie zwischen den beiden Klassen, hier `lcl_reservation` und `lcl_customer`, graphisch dargestellt wird. In der Mitte steht ein Name für die Assoziation, sowie ein kleines Dreieck, das die Leserichtung angibt. Die Assoziation ist hier also als „Kunde tätigt Reservierung“ zu lesen, und nicht etwa als „Reservierung tätigt Kunde“.

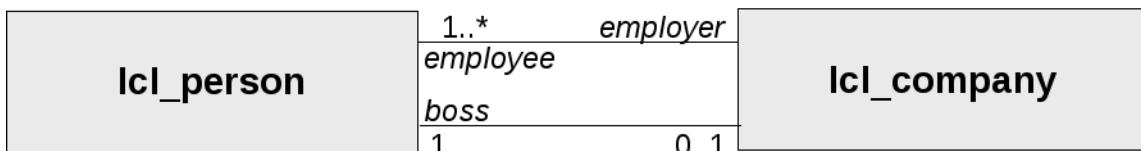
An der rechten Seite der Assoziation sehen Sie eine Pfeilspitze. Diese gibt die Navigierbarkeit an, hier also von der Reservierung zum Kunden. Sollte Ihnen dieser Begriff wenig anschaulich erscheinen, wird sich dies sicherlich im Laufe des Kurses ändern. An beiden Enden der Assoziation finden Sie zusätzlich Zahlen. Diese legen fest, mit wie vielen Objekten ein Objekt in Beziehung stehen kann:

Tabelle 1: Gebräuchliche Kardinalitäten im Überblick

* oder 0..*	Beliebig viele
1	Genau eins
1..*	Mindestens eins
0..1	Höchstens eins

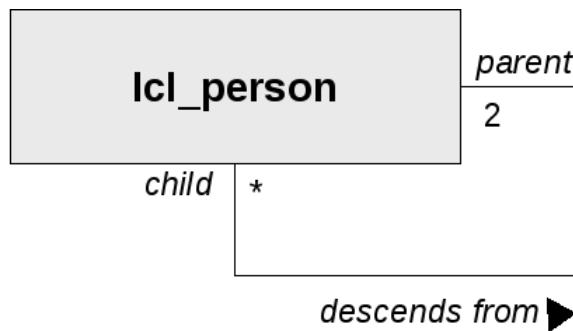
Im vorliegenden Beispiel bedeutet das: eine Reservierung gehört immer zu genau einem Kunden (Angabe der 1 am rechten Ende der Assoziation), während ein Kunde beliebig viele Reservierungen tätigen kann (Angabe 0..* am linken Ende der Assoziation).

Zwischen zwei Klassen kann auch mehr als eine Beziehung bestehen:

**Abbildung 14: Mehrfache Assoziation**

In diesem Beispiel mit einer Klasse für Personen und einer Klasse für Unternehmen bestehen zwei Beziehungen zwischen den beiden Klassen. Um diese besser lesbar zu machen, wurden Rollen angegeben. Dies sind die Bezeichnungen „employer“, „employee“ und „boss“ in der obigen Abbildung. Eine Person kann hier also sowohl als Angestellter mit einem Unternehmen verbunden sein als auch Chef eines Unternehmens sein.

Für die Lesbarkeit hilfreich sind die Rollenbezeichnungen besonders auch bei rekursiven Assoziationen. Als rekursive Assoziation wird eine Assoziation einer Klasse mit sich selbst bezeichnet, wie etwa im folgenden Beispiel:

**Abbildung 15: Rekursive Assoziation**

Hier kann eine Person Kind von anderen Personen sein.

Um bei der Analyse eines Sachverhalts die Eigenschaften einer Beziehung zu visualisieren, kann auch eine spezielle Klasse, die Assoziations-Klasse, verwendet werden:

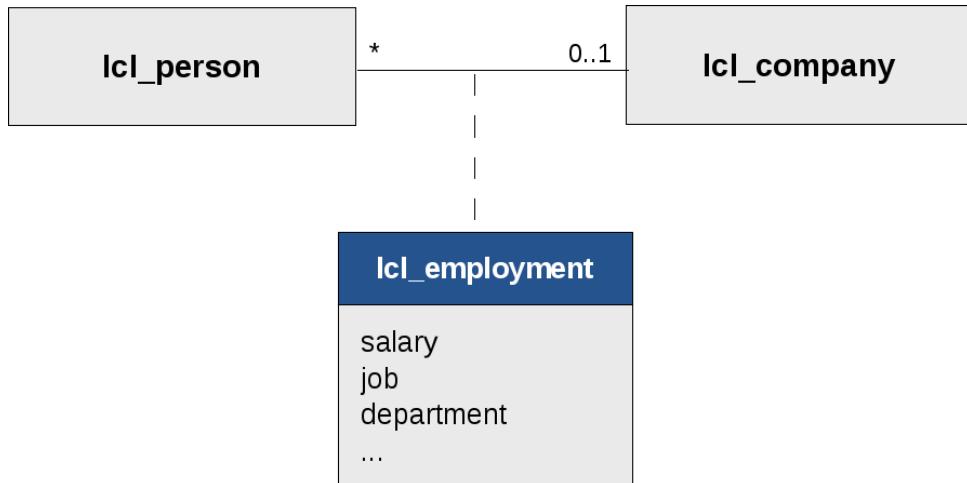


Abbildung 16: Beispiel für eine Assoziationsklasse

Über die Form der Assoziation, die Sie bisher kennen gelernt haben, gibt es zwei besondere Formen derselben: Die *Aggregation* und die *Komposition*, die ihrerseits ein Spezialfall der Aggregation ist.

Die Aggregation stellt eine Ganze-Teile-Beziehung dar, wie etwa im folgenden Beispiel:

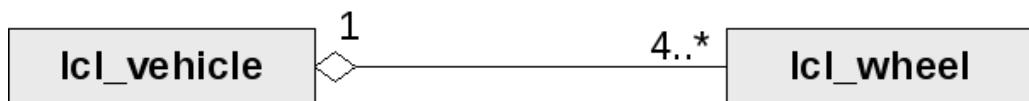


Abbildung 17: Beispiel für eine Aggregation

Hier wird dargestellt, dass das Rad Teil des Autos ist. Gekennzeichnet wird die Aggregation durch die Raute, die sich stets auf Seite des Aggregats, also auf der Seite des Ganzen befindet. Bei der Komposition handelt sich nun um den speziellen Fall, bei dem das Teil nicht ohne das Ganze existieren kann. Ein Beispiel ist das folgende:



Abbildung 18: Beispiel für eine Komposition

Hier wird eine Klasse für Rechnungen und eine Klasse für Rechnungspositionen dargestellt. Eine Rechnungsposition kann nicht ohne die zugehörige Rechnung existieren. So werden bei einer Komposition die enthaltenen Objekte zusammen mit dem Aggregat oder nach dessen Erzeugung erstellt, und vor oder zeitgleich mit dem Aggregat zerstört. Die Komposition wird wie die Aggregation dargestellt, mit dem Unterschied dass die Raute ausgefüllt wird.

3.5.1.2 Generalisierungs- / Spezialisierungsbeziehungen

Bei der Generalisierungs- und Spezialisierungsbeziehung handelt es sich um dieselbe Beziehung, die je nach Bezeichnung aus der Perspektive der allgemeineren oder aus der Perspektive der spezielleren Klasse betrachtet wird. Es wird auch von einer Ober- und einer Unterklasse gesprochen.

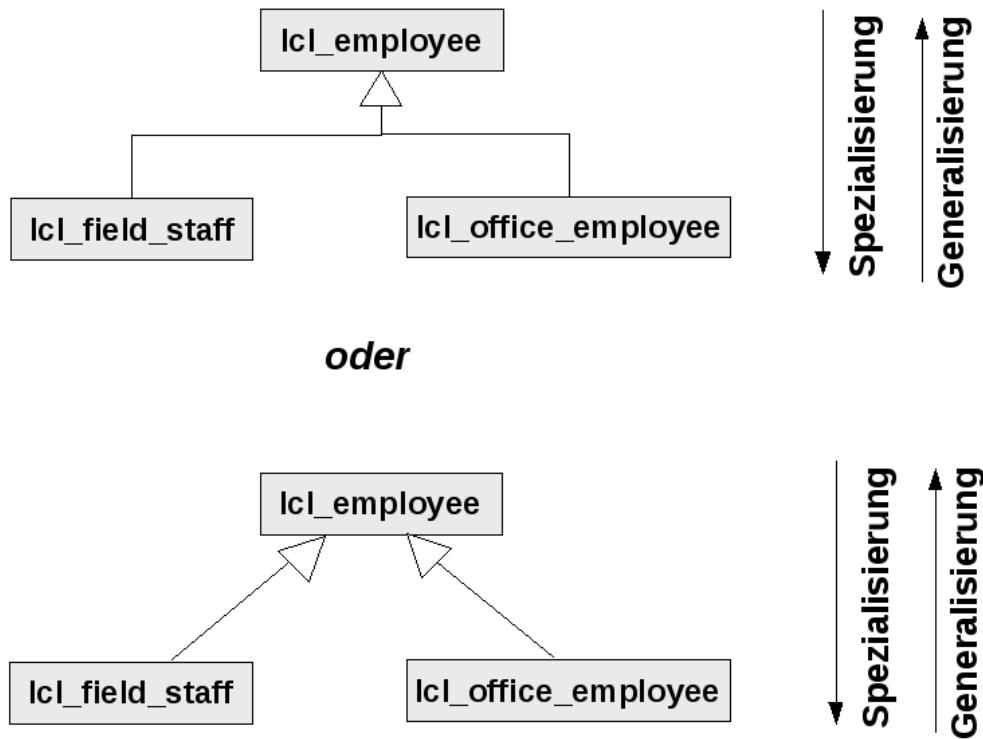


Abbildung 19: Beispiel mit zwei Notationsarten

Das obige Beispiel zeigt eine Klasse für Mitarbeiter und zwei davon abgeleitete Klassen für Außendienst- und Innendienstmitarbeiter. Die beiden unteren Klassen sind Spezialfälle von Mitarbeitern, es handelt sich um Spezialisierungen der Mitarbeiterklasse. Umgekehrt betrachtet handelt es sich bei der Mitarbeiter-Klasse um eine Generalisierung der anderen beiden Klassen.

Die Beziehung wird über einen Pfeil mit Dreieckspitze dargestellt. Beide dargestellten Notationen sind zulässig. Es kann selbstverständlich auch nur eine oder mehr als zwei spezielle Klassen unter einer Klasse geben, und auch die speziellen Klassen könnten weiter abgeleitet werden, wenn dies erforderlich ist.

3.5.2 Das UML-Objektdiagramm

Das Objektdiagramm zeigt den aktuellen Zustand von Objekten, also die aktuelle Belegung von Attributwerten von Objekten und die Beziehungen zwischen den konkreten Objekten. Dies entspricht einem „Schnappschuss“ der aktuellen Instanzen.

Die folgende Abbildung zeigt ein Beispiel:

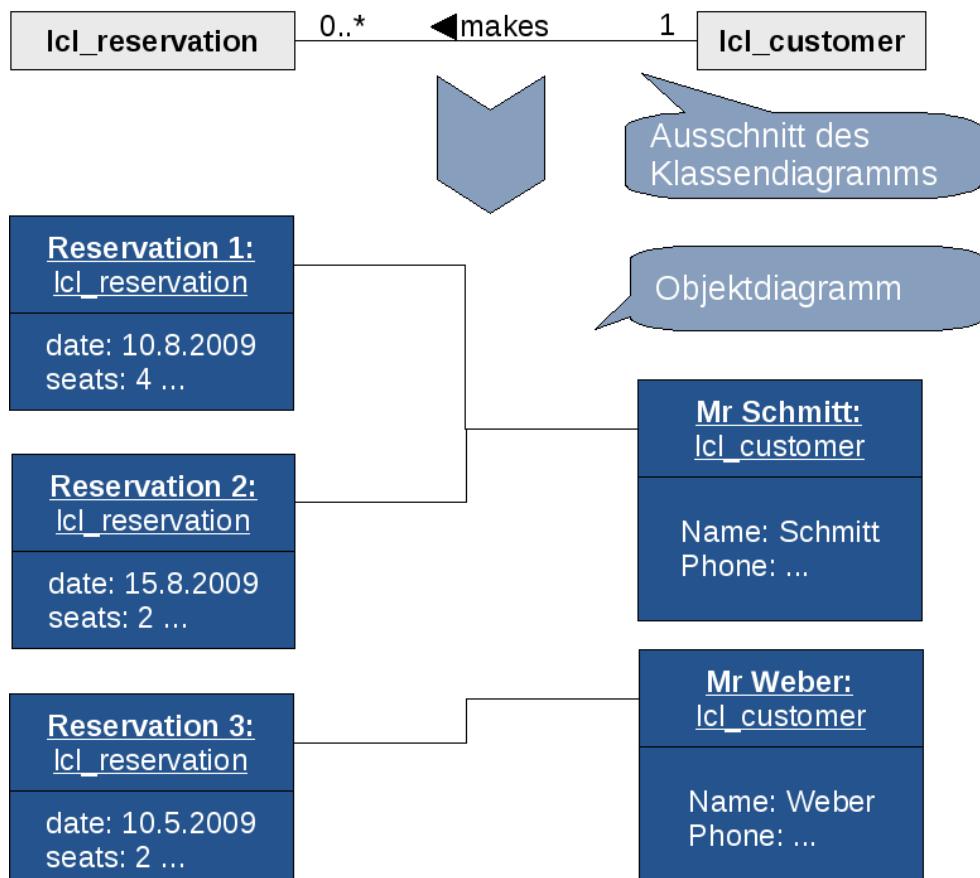


Abbildung 20: Beispiel für ein Objektdiagramm

In diesem Objektdiagramm werden drei Instanzen (Objekte) der Klasse `lcl_reservation` sowie zwei Instanzen der Klasse `lcl_customer` dargestellt, sowie die Beziehungen zwischen diesen.

Das Objektdiagramm muss den Angaben zur Beziehung aus dem Klassendiagramm genügen. So wäre das folgende Objektdiagramm kein mögliches Diagramm zum Klassendiagramm:

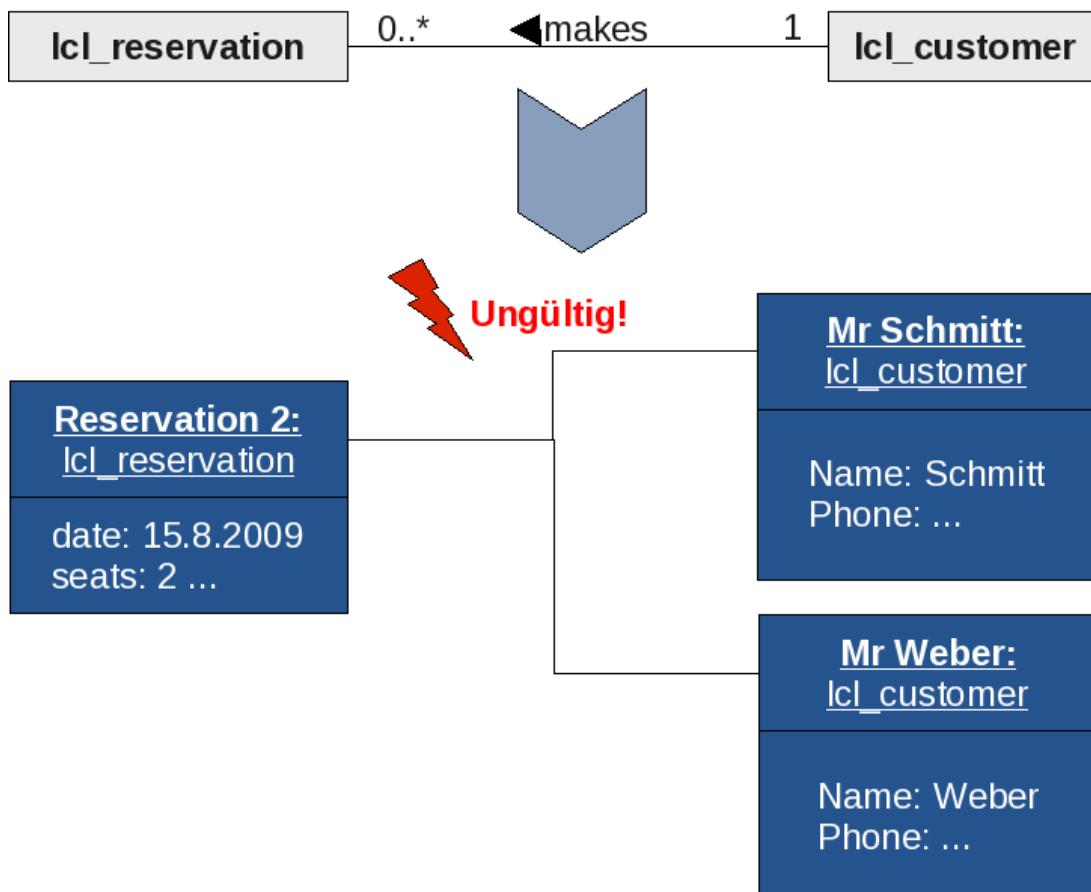


Abbildung 21: Unzulässiges Objektdiagramm

Das Objektdiagramm ist zwar in sich syntaktisch korrekt, es enthält jedoch eine Reservierung, die mehr als einem Kunden zugeordnet ist. Das Klassendiagramm schreibt jedoch vor, dass es zu jeder Bestellung nur einen Kunden gibt.

3.5.3 Das UML-Sequenzdiagramm

Um nun auch zeitliche Abläufe darstellen zu können, wird das UML-Sequenzdiagramm verwendet. Mithilfe eines solchen Diagramms können die Kommunikationsabläufe zwischen den Objekten visualisiert werden.

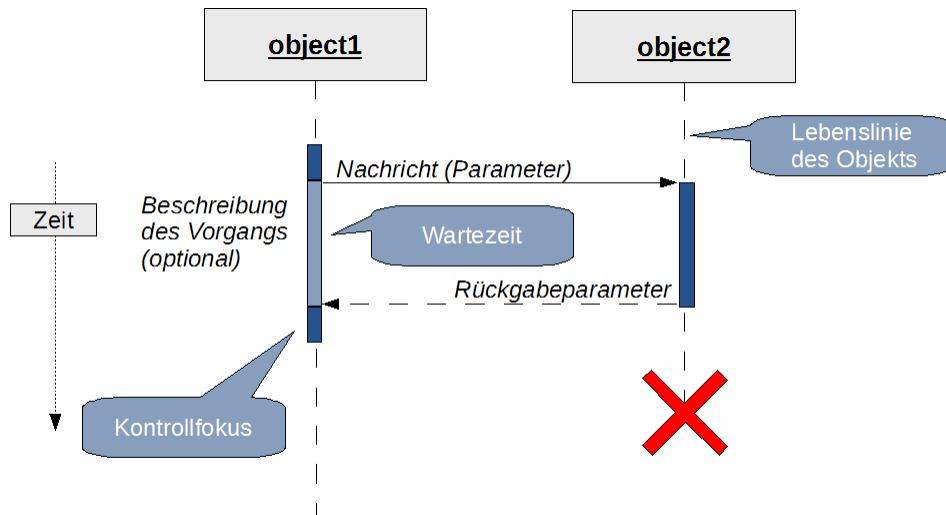


Abbildung 22: Aufbau von Sequenzdiagrammen

Die Abbildung zeigt den allgemeinen Aufbau eines Sequenzdiagramms. Das Diagramm kann als zeitlicher Verlauf von oben nach unten gelesen werden. Die beteiligten Objekte haben jeweils eine **Lebenslinie** (gestrichelte Linie unterhalb des Objekts). Durch die Balken auf der Lebenslinie eines Objekts wird dargestellt, ob dieses **aktiv** ist, also ob es gerade Aktionen ausführt. Weiterhin kann es **indirekt aktiv** sein, wenn es auf die Rückkehr von einem Aufruf wartet. Solche Aufrufe werden durch waagerechte Pfeile dargestellt, auf denen die Nachricht (also die gerufene Methode) notiert wird. Für die Rückkehr wird hier ein gestrichelter Pfeil verwendet, auf dem angegeben werden kann was von der Methode zurückgegeben wird. Neben der Kommunikation wird auch die Lebensdauer eines Objekts dargestellt. Hierzu dient die Lebenslinie. Durch ein Kreuz wird das Ende der Lebensdauer gekennzeichnet, wie hier bei object2.

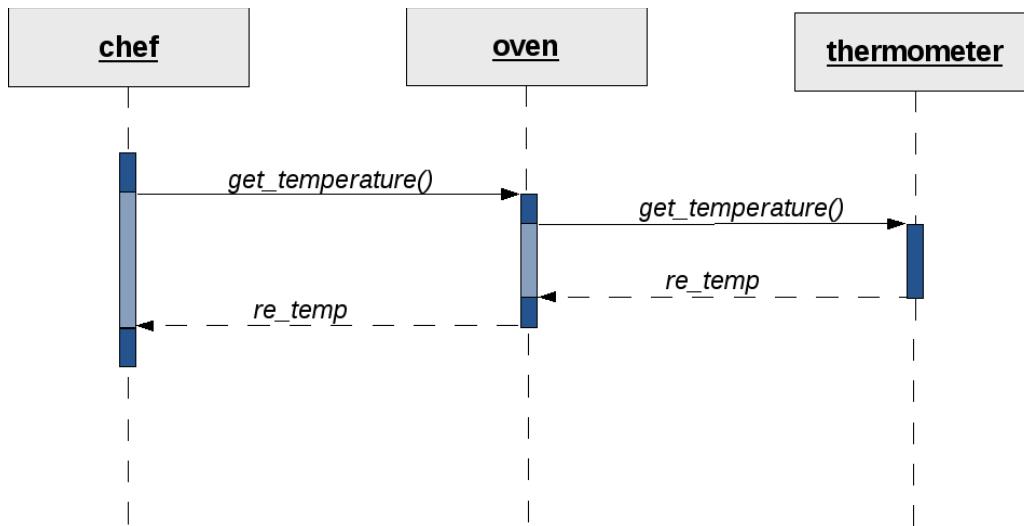


Abbildung 23: Delegationsbeispiel als Sequenzdiagramm

Die obige Abbildung zeigt ein konkretes Beispiel für ein Sequenzdiagramm. Hier wird eine Delegation veranschaulicht: Das Koch-Objekt („chef“) sendet die Nachricht (den Methodenaufruf) `get_temperature()` an das Ofen-Objekt („oven“). Dieses wiederum sendet eine entsprechende Nachricht an den Thermometer, um die Temperatur zu ermitteln. Diese liefert es dann an das Koch-Objekt zurück, möglicherweise führt es dazwischen noch eine Konvertierung von Einheiten o. ä. durch.

Die UML bietet noch eine Vielzahl weiterer Modelltypen, die jedoch für diesen Kurs nicht benötigt werden. Bei Interesse können Sie diese in der entsprechenden Literatur nachlesen oder die Spezifikation auf der zuvor genannten Internetseite anschauen.

3.5.4 Kontrollfragen

Im Folgenden finden Sie einige Kontrollfragen, anhand derer Sie das Verständnis der Inhalte überprüfen können. Die Kontrollfragen können nicht alle Inhalte abdecken, sollen aber sicherstellen dass Grundlegende Dinge verstanden wurden.

1. Welche der folgenden Aussagen ist wahr?
 - a. Eine Klasse ist die Instanz eines Objekts.
 - b. Ein Objekt ist die Instanz einer Klasse.
2. Werden Objekte über ihre Attributwerte Identifiziert?
 - a. Ja
 - b. Nein
3. Eine Klasse beschreibt Merkmale, die alle Objekte der Klasse aufweisen.
 - a. Richtig
 - b. Falsch
4. Das Konzept der Kapselung von Daten wurde in ABAP erst durch ABAP Objects möglich.
 - a. Richtig
 - b. Falsch
5. Wie werden Klassen in UML dargestellt?
6. Wie wird eine Komposition in UML kenntlich gemacht?
 - a. Durch eine dreieckige Pfeilspitze
 - b. Durch eine nicht ausgefüllte Raute
 - c. Durch eine ausgefüllte Raute
7. Wie wird eine Generalisierungs-/Spezialisierungsbeziehung in UML kenntlich gemacht?
 - a. Durch eine dreieckige Pfeilspitze
 - b. Durch eine nicht ausgefüllte Raute
 - c. Durch eine ausgefüllte Raute
8. Bei einer Kompositionsbeziehung...
 - a. ...kann das Ganze nicht ohne seine Teile bestehen
 - b. ...kann das Teil nicht ohne das Ganze bestehen

Die Antworten finden Sie auf der nächsten Seite.

3.5.5 Antworten

1. b
2. b
3. a

4. b
5. Siehe Abschnitt zu UML-Klassendiagrammen.
6. c
7. a
8. b

3.6 Grundlegende objektorientierte Syntax in ABAP

In diesem Abschnitt lernen Sie die Syntax kennen, mit der Sie grundlegende objektorientierte Konzepte in ABAP umsetzen. Zugleich werden Ihnen Attribute und Methoden von Klassen sowie einige weitere Konzepte nähergebracht. Später werden Ihnen weitere objektorientierte Konzepte vorgestellt, die Sie ebenfalls in ABAP-Code umsetzen.

Um ein objektorientiertes Programm entwickeln zu können, müssen Sie zunächst wissen wie eine Klasse und ihre Methoden und Attribute definiert werden.

Um eine Klasse zu erstellen, benötigen Sie in ABAP, im Gegensatz zu vielen anderen Sprachen, einen separaten Definitions- und Implementierungsteil.

```
CLASS lcl_employee DEFINITION.

ENDCLASS.
```

```
CLASS lcl_employee IMPLEMENTATION.

ENDCLASS.
```

Abbildung 24: Definitions- und Implementierungsteil einer Klasse

Im Definitionsteil der Klasse wird definiert, welche Attribute und Methoden die Klasse besitzt. Hinzu kommen weitere Definitionen (Ereignisse, Interfaces), die in späteren Kursteilen vorgestellt werden. Im Implementierungsteil der Klasse werden hingegen lediglich die Methoden der Klasse implementiert. CLASS-Anweisungen dürfen nicht geschachtelt werden. Der Definitionsteil wird, genau wie der Implementierungsteil, durch das Schlüsselwort ENDCLASS abgeschlossen.

Wie bereits zuvor erläutert, ist ein Ziel der Objektorientierung die Kapselung. Um diese zu erreichen, besitzt eine Klasse öffentliche und private Komponenten. Auf die öffentlichen Komponenten kann von außerhalb der Klasse zugegriffen werden, während die privaten Komponenten nur innerhalb der Klasse verwendet werden können. So kann z. B. verhindert werden, dass bestimmte Attribute unkontrolliert beschrieben oder gelesen werden können.

Als Beispiel soll im Folgenden eine Klasse für Mitarbeiter dienen:

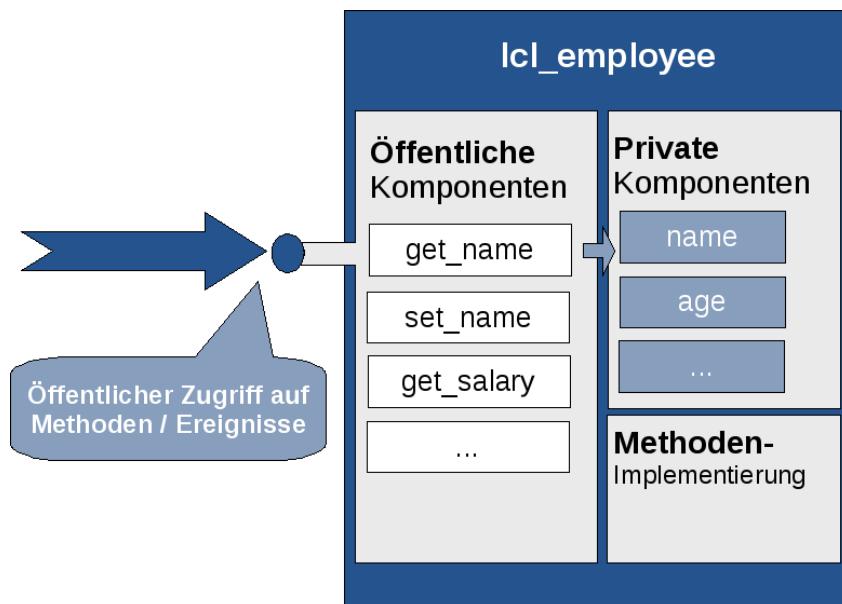


Abbildung 25: Eine Klasse für Mitarbeiter

Auf die öffentlichen Komponenten kann von außen zugegriffen werden. Dies wird durch den Anknüpfungspunkt auf der linken Seite der Klasse veranschaulicht. Die privaten Komponenten sind von außen nicht erreichbar, können aber von den öffentlichen Komponenten verwendet werden. In diesem Fall kann es also eine öffentliche Methode `get_name` geben, die ihrerseits in ihrer Implementierung auf das private Attribut `name` zugreift. Die öffentlichen Komponenten einer Klasse werden auch als **Schnittstelle der Klasse** bezeichnet. Jeder externe Verwender kann nur über diese Schnittstelle auf die Klasse zugreifen.

3.6.1 Definition von Attributen

In der obigen Abbildung sind bereits einige Methoden und Attribute dargestellt. Sicherlich fallen Ihnen einige weitere Möglichkeiten ein, einen Mitarbeiter zu beschreiben.

Attribute werden im Definitionsteil der Klasse spezifiziert. Dazu wird die DATA-Anweisung verwendet, die Sie bereits von der Definition gewöhnlicher Datenobjekte kennen. Hierbei zeigt sich, dass mit der Einführung von ABAP Objects veraltete Sprachelemente ausgeräumt wurden. Im Prozeduralen Kontext, also außerhalb von Klassen, ist es möglich eine Typisierung mit Bezug auf ein Dictionary-Element über `LIKE` durchzuführen:

```
DATA feld LIKE spfli.
```

Dies mag Ihnen ungewohnt erscheinen, da im Kurs „Einführung in ABAP“ die Typisierung mit `TYPE` verwendet wird, wohingegen `LIKE` für den Bezug auf andere Datenobjekte verwendet wird. Dies ist auch die vorgesehene Verwendung in ABAP Objects: Dort wird `LIKE` für den Bezug auf lokale Datenobjekte sowie auf SY-Felder (SY-DATUM usw.) verwendet. Die oben dargestellte Deklaration würde hingegen im objektorientierten Kontext zu einem Syntaxfehler führen:

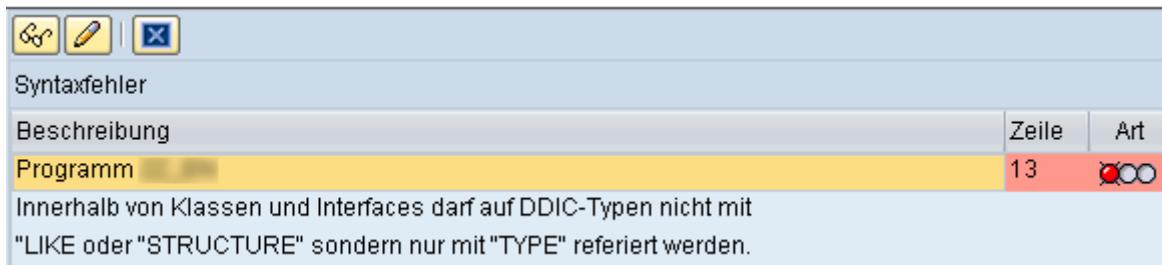


Abbildung 26: Syntaxfehler im objektorientierten Kontext: SAP-System-Screenshot

Die folgende Abbildung zeigt schematisch einige Beispiele für Attributdefinitionen.

```

CLASS lcl_myclass DEFINITION.
  ...
  TYPES: local_type ...,
  ...
  CONSTANTS: ... .
  DATA: att1 TYPE local_type,
        att2 TYPE global_type,
        att3 LIKE field2,
        att4 TYPE i VALUE 3,
        att5 TYPE i READ-ONLY.
ENDCLASS.

```

Abbildung 27: Beispiele für Attributdefinitionen: SAP-System-Screenshot

Die ersten vier Definitionen im DATA-Kettensatz sollten Ihnen vertraut erscheinen und könnten so auch als gewöhnliche Definitionen in einem prozeduralen Programm auftauchen. Bei der fünften Definition wurde der Zusatz READ-ONLY angegeben. Durch diesen wird festgelegt, dass ein Attribut von außen (also von außerhalb der Klasse) nur gelesen, nicht aber geschrieben werden darf. Dies ist nur im öffentlichen Bereich einer Klassendefinition zulässig. Dies wirft die Frage auf, was es mit dem öffentlichen und dem privaten Bereich der Klassendefinition genau auf sich hat.

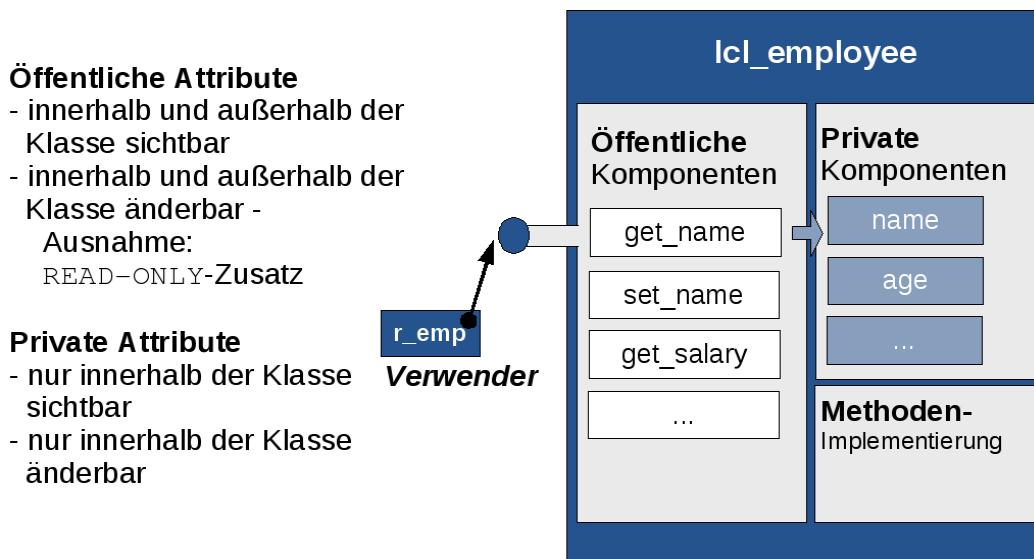


Abbildung 28: Sichtbarkeitsbereiche von Attributen

Der Definitionsteil der Klasse teilt sich in zwei Bereiche auf: den öffentlichen Bereich, der mit PUBLIC SECTION eingeleitet wird, und den privaten Bereich, der mit PRIVATE SECTION eingeleitet wird.

Attribute, die im öffentlichen Bereich stehen, können innerhalb und außerhalb der Klasse verwendet werden. Ein Beispiel für eine Verwendung innerhalb einer Klasse ist der Zugriff aus einer Methode derselben Klasse. Ein Beispiel für die Verwendung außerhalb der Klasse ist ein Objekt, z. B. der Mitarbeiter Schmitt, dessen Attributwert in einem Programm über eine Referenz auf das Objekt ausgelesen wird. Wird der zuvor erwähnte READ ONLY-Zusatz angegeben, kann von außerhalb der Klasse nur lesend zugegriffen werden. So kann eine get-Methode gespart werden, also eine öffentliche Methode, mit der ein privates Attribut gelesen wird. Der Nachteil von öffentlichen Attributen ist, dass bei Änderungen nicht nur die Klasse selbst modifiziert werden muss, sondern auch die Stelle an der auf das Attribut zugegriffen wurde. Wird hingegen eine öffentliche Schnittstelle in Form einer Methode angeboten, kann innerhalb der Klasse alles verändert werden, solange die Schnittstelle nicht verändert wird, und der Verwender ist nicht davon betroffen. Weiterhin kann durch die Methode sicher gestellt werden, dass alle Attribute sich in einem konsistenten Zustand befinden, also keine unzulässigen Werte oder Kombinationen von Werten zugewiesen werden. Es gilt daher als guter Programmierstil, möglichst wenige öffentliche Attribute zu verwenden. Man spricht hier von **Information Hiding** oder Kapselung. Das Thema wird wieder aufgegriffen, wenn Ihnen Methoden vorgestellt werden.

Neben den beiden hier vorgestellten Sichtbarkeitsbereichen Public und Private gibt es noch einen dritten Bereich, den Protected-Bereich. Dieser spielt aber erst im Kontext von Vererbung eine Rolle und wird Ihnen daher erst in Abschnitt 4.7 vorgestellt.

Beachten Sie jedoch schon jetzt, dass bei der Klassendefinition die folgende Reihenfolge der Sichtbarkeitsbereiche vorgeschrieben ist:

```
PUBLIC SECTION.  
...  
PROTECTED SECTION.  
...  
PRIVATE SECTION.  
...
```

Wie an den Beispielen ersichtlich ist, müssen nicht alle Sichtbarkeitsbereiche verwendet werden.

```
CLASS lcl_building DEFINITION.
  PUBLIC SECTION.
    DATA: storeys TYPE i,
          height TYPE i.

  PRIVATE SECTION.
  ...
ENDCLASS.
```



```
CLASS lcl_building DEFINITION.
  PUBLIC SECTION.
  ...
  PRIVATE SECTION.
    DATA: storeys TYPE i,
          height TYPE i.

  ENDCLASS.
```



Abbildung 29: Öffentliche und private Attribute

Die obige Abbildung zeigt die Definition von zwei Attributen für die Etagenzahl (`storeys`) und die Höhe (`height`) von Gebäuden (Klasse `lcl_building`). Im oberen Fall handelt es sich um öffentliche Attribute. Es wäre hier möglich, z. B. negative Zahlen zu speichern, aber auch eine Höhe zu speichern, die nicht mit der Etagenzahl zusammenpassen kann. Bei der unteren Variante sind die Attribute hingegen nicht öffentlich zugreifbar. Es könnte nun eine öffentliche Methode implementiert werden, die die Attributwerte setzt, und zuvor auch prüft ob die zu setzenden Werte zulässig bzw. konsistent sind und im Fehlerfall eine Ausnahme auslöst.

Die Variante eines `READ-ONLY`-Zusatzes (hier nicht abgebildet) wird vornehmlich dann verwendet, wenn besonderer Wert auf die Effizienz gelegt wird, da dann ein Methodenaufruf weniger anfällt, wenn das Attribut nur gelesen werden soll.

Bislang haben Sie lediglich Attribute kennen gelernt, die zur Laufzeit ein einzelnes Objekt einer Klasse beschreiben, etwa die Höhe eines bestimmten Gebäudes oder der Name eines bestimmten Mitarbeiters. Diese Attribute werden auch als **Instanzattribute** bezeichnet, da Sie für jede Instanz definiert sind und einen unterschiedlichen Wert annehmen können. Diese Attribute sind der Regelfall. Es gibt darüber hinaus aber auch Attribute, die es nur einmal für eine gesamte Klasse gibt. Diese Attribute werden in ABAP Objects **statische Attribute** genannt. In anderen Programmiersprachen existiert auch die Bezeichnung **Klassenattribut**, diese ist aber im Bereich von ABAP nicht üblich. Die statischen Attribute sind für alle Instanzen der Klasse sichtbar. Sie werden verwendet, um Informationen zu verwalten, die für alle Instanzen gemeinsam gelten. Dies können etwa gepufferte Daten, Konstanten oder Instanzzähler sein, die festhalten wie viele Instanzen von einer Klasse erzeugt wurden.

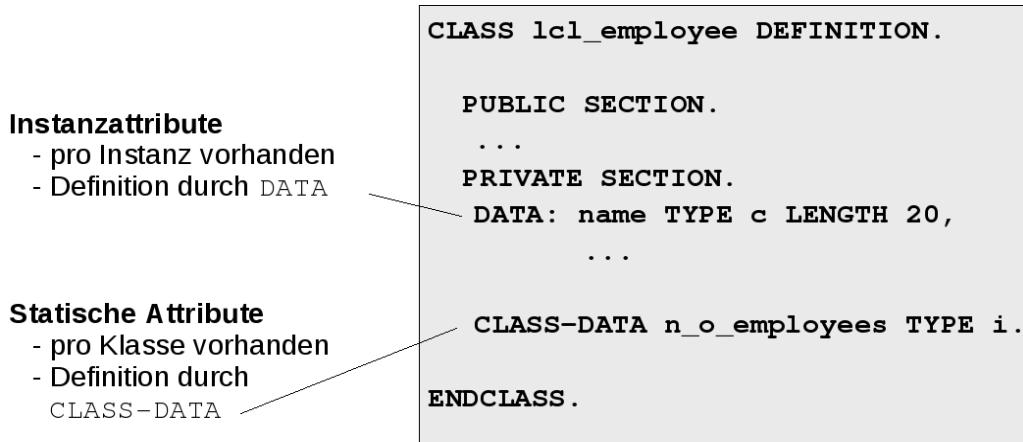


Abbildung 30: Instanz- und statische Attribute

Die obige Abbildung zeigt die Definition eines statischen Attributs `n_o_employees` für die Mitarbeiterklasse. Dieses soll die Anzahl der Instanzen der Klasse enthalten, also die Anzahl der erzeugten Mitarbeiterobjekte.

Zur Laufzeit besitzt dann jedes Objekt ein eigenes Namens-Attribut, es existiert aber nur eine Mitarbeiteranzahl:

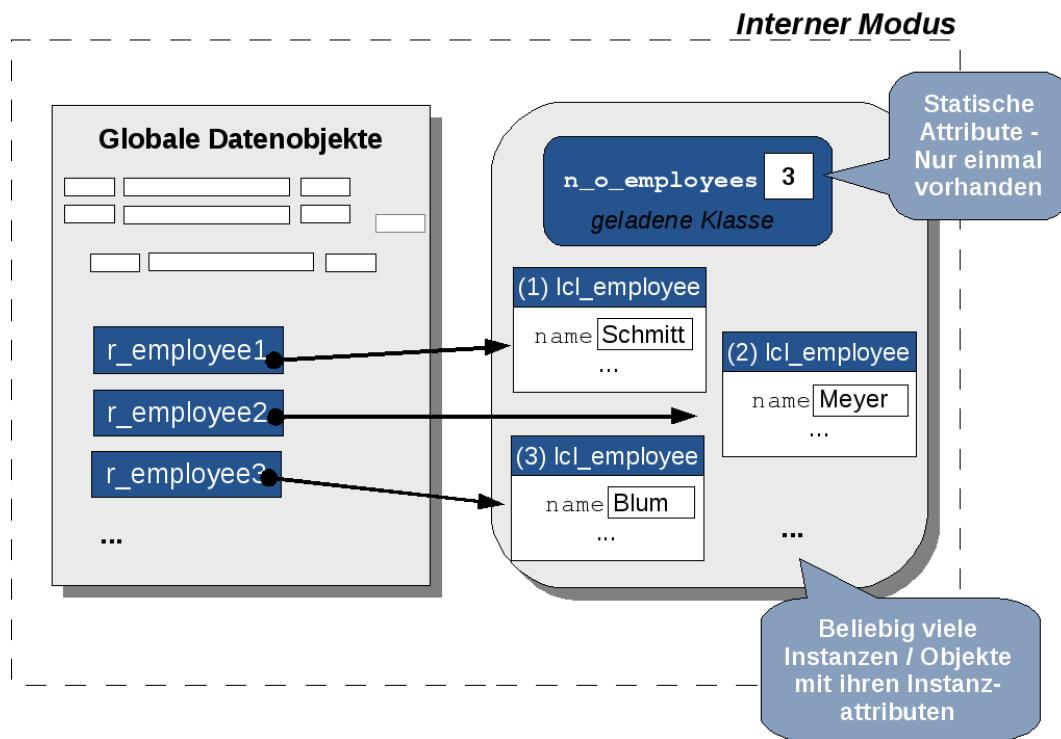


Abbildung 31: Statische Attribute zur Laufzeit

Beachten Sie, dass das statische Attribut manuell gepflegt werden muss, indem es etwa beim Erzeugen der Instanzen erhöht wird. Das System stellt die Anzahl der Instanzen nicht zur Verfügung.

3.6.2 Referenzvariablen

In den Abbildungen wie Abbildung 31: Statische Attribute zur Laufzeit haben Sie bereits Referenzen kennen gelernt. Dabei handelt es sich um Variablen, deren Inhalt ein Zeiger auf ein Objekt ist. Anschaulich vorstellen kann man sich dies durch einen Vergleich mit Luftballons: Die Referenzvariable ist wie ein Faden, an dem der Luftballon (das Objekt) festgehalten wird. Zu einem Luftballon (Objekt) kann es mehrere Fäden geben (mehrere Referenzvariablen können auf dasselbe Objekt zeigen). Enthält jedoch keine Referenzvariable mehr einen Verweis auf das Objekt (existiert kein Faden mehr, an dem der Luftballon festgehalten wird), kann das Objekt nicht mehr erreicht werden (der Luftballon fliegt nach oben). Das Objekt wird dann durch das System zerstört. Dieser Vorgang nennt sich **Garbage Collection**.

```

CLASS lcl_employee DEFINITION.
  PUBLIC SECTION .
  ...
  PRIVATE SECTION.
  ...
ENDCLASS .

CLASS lcl_employee IMPLEMENTATION
  ...
ENDCLASS .

DATA: r_employee1 TYPE REF TO lcl_employee,
      r_employee2 LIKE r_employee1.

START-OF-SELECTION.
  ...

```

Abbildung 32: Definition von Referenzvariablen

Die Definition einer Referenzvariablen erfolgt wie in der obigen Abbildung über die Angabe `REF TO` nach `TYPE` und vor der Angabe der Klasse. Die Angegebene Klasse besagt, dass dieser Zeiger auf Objekte der Klasse `lcl_employee` zeigen soll. Da dem Zeiger aber noch nichts zugewiesen wurde, zeigt er auf nichts. Dies wird auch als *Nullreferenz* bezeichnet. Auch hier ist eine Typisierung über `LIKE` und die Angabe einer anderen Referenzvariablen möglich.

Um den Zeigern ein Objekt zuzuweisen, auf das diese zeigen sollen, kann mit dem Befehl `CREATE OBJECT` ein neues Objekt erzeugt werden:

```
DATA: r_employee1 TYPE REF TO lcl_employee,
      r_employee2 LIKE r_employee1.

CREATE OBJECT r_employee1.
CREATE OBJECT r_employee2.
```

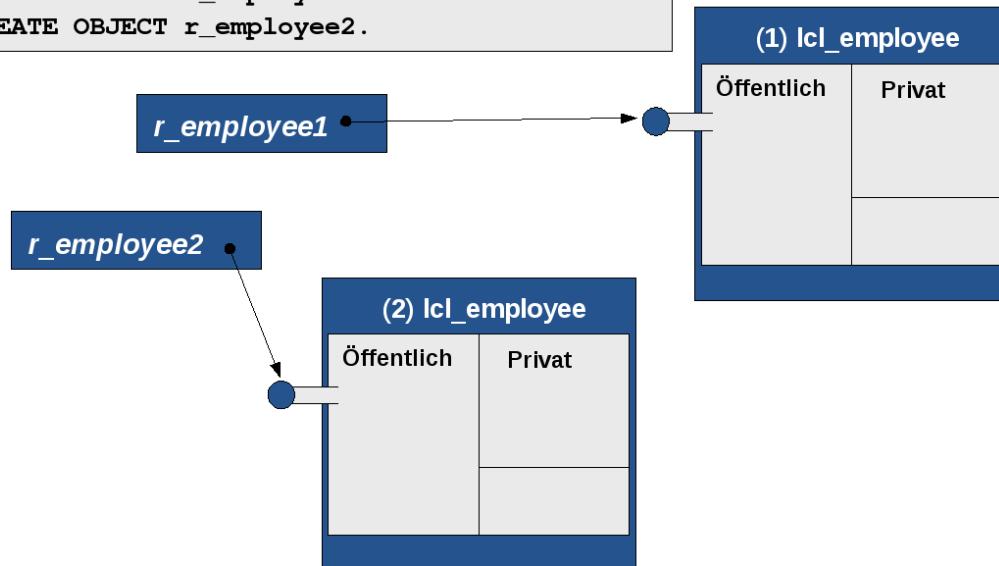


Abbildung 33: Erzeugung neuer Objekte

Die `CREATE OBJECT`-Anweisung erzeugt das Objekt im Hauptspeicher. Durch die beiden abgebildeten `CREATE OBJECT`-Anweisungen zeigen nun die Referenzvariablen `r_employee1` und `r_employee2` auf jeweils ein Objekt.

Wie bereits bei der Veranschaulichung mit den Luftballons beschrieben, ist es möglich zwei Referenzvariablen auf dasselbe Objekt verweisen zu lassen. Dazu können die Referenzvariablen einander zugewiesen werden, wie sie es von anderen Variablen bereits gewohnt sind.

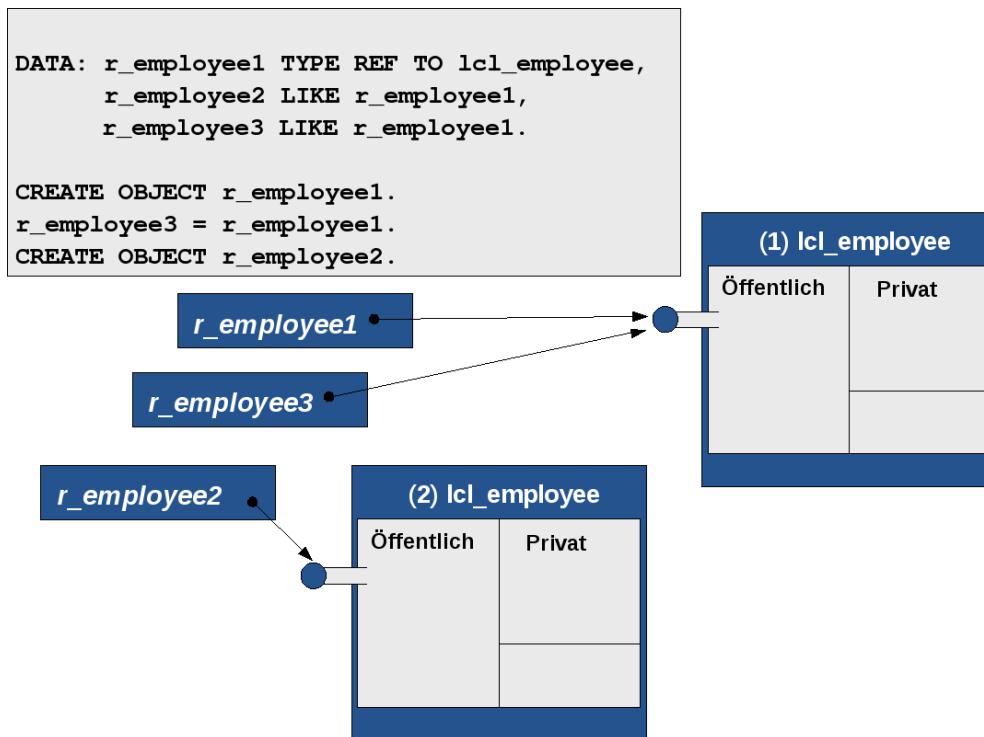


Abbildung 34: Mehrere Referenzen auf ein Objekt

In diesem Beispiel (siehe obige Abbildung) wird der Variablen r_employee3 eine Referenz auf das Objekt, auf das r_employee1 zeigt, zugewiesen. Beachten Sie, dass beide Variablen nun auf das Objekt „(1) lcl_employee“ zeigen. Würde man r_employee1 nun die Referenz von r_employee2 zuweisen, hätte dies auf r_employee3 keinen Einfluss, es würde weiterhin auf dasselbe Objekt zeigen.

Um eine Referenz zu entfernen, also eine Variable nicht mehr auf ein Objekt zeigen zu lassen, kann der CLEAR- oder der FREE-Befehl verwendet werden:

```
CLEAR r_employee1.  
Oder  
FREE r_employee1.
```

In beiden Fällen wird die Variable auf ihren typgerechten Initialwert zurückgesetzt. Im Fall einer Referenzvariablen ist dies die Null-Referenz.

Wenn ein Objekt nicht mehr Referenziert wird, es also keine Referenzvariable mehr gibt, die auf das Objekt verweist, ist dieses nicht mehr erreichbar. Es würde aber ohne spezielle Maßnahmen im Hauptspeicher verbleiben. In der Veranschaulichung würde dies bedeuten, dass ein Luftballon über keinen Faden mehr festgehalten wird und zur Decke steigt. Um nun zu verhindern dass der Speicher überläuft, gibt es den sogenannten **Garbage Collector**. Hierbei handelt es sich um eine Systemroutine, die die nicht mehr erreichbaren Objekte aus dem Speicher entfernt.

- Zeigt auf ein Objekt keine eigenständige Referenz mehr, so ist es syntaktisch nicht mehr zugreifbar.

- Alle Objekte, auf die syntaktisch nicht mehr zugegriffen werden kann, werden vom Garbage Collector automatisch gelöscht.

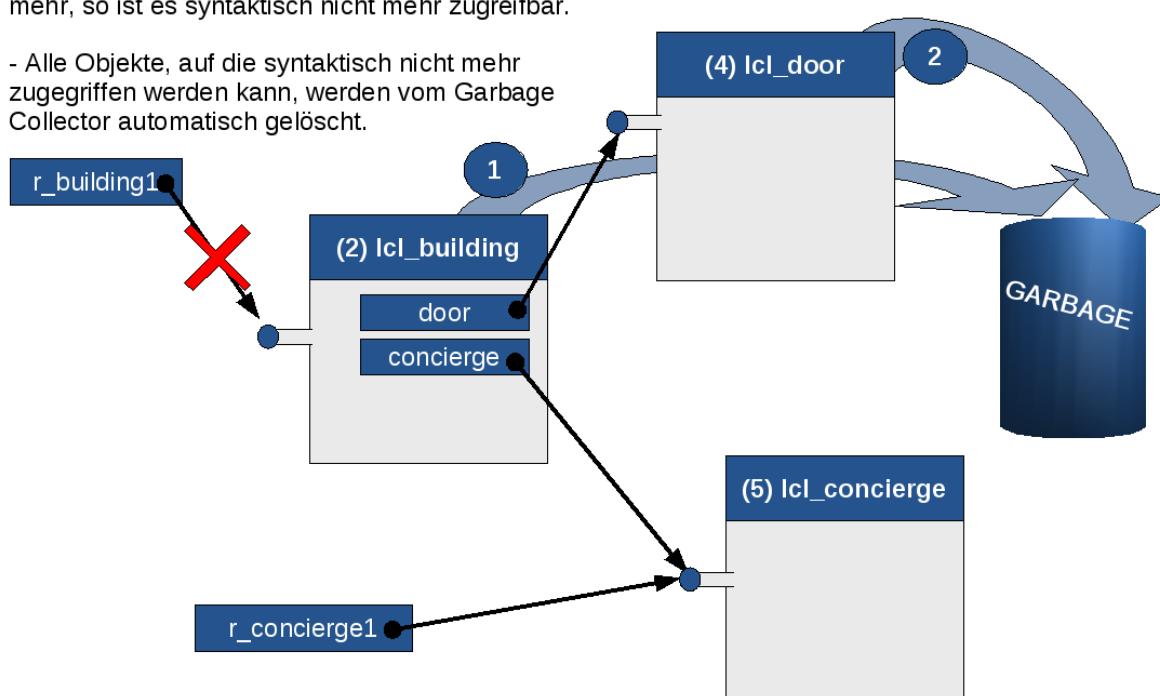


Abbildung 35: Beispiel zum Garbage Collector

Das obige Beispiel zeigt eine Referenzvariable, die auf ein Gebäude-Objekt zeigt. Dieses Objekt hat wiederum ein Attribut vom Referenztyp einer Tür, die auf ein entsprechendes Objekt zeigt, analog dazu hat das Gebäude einen Hausmeister. Zusätzlich existiert eine Referenzvariable auf den Hausmeister. Wird nun mit dem FREE- oder CLEAR-Befehl die Referenzvariable, die zuvor auf das Gebäude zeigte, auf die Null-Referenz initialisiert, ist das Objekt „(2) lcl_building“ nicht mehr erreichbar. Der Garbage-Collector wird dieses Objekt also Löschen (1). Dadurch ist die Referenz auf die Tür aus dem Attribut des Gebäudeobjektes nicht mehr vorhanden, und so ist auch das Türobjekt nicht mehr erreichbar. Es wird daher ebenfalls vom Garbage Collector entfernt (2). Das Hausmeisterobjekt wird hingegen nicht entfernt: Auch nachdem das Gebäudeobjekt nicht mehr existiert, gibt es eine Referenzvariable (r_concierge1), die auf ebendieses zeigt.

Um anzufragen, ob eine Variable die Null-Referenz enthält, kann die Abfrage IF referenzvariable IS INITIAL verwendet werden.

Implizit wird in diesem Beispiel auch gezeigt, dass Attribute auch mit dem REF TO-Zusatz typisiert werden können, also ebenfalls Verweise sein können.

Wie bei gewöhnlichen Typen ist auch bei Referenzvariablen der Aufbau von internen Tabellen möglich:

```

DATA: r_employee  TYPE REF TO lcl_employee,
      itab        TYPE TABLE OF REF TO lcl_employee.

CREATE OBJECT r_employee.
APPEND r_employee TO itab.

CREATE OBJECT r_employee.
APPEND r_employee TO itab.

READ TABLE itab INTO r_employee
  WITH KEY table_line->pub_attr1 = ...
    table_line->pub_attr2 = ...

LOOP AT itab INTO r_employee
  [ WHERE table_line->pub_attr3 ...
    AND   table_line->pub_attr4 ... ] .
  ...
ENDLOOP.

```

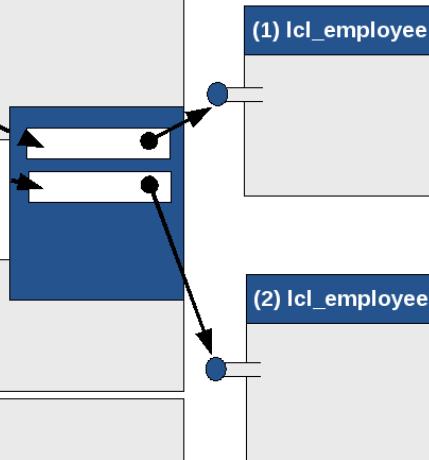


Abbildung 36: Interne Tabellen aus Referenzvariablen

Die Pseudokomponente `table_line` dient der Spezifikation logischer Bedingungen. Hierfür müssen die verwendeten Attribute öffentlich sein.

3.6.3 Zugriff auf Attribute

In der obigen Abbildung 36: Interne Tabellen aus Referenzvariablen sehen sie bereits einen neuen Operator, den einfachen Pfeil `->`, der sich aus einem Minuszeichen und einem Größer-als-Zeichen zusammensetzt. Dieser Operator wird als Komponentenselektor bezeichnet. Vor dem Operator steht der Name einer Referenzvariablen, dahinter der Name der Komponente auf die zugegriffen werden soll.

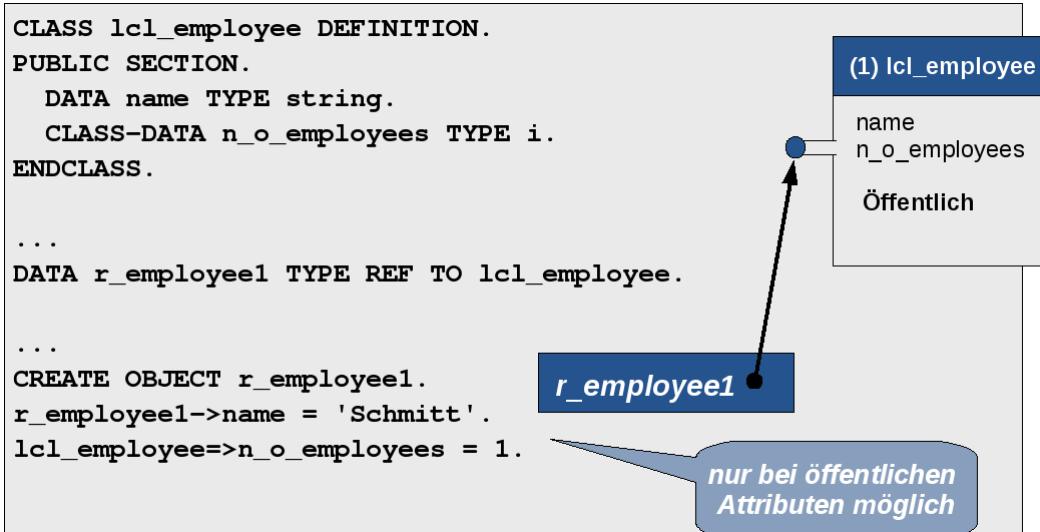


Abbildung 37: Zugriff auf öffentliche Attribute

Im abgebildeten Beispiel ist das Attribut `name` öffentlich und daher zugreifbar.

In der Zeile unterhalb des Zugriffs auf `name` sehen Sie einen weiteren neuen Operator, den Pfeil mit zwei Linien `=>`, der sich aus einem Gleichzeichen und einem Größer-als-Zeichen zusammensetzt. Dieser Operator wird für den Zugriff auf statische Attribute verwendet. Vor dem Operator muss der Name der Klasse, hinter dem Operator der Name des statischen Attributs stehen. Auch hier gilt, dass das statische Attribut öffentlich sein muss, damit ein Zugriff in dieser Form möglich ist.

3.6.4 Definition und Implementierung von Methoden

Nachdem Sie nun die Definition von Variablen kennen gelernt haben, folgt nun die Definition und Implementierung von Methoden. Wie bereits Eingangs skizziert, sind in ABAP Objects die Definition und die Implementierung von Methoden voneinander getrennt.

Methoden bestimmen das Verhalten von Objekten. Es handelt sich um Prozeduren innerhalb von Klassen, folglich haben sie vollen Zugriff auf alle privaten und öffentlichen Komponenten der Klasse.

Bei der Definition der Methode wird lediglich ihre Signatur angegeben, die zunächst aus den IMPORTING-, EXPORTING- und CHANGING-Parametern besteht, die Sie bereits von Funktionsbausteinen kennen. Zusätzlich kann mit RETURNING ein Rückgabewert der Methode definiert werden, der als Wert zurückgegeben werden muss. Hinzu kommen Ausnahmen. Hierbei wird zwischen dem klassischen Ausnahmekonzept, das Sie von Funktionsbausteinen kennen, und dem neuen klassenbasierten Ausnahmekonzept unterschieden. Letzteres werden Sie im Laufe dieses Kurses noch näher kennen lernen. Beachten Sie, dass beide Formen nicht vermischt werden dürfen.

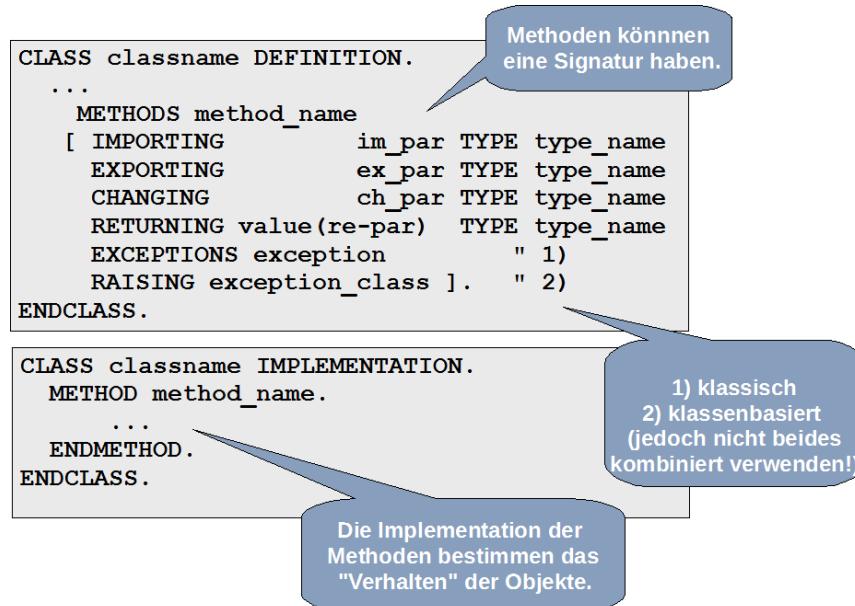


Abbildung 38: Methodendefinition und -implementierung

Den Parametern können die Zusätze OPTIONAL und DEFAULT zugewiesen werden. OPTIONAL kennzeichnet optionale Parameter, die also beim Aufruf nicht zwingend versorgt werden müssen. DEFAULT ermöglicht stattdessen die Angabe eines Standardwerts für den Fall, dass der Parameter nicht übergeben wird. Bei OPTIONAL wird hingegen immer mit dem Typgerechten Initialwert initialisiert.

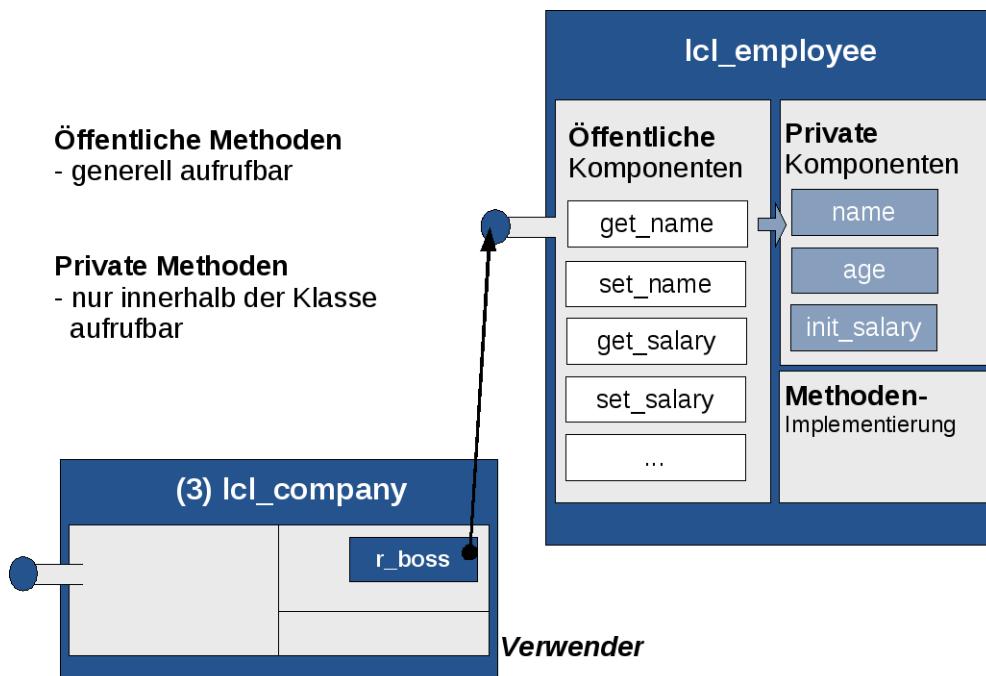


Abbildung 39: Sichtbarkeit von Methoden

Wie die Attribute sind auch die Methoden einem Sichtbarkeitsbereich zugeordnet. So wird festgelegt, ob diese nur innerhalb der Klasse, oder auch von außerhalb aufgerufen werden können.

```

CLASS lcl_employee DEFINITION.
  PUBLIC SECTION.
    METHODS set_salary IMPORTING im_salary TYPE i.
    ...
  PRIVATE SECTION.
    METHODS init_salary, ... .
    DATA: salary TYPE i, ... .
    ...
ENDCLASS.

```

```

CLASS lcl_employee IMPLEMENTATION.
  METHOD init_salary.
    *      Setze Standard-Gehalt
    ENDMETHOD.
  METHOD set_salary.
    IF im_salary IS INITIAL.
    *      Rufe Methode init_salary
    ELSE.
      salary = in_salary.
    ENDIF.
    ENDMETHOD.
    ...
ENDCLASS.

```

Abbildung 40: Zugriff auf eine private Methode

Im obigen Codebeispiel greift die öffentliche Methode `set_salary` auf die private Methode `init_salary` zu, um ein Standardgehalt zu bestimmen und in das Attribut zu speichern. Von Außen ist letztere Methode nicht erreichbar.

Im Definitionsteil der Klasse befindet sich nur die Signatur der Methode. Diese wird durch das Schlüsselwort `METHODS` begonnen (beachten Sie den Plural! Lassen Sie sich nicht durch eine evtl. falsche Wortvervollständigung des ABAP-Editors in die Irre führen). Es gibt hierbei noch syntaktische Varianten, die sich allerdings auf spezielle Konzepte der objektorientierten Programmierung beziehen, die Sie erst in späteren Kursteilen kennen lernen werden. Dort wird die entsprechende Syntax vorgestellt.

Im Implementierungsteil der Klasse befindet sich dann der eigentliche Code der Methode. Dieser wird wie folgt umrahmt:

```

METHOD meth.
  ...
ENDMETHOD.

```

Beachten Sie hier, dass das Schlüsselwort im Singular steht (`METHOD` statt `METHODS`) und der Codebereich mit `ENDMETHOD` abzuschließen ist.

In der obigen Abbildung wird das Attribut der Klasse mit `salary` erreicht. Innerhalb der Methode liegt ein lokaler Namensraum vor, der den Namensraum der umgebenden Klasse überdeckt. Der Namensraum der Klasse umfasst alle Attribut-, Methoden-, Ereignis-, Konstanten-, Typen- und Alias-Namen. Würde in der Methode also eine Variable definiert, die ebenfalls `salary` heißt, würde die lokale Variable die Definition aus der Klasse überdecken.

Analog zu statischen Attributen gibt es auch statische Methoden. Die Syntax ist ähnlich: Bei diesen Methoden wird im Definitionsteil der Klasse METHODS durch CLASS-METHODS ersetzt.

Im Implementierungsteil einer statischen Methode darf nur auf statische Komponenten der Klasse zugegriffen werden. Das hängt damit zusammen, dass eine statische Methode sich auf die Klasse und nicht auf eine spezielle Instanz bezieht. Würde man etwa in einer statischen Methode auf ein nicht-statisches Attribut „Name“ zugreifen, wäre unklar wessen Name gemeint ist, da kein Bezug zu einem Objekt existiert.

Instanzmethoden

- Zugriff auf statische und Instanzkomponenten möglich
- Definition per METHODS

```
CLASS lcl_employee DEFINITION.
  PUBLIC SECTION.
    METHODS ... .
  CLASS-METHODS get_n_o_employees
    EXPORTING ex_count TYPE i.

  PRIVATE SECTION.
    DATA: ... .
    CLASS-DATA n_o_employees TYPE i.
ENDCLASS.
```

Statische Methoden

- Zugriff nur auf statische Komponenten möglich
- Definition per CLASS-METHODS

```
CLASS lcl_employee IMPLEMENTATION.
  ...
  METHOD get_n_o_employees.
    ex_count = n_o_employees.
  ENDMETHOD.

  ...
ENDCLASS.
```

Abbildung 41: Statische Methoden und Instanzmethoden

Das Überladen von Methoden ist in ABAP nicht möglich. Hierunter versteht man das vorhanden sein mehrerer Definitionen zu einer Methode, deren Signatur und Implementierung sich jeweils unterscheidet.

Bei den oben gezeigten Methoden waren die Bezeichner von Parametern der Methoden und die Bezeichner von Attributen der Klasse stets unterschiedlich. Tritt hingegen eine Namensgleichheit auf, so ist zunächst nur der Parameter erreichbar. Um auch das Klassenattribut erreichen zu können, ist eine **Selbstreferenz** erforderlich. Diese heißt in ABAP me. Die folgende Abbildung zeigt ein Beispiel:

```
CLASS lcl_employee DEFINITION.  
  PUBLIC SECTION.  
    METHODS set_name IMPORTING name TYPE string.  
    ...  
  PRIVATE SECTION.  
    DATA name TYPE string.  
    ...  
  ENDCCLASS.  
  
CLASS lcl_employee IMPLEMENTATION.  
  METHOD set_name.  
    me->name = name.  
  ENDMETHOD.  
  ...  
ENDCLASS.
```

Abbildung 42: Verwendung der Selbstreferenz

Hier ist die Selbstreferenz erforderlich, um zwischen dem Parameter `name` und dem Attribut `name` zu unterscheiden. Das Attribut der Klasse wird durch den Parameter der Methode verdeckt und ist daher ohne die Selbstreferenz nicht mehr erreichbar. Auch an Stellen, an denen die Angabe nicht zwingend erforderlich ist, darf sie zur Verbesserung der Lesbarkeit eingesetzt werden.

Ein weiteres Anwendungsbeispiel ist eine Instanzmethode, die beim Aufruf einer anderen Methode eine Referenz auf das eigene Objekt übergeben soll. Hier kann ebenfalls `me` verwendet werden.

3.6.5 Zusätzliche Eigenschaften von Komponenten in UML

Die zuvor dargestellten Eigenschaften der Sichtbarkeit und Statik von Attributen und Methoden sind auch in UML darstellbar.

lcl_employee	
<ul style="list-style-type: none"> - name - salary - <u>n_o_employees</u> ... 	<ul style="list-style-type: none"> + bezeichnet öffentliche Komponenten - bezeichnet private Komponenten
<ul style="list-style-type: none"> + set_name - init_salary + get_n_o_employees ... 	<ul style="list-style-type: none"> statische Komponenten werden <u>unterstrichen</u>

```

CLASS lcl_building DEFINITION.
  PUBLIC SECTION.
    METHODS set_name ...
    CLASS-METHODS get_n_o_employees...

  PRIVATE SECTION.
    DATA: name ... ,
          salary ... .
    CLASS-DATA n_o_employees ... .
    METHODS init_salary ... .
    ...
ENDCLASS.

```

Abbildung 43: Erweiterte Angaben in UML

Öffentliche Komponenten können durch ein vorangestelltes Pluszeichen dargestellt werden, private Komponenten durch ein vorangestelltes Minus. Statische Komponenten werden unterstrichen.

Die Namen und Typen von IMPORTING- und CHANGING-Parametern können bei Bedarf ebenfalls angegeben werden. Dabei wird in einer Klammer nach dem Methodennamen zunächst der Name des Parameters angegeben, und nach einem Doppelpunkt der Typ desselben. Besitzt die Methode einen Rückgabewert (einen RETURNING-Parameter), wird dessen Typ nach einem Doppelpunkt ans Ende der Methodenbeschreibung gesetzt.

Die folgende Abbildung zeigt ein einfaches Beispiel für eine solche Beschreibung.

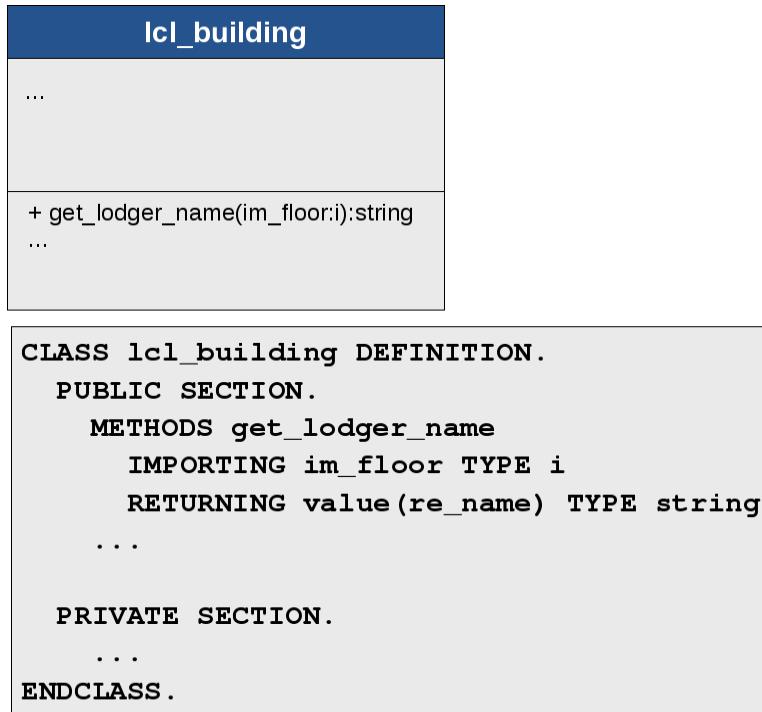


Abbildung 44: Parameter und Rückgabetyp in UML

3.6.6 Methodenaufrufe

Der Aufruf von Methoden entspricht der zu Beginn erläuterten Kommunikation von Objekten durch Nachrichten. Er ist zunächst syntaktisch dem Zugriff auf Attribute ähnlich: Auch hier kommen die Selektoren -> für Instanzmethoden und => für statische Methoden zum Einsatz.

Für den Aufruf gibt es eine längere Form, die mit den Schlüsselwörtern CALL METHOD eingeleitet wird, und seit SAP Web AS 6.10 eine verkürzte Form. Beide Formen werden in der folgenden Abbildung dargestellt:

```
CALL METHOD ref->method_name
  EXPORTING im_par      = val_ex ...
  IMPORTING ex_par      = val_im ...
  CHANGING ch_par      = val_chg ...
  RECEIVING re_par      = val_res
  EXCEPTIONS exception = val_rc ... .
```

Verkürzte Syntax:

```
ref->method_name(
  EXPORTING im_par      = val_ex ...
  IMPORTING ex_par      = val_im ...
  CHANGING ch_par      = val_chg ...
  RECEIVING re_par      = val_res
  EXCEPTIONS exception = val_rc ... ).
```

Beispiel:

```
DATA: r_employee TYPE REF TO lcl_employee,
      name TYPE string, firstname TYPE string.
...
r_employee1->get_name( IMPORTING ex_name = name
                        ex_firstname = firstname ).
```

Abbildung 45: Aufrufsyntax von Instanzmethoden

Wenn Sie zuvor Erfahrungen mit Sprachen wie Java gemacht haben, wird Ihnen die verkürzte Syntax vertrauter vorkommen. Die Zuordnung von Formal- und Aktualparametern erfolgt Analog zu Funktionsbausteinen. Mit RECEIVING wird ggf. der Rückgabewert zugeordnet. Hierfür werden Sie noch eine einfachere Schreibweise kennen lernen. Die Parameterarten IMPORTING-/CHANGING und RECEIVING schließen sich gegenseitig aus. Beachten Sie bei der Syntax die Leerzeichen: In der verkürzten Schreibweise **darf** vor der Klammer **kein** Leerzeichen stehen, hinter der Klammer **muss** hingegen ein Leerzeichen stehen!

Der Aufruf statischer Methoden erfolgt analog, hier wird der Selektor => verwendet und statt einer Referenzvariablen der Name der Klasse angegeben:

```
CALL METHOD class_name=>method_name
  EXPORTING im_par      = val_ex ...
  IMPORTING ex_par      = val_im ...
  CHANGING ch_par      = val_chg ...
  RECEIVING re_par      = val_res
  EXCEPTIONS exception = val_rc ... .
```

Verkürzte Syntax:

```
class_name=>method_name(
  EXPORTING im_par      = val_ex ...
  IMPORTING ex_par      = val_im ...
  CHANGING ch_par      = val_chg ...
  RECEIVING re_par      = val_res
  EXCEPTIONS exception = val_rc ... ).
```

Beispiel:

```
DATA number TYPE i.
...
lcl_employee=>get_n_o_employees( IMPORTING ex_count = number ).
```

Abbildung 46: Aufrufsyntax statischer Methoden

Erfolgt ein Aufruf einer Methode aus einer anderen Methode der Klasse [des Objekts], können bei statischen Methoden der Klassename und der Selektor, bei Instanzmethoden der Name einer Referenz und der Selektor weggelassen werden:

Aufruf einer Instanzmethode außerhalb der Klasse:

```
r_employee1->set_name( EXPORTING im_name = name ).
```

Aufruf derselben Methode innerhalb einer anderen Instanzmethode der Klasse:

```
set_name( EXPORTING im_name = name ).
```

Aufruf einer statischen Methode außerhalb der Klasse:

```
zcl_employee=>get_n_o_employees( IMPORTING ex_count = count ).
```

Aufruf derselben Methode innerhalb einer anderen statischen Methode der Klasse:

```
get_n_o_employees( IMPORTING ex_count = count ).
```

Eine weitere syntaktische Vereinfachung wurde oben bereits angekündigt. Dabei geht es um sogenannte **funktionale Methoden**. Diese zeichnen sich dadurch aus, dass Sie genau einen RETURNING-Parameter und ansonsten IMPORTING-Parameter und Ausnahmen besitzen. Methoden mit RETURNING-Parameter dürfen keine EXPORTING- oder CHANGING-Parameter besitzen.

Aufrufe funktionaler Methoden können in vielen Ausdrücken, ähnlich wie in anderen Sprachen (z. B. Java), direkt an die Stelle gesetzt werden, an denen ihr Rückgabewert ausgewertet werden soll. Im einfachsten Fall handelt es sich um eine Zuweisung dieses Wertes zu einer Variablen per MOVE oder =:

```
result = ref->func_method_name( ... ).
result = class_name=>func_method_name( ... ).
```

Weitere Ausdrücke, die die Verwendung erlauben, sind:

- Logische Ausdrücke (IF, ELSEIF, WHILE, CHECK, WAIT)
- Fallunterscheidungen (CASE, WHEN)
- Arithmetische und Bit-Ausdrücke (COMPUTE)
- Suchklauseln für interne Tabellen (LOOP AT ... WHERE)

```
CLASS lcl_employee DEFINITION.  
  PUBLIC SECTION.  
    METHODS get_name  
      RETURNING value(re_name) TYPE string.  
    ...  
  ENDCLASS.
```

```
DATA: r_employee1 TYPE REF TO lcl_employee,  
      name TYPE string.  
    ...  
  name = r_employee1->get_name( ).
```

Abbildung 47: Beispiel für einen funktionalen Methodenaufruf

Die obige Abbildung zeigt ein Beispiel für einen Aufruf einer funktionalen Methode mit der verkürzten Schreibweise. Beachten Sie, dass die Parameterliste leer ist, aber trotzdem dargestellt wird und ein Leerzeichen enthält.

Eine weitere mögliche Vereinfachung betrifft die Parameterliste:

```
CLASS lcl_employee DEFINITION.  
  PUBLIC SECTION.  
    METHODS set_names  
      IMPORTING im_name TYPE string  
        im_firstname TYPE string.  
    ...  
  ENDCLASS.
```

```
DATA: r_employee1 TYPE REF TO lcl_employee,  
      name TYPE string,  
      firstname TYPE string.  
    ...  
  r_employee1->set_names( im_name = name im_firstname = firstname ).
```

Abbildung 48: Kurzform der Parameterliste

Die hier gezeigte Übergabe ist eine Kurzform für folgenden Aufruf:

```
CALL METHOD r_employee1->set_names EXPORTING im_name = name  
      im_firstname = firstname.
```

Die Methode darf für diese Syntaxvariante beliebige Eingabeparameter, aber nur optionale Ein-/Ausgabeparameter haben.

3.6.7 Konstruktoren

Beim **Konstruktor** handelt es sich um eine spezielle Instanzmethode, die bei der Erzeugung von Objekten einer Klasse aufgerufen wird. Es gibt auch statische Konstruktoren (dazu später mehr); mit dem Begriff Konstruktor ist aber normalerweise der hier beschriebene Instanzkonstruktor gemeint. Ein typischer Einsatzzweck ist das Belegen der Attribute des Objekts. Würde diese nicht durch den Konstruktor gesetzt, hätten Sie nach Erzeugung einen durch den Typ festgelegten Initialwert, es sei denn es wird mit dem VALUE-Zusatz der DATA-Anweisung ein konkreter Initialwert festgelegt.

Der Konstruktor hat den speziellen, vorgeschriebenen Namen `constructor`. Durch Vergabe dieses Namens wird eine Methode automatisch zum Konstruktor. Der Konstruktor darf lediglich `IMPORTING`-Parameter und Ausnahmen besitzen.
Der Aufruf des Konstruktors wird automatisch mit der `CREATE OBJECT`-Anweisung ausgelöst.

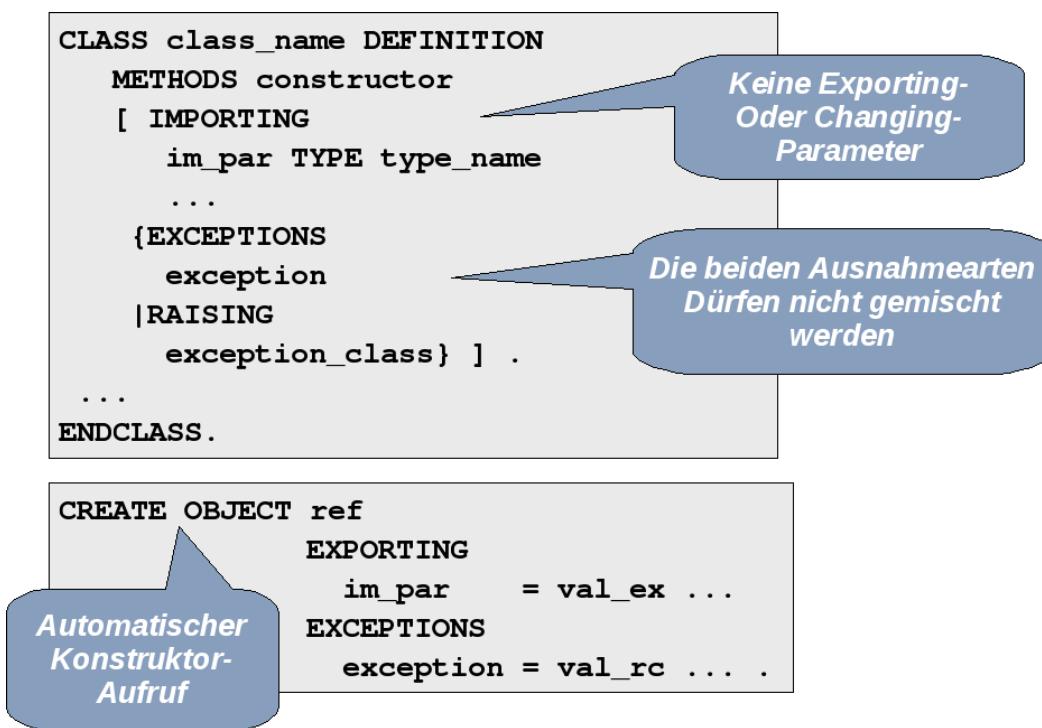


Abbildung 49: Syntax für den Instanzkonstruktor

Bei Konstruktoren in ABAP sind einige Dinge zu beachten, die zum Teil einen Unterschied zu anderen Programmiersprachen darstellen.

- Jede Klasse hat **maximal einen** Instanzkonstruktor
- Der Instanzkonstruktor darf im Normalfall (es gibt Ausnahmen) nur im öffentlichen Bereich der Klasse definiert werden. Es gibt keinen Destruktor, den Sie evtl. aus anderen Programmiersprachen kennen
- Der Konstruktor wird implizit durch `CREATE OBJECT` aufgerufen. Ein expliziter Aufruf ist nur in einem Spezialfall im Kontext der Vererbung möglich, dies wird später noch genauer vorgestellt.

Wenn Sie im Konstruktor Ausnahmen definieren, sollten Sie beachten, dass beim Eintreten einer dieser Ausnahmen das Objekt nicht erzeugt wird.

Im folgenden Beispiel wird ein Konstruktor verwendet, um direkt bei der Erzeugung von Objekten Attributwerte übergeben zu können:

```

CLASS lcl_employee DEFINITION.
  PUBLIC SECTION.
    METHODS constructor IMPORTING im_name TYPE string
              im_firstname TYPE string.

    ...

  PRIVATE SECTION.
    DATA: name TYPE string,
          firstname TYPE string.

    ...

ENDCLASS.

CLASS lcl_employee IMPLEMENTATION.
  METHOD constructor.
    name = im_name.
    firstname = im_firstname.
  ENDMETHOD.

ENDCLASS.

DATA r_employee1 TYPE REF TO lcl_employee.
...
CREATE OBJECT r_employee1
  EXPORTING im_name = 'Schmitt'
            im_firstname = 'Achim'.

```

Abbildung 50: Beispiel für einen Instanzkonstruktor

Neben solchen, über die VALUE-Angabe der DATA-Anweisung hinausgehenden Initialisierungen werden Instanzkonstruktoren typischerweise verwendet um Ressourcen zu allozieren, statische Attribute zu modifizieren oder die Information der Objekterzeugung als Nachricht zu verschicken.

Wie bereits oben angekündigt, gibt es neben gewöhnlichen Instanzkonstruktoren in ABAP auch statische Konstruktoren (Klassenkonstruktoren). Diese Unterliegen noch stärkeren Restriktionen: Sie dürfen überhaupt keine Parameter oder Ausnahmen besitzen. Es ließe sich auch schwer eine Zuordnung der Formalparameter zu Aktualparametern durchführen, da es keine Bestimmte Anweisung gibt, die den Klassenkonstruktor explizit auslöst. Vielmehr wird er **automatisch ausgeführt**, bevor erstmalig auf die Klasse zugegriffen wird. Dieser erstmalige Zugriff wird als das erstmalige eintreten einer der folgenden Aktionen definiert:

- Instanziierung der Klasse (CREATE OBJECT)
- Zugriff auf eine statische Komponente der Klasse
- Registrierung einer Ereignisbehandlermethode für ein Ereignis der Klasse (dazu später mehr)

```

CLASS lcl_employee DEFINITION.
  PUBLIC SECTION.
    CLASS-METHODS class_constructor.
    ...
  PRIVATE SECTION.
  ...
ENDCLASS.

CLASS lcl_employee IMPLEMENTATION.
  METHOD class_constructor.
  ...
ENDMETHOD.

...
ENDCLASS.

```

Falls erster Zugriff auf die Klasse: Zuvor Ausführung des statischen Konstruktors!

```
CREATE OBJECT r_employee ... .
```

Abbildung 51: Beispiel für einen statischen Konstruktor

Die Abbildung zeigt ein Beispiel zur Definition eines statischen Konstruktors. Wenn die am Ende dargestellte Anweisung – es sei angenommen dass r_employee eine Referenzvariable ist, die auf lcl_employee typisiert ist – der erste Zugriff auf die Klasse ist, wird vor Ausführung der Anweisung der statische Konstruktor ausgeführt.

Der erste Aufruf muss nicht zwingend ein Instanzkontruktoraufruf sein. Es ist auch möglich, dass der statische Konstruktor durch den Zugriff auf eine statische Komponente der Klasse ausgelöst wird. Die Tatsache dass der Aufruf implizit mit einer anderen Anweisung verbunden ist, macht klar, dass es keine Parameter für den statischen Konstruktor geben kann.

3.6.8 Hinweise zur Namensgebung

Die Frage nach der Namensgebung von Klassen, Attributen, Methoden usw. spielt eine wichtige Rolle bei der Entwicklung. Eine gute Namensgebung erhöht die Lesbarkeit Ihres Quellcodes für Sie selbst, und ebenso für andere Mitglieder eines Teams, wenn Sie später an einem größeren Entwicklungsprojekt beteiligt sind. Es gibt keine verbindliche Vorgabe dafür, welche Namenskonventionen in ABAP-Programmen zu verwenden sind. In den bisher dargestellten Codefragmenten wurde aber versucht, eine einheitliche Namensgebung zu verwenden, indem zum einen konsequent englische Namen vergeben wurden (im Hinblick auf internationale Entwicklungsumfelder), und jeweils Präfixe verwendet wurden, um die Namen zusätzlich zu beschreiben:

- lcl_ = local class (Lokale Klasse, später lernen Sie auch globale Klassen kennen)
- im_ / ex_ / ch_ / re_ = Import- / Export- / Changing- / Returning-Parameter
- r_ = Referenz
- get_ = Getter (Methode zum Lesen von Attributen)
- set_ = Setter (Methode zum Schreiben von Attributen)

Weitere, ähnliche Präfixe kennen Sie bereits aus dem Grundlagen-Kurs, etwa

- wa_ = workarea (Tabellenarbeitsbereich)
- it_ = interne Tabelle
- pa_ = Parameter
- so_ = select option (Selektionskriterium)

Wenn Sie in einem ABAP-Projekt arbeiten, empfiehlt es sich zunächst die dort geltenden Namenskonventionen zu erfragen. Dies vereinfacht die Zusammenarbeit und erspart es Ihnen in gewissem Maße, bei jedem zu vergebenden Namen über eine gute Benennung nachzudenken.

3.6.9 Kontrollfragen

1. Die Angabe, dass eine Klasse über eine bestimmte Methode verfügt, erfolgt im...
 - a. Definitionsteil der Klasse
 - b. Implementierungsteil der Klasse
2. Wo werden die Attribute einer Klasse festgelegt?
 - a. Im Definitionsteil
 - b. Im Implementierungsteil
3. Wie wird in UML eine öffentliche Komponente gekennzeichnet?
 - a. Durch ein vorangestelltes Minuszeichen
 - b. Durch ein vorangestelltes Pluszeichen
 - c. Durch Unterstrichen
 - d. Durch Einordnen in den unteren Block
4. Kann man in einer statischen Methode ein nicht-statisches Attribut der Klasse verwenden?
 - a. Ja
 - b. Nein
5. Wie wird die Implementierung einer Methode eingeleitet?
 - a. Mit METHOD
 - b. Mit METHODS

Die Antworten finden Sie weiter unten:

3.6.10 Lösungen

1. a (im Implementierungsteil folgt dann die Implementierung der Methode)
2. a
3. b
4. b
5. a (b ist das Schlüsselwort für den Definitionsteil)

3.7 Praxis: Ein objektorientiertes Programm

Bevor Sie mit den tieferen Details der Objektorientierung konfrontiert werden, soll an dieser Stelle zunächst das bisher kennen gelernte praktisch angewendet werden.

Es wird an dieser Stelle vorausgesetzt, dass Sie bereits erfolgreich Ihre IP freigeschaltet haben, sich mit dem SAP-System verbunden haben und sich dort eingeloggt haben. Sollte dies nicht der Fall sein, blättern Sie bitte zurück zur entsprechenden Stelle in den Kursunterlagen.

Vorsicht: Bitte achten Sie darauf, nicht versehentlich die Login-Daten des Grundlagenkurses zu verwenden. Für diesen Kurs gibt es eigene Mandanten / Logins / User-Nummern!

3.7.1 Anlegen von Paket und Transportauftrag

Für Ihre Entwicklungsobjekte werden Sie auch in diesem Kurs ein Paket anlegen. Bevor Sie dies tun, empfiehlt es sich den Object Navigator in die Favoriten Ihres SAP Easy Access-Menüs aufzunehmen. Klicken Sie dazu mit der rechten Maustaste auf den Menüpunkt **Favoriten**, und wählen Sie **Transaktion einfügen**.

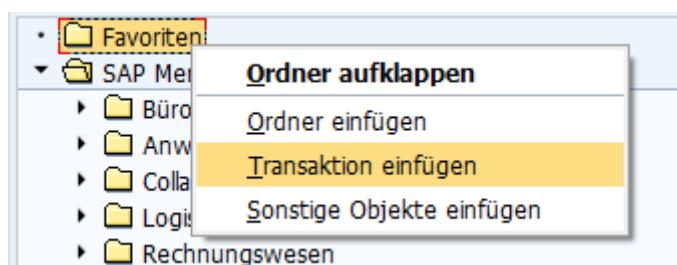


Abbildung 52: Favoriten anlegen: SAP-System-Screenshot

Geben Sie im nun erscheinenden Fenster den Transaktionscode **SE80** ein und bestätigen Sie.



Abbildung 53: Eingabe des Transaktionscodes: SAP-System-Screenshot

Daraufhin befindet sich der Object Navigator in Ihren Favoriten und kann von dort per Doppelklick gestartet werden. Fügen Sie bei Bedarf im Laufe des Kurses weitere Transaktionen hinzu.

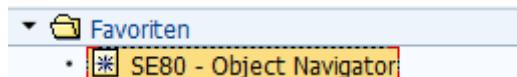


Abbildung 54: Object Navigator im Favoritenmenü: SAP-System-Screenshot

Im obigen Screenshot sehen sie auch den Transaktionscode, was in ihrer SAP GUI möglicherweise noch nicht der Fall ist. Um diese Anzeige zu aktivieren, wählen Sie **Zusätze** -> **Einstellungen** aus dem Menü und aktivieren Sie dort **Technische Namen anzeigen**:

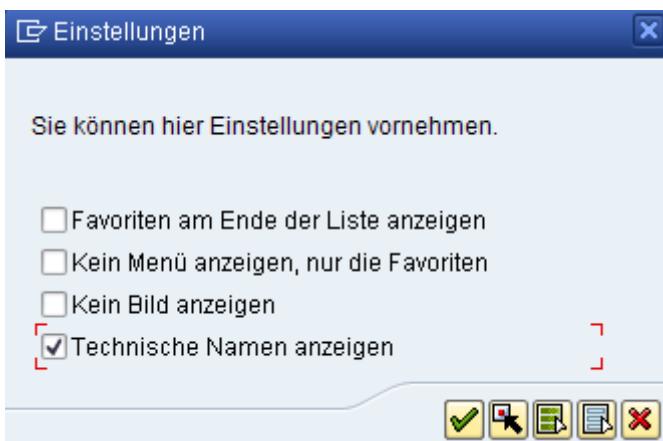


Abbildung 55: Technische Namen aktivieren: SAP-System-Screenshot

Starten Sie nun den Object Navigator durch Doppelklick auf den soeben angelegten Eintrag. Sie werden dort nun ein Paket und einen Transportauftrag anlegen. Die Schritte sollten Ihnen bereits bekannt sein, werden hier aber im Interesse eines reibungslosen Einstieg in diesen Kurs genau dargestellt.

Geben Sie im Eingabefeld unterhalb von **Paket** den Namen Ihres Pakets ein. Dieser wird nach dem Schema **ZZ_####** gebildet. **Achtung!** An allen Stellen des Kurses, an denen Sie die vier Rauten (####) sehen, müssen Sie Ihre Usernummer einsetzen. Diese entspricht den 4 Ziffern aus Ihrem Username. Beispiel: Wenn Ihr Username USER1-234 lautet, ersetzen Sie die vier Rauten durch 1234, Ihr Paket heißt dann also ZZ_1234. **Verwenden Sie niemals eine andere Nummer als Ihre Usernummer. Übernehmen Sie nicht die Nummern aus den Screenshots.** Diese Vorgabe ist unbedingt einzuhalten.

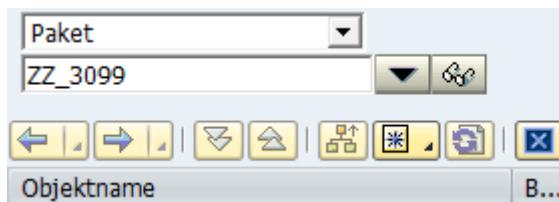


Abbildung 56: Eingabe des Paketnamens (Abbildung ist ein Beispiel - Sie verwenden Ihre eigene Nummer) : SAP-System-Screenshot

Bestätigen Sie die Nachfrage, ob das Objekt angelegt werden soll.

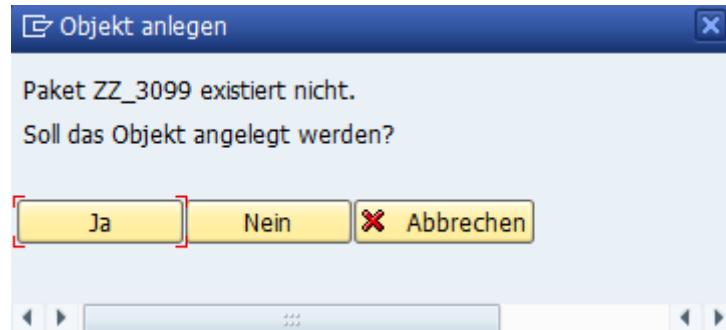


Abbildung 57: Nachfrage des SAP-Systems: SAP-System-Screenshot

Geben Sie im daraufhin erscheinenden Fenster als Kurzbeschreibung für Ihr Paket „Paket von Max Mustermann“ ein, wobei Sie „Max Mustermann“ durch Ihren Vor- und Nachnamen ersetzen. Lassen Sie die anderen Felder unverändert.



Abbildung 58: Angabe einer Kurzbeschreibung: SAP-System-Screenshot

Sie werden daraufhin nach einem Transportauftrag gefragt:

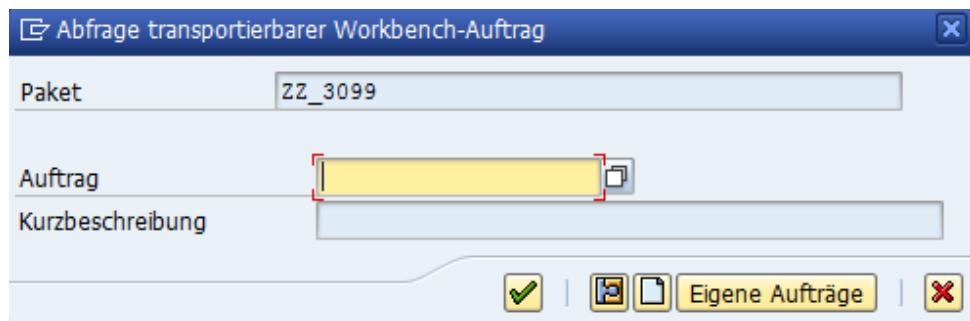


Abbildung 59: Abfrage des Transportauftrags: SAP-System-Screenshot

Um einen neuen Auftrag anzulegen, klicken Sie auf (**Auftrag Anlegen**) oder drücken Sie F8.

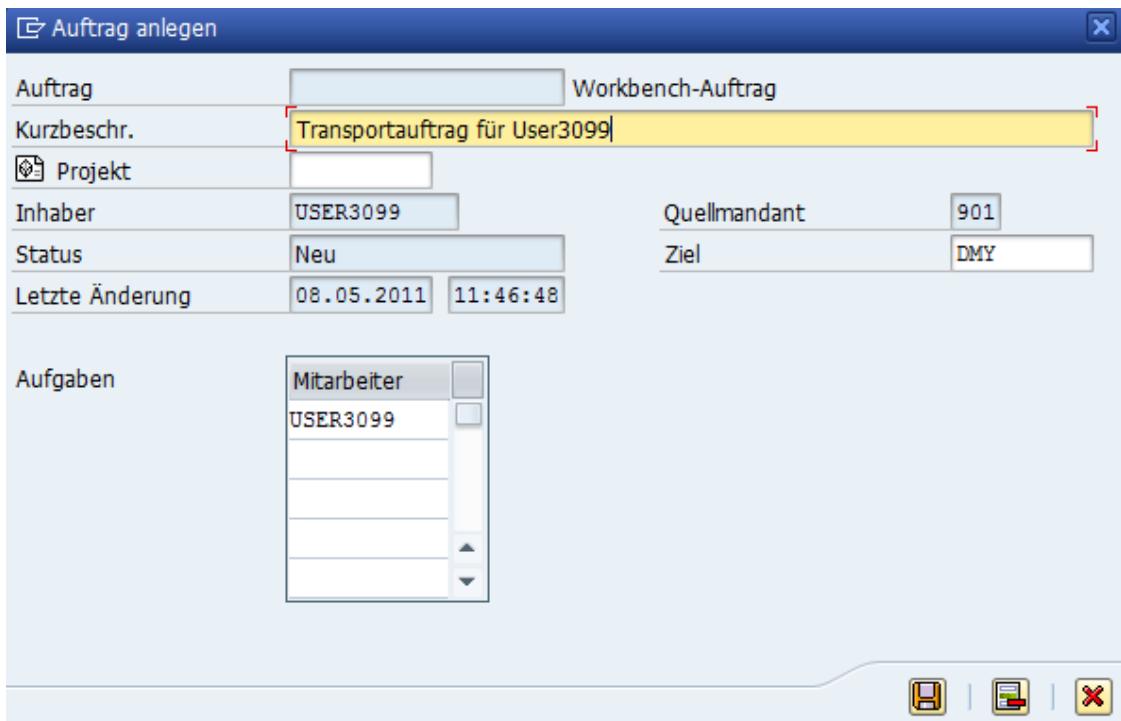


Abbildung 60: Transportauftrag anlegen: SAP-System-Screenshot

Geben Sie für den Transportauftrag einen passenden Namen ein und sichern Sie ihn durch Klick auf (Sichern) oder Betätigen der Enter-Taste. Bestätigen Sie dann auch das Fenster zur Abfrage des Transportauftrags, indem Sie dort auf (Weiter) klicken oder ebenfalls die Enter-Taste verwenden.

Sie haben nun ein Paket und einen Transportauftrag für Ihre Entwicklungsobjekte angelegt.

3.7.2 Erstellen des Programms

Legen Sie in Ihrem Paket ein neues Programm an (Rechtsklick auf das Paket unter **Objektname**, Anlegen -> Programm), und nennen Sie dieses **ZZ_####_COMPANY**. Das Programm soll kein TOP-Include besitzen.



Abbildung 61: Anlegen des Programms: SAP-System-Screenshot

Bestätigen Sie mit (Weiter) oder der Enter-Taste.

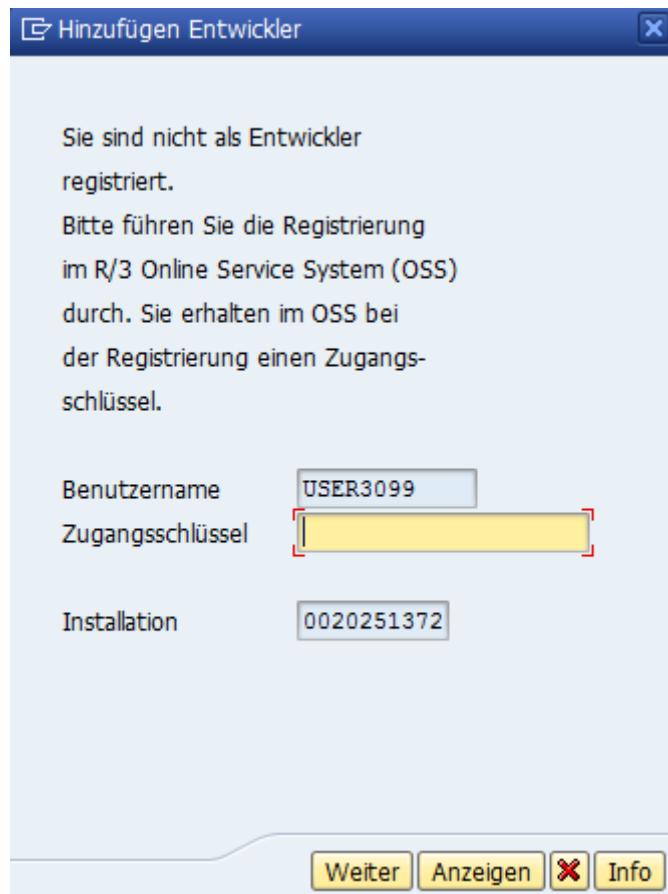


Abbildung 62: Eingabe des Entwicklerschlüssels: SAP-System-Screenshot

Sie werden daraufhin nach Ihrem Entwicklerschlüssel gefragt. Diesen haben Sie zu Beginn des Kurses per E-Mail erhalten. Geben Sie ihn nun ein und bestätigen Sie.

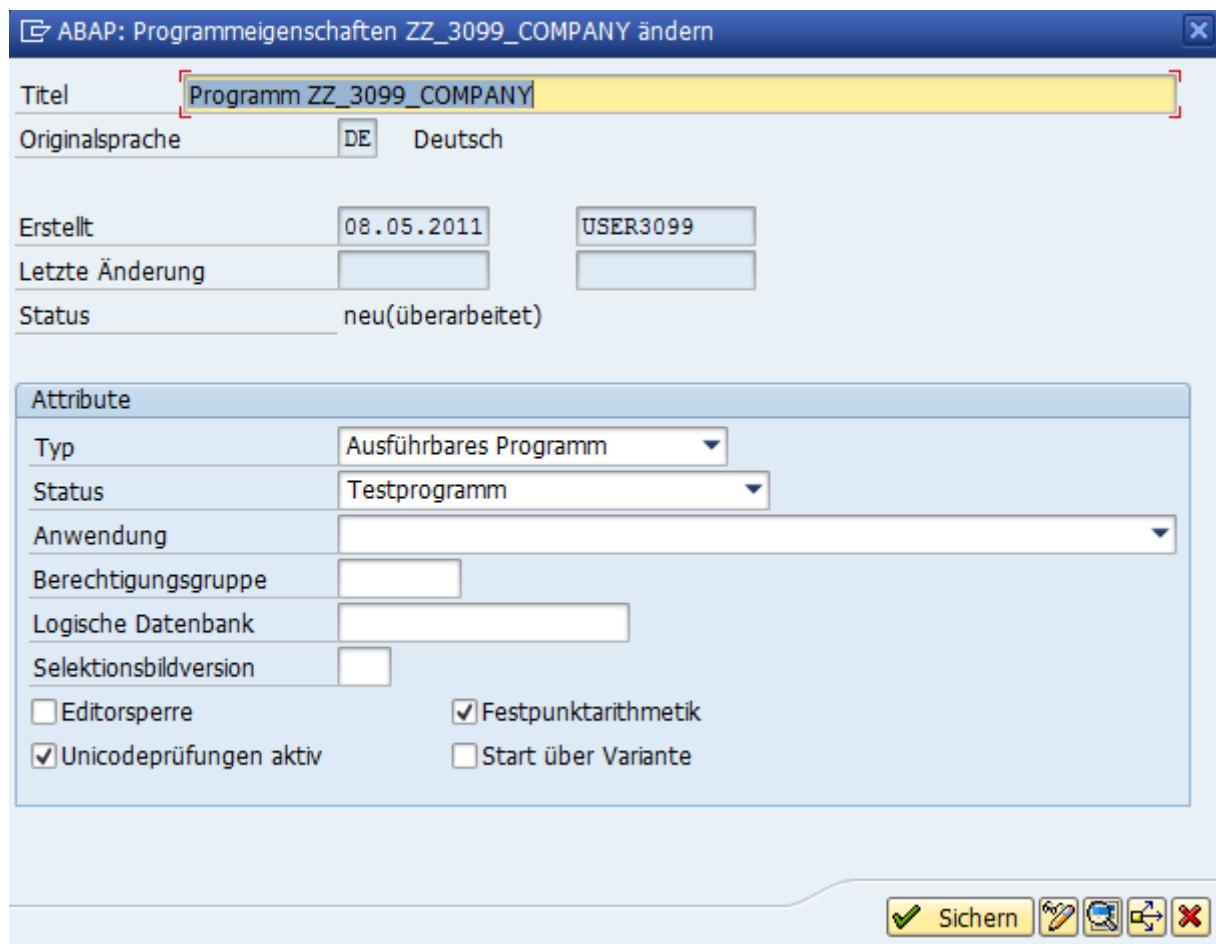


Abbildung 63: Eigenschaften des Programms: SAP-System-Screenshot

Wählen Sie als Typ des Programms **Ausführbares Programm** und als Status

Testprogramm. Sichern Sie das Programm durch Klick auf **Sichern** oder die Entertaste, und bestätigen Sie auch die beiden folgenden Fenster. Ihr Programm ist nun angelegt.

Legen Sie als nächstes ein Include für Ihr Programm an, indem Sie im Navigationsbaum auf der linken Seite des Object Navigators zu Ihrem Programm navigieren, es mit der rechten Maustaste anklicken, und **Anlegen -> Include** wählen. Bestätigen Sie die Nachfrage, ob Ihr Programm gesichert werden soll, mit **Ja**. Geben Sie als Namen des Includes **ZZ_####_COMPANY_CLASSES** an.



Abbildung 64: Anlegen des Includes: SAP-System-Screenshot

Wählen Sie im darauffolgenden Fenster erneut den Status **Testprogramm** aus und sichern Sie. Bestätigen Sie wie gewohnt auch die beiden folgenden Fenster. Danach informiert Sie das SAP-System darüber, dass eine Include-Anweisung in Ihr Programm eingefügt wurde:



Abbildung 65: Meldung des SAP-Systems: SAP-System-Screenshot

Sichern Sie Ihr Include. Das Include soll im Folgenden dazu dienen, die Klassen in dieser Aufgabe aufzunehmen (sowohl den Definitions- als auch den Implementierungsteil). Im Hauptprogramm soll hingegen ein herkömmlicher Report programmiert werden, der die im Include definierten Klassen verwendet. Vom Hauptprogramm können Sie – außer über den Navigationsbaum des Object Navigators – auch über die Include-Anweisung zum Include navigieren. Klicken Sie dazu einfach in der Zeile INCLUDE ZZ_####_COMPANY_CLASSES. auf ZZ_####_COMPANY_CLASSES. Diese Art der Navigation (Vorwärtsnavigation) funktioniert auch an anderen Stellen, wenn Sie etwa zu einer Variable oder einem Typ die zugehörige Definition finden wollen.

3.7.3 Die Klasse lcl_employee

Die erste Klasse die Sie implementieren werden, ist eine Klasse für Mitarbeiter. Die Folgende Abbildung zeigt die Klasse in UML. Lassen Sie sich vom scheinbaren Umfang der Klasse nicht abschrecken. Die einzelnen Komponenten sind nicht all zu aufwändig und werden Schritt für Schritt entwickelt.

Klassenname	Icl_employee
Attribute	- <u>name</u> :string - <u>firstname</u> :string - <u>address</u> :string - <u>dob</u> :d - <u>base_salary</u> :i <u>-n_o_employees</u> :i
Methoden	+constructor(im_name:string, im_firstname:string, im_base_salary:i, im_dob:d) +set_name(im_name:string) +set_firstname(im_firstname:string) +set_address(im_address:string) -set_dob(im_dob:d) +get_name():string +get_firstname():string +get_address():string +get_dob():d +get_base_salary():i +get_salary():i +get_full_name():string -init_base_salary(im_base_salary:i) +change_base_salary(im_amount:i) +print() +get_n_o_employees():i

Abbildung 66: Eine Klasse für Mitarbeiter

Erstellen Sie als erstes die Klasse selbst. Fügen Sie dazu in ihrem Include sowohl einen Definitions- als auch einen Implementierungsteil hinzu.

Ergänzen Sie dann einen Bereich für private und einen Bereich für öffentliche Komponenten, und fügen Sie alle in der Abbildung vorgegebenen Attribute hinzu. Der Instanzzähler n_o_employees soll einen Startwert von 0 erhalten. Dieser kann in der CLASS-DATA-Anweisung gesetzt werden, es wird kein statischer Konstruktor benötigt um die Vorbelegung zu realisieren. dob steht für „date of birth“, also das Geburtsdatum des Mitarbeiters.

Hinweis: Alle vorgegebenen Namen von Attributen, Methoden, Programmen und Includes müssen exakt so verwendet werden, wie sie in den Aufgaben stehen.

Damit die Attribute verwendet werden können, müssen aufgrund der Sichtbarkeitsvorgabe entsprechende Methoden geschaffen werden. Implementieren Sie daher als nächstes alle get- und set-Instanzmethoden. Die get-Methoden sollen als funktionale Methoden implementiert werden, so dass die oben beschriebenen syntaktischen Vereinfachungen erreicht werden. Die Methode get_salary soll in dieser Klasse das Gehalt zurückliefern, das sich aus dem Grundgehalt und einer altersabhängigen Zulage errechnet. Mitarbeiter über 40 Jahren erhalten 5% Zulage, Mitarbeiter über 50 Jahren 10%. Für die Berechnung des Alters können Sie Datumswerte voneinander abziehen, und erhalten als Ergebnis einen Wert in Tagen. Die Methode get_full_name soll den vollständigen Namen zurückliefern. Nutzen Sie hier den CONCATENATE-Befehl, um Strings zu verketten. Die Syntax des Befehls lautet:

```
CONCATENATE {dobj1 dobj2 ...} | {LINES OF itab}
INTO result
```

```
[IN {BYTE | CHARACTER} MODE]
[SEPARATED BY sep]
[RESPECTING blanks].
```

Hierbei werden die Datenobjekte dobj1 dobj2 usw. bzw. die Zeilen der internen Tabelle itab verkettet und das Ergebnis in result gespeichert. Die Angabe IN {BYTE | CHARACTER} MODE wird in einem späteren Kursteil behandelt. Mit SEPARATED BY sep kann ein Trennzeichen zwischen den Datenobjekten gesetzt werden. Die Angabe RESPECTING blanks erzwingt die Beachtung schließender Leerzeichen.

Zum Setzen des Grundgehalts soll der Konstruktor die Methode init_base_salary benutzen. Implementieren Sie daher auch diese Methode. In der Methode soll geprüft werden, ob das übergebene Grundgehalt positiv ist und andernfalls der Wert auf 0 gesetzt wird.

Implementieren Sie als nächstes den Konstruktor. Damit n_o_employees die Anzahl der erzeugten Mitarbeiter enthält, muss dieses Attribut vom Konstruktor für jedes erzeugte Objekt erhöht werden.

Die Methode print soll mithilfe der WRITE-Anweisung eine einfache Ausgabe von vollem Namen, Adresse, Gehalt und Alter realisieren.

Hinweis: Sie werden an einigen Stellen Hilfsvariablen benötigen, bspw. um das berechnete Alter zwischenzuspeichern. Definieren Sie diese Hilfsvariablen immer lokal in der Methode. Kommen sie nicht auf die Idee, diese als Attribute auf Ebene der Klasse zu definieren. Dies wäre semantisch unsinnig und würde leicht zu Fehlern führen: Wenn es ein Attribut für das Alter gäbe, würde dessen Wert bei einer Änderung des Geburtsdatums inkonsistent. Daher sollte das Alter immer aktuell aus dem Geburtsdatum berechnet werden.

Sichern Sie dann und wechseln Sie vom Include zu ihrem Hauptprogramm. Ergänzen Sie unterhalb der Include-Anweisung die Anweisung START-OF-SELECTION. Erstellen Sie in diesem Block ein Objekt der Klasse, indem Sie eine Referenzvariable r_employee definieren und ein Objekt erzeugen. Die Adresse müssen Sie dabei über die set-Methode festlegen, während die anderen Attribute vom Konstruktor gesetzt werden. Geben Sie den Mitarbeiter anschließend mit der print-Methode aus. Speichern , prüfen  und aktivieren  Sie Programm und Include.

Testen Sie das Programm anschließend, und kehren Sie zum Quelltext zurück. Setzen Sie mit  einen Breakpoint in die Zeile, in der das Objekt erzeugt wird. Stellen Sie sicher, dass der neue Debugger aktiviert ist (**Hilfsmittel -> Einstellungen -> ABAP Editor -> Debugging -> ABAP-Debugger -> Neuer Debugger**) und testen Sie das Programm. Sie gelangen in den Debugger. Doppelklicken Sie auf r_employee im Quelltext, woraufhin die Variable auf der rechten Seite erscheint.

Variablen 1	Variablen 2	Locals	Globals
S... Variable	W.. Wert	Ä... Hexadezimaler Wert	

R_EMPLOYEE {0:INITIAL}

Abbildung 67: Referenzvariable im Debugger: SAP-System-Screenshot

Fahren Sie nun mit Einzelschritten fort. Sie können so beobachten, wie der Konstruktor und die Methode zur Initialisierung des Gehalts durchlaufen werden. Danach ist das Objekt erstellt. Durch Doppelklick auf R_EMPLOYEE in der rechten Tabelle (siehe Abbildung oben) gelangen Sie in die Ansicht für Objekte. Dort können Sie die Attributwerte beobachten:

Attribute		Attribut	W...	Wert
St...	Bin...	Sic...		
		OBJECT		
		LCL_EMPLOYEE		
		N_O_EMPLOYEES		1
		NAME		Meier
		FIRSTNAME		Hans
		ADDRESS		Hauptstraße 10
		DOB		19701012
		BASE_SALARY		3200

Abbildung 68: Attribute im Debugger: SAP-System-Screenshot

Wenn Sie zurück zur Quelltextansicht des Debuggers wechseln möchten, wählen Sie die Registrierkarte **Desktop1** aus. Sie können sich nun die weiteren Schritte einzeln anzeigen lassen, oder den Vorgang durch **Weiter** beschleunigen. Entfernen Sie den Breakpoint anschließend wieder.

Erstellen Sie als nächstes eine interne Tabelle mit Referenzen auf Mitarbeiter (`it_employees`), und fügen Sie den erstellten Mitarbeiter in diese Tabelle ein. Erstellen Sie dann mit derselben Referenzvariablen eine weitere Instanz. Die Referenz auf die erste Instanz geht nicht verloren, da sie in der internen Tabelle festgehalten wird. Fügen Sie die zweite Instanz ebenfalls zur internen Tabelle hinzu. Geben Sie anschließend alle Mitarbeiter aus, indem Sie die interne Tabelle durchlaufen und in der Schleife die print-Methode aufrufen (anstatt des bisherigen Aufrufs außerhalb der Schleife). Speichern, prüfen, aktivieren und testen Sie Ihr Programm.

Implementieren Sie nun noch die fehlenden Methoden: Die Methode `change_base_salary` soll das Grundgehalt um den übergebenen Betrag ändern. Der übergebene Betrag darf negativ sein, nicht aber das Grundgehalt nach der Änderung. Setzen Sie es in diesem Fall auf 0. Die Methode `get_n_o_employees` soll den Instanzzählerstand zurückliefern.

Testen Sie die Methoden im Hauptprogramm, indem Sie das Grundgehalt der erzeugten Mitarbeiter vor der Ausgabe ändern, und den Instanzzählerstand nach der Objekterzeugung ausgeben.

Die Ausgabe Ihres Programms sollte nun etwa wie folgt aussehen:

Programm ZZ_3099_COMPANY				
Programm ZZ_3099_COMPANY				
Name: Hans Meier	Adresse: Hauptstraße 10	Gehalt: 3.360	Geburtsdatum: 12.10.1970	
Name: Gerd Schmitt	Adresse: Wiesenweg 1	Gehalt: 12.870	Geburtsdatum: 12.10.1960	
Name: Dirk Schulz	Adresse: Adenauerstr. 13	Gehalt: 7.700	Geburtsdatum: 12.10.1980	
Es wurden 3 Objekte erzeugt.				

Abbildung 69: Ausgabe des Programms: SAP-System-Screenshot

3.7.4 Die Klasse lcl_department

In dieser Übung sollen Sie eine Klasse für Abteilungen (lcl_department) entwickeln. Jeder Abteilung sind Mitarbeiter zugeordnet:



Abbildung 70: Abteilung und Mitarbeiter

Die Abteilungsklasse soll einen Namen und eine Tabelle mit Referenzen auf ihre Mitarbeiter beinhalten.

Erstellen Sie auch für diese Klasse den entsprechenden Definitions- und Implementierungsteil in ihrem Include. Definieren Sie dann zunächst ein Attribut `name` für den Namen der Abteilung sowie entsprechende get- und set-Methoden (analog zur Mitarbeiterklasse).

Definieren Sie dann ein privates Attribut `it_employees`, das Referenzen auf die Mitarbeiter aufnehmen soll. Fügen Sie eine Methode `add_employee` hinzu, die als Parameter die Referenz auf einen Mitarbeiter übergeben bekommt und diesen in die Tabelle `it_employees` einfügt.

Die Abteilungs-Klasse soll ebenfalls über eine `print`-Methode verfügen. Diese soll öffentlich sein und sowohl den Namen der Abteilung als auch die Mitarbeiter der Abteilung ausgeben.

Implementieren Sie die Methode und verwenden Sie zur Ausgabe der Mitarbeiter die `print`-Methode der Mitarbeiter-Klasse (durchlaufen Sie die Mitarbeiter-Tabelle genau wie sie es zuvor im Hauptprogramm getan haben). Fügen Sie schließlich noch einen Konstruktor hinzu, der den Namen der Abteilung setzt. Speichern, prüfen und aktivieren Sie Ihr Include und wechseln Sie wieder zum Hauptprogramm.

Entfernen Sie dort die interne Tabelle, und erzeugen Sie stattdessen mit einer Referenz `r_department` eine Abteilung, der Sie die Mitarbeiter über die entsprechende Methode hinzufügen. Ersetzen Sie die Ausgabe der Mitarbeiter durch einen Aufruf der `print`-Methode des Abteilungsobjekts.

Die Ausgabe des Programms könnte nun wie folgt aussehen:

Programm ZZ_3099_COMPANY							
<hr/>							
Programm ZZ_3099_COMPANY							
Abteilung:	Einkauf						
Name:	Hans Meier	Adresse:	Hauptstr. 19	Gehalt:	3.360	Geburtsdatum:	12.10.1970
Name:	Gerd Schmitt	Adresse:	Lindenplatz 7	Gehalt:	12.285	Geburtsdatum:	12.10.1965
Name:	UweSchulz	Adresse:	Kalorastr. 6	Gehalt:	7.700	Geburtsdatum:	12.10.1980
Es wurden		3 Objekte erzeugt.					

Abbildung 71: Ausgabe des Programms mit Abteilungsklasse: SAP-System-Screenshot

Ergänzen Sie nun eine Methode `avg_salary` zur Berechnung des Durchschnittsgehalts der Mitarbeiter einer Abteilung. Entscheiden Sie selbständig, in welche Klasse diese Methode gehört, und ob sie statisch oder nicht-statisch sein muss. Rufen Sie die Methode im Hauptprogramm auf und geben Sie das Ergebnis aus.

Tipp: Sie dürfen hier nicht die statische Methode `get_n_o_employees` benutzen. Diese Methode würde die Anzahl **aller** Mitarbeiter liefern – hier geht es aber um die Anzahl der Mitarbeiter der betrachteten Abteilung.

4 Spezielle Objektorientierte Konzepte in ABAP

Sie haben nun bereits einige grundlegende Konzepte der Objektorientierung in ABAP umsetzen gelernt. In diesem Teil werden nun weitere Konzepte vorgestellt. Im Wesentlichen sind dies die Vererbung, Polymorphie, Interfaces und Ereignisse.

4.7 Vererbung

Das Konzept der Vererbung ist Ihnen bereits im Abschnitt 3.5.1.2 (Generalisierungs- / Spezialisierungsbeziehungen) begegnet. Die Vererbung stellt eine Generalisierungs- bzw. Spezialisierungsbeziehung dar. Zunächst bedeutet dies, dass in den abgeleiteten Klassen, also den erbenden Klassen, speziellere Attribute und Methoden hinzugefügt werden. Die in der Oberklasse (auch „Superklasse“ genannt) definierten Attribute und Methoden stehen auch in den Unterklassen zur Verfügung, wobei es durch die Sichtbarkeit zu Einschränkungen kommen kann.

Die gezeigten Beispiele sollen nicht zu der Annahme führen, es gebe immer genau zwei Unterklassen. Eine Klasse kann beliebig viele Unterklassen besitzen, jedoch stets nur eine Oberklasse (Das bilden von Unterklassen kann allerdings auch unterbunden werden, dazu mehr in einem späteren Abschnitt des Kurses). Es gibt andere Programmiersprachen, in denen letztere Restriktion nicht gilt, so dass Klassensysteme mit „Mehrfachvererbung“ möglich sind, in denen also Klassen vorkommen, die von mehreren Klassen erben. In ABAP Objects können in solchen Anwendungsfällen Interfaces helfen, die Ihnen im Laufe des Kurses noch vorgestellt werden. Eine Klasse kann zugleich die Rolle einer Ober- und die einer Unterklasse einnehmen, wenn Sie von einer Oberklasse abgeleitet ist, aber zugleich von Ihr weitere Klassen abgeleitet wurden. So können mehrstufige Systeme realisiert werden. Die folgende Abbildung zeigt ein Beispiel für eine einfache Vererbung mit zwei Unterklassen:

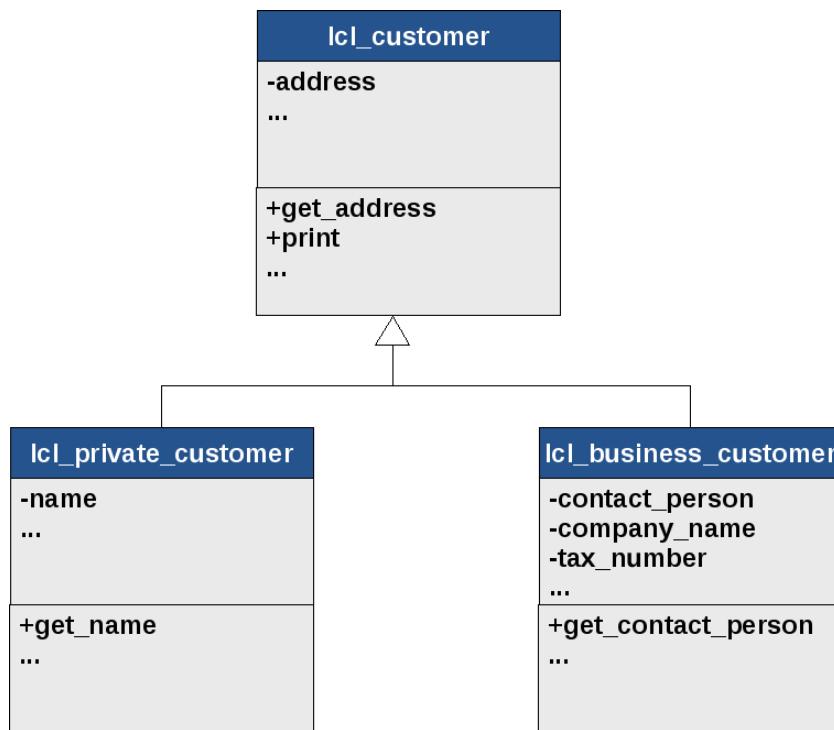


Abbildung 72: Beispiel zum Vererbungskonzept

Das Beispiel zeigt eine Klasse für Kunden (lcl_customer) und zwei von dieser Klasse abgeleitete, speziellere Klassen für Privatkunden (lcl_private_customer) bzw. Geschäftskunden (lcl_business_customer). Die gemeinsamen Komponenten beider Klassen wurden in der Oberklasse lcl_customer abgebildet: Alle Kunden verfügen über eine Adresse und können ausgegeben werden (print-Methode). Die Komponenten, die es nur für Privat- oder nur für Geschäftskunden gibt, liegen hingegen in der jeweiligen Unterklassse. Auch die in der Oberklasse definierten Attribute und Methoden stehen (bei entsprechender Sichtbarkeit) in den Unterklassen zur Verfügung, auch wenn sie im Diagramm nicht dargestellt werden.

Der Vorteil dieser Abstraktion liegt darin, dass gemeinsame Komponenten der Unterklassen nur einmal erstellt werden müssen und zentral gewartet werden können. Der Nachteil liegt in einer hohen Kopplung: Die Unterklassen sind von der Oberklasse abhängig (daher wird auch die Bezeichnung **white box reuse** verwendet), werden dort Änderungen durchgeführt, wirken sich diese unmittelbar auf alle Unterklassen aus. Daher sind Änderungen an Oberklassen stets vorsichtig und vor allem **unter Beibehaltung der Semantik** durchzuführen.

Bei der Entwicklung der Oberklasse sind Bedürfnisse der Unterklassen zu berücksichtigen. Es kommt vor dass diese erst bei der Entwicklung der Unterklasse zutage treten und so nachträgliche Anpassungen der Oberklasse durch Anforderungen der Unterklasse erforderlich werden. Die Vererbung unterstützt die Wiederverwendung, gemäß dem Leitsatz „Programming by difference“ können neue Klassen durch Ableitung existierender Klassen entwickelt werden.

```
CLASS lcl_customer DEFINITION.
  PUBLIC SECTION.
    METHODS get_address RETURNING re_address TYPE string,
            set_address IMPORTING im_address TYPE string,
            print,
            ...
  PRIVATE SECTION.
    DATA: address TYPE string,
    ...
ENDCLASS.

CLASS lcl_private_customer DEFINITION INHERITING FROM lcl_customer.
  PUBLIC SECTION.
    METHODS get_name RETURNING re_name TYPE string,
            set_name IMPORTING ex_name TYPE string,
            ...
  PRIVATE SECTION.
    DATA name TYPE string,
    ...
ENDCLASS.
```

Abbildung 73: Beispiel für die Definition einer Vererbung

Um eine Vererbung zu definieren, muss im Definitionsteil der Unterklasse die einleitende CLASS-Anweisung um den Zusatz INHERITING FROM oberklasse ergänzt werden, wobei oberklasse der Name der Oberklasse ist, von der die Klasse abgeleitet werden soll.

Es kann stets nur eine Oberklasse angegeben werden. Die Oberklasse wird syntaktisch nicht verändert. Sie kennt die Unterklassen nicht, was das beachten von Anforderungen der Unterklassen umso schwieriger machen kann.

Es ist nicht möglich, bei der Definition einer Unterklassie Komponenten der Oberklasse zu entfernen. Über die Möglichkeit des Hinzufügens von Komponenten hinaus gibt es jedoch die Möglichkeit der **Redefinition** von Methoden. Dabei wird die Implementierung einer Methode durch eine neue Implementierung für die Unterklassie ersetzt.

Im hier verwendeten Beispiel-Klassensystem böte sich die Methode `print` für eine Redefinition an. Die Methode soll die Attributwerte des jeweiligen Objekts ausgeben. Da in den Unterklassen spezifische Attribute hinzu kommen, sollten diese beim Aufruf der Methode für ein Objekt einer der Unterklassen auch diese mit ausgeben.

Im UML-Diagramm werden Redefinitionen auch dargestellt. Dies geschieht, indem die Methode für die Unterklassie erneut aufgeführt wird. Geerbte, nicht redefinierte Methoden werden hingegen nicht erneut in der Unterklassie dargestellt. Die folgende Abbildung zeigt das zuvor verwendete Diagramm ergänzt um die Redefinitionen:

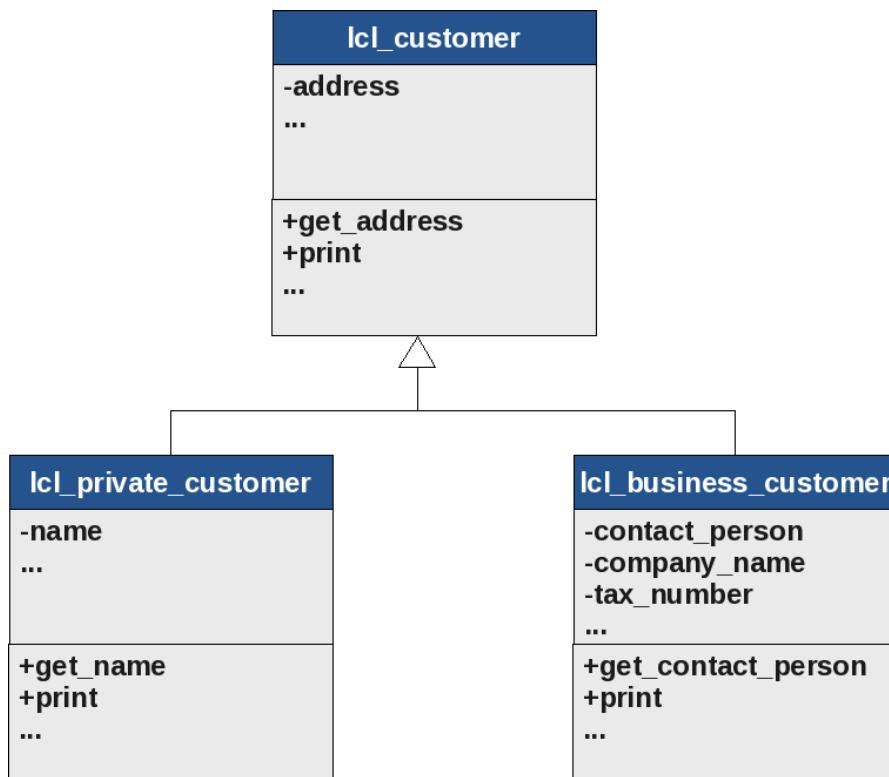


Abbildung 74: Vererbung mit Redefinition einer Methode

Die Redefinition einer Methode in einer Unterklassie erfordert, dass die Signatur der Methode beibehalten wird. Es muss in der Unterklassie daher lediglich im Definitionsteil der Klasse in der METHODS-Anweisung der Methodenname mit dem Zusatz REDEFINITION angegeben werden. Eine erneute Angabe der Parameter und Ausnahmen der Methode in der Unterklassie ist nicht notwendig, da die Signatur ohnehin mit der der Oberklassie übereinstimmen muss, und daher auch syntaktisch nicht zulässig (eine gewisse Sonderrolle nimmt der Konstruktor ein, dazu unten mehr). An der Oberklassie sind keine Veränderungen erforderlich.

```
CLASS lcl_customer DEFINITION.  
  PUBLIC SECTION.  
    METHODS ...,  
          print,  
          ...  
  ENDCLASS.
```

```
CLASS lcl_private_customer DEFINITION INHERITING FROM lcl_customer.  
  PUBLIC SECTION.  
    METHODS print REDEFINITION,  
          ...  
  ENDCLASS.
```

*Etwaige Parameter /
Ausnahmen nicht
erneut angeben!*

Abbildung 75: Beispiel zur Redefinition von Methoden

Im oben abgebildeten Beispiel wurde die Methode `print` redefiniert. Die Signatur enthält hier ohnehin keine Parameter oder Ausnahmen. Bei der Redefinition muss auch der Sichtbarkeitsbereich stets übereinstimmen, d. h. die Redefinition muss im gleichen Sichtbarkeitsbereich wie die ursprüngliche Definition geschehen. Da private Komponenten außerhalb ihrer Klasse nicht sichtbar sind, ist in diesem Bereich folglich auch keine Redefinition möglich.

Es ist zu beachten, dass die Semantik einer Methode bei der Redefinition erhalten bleibt. Es könnte sonst etwa im Zuge der Polymorphie zu unerwarteten Effekten führen. Dort wird zwar erwartet, dass zwei Objekte auf eine Nachricht unterschiedlich reagieren, die Bedeutung der Nachricht sollte aber gleich sein. Im vorliegenden Beispiel soll die Methode `print` in der Unterklasse zwar andere Ausgaben liefern, aber es bleibt weiterhin eine Methode zur Ausgabe des Objekts.

Bei der Redefinition kann es nötig oder sinnvoll sein, die bereits in der Oberklasse vorhandene Implementierung der Methode wiederzuverwenden. Das kann der Fall sein wenn etwa ein von der Oberklassmethode berechnetes Ergebnis in der Unterklasse nur leicht verändert werden muss, oder wenn die Oberklasse eine Aufgabe erfüllt, die in der Unterklasse lediglich um eine Zusatzfunktionalität ergänzt werden muss. In diesen Fällen kann die entsprechende Oberklassenmethode durch voranstellen von `super->` erreicht werden. Dieses Präfix lässt sie Oberklassenkomponenten erreichen. Bei der Implementierung des obigen Beispiels könnte dies etwa so aussehen:

```
CLASS lcl_private_customer IMPLEMENTATION.  
  METHOD print.  
    super->print.  
    ...  
  ENDMETHOD  
ENDCLASS.
```

*Aufruf der Methode
der Oberklasse*

Abbildung 76: Zugriff auf Oberklassenkomponenten

Eine Ausnahme bei der Redefinition stellt der Konstruktor dar. Hier wird davon ausgegangen, dass im Fall einer Redefinition, wenn also der Oberklassenkonstruktor nicht unverändert übernommen werden kann, eine Änderung der Signatur erforderlich ist. Der Konstruktor kann daher nicht redefiniert, aber **überschrieben** werden, indem in der Unterkasse ein neuer Konstruktor definiert wird. Entsprechend wird bei dieser Definition eine Signatur, nicht aber das Schlüsselwort REDEFINITION angegeben.

Innerhalb der Implementierung des Konstruktors muss zwingend ein Aufruf des Konstruktors der (direkten) Oberklasse erfolgen. So wird sichergestellt, dass ein Konstruktor auch für Objekte aller Unterklassen stets ausgeführt wird. Bei statischen Konstruktoren besteht die Problematik nicht, da das System von selbst vor dem Aufruf eines statischen Konstruktors die der Oberklassen ausführt.

```

CLASS lcl_customer DEFINITION.
  PUBLIC SECTION.
    METHODS ...,
      constructor IMPORTING im_address TYPE string,
      ...
  ENDCLASS.

CLASS lcl_private_customer DEFINITION INHERITING FROM lcl_customer.
  PUBLIC SECTION.
    METHODS ...,
      constructor IMPORTING im_address TYPE string
                    im_name TYPE string,
      ...
  ENDCLASS.

CLASS lcl_private_customer IMPLEMENTATION.
  METHOD constructor.
    CALL METHOD super->constructor( im_address = im_address ).
    name = im_name.
  ENDMETHOD.
  ...
ENDCLASS.

```

Abbildung 77: Überschreiben eines Konstruktors

Die Abbildung zeigt ein Beispiel für einen überschriebenen Konstruktor. Bei der Erzeugung eines Objekts sucht das Laufzeitsystem in der Klassenhierarchie den zuständigen Konstruktor. Hat dabei eine Klasse keinen Konstruktor, wird der Konstruktor der Oberklasse unverändert verwendet. Dies kann an einer Vererbungshierarchie mit 3 Klassen gezeigt werden:

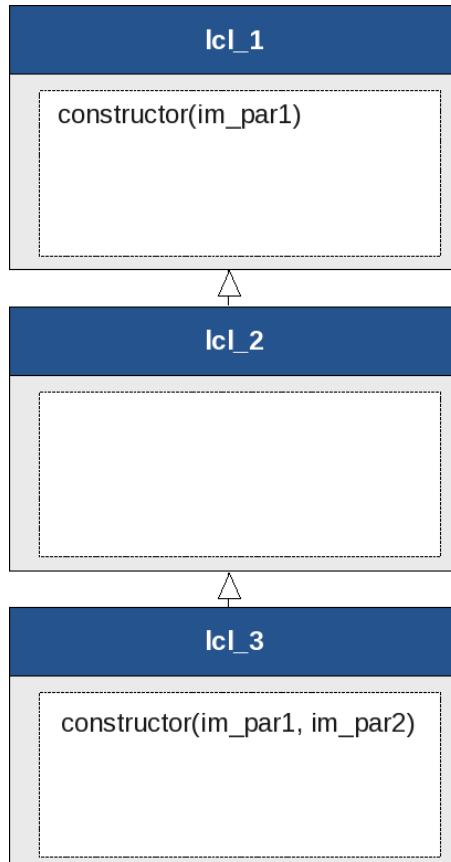


Abbildung 78: Vererbungshierarchie mit 3 Klassen

Die Klassen lcl_1 und lcl_3 besitzen dabei eine Konstruktordefinition. Wird nun ein Objekt der Klasse lcl_3 instanziert, wird dort zunächst eine Konstruktordefinition gefunden, von der aus ein Oberklassenkonstruktor aufgerufen werden muss. Die direkte Oberklasse besitzt keinen Konstruktor, also wird in der Vererbungshierarchie weiter nach oben gesprungen und der Konstruktor von lcl_1 mit dem Parameter versorgt.

In Abschnitt 3.6.1 wurde angedeutet, dass es neben dem public- und dem private-Bereich einen dritten Sichtbarkeitsbereich gibt, der erst im Zusammenhang mit der Vererbung eine Rolle spielt. Analog zu den anderen Sichtbarkeitsbereichen wird der protected-Bereich wie folgt eingeleitet:

PROTECTED SECTION.

Komponenten, die im protected-Bereich definiert werden, sind innerhalb der Klasse sowie in allen Unterklassen zugreifbar. Dies wirft die Frage auf, welchen Unterschied dies zum private-Bereich darstellt. Komponenten die im private-Bereich definiert wurden, stehen schließlich auch in der entsprechenden Unterklasse zur Verfügung. So besitzt im dargestellten Beispiel auch die Privatkunden-Klasse einen Namen. Der Unterschied liegt in der syntaktischen Ansprechbarkeit. Der oben dargestellte Konstruktor der Unterklasse könnte nicht direkt auf das in der Oberklasse definierte, private Attribut address zugreifen, sondern müsste die öffentliche get- oder set-Methode benutzen. So ist sichergestellt, dass in der Oberklasse die privaten Komponenten bedenkenlos geändert werden können, ohne dass es unmittelbare Abhängigkeiten gibt. Das Attribut könnte dort intern also auch z. B. in contact umbenannt werden, solange die öffentlichen Methoden get_address und set_address weiterhin zur Verfügung stehen. Wäre das Attribut hingegen in der Oberklasse als protected definiert worden, könnte der Konstruktor direkt darauf zugreifen und

wäre nicht mehr auf die get-/set-Methoden angewiesen. Der Nachteil bestünde dann darin, dass das Attribut in der Oberklasse nicht mehr bedenkenlos geändert werden könnte.

In UML wird die Protected-Sichtbarkeit mit einer Raute (#) visualisiert.

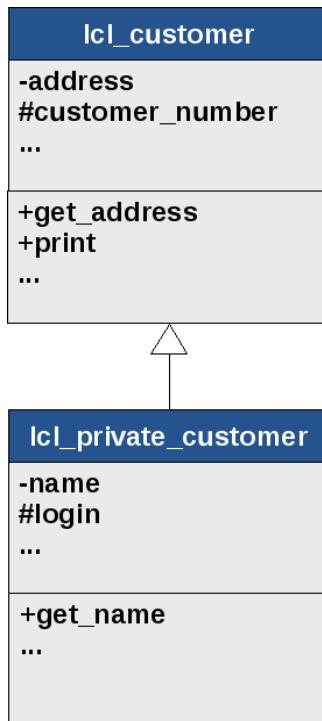


Abbildung 79: Beispiel mit Protected-Attributen

Betrachtet man nun ein Objekt der Klasse `lcl_private_customer`, ergibt sich folgendes Bild:

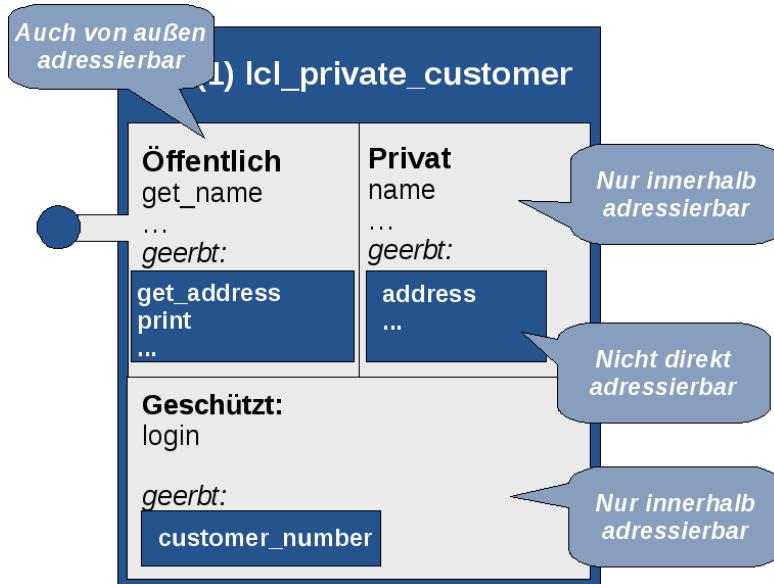


Abbildung 80: Ein Objekt und die Adressierbarkeit seiner Komponenten

Dies veranschaulicht noch einmal die Ausführungen von oben: Die private Komponente `address` ist nicht direkt adressierbar, also auch nicht aus Methoden innerhalb der Klasse; es muss die Methode `get_address` verwendet werden.

Beachten Sie, dass statische Komponenten bei der Vererbung ein besonderes Verhalten zeigen: Sie existieren nur ein einziges Mal pro Programmkontext. Eine Klasse, die ein statisches Attribut im Sichtbarkeitsbereich PUBLIC oder PROTECTED besitzt, teilt dieses daher mit allen Unterklassen. Weiterhin ist eine Redefinition von statischen Methoden nicht möglich.

4.7.1 Casting und Polymorphie

Sie wissen bereits, dass Variablen, die als Zeiger auf Objekte fungieren sollen, über TYPE REF TO typisiert werden. Solange nur eine einzige Klasse betrachtet wurde, stellte die Arbeit mit diesen Variablen keine besondere Herausforderung dar, es konnten derselben Variable verschiedene Objekte einfach zugewiesen werden und gemäß der Definition der zugehörigen Klasse wurden die öffentlichen Komponenten von außen verwendet. Mit der Einführung der Vererbung wird die Lage nun etwas komplizierter: das referenzierte Objekt muss nicht aus genau der Klasse stammen, aus der die Referenz stammt. Eine Referenz, die mit einer Oberklasse typisiert ist, kann zur Laufzeit auch auf Objekte einer Unterklassse dieser Klasse zeigen. Beispielsweise kann eine Variable, die Kunden referenziert, auch Objekte der Privatkunden-Klasse referenzieren. Jeder Privatkunde ist ein Kunde, somit kann er überall dort verwendet werden, wo ein Kunde erwartet wird. Über die Referenz, die mit der Oberklasse typisiert ist, können allerdings nur die Komponenten adressiert werden, die zum öffentlichen Teil dieser Klasse gehören (ggf. auch von einer etwaigen darüber liegenden Oberklasse bei entsprechender Sichtbarkeit geerbt wurden).

Bei der Zuweisung von Referenzvariablen zueinander kann es dazu kommen, dass die Klassen, über die die Referenzvariablen typisiert sind, nicht übereinstimmen. Wird der Inhalt einer durch eine Unterklassse typisierten Referenzvariablen einer mit der Oberklassse typisierten Referenzvariable zugewiesen, spricht man von einem **Up-Cast (Widening Cast)**, wird hingegen umgekehrt der einer durch eine Oberklassse typisierten Referenzvariablen einer mit der Unterklassse typisierten Referenzvariable zugewiesen, spricht man von einem **Down-Cast (Narrowing Cast)**. Der Up-Cast ist hierbei weniger Problematisch, da er in jedem Fall gültig ist. Beim Down-Cast kann es hingegen zu Problemen kommen, wenn das referenzierte Objekt nicht zu der Klasse passt, mit der die Referenzvariable typisiert ist, der die Referenz auf das Objekt zugewiesen werden soll. Es wird grundsätzlich gefordert, dass nach einer Zuweisung alle Komponenten, auf die syntaktisch zugegriffen werden kann, auch in der referenzierten Instanz vorhanden sind.

```
DATA: r_customer TYPE REF TO lcl_customer,
      r_pccustomer TYPE REF TO lcl_private_customer.
```

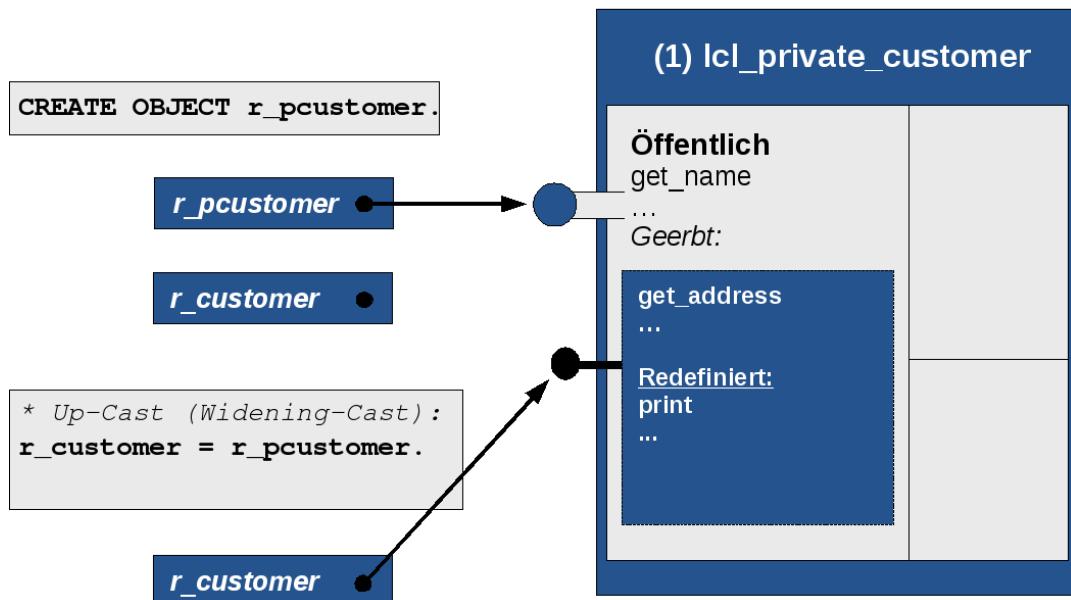


Abbildung 81: Beispiel für einen Up-Cast

Die Abbildung zeigt zunächst, wie ein Up-Cast durchgeführt wird: Über die mit der Unterklasse typisierte Referenzvariable `r_pccustomer` wird ein Objekt vom Typ der Unterklasse erzeugt. Anschließend wird die Referenz der mit der Oberklasse typisierten Referenzvariablen `r_customer` zugewiesen.

Hier passiert der Up-Cast: `r_customer` verweist nun auf ein Objekt der Klasse `lcl_private_customer`. Wie bereits zuvor erläutert, sind über die Referenzvariable `r_customer` aufgrund ihrer Typisierung nur die öffentlichen Komponenten der Oberklasse `lcl_customer` erreichbar (veranschaulicht durch das blaue Rechteck im öffentlichen Bereich der Klasse). Wenn die Klasse `lcl_private_customer` jedoch eine Methode redefiniert hat, wird beim Zugriff auf die Methode die redefinierte Version aus der Unterklasse verwendet. Das ist hier im Falle der `print`-Methode der Fall.

Um den Typ einer Referenzvariable und des referenzierten Objekts zu beschreiben, spricht man auch vom **statischen** bzw. **dynamischen** Typ der Referenzvariablen. Der statische Typ ist der in der `TYPE`-Angabe festgelegte Typ, während der dynamische Typ der Typ des zur Laufzeit referenzierten Objekts ist. Für die syntaktische Zugreifbarkeit auf eine Komponente ist der statische Typ ausschlaggebend: Über mit der Klasse `lcl_customer` typisierte Referenzvariablen kann nur auf Komponenten (inkl. geerbter Komponenten) dieser Klasse zugegriffen werden, auch wenn das tatsächlich referenzierte Objekt möglicherweise der Unterklasse `lcl_private_customer` entstammt.

Die Bezeichnung Widening Cast für den Up-Cast röhrt daher, dass die Referenzvariable der hier etwas zugewiesen wird mehr dynamische Typen annehmen kann als die Variable von der zugewiesen wird. Nicht zu verwechseln ist dies mit der Zugreifbarkeit der Komponenten: Über die Oberklassen-typisierte Referenzvariable kann auf weniger Komponenten zugegriffen werden als über die Referenzvariable mit dem statischen Typ der Unterklasse.

Up-Casts finden häufig in Szenarien wie dem Folgenden Anwendung:

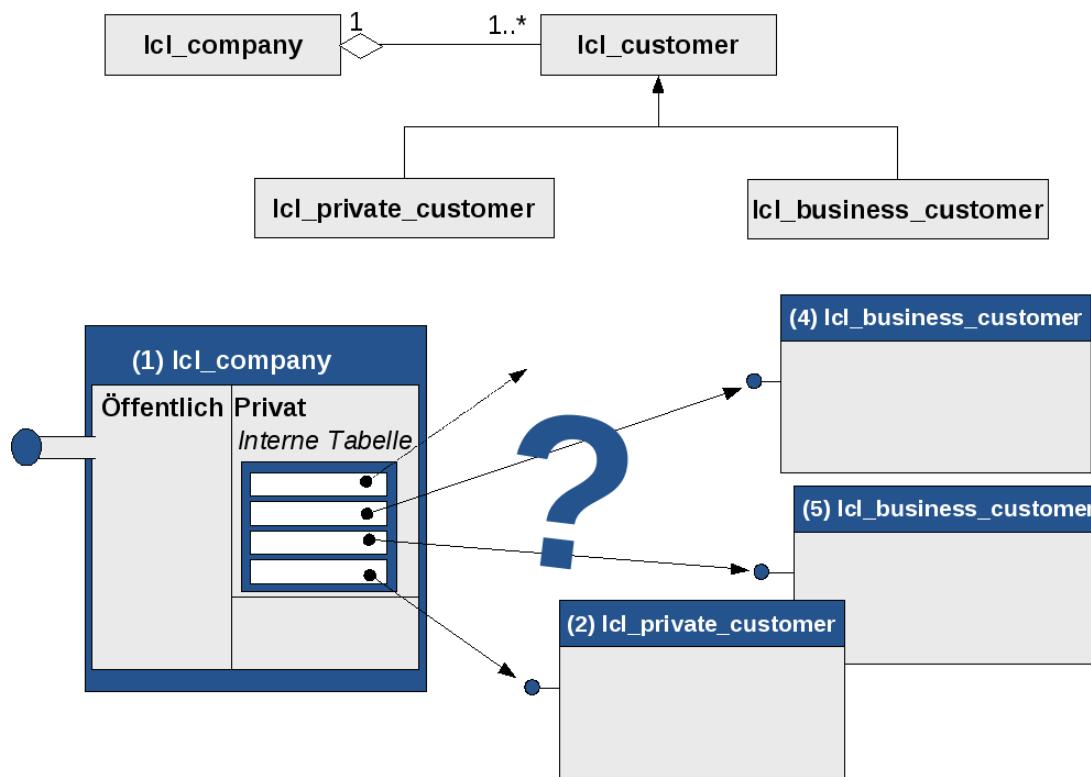


Abbildung 82: Beispielszenario

Die interne Tabelle in der Firmenklasse **lcl_company** kann als Tabelle mit Referenzen zu Kunden (**lcl_customer**) typisiert werden. Auch die Parameter der Zugriffsmethoden sind hierfür mit der Oberklasse **lcl_customer** zu typisieren:

```

CLASS lcl_company DEFINITION.
  PUBLIC SECTION.
    METHODS add_customer IMPORTING im_customer TYPE REF TO lcl_customer.
    ...
  PROTECTED SECTION.
    DATA customer_list TYPE TABLE OF REF TO lcl_customer.
    ...
ENDCLASS.

CLASS lcl_company IMPLEMENTATION.
  METHOD add_customer.
    APPEND im_customer TO customer_list.
  ENDMETHOD.
  ...
ENDCLASS.

```

Abbildung 83: Implementierung einer Methode im Szenario

Über die Referenzen der internen Tabelle kann nur auf die Komponenten der Oberklasse **lcl_customer** zugegriffen werden, wobei ggf. vorhandene Implementierungen aus der Unterklasse des jeweiligen Objekts verwendet werden. Ein Beispiel könnte hier ein Zugriff auf die **print**-Methode sein:

```

CLASS lcl_company DEFINITION.
PUBLIC SECTION.
METHODS: add_customer IMPORTING im_customer TYPE REF TO lcl_customer,
          print_customers.

...
PROTECTED SECTION.
DATA customer_list TYPE TABLE OF REF TO lcl_customer.

...
ENDCLASS.

CLASS lcl_company IMPLEMENTATION.
METHOD print_customers.
  DATA r_customer TYPE REF TO lcl_customer.
  LOOP AT customer_list INTO r_customer.
    r_customer->print( ).
  ENDLOOP.
ENDMETHOD.
...
ENDCLASS.

```

Abbildung 84: Methodenaufruf im Szenario

Sowohl die Klasse lcl_customer als auch die Klassen lcl_private_customer und lcl_business_customer verfügen über eine print-Methode. Durch den Aufruf `r_customer->print()` wird eine Methodenimplementierung verwendet, die nicht zwingend die aus der Klasse lcl_customer sein muss. Vielmehr **kommt es auf den dynamischen Typ an**, also auf das gerade betrachtete Objekt. Handelt es sich also z. B. um einen Privatkunden, wird die redefinierte Methode aus der Klasse lcl_private_customer verwendet. Im vorliegenden Beispiel haben alle Unterklassen die Methode redefiniert. Gäbe eine Unterklasse, die die Methode nicht redefiniert, würde für Objekte dieser Klasse stattdessen die geerbte Oberklassenmethode verwendet.

Diese unterschiedliche Reaktion von Objekten verschiedener Klassen auf den gleichen Methodenaufruf wird als **Polymorphie** bezeichnet und stellt ein wichtiges Konzept der Objektorientierung dar. Der Entwickler der Oberklasse muss sich keine Gedanken darüber machen, ob für bestimmte Objekte eine andere Implementierung einer Methode verwendet werden sollte. Das System ermittelt stattdessen, welche Implementierung zum jeweiligen Objekt gehört. Problemlos wäre auch die Implementierung weiterer Kundenklassen möglich, ohne dass die Klasse lcl_company verändert werden müsste. Die Klasse müsste lediglich im Vererbungsbaum unterhalb der Klasse lcl_customer eingeordnet werden. So lassen sich sehr **generische Programme** entwickeln.

Eine solche Generizität wäre in Prozedurelem ABAP nur sehr beschränkt realisierbar. Für die verschiedenen Kundentypen müssten separate Funktionsbausteine in einer Funktionsgruppe angelegt werden, und der Strukturtyp für Kunden müsste ein Feld besitzen, das den Namen des für diesen Kunden zu verwendenden Funktionsbausteins enthält. Diese Lösung besäße aber klare Schwächen, so ist es nicht möglich, zum Zeitpunkt der Syntaxprüfung festzustellen, ob zur Laufzeit in den entsprechenden Feldern auch tatsächlich nur Namen existierender Funktionsbausteine stehen:

```

FUNCTION-POOL s_customer.

...
FUNCTION print_private_customer.

...
ENDFUNCTION.

FUNCTION print_business_customer.

...
ENDFUNCTION.

```

Funktionsgruppe

```

DATA: customer_list TYPE TABLE OF str_customer,
      customer      TYPE str_customer.

...
LOOP AT customer_list INTO customer.
  CALL FUNCTION customer-func_name.
ENDLOOP.
...

```

Dynamische Angabe des Namens des Funktionsbausteins

name	...	func_name
Meier	print_private_customer
Müller	print_private_customer
Schmidt	print_business_customer
Schulz	print_private_customer

Abbildung 85: Generische Aufrufe ohne objektorientiertes ABAP

Es könnte durch einen Fehler passieren, dass bei einem Eintrag in der internen Tabelle `customer_list` das Feld `func_name` nicht mit einem Funktionsbausteinnamen belegt ist. Somit ist das prozedurale Programmiermodell an dieser Stelle dem objektorientierten Modell unterlegen.

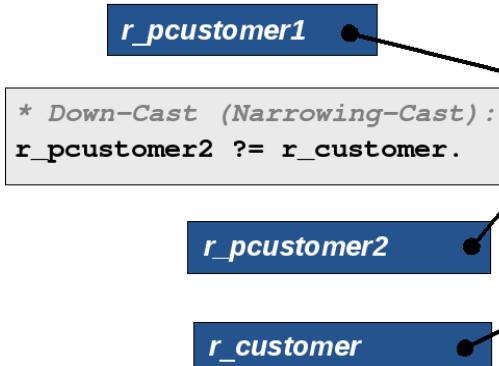
Hinweis: Sie werden sich vielleicht gefragt haben, ob Instanzen der Klasse `lcl_customer`, die weder Privat- noch Geschäftskunden sind, überhaupt sinnvoll sind. Diesem Thema werden wir uns im Abschnitt zu abstrakten Klassen widmen.

Im Beispiel aus Abbildung 84: Methodenaufruf im Szenario werden alle Typen von Kunden in einer Tabelle über Referenzen des Typs der Oberklasse verwaltet. Ein Zugriff auf Komponenten der Unterklassse ist so nicht möglich. Es könnte daher der Bedarf entstehen, die Referenz an eine Referenzvariable, die auf die Unterklassse typisiert ist zuzuweisen. Über diese wäre daraufhin ein Zugriff auf die Komponenten der Unterklassse wieder möglich.

Eine solche Zuweisung ist nicht ganz so unproblematisch wie die zuvor kennen gelernte Zuweisung in umgekehrter Richtung. Etwa könnte eine Referenz, die über die Klasse `lcl_customer` typisiert ist, einer Referenz zugewiesen werden, die über die Klasse `lcl_private_customer` typisiert ist. Verweist die erste Referenzvariable tatsächlich auf ein Objekt der Klasse `lcl_private_customer` (oder einer Unterklassse davon), besteht kein Problem. Es könnte aber auch passieren, dass zur Laufzeit die Referenzvariable ein Objekt der Klasse `lcl_customer` oder der Klasse `lcl_business_customer` referenziert. In diesem Fall wäre die Zuweisung unzulässig, denn Komponenten wie die Methode `get_name` wären syntaktisch über die Referenzvariable ansprechbar, beim Objekt aber gar nicht vorhanden. Eine solche Zuweisung erfolgt daher nicht wie gewohnt über den „=“-Operator bzw. die `MOVE ... TO ...`-Anweisung, sondern über den „?=”-Operator oder `MOVE ... ?TO` So wird deutlich gemacht, dass es sich um eine solche „riskante“ Zuweisung handelt.

```
DATA: r_customer TYPE REF TO lcl_customer,
      r_pcUSTOMER1 TYPE REF TO lcl_private_customer,
      r_pcUSTOMER2 TYPE REF TO lcl_private_customer.
```

```
CREATE OBJECT r_pcUSTOMER1.
r_customer = r_pcUSTOMER1.
```



(5) lcl_private_customer

Öffentlich
get_name
..

Geerbt:
get_address
...
redefiniert:
print
...

Abbildung 86: Down-Cast

Im Beispiel wird über die Unterklassen-Referenzvariable `r_pcUSTOMER1` zunächst ein Objekt der Unterklasse erzeugt. Diese Referenz wird dann der Oberklassen-Referenzvariable `r_customer` zugewiesen (Up-Cast). Anschließend folgt die hier entscheidende Zuweisung: Der Unterklassen-Referenzvariable `r_pcUSTOMER2` wird die Referenz von `r_customer`, also einer Oberklassen-Referenzvariable zugewiesen. Eine solche Zuweisung wird als *Down-Cast* bezeichnet. Die Zielvariable kann weniger dynamische Typen annehmen, so dass hier auch vom *Narrowing-Cast* gesprochen wird. Es sind aber mehr Komponenten erreichbar, nämlich zusätzlich die Komponenten der Unterklasse (und ggf. geerbte Komponenten von Klassen, die in der Vererbungshierarchie dazwischen liegen).

Ein Anwendungsbeispiel im Kontext des betrachteten Szenarios stellt ein Zugriff über die Kundenliste in der Unternehmens-Klasse `lcl_company` dar, bei der eine Komponente benötigt wird, die nur in einer Unterklasse verfügbar sind.

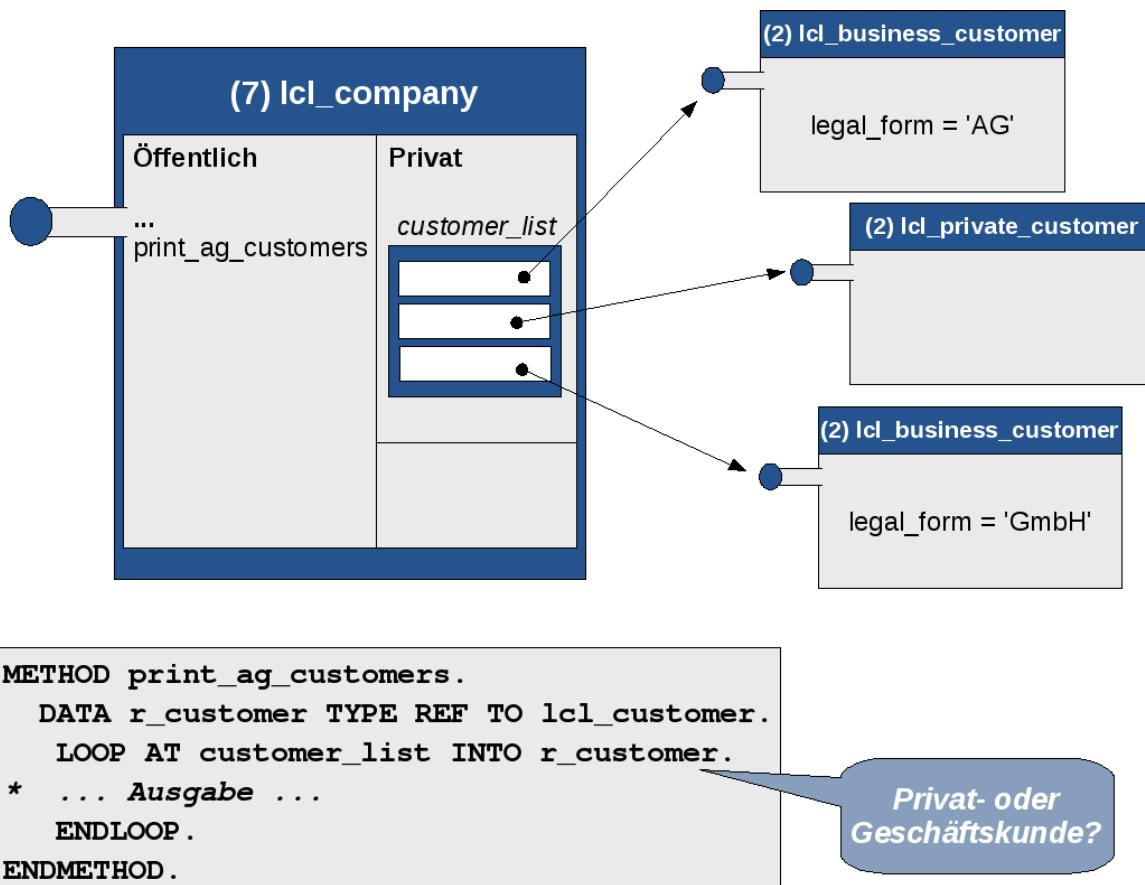


Abbildung 87: Methode, für die ein Down-Cast benötigt wird

Die Abbildung zeigt eine Methode in der Unternehmens-Klasse, die alle Aktiengesellschaften ausgeben soll, die Kunden des Unternehmens sind. Hierfür wurde der Geschäftskunden-Klasse ein Attribut für die Rechtsform (legal_form) hinzugefügt. Um auf diese Komponente (oder, falls entsprechend sauber implementiert, die zugehörige get-Methode) zugreifen zu können, ist eine Down-Cast-Zuweisung mit dem „?“-Operator erforderlich.

Würde eine solche Zuweisung der Referenz aus `r_customer` an eine Referenzvariable der Klasse `lcl_business_customer` ausgeführt, ohne dass es sich um ein Geschäftskundenobjekt handelt, würde eine behandelbare Ausnahme des Systems ausgelöst und das Programm würde nicht weiterlaufen. Daher muss dieser Fall abgefangen werden, indem auf die entsprechende Ausnahme reagiert wird.

Die Zuweisung könnte dann folgendermaßen aussehen:

```

METHOD print_ag_customers.
  DATA: r_customer TYPE REF TO lcl_customer,
        r_bcustomer TYPE REF TO lcl_business_customer.
  LOOP AT customer_list INTO r_customer.
    TRY.
      r_bcustomer ?= r_customer.
      * Bestimmung der Rechtsform und ggf. Ausgabe
      CATCH cx_sy_move_cast_error.
      * Fehlerbehandlung
    ENDTRY.
  ENDLOOP.
ENDMETHOD.

```

Abbildung 88: Implementierung des Down-Casts

Wie Sie sehen, ist hier die Zuweisung in einen Block der Form

```

TRY.
  anweisungen
  CATCH ausnahme.
    fehlerbehandlung
ENDTRY.

```

eingebunden. Hierbei handelt es sich um ein neues Konzept zur Behandlung von Ausnahmen, das über die Möglichkeiten des sy-subrc hinaus geht. Dieses Konzept wird Ihnen später näher erläutert. An dieser Stelle genügt es zu wissen, dass bei einem Fehler bei der Zuweisung im TRY-Teil in den CATCH-Block gesprungen wird, der diese Ausnahme behandelt. Im Fall des ungültigen Down-Casts handelt es sich um eine Ausnahme der Klasse `cx_sy_move_cast_error`.

4.7.2 Fazit: Vererbung

Durch die vorgestellten Möglichkeiten stellt die Vererbung ein sehr mächtiges Werkzeug bei der Entwicklung objektorientierter Software dar. Durch ihre Anwendung kann ein hohes Maß an Generizität und eine intensive Wiederverwendung bereits implementierter Funktionalität realisiert werden. Dies wirkt sich positiv auf die Wartbarkeit und meist auch auf die Lesbarkeit des Softwarequelltextes aus.

Diese Vorteile sollten Sie dennoch nicht dazu verleiten, Vererbung überall und um jeden Preis einzusetzen. Die Vererbung stellt semantisch eine „ist ein“-Beziehung von der Unter- zur Oberklasse dar. Ist eine solche Beziehung nicht gegeben, sollte die Vererbung nicht eingesetzt werden, auch wenn sich möglicherweise in der Oberklasse Komponenten befinden, die in der Unterklasse wiederverwendet werden könnten. Die Lesbarkeit des Quellcodes der Software und des zugrundeliegenden Modells würde darunter leiden.

Ein weiterer, häufig auftretender Fehler sind Vererbungen, die besser über ein Attribut abgebildet werden sollten. Würde man etwa von der Klasse `lcl_building` für Gebäude die Klassen `lcl_red_building`, `lcl_green_building`, `lcl_white_building` usw. ableiten, um rote, grüne, weiße Gebäude usw. abzubilden, wäre zwar die „ist ein“-Semantik gegeben, und auch die Eigenschaft, dass die Unterklasse ein speziellerer Fall der Oberklasse ist. Die Unterklassen würden sich aber inhaltlich vermutlich nicht von der Oberklasse unterscheiden,

und wenn die Oberklasse eine Methode zum Anstreichen des Gebäudes erhalten würde, stünde man vor einem Problem. Es müsste nämlich bereits in der Oberklasse ein Attribut für die Farbe geben, denn es gibt kein Haus ohne eine Farbe (es sei hier von besonderen Designs abgesehen).

Sinnvoll modelliert?

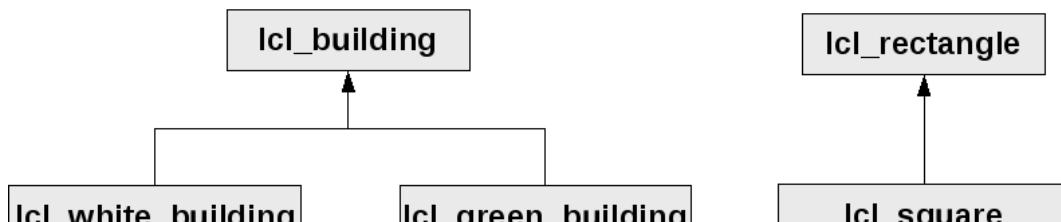


Abbildung 89: Fragwürdige Vererbungsbeziehungen

Das rechte Beispiel in der obigen Abbildung zeigt eine Vererbung von einer Klasse für Rechtecke zu einer Klasse für Quadrate. Auch diese Vererbung ist nur auf den ersten Blick sinnvoll. Wenn die Rechteck-Klasse separate Methoden zum Ändern von Länge und Breite besitzt, müssten diese Methoden in der UnterkLASSE zusätzlich die jeweils andere Seite verändern. Hierdurch wäre die eingangs geforderte Semantikerhaltung bei der Redefinition von Methoden nicht mehr gegeben.

4.7.3 Praxis: Übung zu Vererbung, Casting und Polymorphie

Legen Sie für diese Übung ein neues Programm `ZZ_####_COMPANY_2` an. Auch dieses Programm soll kein TOP-Include besitzen.



Abbildung 90: Anlegen des Programms: SAP-System-Screenshot

Wählen Sie im folgenden Fenster – wie beim letzten Programm – den Status **Testprogramm**, und bestätigen Sie dieses Fenster genau wie die Nachfrage nach Ihrem Paket und Ihrem Transportauftrag.

Speichern Sie das Programm und legen Sie auch hier ein Include an. Das Include soll den Namen `ZZ_####_COMPANY_2_CLASSES` bekommen. Wählen Sie auch hier den Status **Testprogramm** und bestätigen Sie die Nachfragen nach Paket und Transportauftrag wie auch die Bestätigung dass die `INCLUDE`-Anweisung eingefügt wurde. Sichern Sie das Programm und öffnen Sie das Include aus ihrem ersten Programm. Markieren Sie dort alle Klassendefinitionen und kopieren Sie diese, indem Sie Strg-c drücken, oder mit der rechten Maustaste das Kontextmenü öffnen und **Kopieren** wählen. Öffnen Sie dann wieder das Include des neuen Programms, wechseln sie mit oder Strg-F1 in den Bearbeitungsmodus

und fügen Sie den kopierten Code unterhalb der einleitenden Kommentarzeilen mit Strg-v oder rechter Maustaste und **Einfügen** aus dem Kontextmenü wieder ein.

Hinweis: Es gibt auch Bordmittel des SAP-Systems zum Kopieren, diese verwenden wir wegen technischer Probleme aber nicht. Beachten Sie ferner, dass die Tastenkombinationen auf nicht-Windows-Systemen von den oben angegebenen Kombinationen abweichen können.

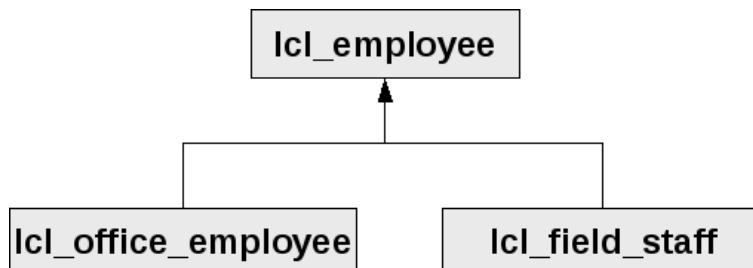


Abbildung 91: Vererbungsbeziehung für die Übung

Die Klasse **lcl_employee** für Mitarbeiter soll in dieser Übung als Ausgangspunkt für eine Vererbung verwendet werden. Es soll ermöglicht werden, Innen- und Außendienstmitarbeiter abzubilden. Erstellen Sie für die Innendienstmitarbeiter eine Klasse **lcl_office_employee**, für die Außendienstmitarbeiter eine Klasse **lcl_field_staff**. Die Klassen sollen von der Mitarbeiterklasse abgeleitet werden.

Die Innendienstmitarbeiter-Klasse soll ein Attribut **office** zur Angabe des Büros (z. B. „Raum A17“) erhalten. Implementieren Sie dieses mit privater Sichtbarkeit und fügen Sie eine entsprechende get- und set-Methode hinzu. Erstellen Sie außerdem einen Konstruktor, über den auch die Büroangabe direkt beim Erzeugen des Objekts gesetzt werden kann.

Die Klasse für die Außendienstmitarbeiter soll ein Attribut **percentage** erhalten. Dieses soll die Provision des Mitarbeiters enthalten, also den Prozentsatz den er von seinen Umsätzen erhält. Fügen Sie ferner ein Feld **sales** hinzu das den aktuellen Umsatz enthält. Beide Attribute sollen privat und über get- bzw. set-Methoden zugreifbar sein.

Redefinieren Sie die Methode **get_salary** in der Klasse für Außendienstmitarbeiter. Das Gehalt des Außendienstmitarbeiters berechnet sich aus dem Grundgehalt zuzüglich dem prozentualen Anteil am Umsatz. Einen altersbezogenen Zuschlag gibt es in diesem Fall nicht. Implementieren Sie die Methode entsprechend.

Fügen Sie außerdem in beide abgeleitete Klassen eine Redefinition der **print**-Methode ein. Die redefinierten Methoden sollen auch die Attribute der jeweiligen spezielleren Klasse ausgeben. Für die Ausgabe der allgemeinen Attribute aus der Oberklasse soll deren **print**-Methode wiederverwendet werden. Rufen Sie daher aus der Unterklassenmethode über den super-Verweis die **print**-Methode der Oberklasse auf und ergänzen Sie danach die spezifischen Ausgaben der Unterklassenattribute. Ergänzen Sie außerdem Konstruktoren, die zusätzlich zu den Attributen die für den Oberklassenkonstruktor benötigt werden, auch die speziellen Attribute der jeweiligen Unterklasse entgegennehmen.

Speichern, prüfen und aktivieren Sie das Include. Wechseln Sie nun zum Hauptprogramm **ZZ_####_COMPANY_2**. Definieren Sie dort je eine Referenzvariable vom Typ der Abteilung, vom Typ Innen- und vom Typ Außendienstmitarbeiter. Erzeugen Sie eine Instanz der Abteilung, und fügen Sie dann über die anderen Variablen je zwei Innen- und

Außendienstmitarbeiter ein, indem Sie den CREATE OBJECT-Befehl verwenden und die Methode ADD_EMPLOYEE aufrufen. Rufen Sie Anschließend die print-Methode der Abteilungsklasse auf.

Vergessen Sie nicht die Anweisung START-OF-SELECTION. Andernfalls erhalten Sie eine solche Fehlermeldung:

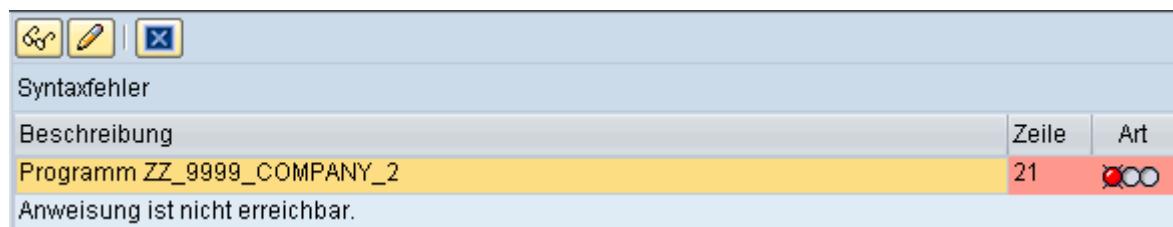


Abbildung 92: Nicht erreichbare Anweisung: SAP-System-Screenshot

Speichern, prüfen und aktivieren Sie das Programm. Beim Testen sollten sie eine Ausgabe ähnlich der Folgenden erhalten:

Programm ZZ_3099_COMPANY_2									
Programm ZZ_3099_COMPANY_2									
Abteilung: Abteilung 1									
Name: Gerd Schulz	Adresse: Astrasse 1	Gehalt:	5.600	Geburtsdatum: 11.11	Verkäufe: 6.000	Prov. 10			
Name: Andreas Gora	Adresse: Bplatz 1	Gehalt:	6.480	Geburtsdatum: 12.12	Verkäufe: 6.500	Prov. 20			
Name: Simon Meri	Adresse: Akalstraße 4	Gehalt:	6.340	Geburtsdatum: 11.12	Büro 10				
Name: Dennis Müller	Adresse: Rudolfplatz 4	Gehalt:	3.180	Geburtsdatum: 11.12	Büro 2				

Abbildung 93: Ausgabe des Programms: SAP-System-Screenshot

Sie haben durch die Verwendung von Unterklassenreferenzen für die Übergabe an die add_employee-Methode, deren Parameter als Referenz der Klasse lcl_employee typisiert ist, einen Up-Cast durchgeführt. Beim Aufruf der print-Methode der einzelnen Mitarbeiter in der Methode print der Klasse lcl_department wird die Methode der jeweiligen Unterklasse verwendet, so dass hier je nach dynamischem Typ Büro- bzw. Verkaufsinformationen ausgegeben werden. Beachten Sie, dass dafür keine Veränderungen an der Klasse lcl_department erforderlich waren.

Im nächsten Schritt soll die Abteilungsklasse hingegen verändert werden: Sie soll eine Methode get_avg_percentage erhalten. Diese Methode soll den durchschnittlichen Provisionssatz aller Außendienstmitarbeiter der Abteilung zurückliefern. Für diese Berechnung benötigen Sie einen Down-Cast auf die Außendienstmitarbeiter-Klasse, da in der Abteilungsklasse nur über Referenzen der Klasse lcl_employee auf die Mitarbeiterobjekte zugegriffen wird, so jedoch die benötigte get-Methoden zum Zugriff auf die Provision nicht erreicht werden kann. Der Down-Cast muss in eine TRY/CATCH-Anweisung eingebettet werden, da es sonst zu einem cx_sy_move_cast_error kommen würde, wenn der Cast für einen der Innendienstmitarbeiter ausgeführt würde. Bei erfolgreichem Cast kann die Provision über die Außendienstmitarbeiter-Referenz und ihre get-Methode ausgelesen und in die Berechnung einbezogen werden. Der CATCH-Teil kann hingegen leer bleiben, da Innendienstmitarbeiter schlicht ignoriert werden. Zum Erfassen von Summe und Anzahl der Provisionssätze werden Sie Hilfsvariablen benötigen.

Rufen Sie die Methode anschließend im Hauptprogramm auf, und geben Sie das von ihr berechnete Ergebnis anschließend aus:

```

Abteilung: Abteilung 1
Name: Gerd Schulz      Adresse: Astrasse 1
Name: Andreas Gora     Adresse: Bplatz 1
Name: Simon Meri       Adresse: Akalstraße 4
Name: Dennis Müller    Adresse: Rudolfplatz 4
Durchschnittlicher Provisionssatz: 15

```

Abbildung 94: Ausgabe des durchschnittlichen Provisionssatzes: SAP-System-Screenshot

Sie haben nun in Ihrem Programm sowohl den Up-Cast als auch den Down-Cast verwendet.

4.8 Abstrakte Klassen und Interfaces

In diesem Abschnitt lernen Sie eine besondere Art von Klassen kennen, von denen es keine Instanzen gibt. Weiterhin werden Interfaces behandelt.

4.8.1 Abstrakte Klassen

Es gibt Situationen, in denen das Erzeugen einer Instanz einer Klasse nicht erwünscht ist. Dies könnte auch im betrachteten Szenario der Fall sein, wenn ein Programm zwar Objekte erzeugen können soll, die entweder Instanzen der Klasse für Innendienst- oder der Klasse für Außendienstmitarbeiter sind, nicht aber reine Mitarbeiter-Instanzen. Diese Anforderung kann mit Abstrakten Klassen umgesetzt werden.

Abstrakte Klassen sind besondere Klassen, von denen keine Objekte erzeugt werden können. Sie dienen also vorrangig dazu, ihren Unterklassen Komponenten zu vererben. Auch hier ist Mehrfachvererbung unzulässig. Grundsätzlich kann so eine herkömmliche Klasse einfach in eine abstrakte Klasse umgewandelt werden. Sie muss dafür lediglich im Definitionsteil wie folgt definiert werden:

```

CLASS klassename DEFINITION ABSTRACT.
...
ENDCLASS.

```

In UML werden abstrakte Klassen wie herkömmliche Klassen dargestellt, erhalten aber zusätzlich die Angabe abstract in geschweiften Klammern. Das Szenario könnte also wie folgt aussehen:

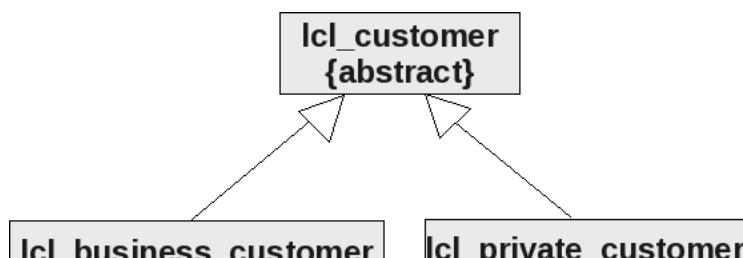


Abbildung 95: Abstrakte Klasse in UML

Zusätzlich zu den Komponenten, die eine herkömmliche Klasse aufweisen kann, kann eine abstrakte Klasse auch abstrakte Methoden aufweisen.

Abstrakte Methoden sind Methoden, die keine Implementierung besitzen. Das mag zunächst paradox erscheinen, da eine Methode ohne Implementierung nicht ausgeführt werden kann. An dieser Stelle kommt jedoch wieder die Polymorphie ins Spiel: Es gibt ohnehin keine Objekte der abstrakten Klasse selbst, über die ein Methodenaufruf versucht werden könnte. Ein Objekt einer Klasse, die sich in der Vererbungshierarchie unterhalb der Klasse befindet (Unterklasse), könnte jedoch eine Redefinition dieser Methode verwenden. Für den Verwender, der einen Verweis statisch mit der abstrakten Klasse typisiert, wäre der Ort der Implementierung nicht wichtig. Es muss jedoch sichergestellt sein, dass in einer Unterklasse die Methode (in Form einer Redefinition) implementiert ist, wenn ein entsprechendes Objekt erzeugt wird. So gesehen legt die abstrakte Methode fest, dass eine solche Methode (in einer erbenden Klasse) implementiert werden muss.

Zur Deklaration einer abstrakten Methode ist im METHODS-Statement hinter dem Methodenamen der Zusatz ABSTRACT anzugeben.

Abstrakte Klassen können auch wiederum abstrakte Klassen als Unterklassen besitzen. Die Implementierung einer abstrakten Methode kann dann entweder in der abstrakten Unterklassie erfolgen, oder erst in einer weiter unten in der Vererbungshierarchie befindlichen Klasse. Eine Klasse in der Vererbungshierarchie muss entweder alle in den darüber liegenden abstrakten Klassen definierten abstrakten Methoden redefinieren, die auf diesem Pfad durch die Vererbungshierarchie noch nicht redefiniert wurden, oder selbst als abstrakt definiert werden.

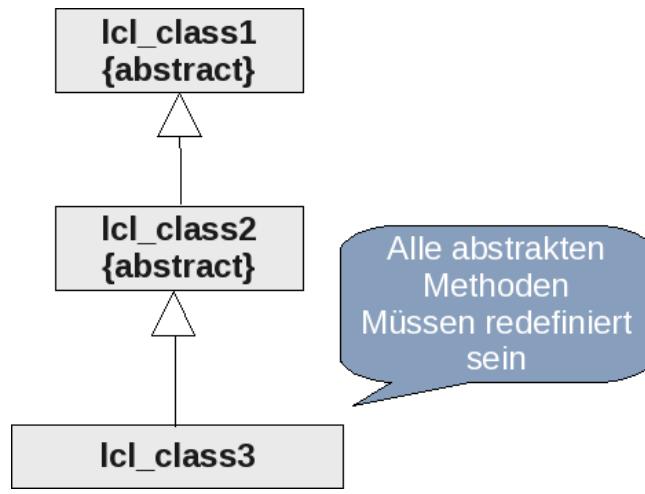


Abbildung 96: Mehrere abstrakte Klassen

Im hier abgebildeten Beispiel müsste die Klasse **lcl_class3** alle abstrakten Methoden der Klasse **lcl_class2** redefinieren, sowie alle abstrakten Methoden der Klasse **lcl_class1**, die nicht von **lcl_class2** redefiniert werden.

Abstrakte Methoden sind immer Instanzmethoden. Statische Methoden können nicht redefiniert werden, und so könnte es für eine statische abstrakte Methode nirgendwo eine Implementierung geben.

4.8.2 Praxis: Übung zu abstrakten Klassen

Legen Sie für diese Übung auf die gewohnte Weise ein neues Programm **ZZ_####_COMPANY_3** an, auch diesmal soll das Programm kein TOP-Include besitzen. Legen Sie dann ein Include **ZZ_####_COMPANY_3_CLASSES** für dieses Programm an, und kopieren Sie die Klassen aus dem Include des Programms **ZZ_####_COMPANY_2** hier hinein.

In dieser Übung soll zunächst dafür gesorgt werden, dass es keine Objekte der Mitarbeiterklasse, sondern nur Objekte der spezielleren Klassen für Innen- und Außendienstmitarbeiter gibt. Wandeln Sie dazu die Klasse `lcl_employee` in eine abstrakte Klasse um.

Speichern, prüfen und aktivieren Sie Ihr Include und wechseln Sie zum Hauptprogramm. Erstellen Sie dort eine Referenzvariable `r_employee` vom Typ der Mitarbeiterklasse. Versuchen Sie diese mit `CREATE OBJECT` zu instanziieren, speichern und prüfen Sie. Es sollte folgende Meldung erscheinen:

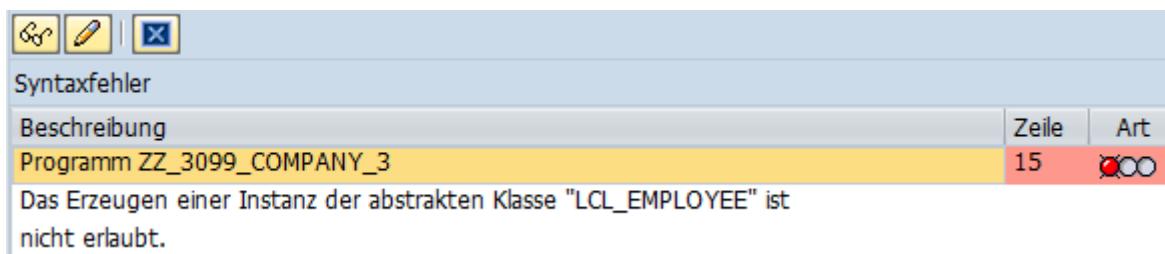


Abbildung 97: Fehler bei Instanziierungsversuch: SAP-System-Screenshot

Die unerlaubte Instanziierung kann also bereits statisch zum Zeitpunkt der Prüfung des Programms festgestellt werden.

Entfernen Sie die Zeile zur Objekterzeugung, die den Fehler auslöst. Definieren Sie stattdessen je eine Referenzvariable der Innen-, der Außendienstmitarbeiter- und der Abteilungsklasse. Instanziieren Sie die Abteilung und einige Mitarbeiterobjekte und fügen Sie diese der Abteilung hinzu.

Führen Sie nun die `print`-Methode der Abteilung aus. Diese sollte wie gewohnt die Daten der Mitarbeiter ausgeben.

Es soll nun zusätzlich eine `print_type`-Methode realisiert werden, die den Typ des Mitarbeiters (Innen- oder Außendienst) in lesbbarer Form ausgibt. Da die Oberklasse `lcl_employee` keinen solchen Typ besitzt, aber jede Unterklasse eine entsprechende Methode implementieren soll, erzeugen Sie eine abstrakte öffentliche Methode in der Oberklasse. Speichern und prüfen Sie den Code. Es erscheint eine Fehlermeldung folgender Art:

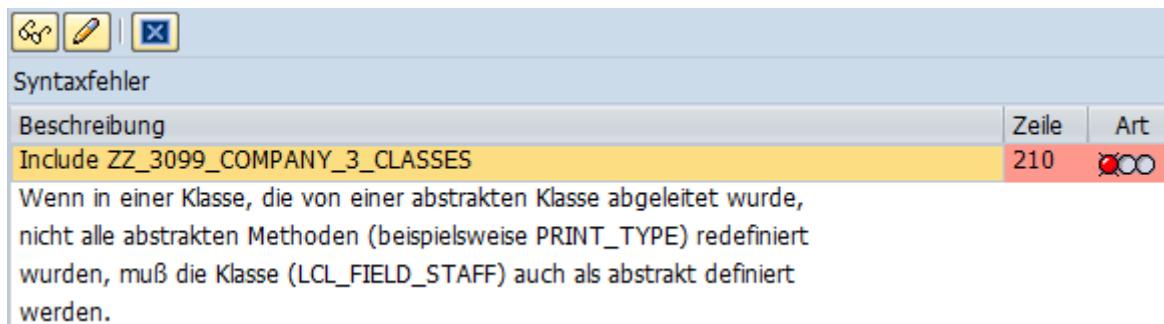


Abbildung 98: Fehlende Implementierung einer abstrakten Methode: SAP-System-Screenshot

Das System bemängelt, dass die konkrete Klasse lcl_field_staff nicht alle abstrakten Methoden implementiert, und daher selbst abstrakt sein muss. In unserem Fall wollen wir aber, dass lcl_field_staff eine konkrete Klasse bleibt. Fügen Sie daher eine Redefinition der abstrakten Methode print_type in der Klasse lcl_field_staff durch und implementieren Sie diese so, dass eine lesbare Bezeichnung des Mitarbeitertyps ausgegeben wird. Verfahren Sie anschließend genau so mit der Klasse lcl_office_employee.

Speichern und prüfen Sie ihr Programm. Es sollte nun kein Fehler mehr auftreten.

Die Methode soll nun verwendet werden, um die Ausgaben um die Angabe des Mitarbeitertyps zu erweitern. Dazu muss **nicht** die print-Methode jeder Unterklasse verändert werden. Stattdessen kann bereits in der Oberklassen-print-Methode, die ja in den Unterklassen aufgerufen wird, die abstrakte print_type-Methode aufgerufen werden. Bearbeiten Sie daher die print-Methode der Klasse lcl_employee und fügen Sie dort einen Aufruf der print_type-Methode ein.

Speichern, prüfen und aktivieren Sie Ihr Programm. Testen Sie es anschließend. Sie sollten eine Ausgabe erhalten, die in etwa wie die Folgende aussieht:

```
Report ZZ_3099_COMPANY_3

Abteilung: Abteilung 1
Innendienst-MA
Name: Andreas Schmidt Adresse:
Innendienst-MA
Name: Gerd Schulz     Adresse:
Aussendienst-MA
Name: Simon Meier      Adresse:
Aussendienst-MA
Name: Armin Müller    Adresse:
```

Abbildung 99: Erweiterte Ausgabe des Programms: SAP-System-Screenshot

Sie können an diesem Beispiel sehen, dass auch eine abstrakte Methode problemlos aufgerufen werden kann: Es ist sichergestellt, dass jede Instanz, für die die entsprechenden Codeelemente ausgeführt werden, einer konkreten Klasse angehören muss, für die eine Implementierung der abstrakten Methode (in Form einer Redefinition) vorhanden ist. Sie haben nun gelernt, wie abstrakte Klassen definiert werden und wie innerhalb dieser abstrakten Methoden definiert werden. Ferner haben Sie gelernt dass eine abstrakte Methode aufgerufen werden kann, ohne dass bekannt ist, in welcher konkreten Klasse sich später die Implementierung befindet.

4.8.3 Interfaces

Wie bereits zuvor dargestellt, ist in ABAP keine Mehrfachvererbung möglich. Darunter versteht man nicht die, problemlos mögliche, Bildung von Unterklassen unterhalb von Klassen, die bereits Unterklassen anderer Klassen sind, sondern Modelle, in denen eine Klasse mehr als eine direkte Oberklasse besitzt.

Ein solches Szenario wäre in ABAP also nicht implementierbar:

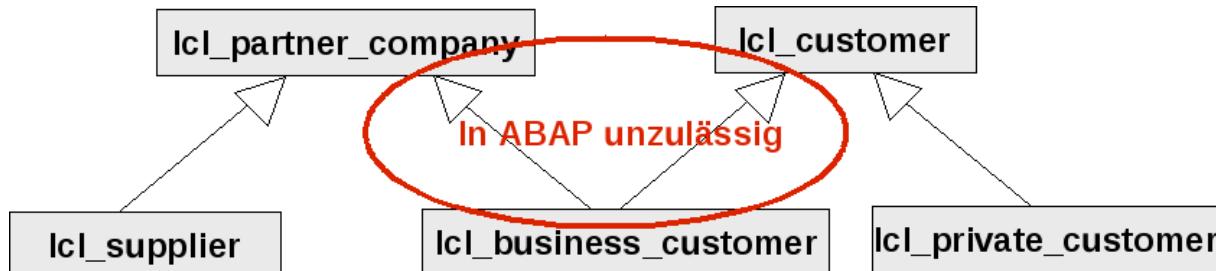


Abbildung 100: Beispiel für in ABAP unzulässige Mehrfachvererbung

Auch in vielen anderen Programmiersprachen (z. B. Java) ist eine solche Vererbung nicht zulässig. Es gibt jedoch Szenarien, in denen die Eigenschaften einer solchen Mehrfachvererbung wünschenswert wären:

- Es könnte eine Klasse geben, die Objekte aggregiert, die aus verschiedenen Teilen der Vererbungshierarchie stammen. Wäre Mehrfachvererbung zulässig, könnte eine Oberklasse eingeführt werden, mit der in der Verwenderklasse die Referenzen typisiert würden. Wie Sie im vorangegangenen Abschnitt gelernt haben, ist dank des Up-Casts ein Zugriff auf Objekte einer Unterklassie über Referenzen vom statischen Typ einer Oberklasse möglich.
- Es könnte der Bedarf entstehen, dass in verschiedenen Klassen ein bestimmter, einheitlicher Satz von Methoden für einen homogenen Zugriff implementiert wird. Wäre Mehrfachvererbung möglich, könnte man hierfür, für die betroffenen Klassen, eine zusätzliche Oberklasse definieren, die die benötigten Methoden enthält. Bei Bedarf könnten diese dann in den Unterklassen redefiniert werden.

Obwohl in ABAP keine Mehrfachvererbung möglich ist, können Eigenschaften wie die oben genannten dennoch umgesetzt werden. Hierfür steht das Konzept der **Interfaces** zur Verfügung. Ein Interface ist, wie der Name bereits nahelegt, eine Schnittstellenspezifikation. Die Schnittstelle wird in der Regel vom Verwender definiert und dann von Anbieter-Klassen implementiert.

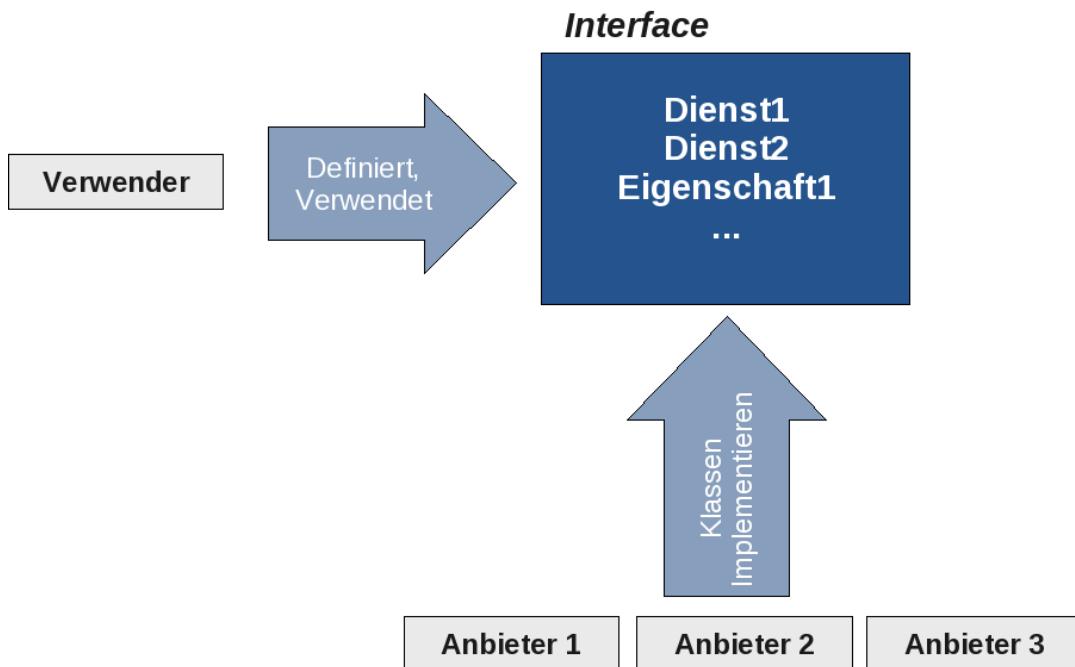


Abbildung 101: Protokollfestlegung durch den Verwender

Interfaces können von Klassen implementiert werden. Dafür erklärt eine Klasse, dass sie die von dem Interface vorgegebenen Methoden besitzt, und implementiert diese individuell. Aus Sicht der Verwenderklasse muss daraufhin lediglich gewährleistet sein, dass eine verwendete Klasse das Interface implementiert, das für den Zugriff auf die Klasse verwendet werden soll.

Innerhalb des Interfaces befinden sich lediglich Signaturen von Methoden. Es ist nicht möglich, den Code der Methoden bereits im Interface zu implementieren. Es dient stattdessen nur als Spezifikation der Schnittstelle, während die Realisierung der geforderten Funktionalität der implementierenden Klasse obliegt. Hier zeigt sich ein Unterschied zu den Möglichkeiten, die eine Mehrfachvererbung theoretisch möglich machen würde.

In der Unified Modeling Language werden Interfaces wie Klassen dargestellt, jedoch durch <<interface>> kenntlich gemacht. Derartige Kennzeichnungen werden in UML als Stereotype bezeichnet. Die Implementierung von Interfaces wird analog zu einer Vererbungsbeziehung dargestellt, die Verwendung analog zu einer gewöhnlichen Verwendungsbeziehung, die Linien sind jedoch gestrichelt. Die optische Ähnlichkeit zur Vererbung ist beabsichtigt, da wie oben dargestellt zahlreiche Ähnlichkeiten bestehen. Zusätzlich kann die Verwendung durch *uses*, die Implementierung durch *implements* gekennzeichnet werden, diese Begriffe werden dann am Pfeil notiert.

Um das obige Beispiel ABAP-kompatibel zu modellieren, könnte ein Interface eingesetzt werden:

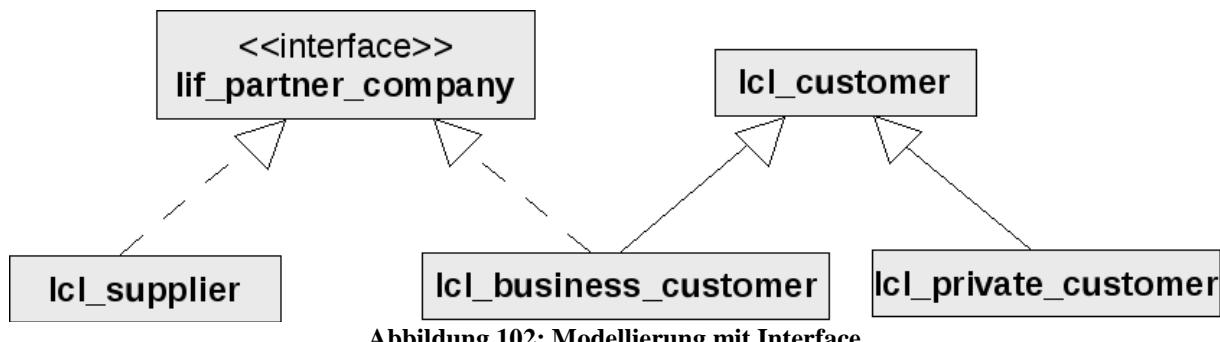


Abbildung 102: Modellierung mit Interface

Die Definition eines Interfaces erfolgt ähnlich einer Klassendefinition.

```

INTERFACE interfacename.
...
ENDINTERFACE .
  
```

An der Stelle der drei Punkte in der obigen Syntaxdarstellung können die Komponenten des Interfaces definiert werden. Im Gegensatz zu einer Klassendefinition erlaubt die Interfacedefinition keine Unterscheidung von Sichtbarkeitsbereichen. Das liegt daran, dass das Interface ohnehin eine Schnittstelle definiert, über die ein Zugriff von außen erfolgen soll. Dementsprechend sind Komponenten eines Interface immer öffentlich, wodurch sich eine Sichtbarkeitsangabe erübrigkt.

Ein weiterer Unterschied zur Klassendefinition ist das Fehlen eines Implementierungsteils. Das Interface definiert lediglich, welche Komponenten eine Klasse, die das Interface implementiert, besitzen muss. Es ist jedoch nicht möglich, bereits im Interface eine Implementierung vorzunehmen.

Um anzugeben, dass eine Klasse ein Interface implementiert, wird in der Klasse das Schlüsselwort **INTERFACES** benutzt (beachten Sie den Plural):

```

CLASS classname DEFINITION.
  PUBLIC SECTION.
    INTERFACES interfacename.
  ...
ENDCLASS .
  
```

Beachten Sie, dass die Angabe stets im öffentlichen Bereich der Klassendefinition erfolgt. Der Grund ist derselbe wie bei der fehlenden Sichtbarkeitsdefinition im Interface: Ein Interface stellt immer eine öffentliche Schnittstelle dar. Somit wäre die Implementierung im geschützten oder privaten Bereich nicht sinnvoll.

Die Klasse benötigt in ihrem Implementierungsteil nun eine Implementierung der Methoden aus dem Interface:

```

CLASS classname IMPLEMENTATION.
  METHOD interfacename~methodname.
  ...
  
```

```
ENDMETHOD.  
ENDCLASS.
```

In der Syntaxdarstellung sehen Sie, dass vor den Methodennamen einer Interfacemethode, mit einer Tilde (~) getrennt, der Name des Interfaces steht, auf das sich die Methodendefinition bezieht. Die Tilde wird auch als Interface-Resolution-Operator bezeichnet. Dieser Operator ist auch dann erforderlich, wenn über eine Referenzvariable eine Interface-Methode eines Objekts aufgerufen werden soll:

```
referenzvariable->interfacename~methodenname( ).
```

Die folgende Abbildung zeigt die Syntax zu Interfaces anhand des oben skizzierten Beispiels:

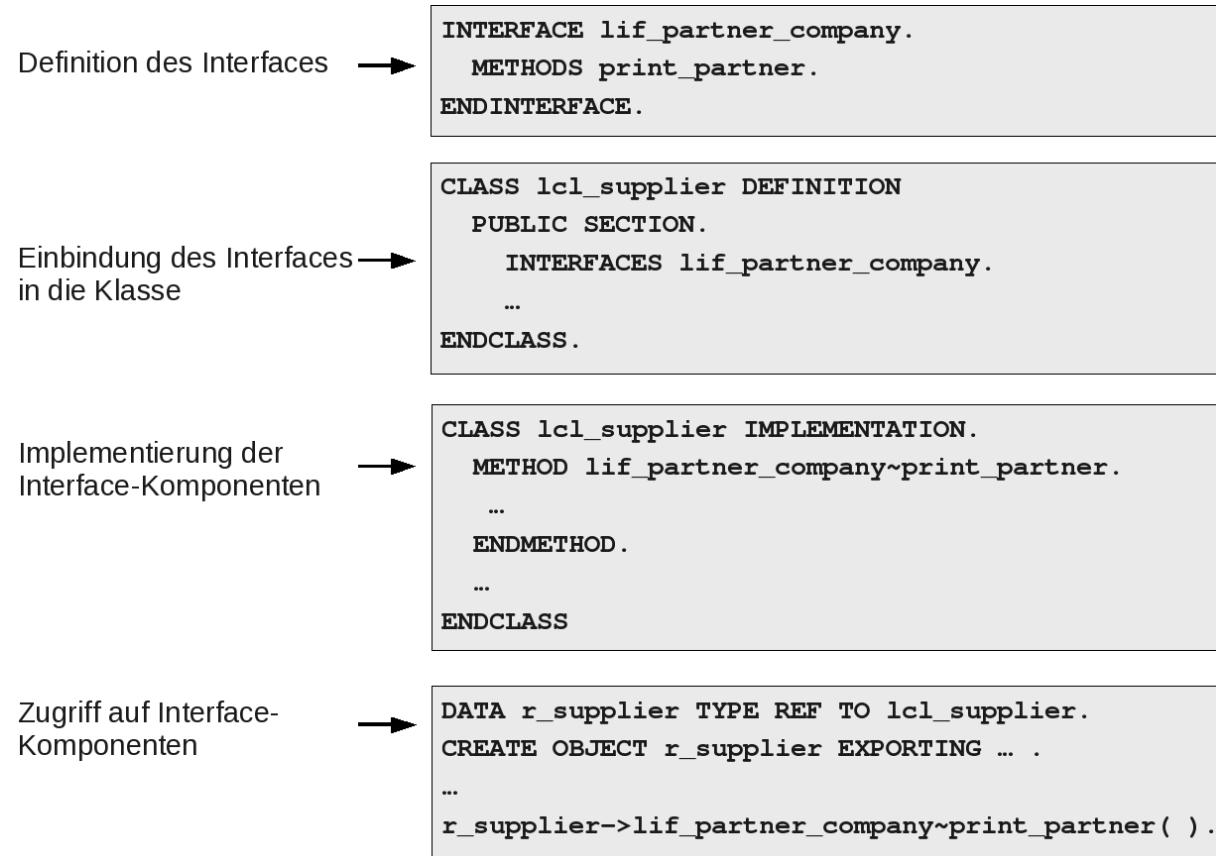


Abbildung 103: Syntaxbeispiel zu Interfacedefinition und -verwendung

Beachten Sie hier, dass die im Interface definierten Methoden nicht noch mittels METHODS definiert werden müssen.

Insbesondere die letzte Zeile in der Abbildung mag recht sperrig erscheinen. Auch bei Zugriffen innerhalb der Klasse muss der Interfacename angegeben werden, hier also etwa

```
lif_partner_company~print_partner( ).
```

Um diese Schreibarbeit zu vermeiden, bietet die Sprache ABAP auch die Möglichkeit, Aliasnamen zu definieren, so dass ein syntaktisch einfacherer Zugriff verwendet werden kann. Die Syntax für diese Aliasdefinition lautet:

```
ALIASES aliasname FOR interfacename~interfacekomponente.
```

Die Aliasdefinition wird im Definitionsteil der Klasse vorgenommen (oder in einem Interface, das das Interface interfacename einbindet, dazu später mehr). Ein Zugriff auf die Komponente ist dann wie folgt möglich:

referenzvariable->aliasname.

Entsprechend würde sich der sperrige Aufruf aus der obigen Abbildung verkürzen:

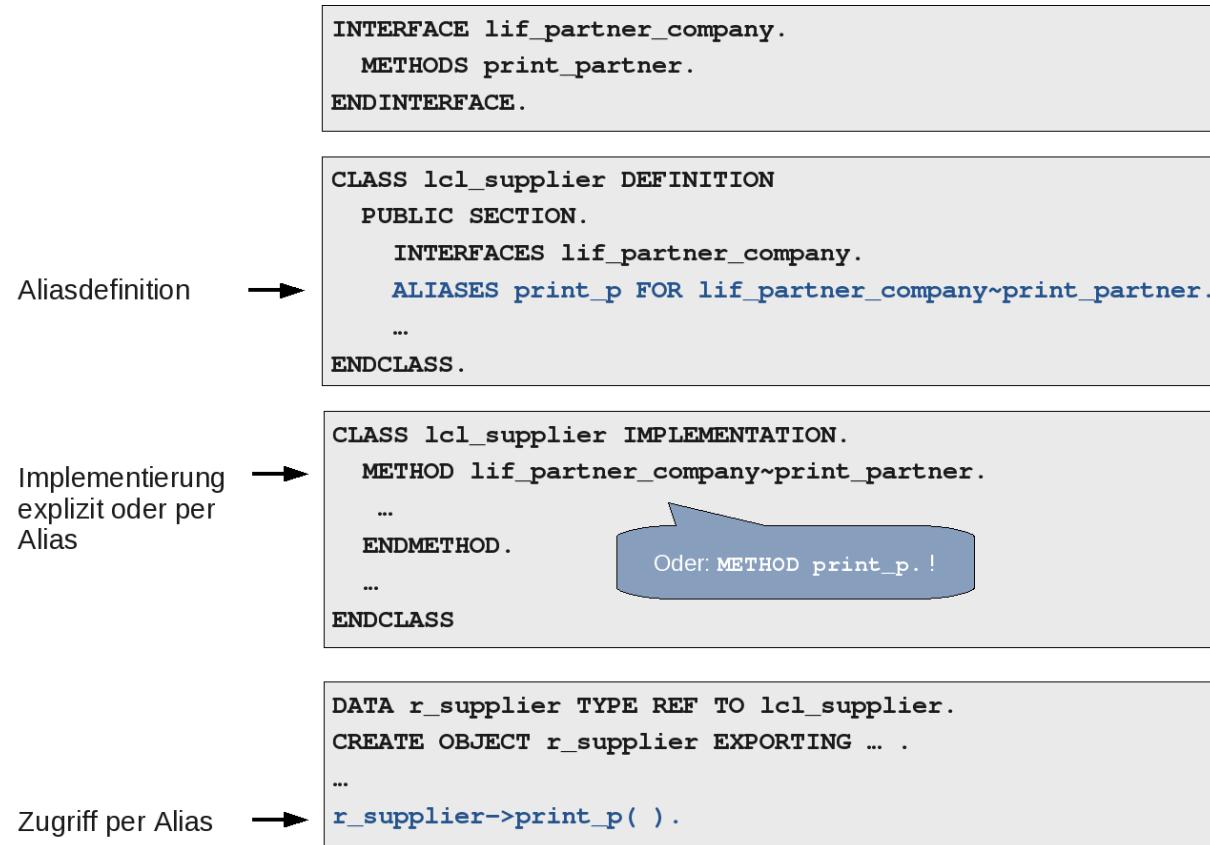


Abbildung 104: Vereinfachung durch Aliasdefinition

Der verminderte Schreibaufwand geht einher mit einer etwas verschlechterten Lesbarkeit. Durch den Alias ist nicht mehr unmittelbar erkennbar, aus welchem Interface eine Komponente stammt, bzw. dass sie überhaupt in einem Interface und nicht in der Klasse selbst definiert wurde. Weiterhin besteht die Gefahr, eine uneinheitliche Schreibweise zu verwenden. Um letzteres zu verhindern gibt es bei einer Durchmischung bei neueren Releases eine Warnmeldung.

Interfaces können zur Typisierung von Referenzvariablen verwendet werden, genau wie dies bei Klassen der Fall war. Von Interfaces selbst kann keine Instanz, also kein Objekt erzeugt werden, eine Referenzvariable, die mit einem Interface typisiert ist, kann jedoch auf Objekte jeder Klasse verweisen, die das Interface implementiert. Eine Referenz kann also per Up-Cast einer mit einem geeigneten Interface typisierten Referenzvariable zugewiesen werden. Über die Referenzvariable kann dann auf alle Interfacekomponenten zugegriffen werden. Da bekannt ist, dass die Klasse das Interface implementiert, ist auch stets gewährleistet, dass eine Implementierung der einzelnen Komponenten existiert.

```
DATA: r_supplier    TYPE REF TO lcl_supplier,
      r_partner_company TYPE REF TO lif_partner_company.
```

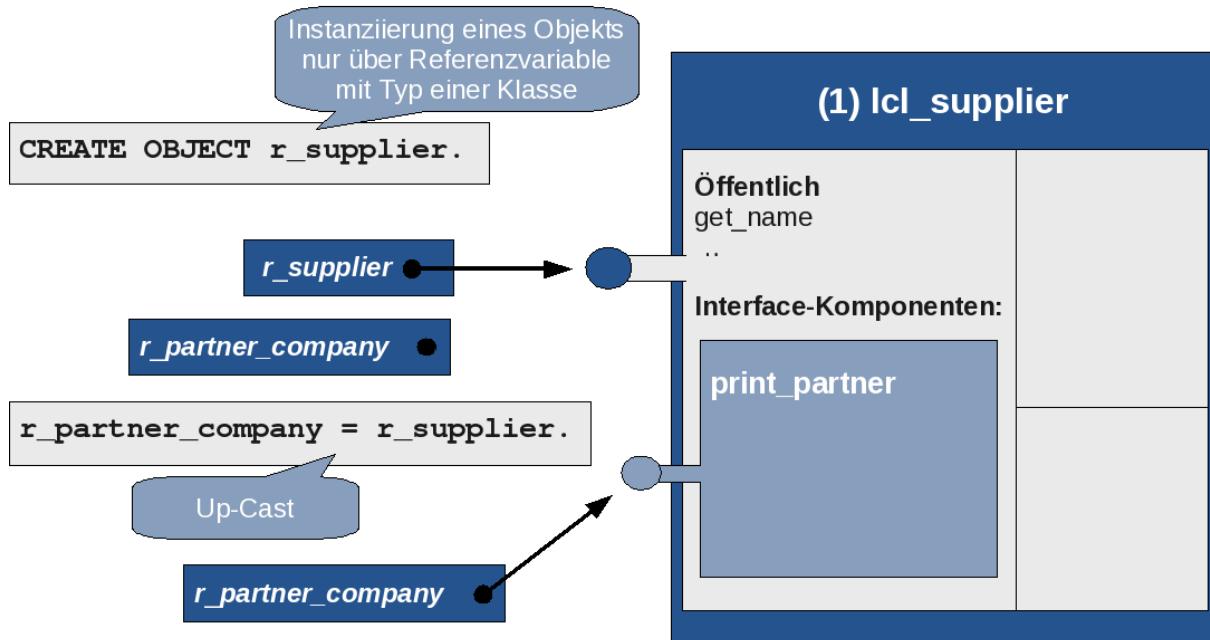


Abbildung 105: Verwendung über Interfaces typisierter Referenzvariablen

Die obige Abbildung zeigt, wie ein Objekt der Lieferantenklasse `lcl_supplier` erzeugt wird. Dieses wird anschließend (per Up-Cast) einer Referenzvariable zugewiesen, die mit dem Interface `lif_partner_company` typisiert ist, das von der Klasse `lcl_supplier` implementiert wird. Analog zum Verhalten bei mit Klassen typisierten Referenzvariablen kann hier über die Referenzvariable `r_partner_company` nur auf die Komponenten des Interfaces zugegriffen werden.

Die folgende Abbildung zeigt, wie durch den Zugriff über die mit dem Interface typisierte Variable auch eine syntaktische Vereinfachung beim Aufruf der Interface-Komponenten einhergeht. Dort kann direkt der Name der Komponente verwendet werden, ohne zusätzlich den Interfacenamen angeben oder eine Aliasdefinition vornehmen zu müssen.

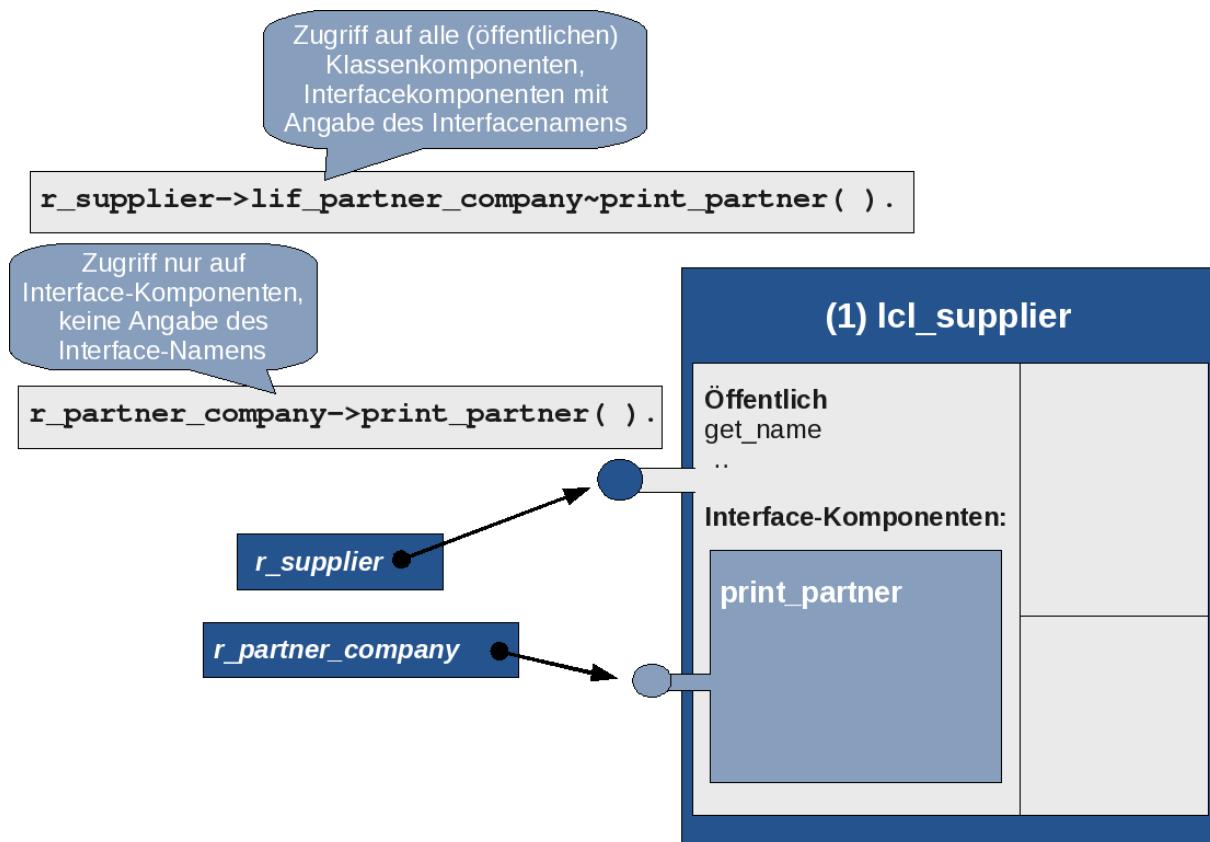


Abbildung 106: Komponentenzugriff über Interfaces

Auch bei den Zugriffen über die Interface-typisierte Referenz kommt Polymorphie zum Tragen: Je nachdem zu welcher Klasse das Objekt gehört, das über die Referenz `r_partner_company` erreicht wird, kann eine andere Implementierung der Methode zum Tragen kommen. Hier ist das referenzierte Objekt eine Instanz der Klasse `lcl_supplier`, die eine Implementierung besitzt, in der die Lieferantenbezogenen Daten ausgegeben werden. Handelte es sich hingegen um eine Instanz der Klasse `lcl_business_customer`, die ebenfalls das Interface implementiert und eine eigene Implementierung der Methode besitzt, würde diese Implementierung verwendet.

Das System sucht die zu verwendende Implementierung der Methode über den dynamischen Typ der Referenzvariablen, also über den Typ des referenzierten Objekts. Auch hier zeigt sich der Vorteil der Polymorphie, also des unterschiedlichen Verhaltens auf gleiche Methodenaufrufe. Der Anwender muss sich darum kümmern, dass die Objekte unterschiedlicher Klassen richtig behandelt werden, sondern das System sucht automatisch die Implementierung einer Methode, die zum jeweiligen Objekt gehört. Es können problemlos weitere Klassen hinzugefügt werden, die ebenfalls ein bereits existierendes Interface wie hier `lif_partner_company` implementieren. An allen Stellen, an denen im Programm Objekte einer Klasse erwartet werden, die das Interface implementiert, können dann auch Objekte der neuen Klasse verwendet werden, ohne dass dabei Änderungen am existierenden Code des Interfaces oder der anderen Klassen erforderlich ist. So kann Software sehr elegant um zusätzliche Fähigkeiten erweitert werden, ohne dass Eingriffe in andere Komponenten vorgenötigt sind.

Analog zum oben gezeigten Up-Cast sind auch hier Down-Cast-Zuweisungen möglich. Diese werden dann benötigt, wenn der Verwender sich für die speziellen Komponenten eines Objekts interessiert, die über die allgemeinere Interface-Referenz nicht erreichbar sind. Eine Referenz von einer Interface-typisierten Referenzvariable kann einer über eine Klasse

typisierte Referenzvariablen zugewiesen werden, die das Interface implementiert. Ein Beispiel wäre die umgekehrte Zuweisung zu vorangegangenen Beispiel:

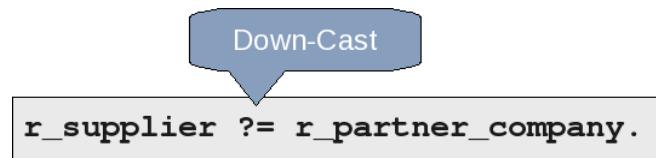


Abbildung 107: Down-Cast mit Interfaces

Nach der Zuweisung könnte der Verwender über `r_supplier` auf die Komponenten der Lieferantenklasse `lcl_supplier` zugreifen, die nicht Teil des Interfaces `lif_partner_company` sind.

Im Allgemeinen ist zur Laufzeit nicht sichergestellt, dass die Zuweisung erfolgreich ist: Wenn die Referenzvariable `r_partner_company` hier gerade auf ein Objekt der Klasse `lcl_business_customer` zeigt, entsteht ein Fehler, da die Variable `r_supplier` dieses Objekt nicht referenzieren kann. Daher sollte auch hier der ggf. entstehende Fehler mit TRY und CATCH abgefangen werden:

```
...
TRY.
  r_supplier ?= r_partner_company.
  CATCH cx_sy_move_cast_error.
  * Behandlung des Fehlers
ENDTRY.
...
```

Abbildung 108: Down-Cast mit Fehlerbehandlung

Im Kursverlauf werden Sie mit RTTI noch eine andere Möglichkeit kennenlernen, um erfolgreiche Zuweisungen sicher zu stellen, indem Sie den dynamischen Typ zur Laufzeit überprüfen.

Im oben erläuterten Beispiel zum Down-Cast wurde angenommen, dass die Zuweisung zu einer Referenzvariablen (hier `r_supplier`) erfolgt, die über eine Klasse typisiert ist, die das Interface implementiert. Es sind aber auch Zuweisungen zwischen Referenzvariablen möglich, die beide über Interfaces typisiert sind, ohne dass die Interfaces zueinander in Beziehung stehen. Es könnte im Szenario etwa ein zweites Interface geben, das die Klassen `lcl_supplier` und `lcl_business_customer` implementieren:

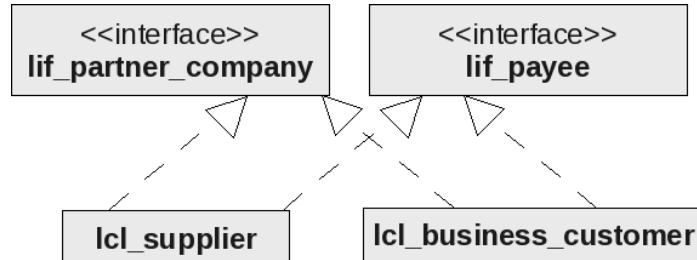


Abbildung 109: Szenario mit 2 Interfaces

Hier könnte es nun eine Referenzvariable geben, die über das Interface `lif_partner_company` typisiert ist, und eine andere Referenzvariable die wiederum über das Interface `lif_payee` typisiert ist.

```
DATA: r_supplier    TYPE REF TO lcl_supplier,
      r_partner_company TYPE REF TO lif_partner_company.
      r_payee TYPE REF TO lif_payee.
```

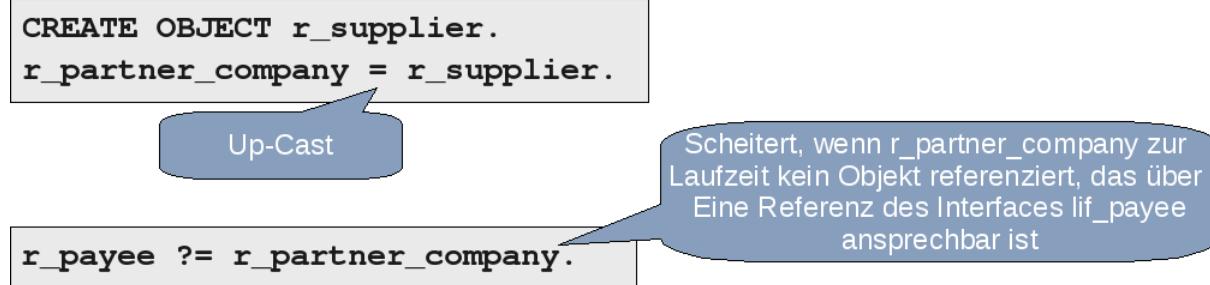


Abbildung 110: Zuweisungen zwischen Referenzvariablen unterschiedlicher Interfaces

Für die Zuweisung (letzte Zeile der obigen Abbildung) ist ebenfalls der Down-Cast-Operator `?=` erforderlich, da statisch nicht überprüft werden kann ob die Zuweisung funktioniert. Ein Versuch, den gewöhnlichen Zuweisungsoperator `=` zu benutzen, führte hingegen zu einer Fehlermeldung:

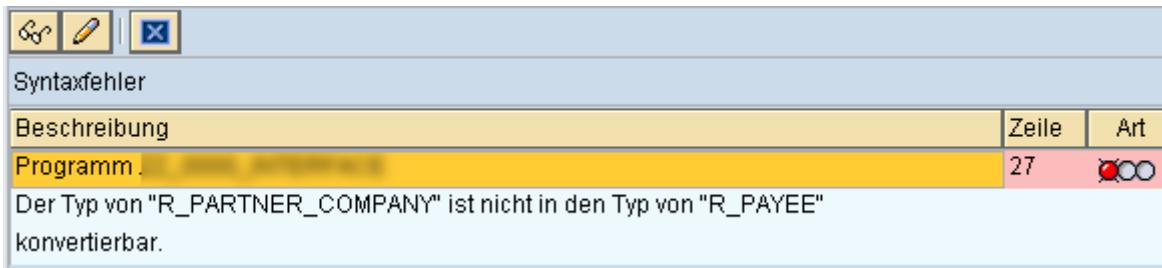


Abbildung 111: Fehlermeldung bei falscher Zuweisung: SAP-System-Screenshot

Neben dem obigen Fall, in dem die Interfaces nicht zueinander in Beziehung stehen, gibt es auch Fälle in denen dies der Fall ist. Interfaces können andere Interfaces einbinden, genau wie dies Klassen tun können. So lässt sich etwa eine Hierarchie, die einer Vererbungshierarchie unter Klassen ähnelt, konstruieren. Allgemeinere Interfaces können zu spezielleren Interfaces verfeinert werden. Es lassen sich Interfaces als Teile anderer Interfaces wiederverwenden, so dass die Wartbarkeit erhöht wird.

In UML wird die Einbindung eines Interfaces durch ein anderes Interface genau so dargestellt, wie die Einbindung durch eine Klasse, wie das folgende Beispiel zeigt.

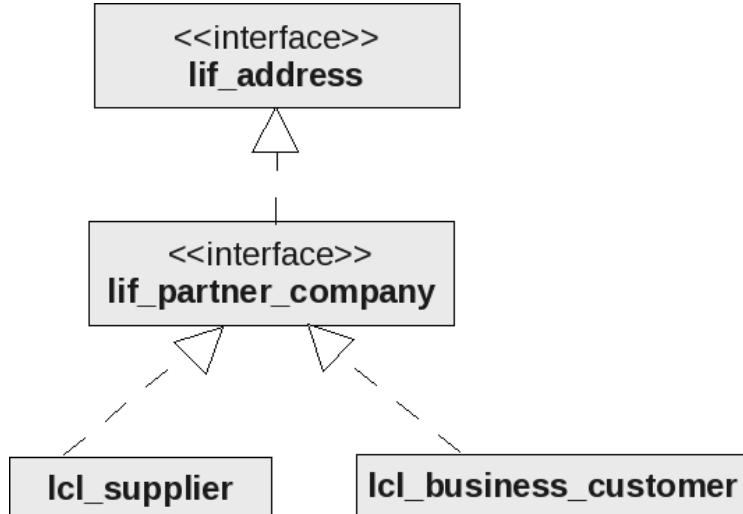


Abbildung 112: Interface-Hierarchie

Hier wird ein Interface für Adressen in das Interface für Partnerunternehmen eingebunden. Im Kontext von Interfaces lassen sich nun folgende Begriffe unterscheiden:

- **Elementare Interfaces:** Interfaces, die keine Interfaces enthalten, wie hier das Interface `lif_address`
- **Zusammengesetzte Interfaces:** Interfaces, die als Komponente andere Interfaces einbinden, wie hier das Interface `lif_partner_company`
- **Komponenten-Interfaces:** Interfaces, die als Komponente in ein anderes Interface eingebunden sind, wie hier das Interface `lif_address`

Eine Klasse, die nun das zusammengesetzte Interface einbindet, bindet dadurch implizit auch dessen Komponenteninterfaces ein. Die Komponenten des Komponenteninterfaces werden dann nicht wie Komponenten des zusammengesetzten Interfaces angesprochen, sondern über ihren eigenen Interfacenamen:

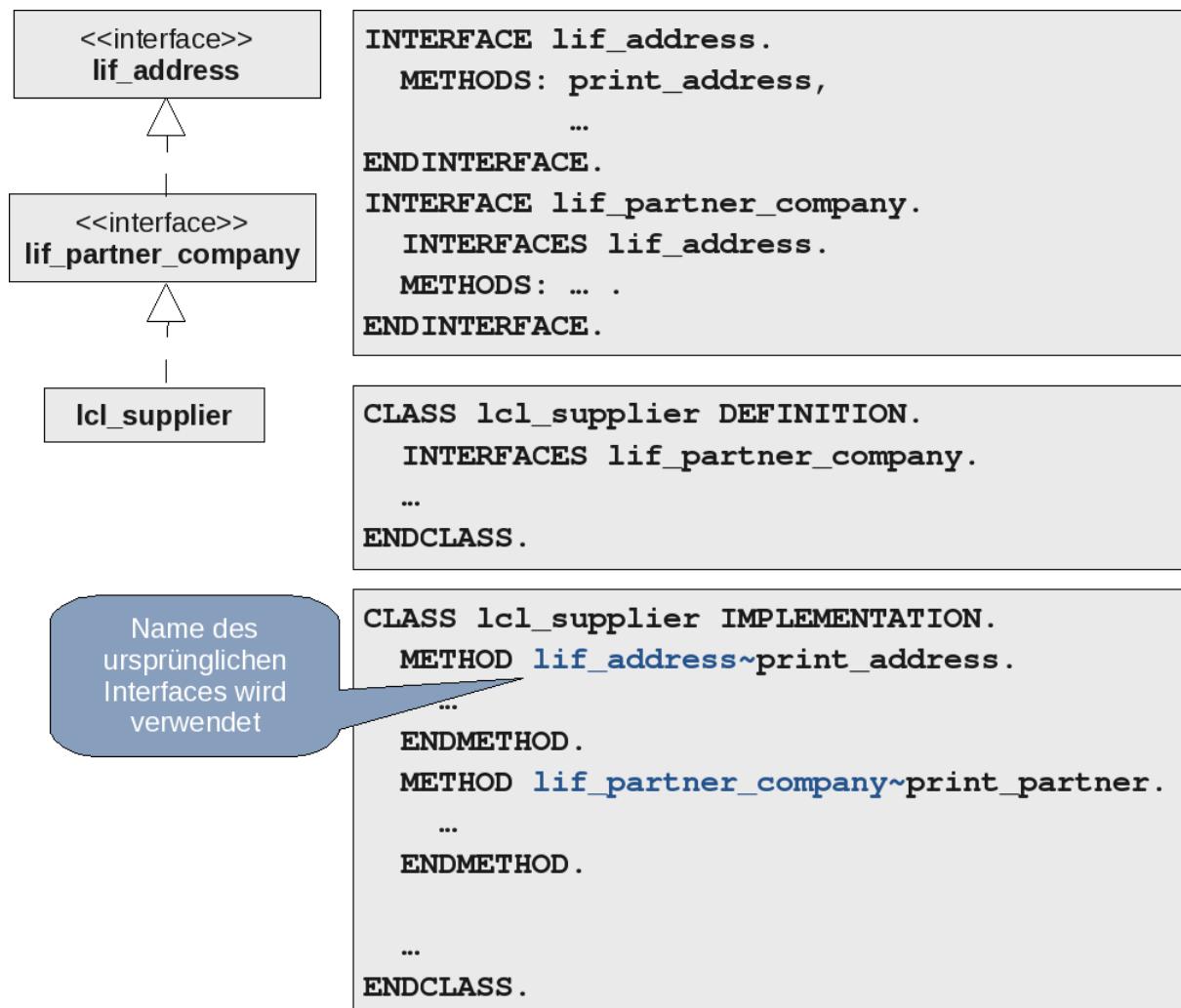


Abbildung 113: Zugriff auf Komponenten von Komponenten-Interfaces

Es dürfte in der Implementierung also insbesondere keine der folgenden Schreibweisen verwendet werden:

```

CLASS lcl_supplier IMPLEMENTATION.
METHOD lif_partner_company~print_address.
...
ENDMETHOD.

...
ENDCLASS.

```

```

CLASS lcl_supplier IMPLEMENTATION.
METHOD lif_partner_company~lif_address~print_address.
...
ENDMETHOD.

...
ENDCLASS.

```

Abbildung 114: Unzulässige Definitionen

Um hierbei eine schlechte Lesbarkeit zu vermeiden, bietet es sich an Aliasnamen zu verwenden.

Die Zuweisungen von Referenzen funktionieren hier analog zu den bekannten Fällen. Ein Up-Cast ist ohne explizite Kenntlichmachung möglich, hier also etwa die Zuweisung einer mit `lif_partner_company` typisierten Referenzvariablen zu einer Referenzvariablen, die mit `lif_address` typisiert ist. Bei einer Zuweisung einer Referenzvariablen des Typs `lif_address` zu einer Referenzvariable des Typs `lif_partner_company` handelt es sich hingegen um einen Down-Cast, für den der Operator `?=` benutzt werden muss. Die Zuweisung kann zu einem Fehler führen, wenn das referenzierte Objekt von einer Klasse stammt, die zwar das Interface `lif_address` einbindet, nicht aber das zusammengesetzte Interface `lif_partner_company`.

4.8.4 Praxis: Übung zu Interfaces

Erstellen Sie mit den gewohnten Schritten ein neues Programm mit dem Namen **ZZ_####_COMPANY_4**. Fügen Sie auch diesem Programm ein Include hinzu, das den Namen **ZZ_####_COMPANY_4_CLASSES** tragen soll und übernehmen Sie die Inhalte des Includes des Programms **ZZ_####_COMPANY_3_CLASSES**.

4.8.4.1 Erzeugen des Interfaces

Das hier erstellte Programm soll eine rudimentäre Unterstützung bei der Vergabe von Räumen bieten. Erstellen Sie im Include des neuen Programms ein Interface **lif_room_requirement**. Dieses soll eine Methode `get_roomsize` enthalten, die einen Rückgabewert `re_roomsize` vom Typ `i` zurückliefert. Dieser steht für die Raumgröße, die benötigt wird und in diesem einfachen Szenario zwischen 1 (kleiner Raum) und 3 (großer Raum) liegen wird. Außerdem soll das Interface eine Methode `print_requirement` enthalten.

Jede Klasse, die das Raumbedarf-Interface besitzt, stellt somit einen Raumbedarf dar. Dies sollen in diesem Szenario einerseits die Mitarbeiter sein (diese brauchen jeweils ein Büro), andererseits werden Orte für regelmäßige Besprechungen benötigt, die ebenfalls einen Raumbedarf darstellen.

4.8.4.2 Das Interface in Klassen implementieren

Erstellen Sie daher nun zunächst eine Klasse **lcl_conference_location**. Diese steht für Besprechungen (bzw. deren Orte) und soll das soeben definierte Interface einbinden. Bei der Implementierung der `get_roomsize`-Methode aus dem Interface soll ein fester Wert von 3 zurückgegeben werden, da für Besprechungsorte stets große Räume nötig sind. Definieren Sie in dieser Klasse auch ein privates Attribut **name** (Typ `string`) und implementieren Sie für dieses je eine öffentliche `get-` und `set-`Methode sowie einen Konstruktor.

Binden Sie das Interface anschließend auch in die Klasse **lcl_office_employee** ein. Hier soll die `get_roomsize`-Methode aus dem Interface die Raumgröße in Abhängigkeit vom Gehalt zurückliefern. Legen Sie eine Gehaltsgrenze ihrer Wahl fest, die unterscheidet ob der Mitarbeiter einen kleinen Raum (Rückgabewert 1) oder einen mittleren Raum (Rückgabewert 2) erhält. Implementieren Sie die Methode entsprechend. Implementieren Sie in beiden Klassen, die das Interface verwenden, die `print_requirement`-Methode. Bei den Besprechungsorten reicht die Ausgabe des Namens, beim Innendienstmitarbeiter können Sie die vorhandene Ausgabemethode aufrufen.

Implementieren Sie als nächstes die Klasse **lcl_building**. Diese repräsentiert Gebäude und soll die privaten Attribute `name` (Typ `string`) und `requirements` erhalten. Letzteres ist eine

Liste gemeldeter Raumbedarfe, die als interne Tabelle von Referenzen vom Typ des soeben definierten Interfaces implementiert werden soll. Implementieren Sie für das Namensattribut öffentliche get- und set-Methoden, und für das Attribut requirements eine Methode **add_requirement**, die eine neue Raumanforderung entgegennimmt und in die Liste einfügt.

Sie haben nun eine Gebäudeklasse erstellt, in der repräsentiert werden kann wofür in diesem Gebäude alles Räume benötigt werden, d. h. welche Mitarbeiter mit ihren Büros und Besprechungs-Orte in diesem Gebäude untergebracht werden sollen. Damit das Gebäude seine Auslastung bestimmen kann, implementieren Sie zunächst private Attribute für die Anzahl großer, mittlerer und kleiner Räume (available_small_rooms, available_medium_rooms und available_large_rooms). Implementieren Sie einen Konstruktor der den Namen und die drei Anzahlen bei der Objekterzeugung setzen kann. Füren Sie danach eine Methode **print_utilisation** hinzu. Die Methode soll ausgeben, welche Bedarfe dem Gebäude bereits zugewiesen wurden, und ob der Platz im Gebäude dafür ausreicht oder nicht. Erstes geben Sie durch Aufruf der print_requirement-Methode aus dem Interface für alle Bedarfe aus. Wenn Sie unsicher sind, wie die Ausgabe aussehen soll, sehen Sie sich den Screenshot auf der nächsten Seite an.

Letzteres müssen Sie berechnen, indem Sie für jedes Element der Bedarfsliste die Raumgröße über die Interfacemethode abrufen und schließlich prüfen, ob genügend große, mittlere und kleine Räume zur Verfügung stehen. Beachten Sie, dass hierfür kein down-cast nötig ist: Alle benötigten Methoden sind Teil des Interfaces, so dass es genügt die Tabelle mit einer über das Interface typisierten Referenzvariable zu durchlaufen.

Wenn Sie möchten, können Sie auch berücksichtigen, dass ein größerer Raum als eigentlich benötigt wird, auch zulässig ist.

Speichern, prüfen und aktivieren Sie das Include und wechseln Sie ins Hauptprogramm.

4.8.4.3 Erstellen des Hauptprogramms

Erstellen Sie dort Referenzvariablen vom Typ lcl_building, lcl_office_employee und lcl_conference_location. Instanziieren Sie ein Gebäude, mehrere Innendienstmitarbeiter (ober- und unterhalb Ihrer Gehaltsgrenze) und Konferenzorte und melden Sie deren Raumbedarfe bei der Gebäudeinstanz, indem Sie sie mit der add_requirement-Methode einfügen. Setzen Sie auch sinnvolle Raumanzahlen für das Gebäude. Rufen Sie die print_utilization-Methode zunächst vor überschreiten der Verfügbaren Räume aus, fügen Sie dann nach der Ausgabe weitere Bedarfe hinzu so dass die Raumzahlen nicht mehr ausreichen, und rufen Sie die print_utilization-Methode anschließend erneut aus. Stellen Sie sicher, dass die Ausgabe **lesbar** ist und dass sowohl die Ausgabe der Situation vor der „Überfüllung“ als auch die Ausgabe der Situation danach sichtbar sind.

Speichern, prüfen, aktivieren und testen Sie Ihr Programm.

Program ZZ_9999_COMPANY_4

Program ZZ_9999_COMPANY_4

Für das Gebäude Büroturm wurden folgende Raumbedarfe registriert:

Innendienstmitarbeiter

Name: John Doe	Addresse: Weststr. 123, Berlin	Gehalt: 1,050	Alter40	Büro: C 36
Innendienstmitarbeiter				
Name: Bengt Olafsson	Addresse: Storgatan 734, Stockholm	Gehalt: 1,100	Alter50	Büro: A 37c
Innendienstmitarbeiter				
Name: John Smith	Addresse: 123 East Street, NY	Gehalt: 1,760	Alter52	Büro: C 39
Besprechungsraum Teambesprechungsraum				

Dies führt zu folgender Auslastung:

Kleine Räume:	2 benötigt,	3 verfügbar.
Mittlere Räume:	1 benötigt,	3 verfügbar.
Große Räume:	1 benötigt,	2 verfügbar.

Ergebnis: OK - genügend freie Räume gefunden.

Für das Gebäude Büroturm wurden folgende Raumbedarfe registriert:

Innendienstmitarbeiter

Name: John Doe	Addresse: Weststr. 123, Berlin	Gehalt: 1,050	Alter40	Büro: C 36
Innendienstmitarbeiter				
Name: Bengt Olafsson	Addresse: Storgatan 734, Stockholm	Gehalt: 1,100	Alter50	Büro: A 37c
Innendienstmitarbeiter				
Name: John Smith	Addresse: 123 East Street, NY	Gehalt: 1,760	Alter52	Büro: C 39
Besprechungsraum Teambesprechungsraum				
Besprechungsraum Vorstandssitzungsraum				
Besprechungsraum Interne Besprechungsraum				

Dies führt zu folgender Auslastung:

Kleine Räume:	2 benötigt,	3 verfügbar.
Mittlere Räume:	1 benötigt,	3 verfügbar.
Große Räume:	3 benötigt,	2 verfügbar.

Ergebnis: Die verfügbaren Räume reichen NICHT aus!

Abbildung 115: Ausgabe des Programms: SAP-System-Screenshot / Montage

Die Abbildung zeigt eine mögliche Ausgabe des Programms. Bei der oberen Ausgabe reichen die räumlichen Kapazitäten aus. Bei der zweiten Ausgabe sind hier Bedarfe für große Räume hinzugekommen die nicht gedeckt werden können, die Räume im Gebäude reichen nun nicht mehr.

4.8.4.4 Verwenden von Aliasen

Öffnen Sie nun erneut ihr Hauptprogramm. Gehen Sie ganz ans Ende des Codes, und rufen Sie für den zuletzt erzeugten Besprechungsraum die Ausgabemethode auf. Versuchen Sie dies zunächst mit folgender, inkorrektener Syntax:

```
r_conference_location->print_requirement( ).
```

Sie erhalten beim Prüfen des Programms folgende Fehlermeldung:



Figure 116: Syntaxfehler: SAP-System-Screenshot

Die Fehlermeldung erscheint, da es keinen Alias für die Methode gibt. Deshalb müsste derzeit noch das Interface beim Aufruf mit angegeben werden. Der korrekte Aufruf würde also wie folgt lauten:

```
r_conference_location->lif_room_requirement~print_requirement( ).
```

Diese Zeile ist sehr lang und es bietet sich daher an, durch die Definition eines Aliases die Schreibarbeit solcher Aufrufe zu vereinfachen. Definieren Sie daher nun für die Interfacemethode einen Aliasnamen, so dass der Kürzere Aufruf funktionsfähig wird, und **belassen** Sie diesen Aufruf anschließend in Ihrem Programm.

```
r_conference_location->print_requirement( ).
```

4.8.5 Fazit: Abstrakte Klassen und Interfaces

Sowohl Abstrakte Klassen als auch Interfaces können nicht selbst instanziert werden. Im Gegensatz zum Interface kann eine Abstrakte Klasse Implementierungen von Methoden enthalten. Sie ähneln stark herkömmlichen Klassen, es ist keine Mehrfachvererbung zulässig. Eine Klasse oder ein Interface kann jedoch mehrere Interfaces einbinden. Auf diese Weise lässt sich Mehrfachvererbung simulieren.

Interfaces bieten Sich an, wenn sich keine geeignete Klasse im Modell einbinden lässt, die dem gewünschten Zweck dient und keine Methoden implementiert werden sollen (im Interface selbst). Durch das Interface wird eine Schnittstelle zwischen Verwender und Anbieter einer Dienstleistung geschaffen. So wird eine starke Kopplung vermieden und Programme sind leicht erweiterbar. Es wird ein hohes Maß an Generizität erreicht. Durch die Möglichkeit, mehrere Interfaces in einer Klasse oder in einem Interface einzubinden, können separate Interfaces für verschiedene Teilaufgaben entworfen und flexibel wiederverwendet werden.

4.9 Ereignisse

In Szenarien, wie dem aus Unternehmen und Kunden, kann es wünschenswert sein, dass bestimmte, regelmäßig anfallende Aufgaben automatisch durchgeführt werden. In diesem Szenario wäre dies etwa das Hinzufügen von neuen Kundenobjekten zum Unternehmen, also in die interne Tabelle in einem Unternehmensobjekt. Es liegt nahe, hier den Konstruktor der Kundenklasse zu verwenden, jedoch müsste dieser jeweils wissen, zu welchem Unternehmensobjekt das neue Kundenobjekt dann hinzugefügt werden soll. Dieser Weg ist also wenig attraktiv.

Stattdessen bietet ABAP ein Ereigniskonzept an, mit dem für relevante Situationen Ereignisse definiert und ausgelöst werden. Verwender können diese Ereignisse dann behandeln, indem sie eine Behandlermethode bereit stellen und diese zur Laufzeit für die entsprechenden Ereignisse registrieren.

Bei der Erzeugung eines neuen Kunden müsste also ein Ereignis ausgelöst werden, während das Unternehmen eine Behandlermethode registriert, die dann automatisch vom System bei jedem Auslösen des Ereignisses aufgerufen wird. Im Unterschied zu einer Implementierung, wie oben skizziert, muss der Auslöser des Ereignisses nicht wissen, wer das Ereignis behandeln soll. Das System sucht beim Auslösen eines Ereignisses alle auf dieses Ereignis registrierten Behandler und führt die entsprechenden Behandlernethoden aus. Damit ist hier eine Kapselung gelungen: Ein Wechsel des Behandlers erfordert keinerlei Änderungen am Auslöser des Ereignisses. So könnten nachträglich weitere Behandler hinzukommen, die sich ebenfalls für das Ereignis „Kundenobjekt erzeugt“ interessieren.

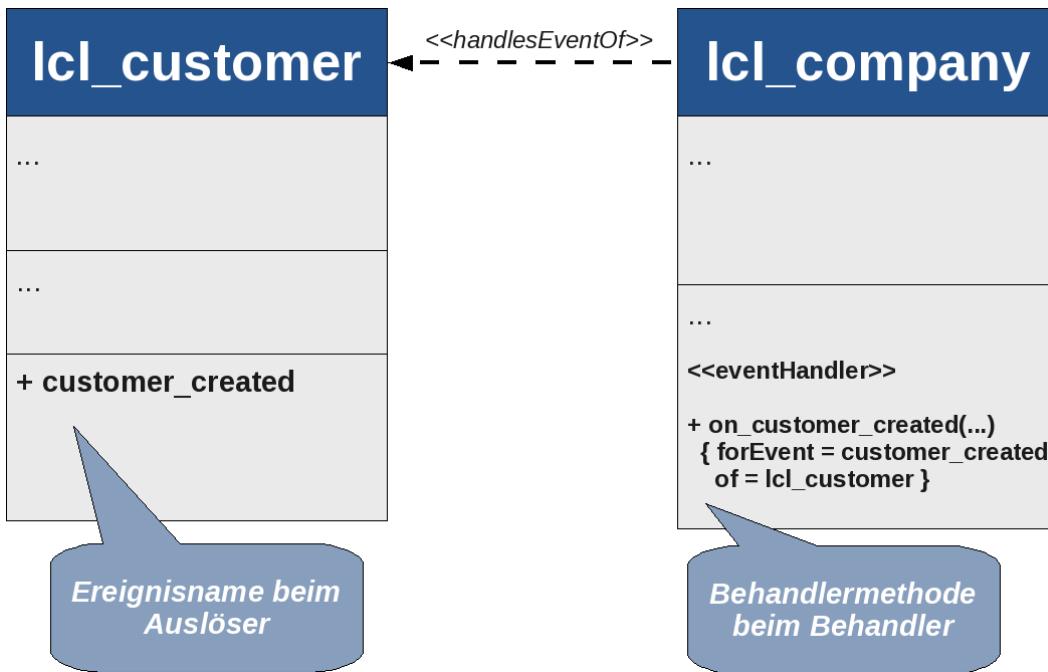


Abbildung 117: Ereignisbehandlung in UML

Die obige Abbildung zeigt die Darstellung von Ereignissen in UML. Der Pfeil vom Behandler zum Auslöser wird mit `<<handlesEventOf>>` beschriftet, die Behandlermethoden mit `<<eventHandler>>` von den normalen Methoden getrennt.

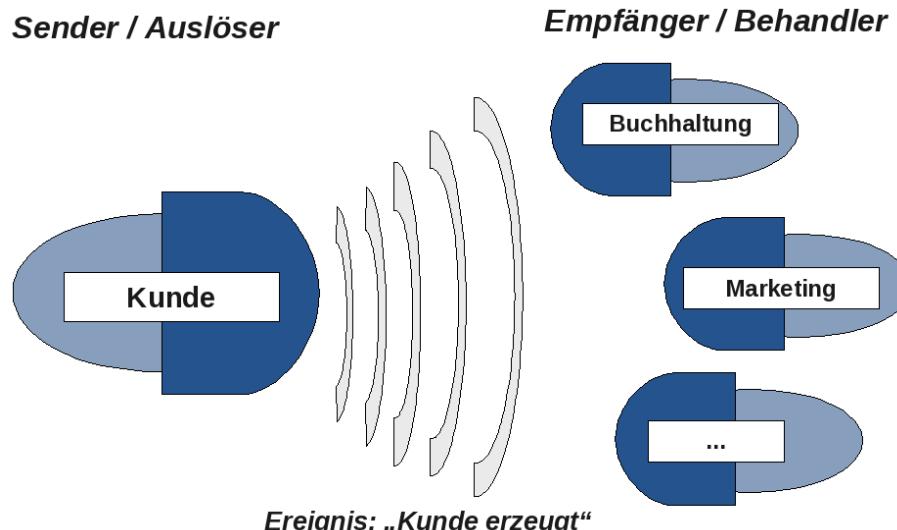


Abbildung 118: Ein Ereignis, viele Behandler

Es gibt sowohl statische Ereignisse, die von der Klasse selbst ausgelöst werden, als auch Instanzereignisse, die von einem Objekt der Klasse ausgelöst werden. Die Definition kann auch in einem Interface erfolgen.

Es sind nun vier Syntaxbereiche zu unterscheiden: Die Definition von Ereignis bzw. Behandlermethode, sowie zur Laufzeit die Registrierung der Behandlermethode und das Auslösen des Ereignisses. Die folgende Matrix zeigt die Syntaxelemente, die hier Verwendung finden.

	Auslöser	Behandler
Definition	<p>Klasse definiert Ereignis (EVENTS, CLASS-EVENTS)</p>	<p>Behandlerklasse definiert und implementiert die Behandlermethode ([CLASS-]METHODS ... FOR EVENT ... OF ...)</p>
Zur Laufzeit	<p>Objekt oder Klasse "lässt" Ereignis aus (RAISE EVENT)</p>	<p>"Behandlerobjekt" oder Behandlerklasse wird zur Laufzeit auf Ereignisse registriert (SET HANDLER)</p>

Abbildung 119: Syntaxelemente zum Auslösen und Behandeln von Ereignissen

Auf der Auslöserseite (in der obigen Matrix die linke Seite) muss zunächst das Ereignis in der Klasse definiert werden. Die Syntax dafür lautet im Fall eines Instanzereignisses:

```
EVENTS eventname [ EXPORTING value(ex_par) TYPE typename ... ].
```

Bei einem Klassenereignis lautet die Syntax hingegen wie folgt:

```
CLASS-EVENTS eventname [ EXPORTING value(ex_par) TYPE typename ... ].
```

In beiden Fällen sind ausschließlich Export-Parameter erlaubt, die als Wertekopie übergeben werden.

Zum Auslösen des Ereignisses wird dann der RAISE EVENT-Befehl benutzt, der folgende Syntax besitzt:

```
RAISE EVENT eventname [ EXPORTING ex_par = par ... ].
```

In diesem Fall gibt es keine syntaktische Unterscheidung zwischen Klassen- und Instanzereignissen. Beachten Sie hierbei das Schlüsselwort EVENT, da es auch einen RAISE-Befehl ohne EVENT gibt, den Sie in einem anderen Zusammenhang noch kennen lernen werden.

Im Kontext des betrachteten Beispiels könnten die Befehle wie in der folgenden Abbildung dargestellt angewendet werden:

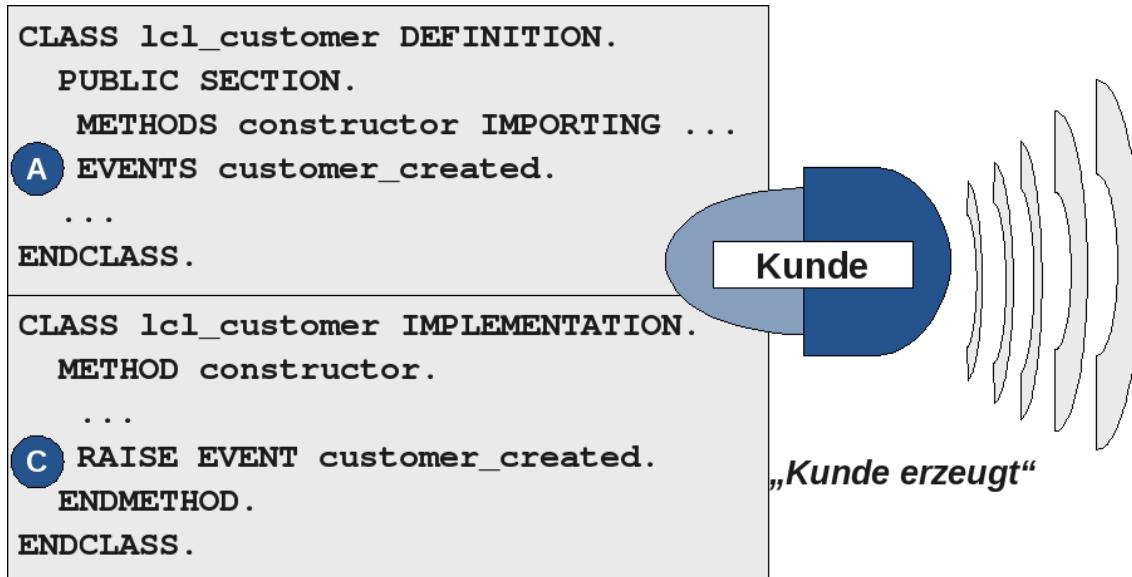


Abbildung 120: Auslöserseite des Szenarios

Die Buchstaben A und C symbolisieren die Felder aus der zuvor gezeigten Matrix. In diesem Beispiel wurde das Ereignis im Public-Sichtbarkeitsbereich definiert. Das hat zur Folge, dass es überall **behandelt** werden darf. Im Gegensatz dazu dürfen private Ereignisse nur in der Klasse selbst, geschützte Ereignisse (Sichtbarkeitsbereich Protected) nur in der Klasse und in Unterklassen behandelt werden.

Auf der Behandlerseite muss zunächst eine Behandlermethode definiert werden. Dies sieht syntaktisch wie folgt aus:

```
METHODS on_eventname FOR EVENT eventname
  OF raising_class | interface
  [IMPORTING ex_par_1 ... ex_par_n [sender] ].
```

Es wird also zur Methoden angegeben, auf welches Event sie sich bezieht und in welcher Klasse oder welchem Interface dieses definiert ist. Als Importing-Parameter können nur die Exporting-Parameter des Ereignisses angegeben werden. Durch die Angabe der Importing-Parameter sind diese dann in der Behandlermethode verfügbar. Eine Typisierung ist hier nicht vorzunehmen, da diese bereits in der Eventdefinition erfolgt ist. Zusätzlich zu den expliziten Parametern kann bei Instanzereignissen noch `sender` als Importparameter verwendet werden. Hierüber erhält die Behandlermethode eine Referenz auf das Objekt, das das Ereignis ausgelöst hat.

Im vorliegenden Szenario könnte dies wie folgt angewendet werden:

```

CLASS lcl_company DEFINITION

...
B METHODS on_customer_created FOR EVENT customer_created
      OF lcl_customer
      IMPORTING sender.
ENDCLASS.

```



Abbildung 121: Behandlerdefinition im Szenario

Durch diese Definition besitzt die Unternehmensklasse eine Methode, die das Ereignis `customer_created` behandeln kann. Eine tatsächliche Behandlung würde so aber noch nicht stattfinden. Erst zur Laufzeit wird durch die **Registrierung** festgelegt, ab welchem Zeitpunkt welche Behandlermethode von welcher Instanz (sofern es sich um eine Instanzmethode handelt) auf ein Ereignis reagieren soll oder nicht mehr reagieren soll. Zu Anfang dieses Abschnitts haben Sie bereits ein Beispiel gesehen, in dem mehrere Behandlerklassen vorgesehen waren. Die Behandlermethoden müssen zur Laufzeit beim Auslöser registriert werden, um dann im Ereignisfall aufgerufen zu werden.

Zur Registrierung von Behandlermethoden wird der `SET HANDLER`-Befehl verwendet. Er hat folgende Syntax:

```
SET HANDLER ref_handler->on_eventname ...
  [ FOR ref_sender | FOR ALL INSTANCES ] [ACTIVATION flag].
```

Hierbei ist `ref_sender` eine Referenz auf ein bestimmtes Objekt, dessen ausgelöste Ereignisse behandelt werden sollen. `FOR ALL INSTANCES` hingegen registriert den Behandler für alle Instanzen. Dabei spielt es keine Rolle, wann diese Instanzen erzeugt werden, die Registrierung gilt also auch für Objekte die erst nach der Registrierung erzeugt werden. Bei statischen Ereignissen entfällt die Angabe ganz.

Die Angabe der `flag` ist beim Registrieren optional (und muss dann 'X' lauten), beim Deregistrieren muss sie erfolgen und ' ' lauten.

Die drei Punkte nach dem Behandler zeigen an, dass mit einem `SET HANDLER`-Aufruf auch gleich mehrere Behandler für dieselben Ereignisse registriert werden können.

Wo die **Registrierung** erfolgen darf (d. h. wo der `SET HANDLER`-Befehl stehen darf), hängt von der Sichtbarkeit der Behandlermethode ab: Bei einer öffentlichen Behandlermethode kann die Registrierung im gesamten Programm erfolgen, bei einer privaten Behandlermethode nur in der Klasse selbst, bei einer geschützten Behandlermethode nur in der Klasse selbst oder deren Unterklassen.

Alle Behandler, die für ein Objekt oder eine Klasse registriert sind, werden in einer systeminternen Tabelle festgehalten. Infolgedessen werden Objekte nicht vom Garbage Collector gelöscht, wenn keine anderweitige Referenz mehr auf sie existiert, solange sie als Behandler registriert sind. Die Reihenfolge der Registrierung ist nicht relevant: Es ist keine Reihenfolge für die Abarbeitung der Behandler eines Ereignisses definiert. Wird jedoch in einer Behandlermethode ein weiterer Behandler für dasselbe Ereignis registriert, wird dieser für das aktuelle Ereignis noch ausgeführt. Die Ausführung der Behandlernethoden erfolgt stets sequentiell.

4.9.1 Praxis: Übung zu Ereignissen

Auch für diese Übung benötigen Sie ein neues Programm. Erstellen Sie dieses mit dem Namen **ZZ_####_COMPANY_5** und wie gewohnt ohne TOP-Include. Erzeugen Sie ein Include **ZZ_####_COMPANY_5_CLASSES** für das Programm, und fügen Sie dort diesmal nicht den Inhalt des letzten Includes, sondern den Codes des Includes **ZZ_####_COMPANY_3_CLASSES** ein.

Ziel dieser Übung ist, das Einfügen von Mitarbeitern in die Abteilung nicht mehr explizit durch Aufruf der add-Methode durchführen zu müssen, sondern implizit beim Anlegen eines Mitarbeiters diesen zu der Abteilung hinzuzufügen, indem ein Ereignis ausgelöst und entsprechend behandelt wird.

Identifizieren Sie die Klasse, in der das Ereignis definiert werden sollte. Definieren Sie dort dann ein öffentliches Ereignis **employee_created**. Definieren Sie ferner eine Behandlernmethode in der Abteilungsklasse. Die Behandlernmethode soll das Objekt, dass das Ereignis ausgelöst hat, in die Liste der Mitarbeiter aufnehmen. Implementieren Sie diese Methode entsprechend. Sorgen Sie anschließend dafür, dass das Ereignis **employee_created** bei der Erzeugung jedes Mitarbeiterobjekts ausgelöst wird.
Speichern, prüfen und aktivieren Sie das Include und wechseln Sie zum Hauptprogramm.

Definieren Sie dort Referenzvariablen vom Typ der Abteilungs-, Innendienstmitarbeiter- und Außendienstmitarbeiterklasse. Instanziieren Sie zunächst die Abteilung. Machen Sie das instanzierte Objekt zum Behandler des Ereignisses. Diese Registrierung soll für alle Mitarbeiterobjekte gelten, die im Programmverlauf erzeugt werden.

Erzeugen Sie dann einige Objekte der beiden Mitarbeiter-Referenzvariablen. Fügen Sie die Mitarbeiter *nicht* explizit der Abteilung hinzu, denn dies sollte jetzt durch die Ereignisbehandlung geschehen. Überprüfen Sie dies, indem Sie die **Ausgabemethode** der Abteilungsklasse aufrufen, Ihr Programm speichern, prüfen, aktivieren und testen.

Sorgen Sie anschließend dafür, dass am Ende des bisherigen Hauptprogramms die Registrierung wieder **aufgehoben** wird. Vergewissern Sie sich, dass das Aufheben funktioniert hat, indem Sie danach ein weiteres Mitarbeiterobjekt erzeugen und erneut die Ausgabemethode der Abteilung aufrufen. Bei dieser Ausgabe sollte das zusätzlich erzeugte Mitarbeiterobjekt nicht erscheinen. Die folgende Abbildung zeigt ein Beispiel für eine mögliche Ausgabe.

Report ZZ_3099_COMPANY_6

Report ZZ_3099_COMPANY_6

*** Ausgabe nach Registrierung ***

Mitarbeiter der Abteilung: Abteilung 1

Aussendienstmitarbeiter:

Name: Andreas Schmidt	Adresse:	Gehalt:	4.300	Geburtsdatum: 01.01.1975
Aussendienstmitarbeiter:				
Name: Gerd Schulz	Adresse:	Gehalt:	5.710	Geburtsdatum: 01.12.1975
Innendienstmitarbeiter:				
Name: Simon Meier	Adresse:	Gehalt:	6.000	Geburtsdatum: 01.06.1980 Büro: A1.11
Innendienstmitarbeiter:				
Name: Armin Müller	Adresse:	Gehalt:	4.455	Geburtsdatum: 01.07.1960 Büro: A1.05

*** Ausgabe nach Deregistrierung und Erzeugung eines weiteren Objekts ***

Mitarbeiter der Abteilung: Abteilung 1

Aussendienstmitarbeiter:

Name: Andreas Schmidt	Adresse:	Gehalt:	4.300	Geburtsdatum: 01.01.1975
Aussendienstmitarbeiter:				
Name: Gerd Schulz	Adresse:	Gehalt:	5.710	Geburtsdatum: 01.12.1975
Innendienstmitarbeiter:				
Name: Simon Meier	Adresse:	Gehalt:	6.000	Geburtsdatum: 01.06.1980 Büro: A1.11
Innendienstmitarbeiter:				
Name: Armin Müller	Adresse:	Gehalt:	4.455	Geburtsdatum: 01.07.1960 Büro: A1.05

Abbildung 122: Ausgabe des Programms: SAP-System-Screenshot

4.9.2 Kontrollfragen

1. Ist in ABAP Mehrfachvererbung erlaubt?
2. Wie drücken Sie syntaktisch aus, dass eine Klasse von einer anderen Klasse erbt?
3. Wie bezeichnet man eine Zuweisung, bei der der Typ der Referenzvariable, von der zugewiesen wird, eine Unterklasse des Typs der Referenzvariable ist, der zugewiesen wird?
4. Was ist bei einem Down-Cast zu beachten?
5. Kann eine abstrakte Klasse instanziert werden?
6. Kann eine abstrakte Klasse konkrete Methoden enthalten?
7. Kann eine konkrete Klasse abstrakte Methoden enthalten?
8. Kann ein Interface eine Methodenimplementierung enthalten?
9. Können Interfaces selbst Interfaces einbinden?
10. Kann eine Referenzvariable mit einem Interface statt einer Klasse typisiert werden?
11. Ist eine Zuweisung zwischen zwei mit unterschiedlichen Interfaces typisierten Referenzvariablen möglich, wenn die Interfaces nicht miteinander in Beziehung stehen?
12. Kann es in einer Klasse zu einer Methode mehrere Signaturen mit unterschiedlichen Parameterlisten geben?
13. Welchen Einfluss hat der Sichtbarkeitsbereich eines Ereignisses?
14. Welchen Einfluss hat der Sichtbarkeitsbereich einer Behandlermethode?
15. Wie kann eine Behandler-Registrierung rückgängig gemacht werden?

Die Antworten finden Sie auf der folgenden Seite.

4.9.3 Antworten

1. Nein
2. INHERITING FROM oberklassenname in der Klassendefinition
3. Up-Cast
4. Möglicherweise ist die Zuweisung zur Laufzeit nicht zulässig, Operator ?=, einbetten in TRY-Block
5. Nein
6. Ja
7. Nein
8. Nein
9. Ja
10. Ja
11. Ja, es handelt sich um einen Down-Cast, der erfolgreich ist, wenn das referenzierte Objekt über beide Interfaces verfügt
12. Nein
13. Er legt fest, wo das Ereignis behandelt werden darf
14. Er legt fest, wo die Registrierung durchgeführt werden darf
15. Mit dem SET HANDLER-Befehl und ACTIVATION ' '

4.10 Kapitelabschluss

Sie befinden sich am Ende dieses Abschnitts. Bevor sie die im folgenden Absatz beschriebene E-Mail verfassen, beachten Sie bitte die folgenden Hinweise:

1. Prüfen Sie, ob sie wirklich alle Aufgaben ab dem Kursbeginn bearbeitet haben. Diese sind mit „Praxis:“ in der Überschrift gekennzeichnet (3.7, 4.7.3, 4.8.2, 4.8.4, 4.9.1).
2. Prüfen Sie bitte noch einmal genau ob alle ihre Repository-Objekte korrekt funktionieren
3. Stellen Sie sicher, dass alle Repository-Objekte aktiviert sind. Um Objekte zu finden, die noch nicht aktiviert sind, wählen Sie aus dem Drop-Down-Menü oberhalb des Navigationsbaums im Object Navigator **Inaktive Objekte** aus. Geben Sie anschließend im darunter befindlichen Feld ihren Benutzernamen **USER#-###** ein und bestätigen Sie. Anschließend werden im Navigationsbaum die inaktiven Objekte dargestellt, die noch aktiviert werden müssen. Aktivieren Sie diese nun. Beachten Sie, dass sie die Zweige des Baums ggf. noch aufklappen müssen. Um zu ihrem Paket zurückzukehren, wählen Sie im Drop-Down-Menü wieder **Paket** aus und bestätigen Sie ihren Paketnamen.
4. Stellen Sie weiterhin sicher, dass die Namen ihrer Entwicklungsobjekte genau den Vorgaben im Skript entsprechen. Sollten Sie sich vertippt haben, können Sie Programme umbenennen, indem Sie diese mit der rechten Maustaste im Navigationsbaum des Object Navigators anklicken und **Umbenennen...** auswählen.

Wenn Sie den Kurs bis zu dieser Stelle bearbeitet haben, senden Sie bitte eine formlose E-Mail an die vom Kursbetreuer für diesen Kurs genannte Adresse mit dem Betreff „ABAP2: Abschluss Kapitel 4 User #####“ (die Anführungszeichen gehören nicht mit zum Betreff). Sie erhalten dann in Kürze Feedback (je nach Ergebnis entweder über den Fortschrittsbericht, wenn alles in Ordnung ist, oder per E-Mail, wenn noch Korrekturen nötig sind) und können Mängel ggf. noch nachbessern. Bitte achten Sie darauf, den Betreff genau wie angegeben zu formulieren, um eine effiziente Verarbeitung der Mail zu ermöglichen.

Sollten Sie Fragen haben, formulieren Sie diese bitte in einer **separaten E-Mail** mit aussagekräftigem Betreff, da die Kapitelabschlussmails meist nur über den Betreff verarbeitet werden!