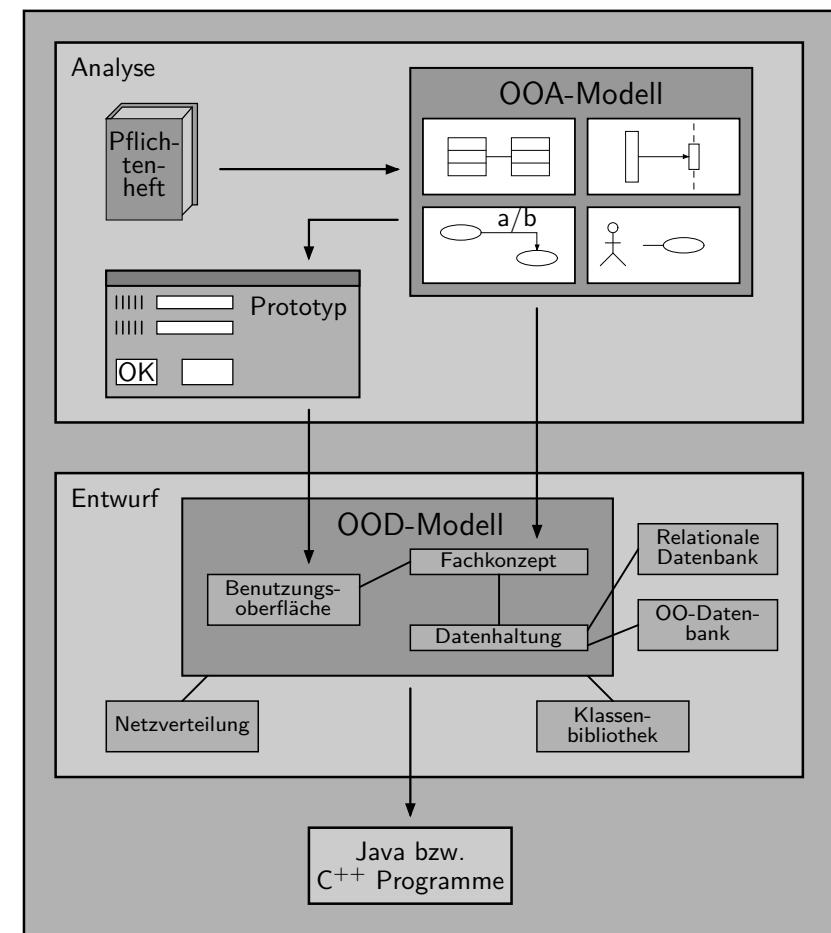


Gregor Tielsch

Systemanalyse

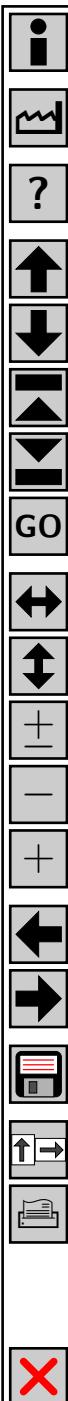
WWI 16 SEA

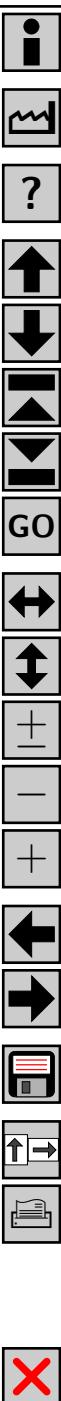
Studienhalbjahr 2



INHALTSVERZEICHNIS ZUR VORLESUNG SYSTEMANALYSE

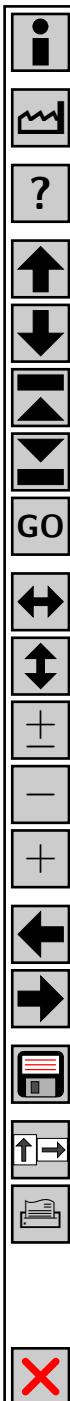
Literatur	8
Objektorientierte Software-Entwicklung im Überblick	11
Der Analyseprozess	80
Basiskonzepte der Objektorientierten Analyse	117
Statische Konzepte der objektorientierten Analyse	180
Dynamische Konzepte der objektorientierten Analyse	219
Analyseschritte für Geschäftsprozesse	308
Gesamtglossar	325



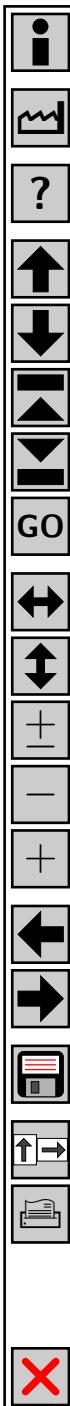


Literatur	8
Objektorientierte Software-Entwicklung im Überblick	11
Lernziele	11
Die Phasen der Software-Entwicklung [LL98]	13
Der Software-Lebenszyklus	13
Modelle für den Software-Entwicklungsprozess	19
Testtechniken	37
Was ist eine objektorientierte Methode?	42
Der Methodenbegriff	42
Objektorientierte Methoden	44
Die Methode nach [Bal99]	49
Objektorientierte Analyse	53
Aufgabenstellung der Systemanalyse	53
Das OOA-Modell	56
Produkte der Analysephase	58
Erstellung des OOA-Modells	65
Objektorientierter Entwurf	67
Aufgabenstellung des Systementwurfs	67
Die Drei-Schichten-Architektur	68
Die Produkte der Entwurfsphase	73
Zusammenfassung	76

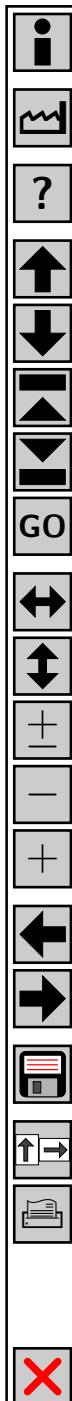
Glossar	77
Der Analyseprozess	80
Lernziele	80
Grundlage: Das Pflichtenheft	81
Überblick	81
Gliederungsschema für ein Pflichtenheft	82
Methodische Vorgehensweise	93
Überblick	93
Die Struktur des Analyseprozesses	96
Zu beachtende Grundsätze	98
Balancierter Makroprozess	99
Struktur des verwendeten Analyseprozesses	99
Aufgabenbereiche des balancierten Makroprozesses	105
Alternative Makroprozesse	111
Häufige Fehler beim Analyseprozess	113
Zusammenfassung	115
Glossar	116
 Basiskonzepte der Objektorientierten Analyse	117
Lernziele	117
Objekte	119



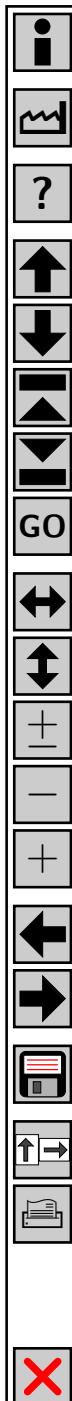
Begriffsdefinitionen	119
UML-Notation für Objekte	123
Objektidentität, -gleichheit und -namen	128
Das Geheimnisprinzip	132
Externe und interne Objekte	134
Klassen	136
Begriffsdefinitionen	136
UML-Notation für Klassen	139
Abstrakte Klassen	145
Klassenzugehörigkeit und Objektverwaltung	147
Attribute	150
Begriffsdefinition	150
UML-Notation für Attribute	152
Klassenattribute	154
Attributtypen	155
Operationen	158
Begriffsdefinition	158
Operationstypen	159
UML-Notation für Operationen	166
Externe und interne Operationen	169
Klassifikation von Operationen nach ihrem Verwendungszweck	170
Basis- und Verwaltungsoperationen	173



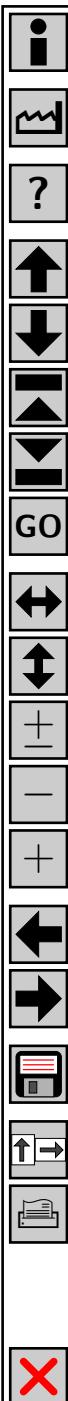
Spezifikation von Operationen	175
Zusammenfassung	176
Glossar	177
 Statische Konzepte der objektorientierten Analyse	 180
Lernziele	180
Assoziationen	182
Begriffsdefinitionen	182
UML-Notation für binäre und reflexive Assoziationen	186
Rollen	192
Assoziative Klassen	195
Aggregation und Komposition	197
Assoziationen in Objektdiagrammen	201
Das Vererbungskonzept	202
Definition einer Klassenhierarchie	202
Der Mechanismus der Vererbung	210
Bewertung des Vererbungskonzepts	215
Zusammenfassung	216
Glossar	217
 Dynamische Konzepte der objektorientierten Analyse	 219
Lernziele	219

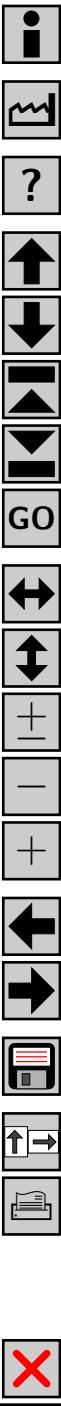


Geschäftsprozess	221
Der Begriff des Geschäftsprozesses	221
Spezifikation von Geschäftsprozessen	229
Das Geschäftsprozessdiagramm	239
Botschaft	246
Szenario	249
Begriffsdefinition	249
Sequenzdiagramm	251
Kommunikationsdiagramm	264
Kommunikationsdiagramm im Vergleich zum Objektdiagramm	269
Sequenzdiagramm im Vergleich zum Kommunikationsdiagramm	271
Zustandsdiagramm	273
Zustandsautomat	273
Darstellung von Zustandsautomaten durch Zustandsdiagramme	276
Aktivitätsdiagramm	297
Eigenschaften	297
UML-Notation für Aktivitätsdiagramme	299
Beispiel: Spezifikation eines Geschäftsprozesses	301
Zusammenfassung	303
Glossar	305
 Analyseschritte für Geschäftsprozesse	308



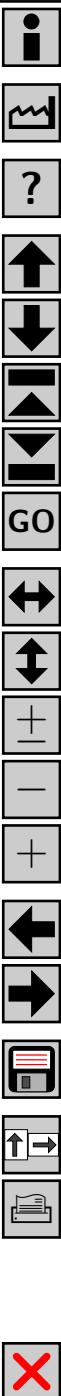
Lernziele	308
Identifikation von Geschäftsprozessen	309
Konstruktive Schritte	309
Analytische Schritte zur Validierung von Geschäftsprozessen	320
Gesamtglossar	325



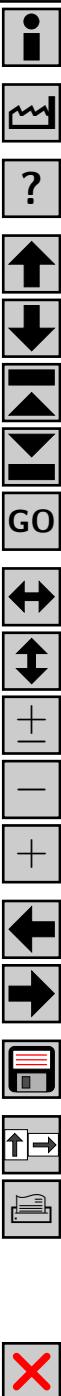


LITERATUR

- [Bal99] BALZERT, HEIDE: *Lehrbuch der Objektmodellierung: Analyse und Entwurf*. Spektrum Akademischer Verlag, 1999. ISBN 978-3-8274-0285-1.
- [Bal00] BALZERT, HELMUT: *Lehrbuch der Software-Technik: Software-Entwicklung*. Spektrum Akademischer Verlag, 2. Auflage, 2000. ISBN 978-3-8274-0480-0.
- [Bal04] BALZERT, HEIDE: *Lehrbuch der Objektmodellierung: Analyse und Entwurf mit der UML 2*. Spektrum Akademischer Verlag, 2. Auflage, 2004. ISBN 978-3-8274-1162-4.
- [Ben56] BENINGTON, H. D.: *Production of Large Computer Programs*. In: *Proceedings of the ONR Symposium on Advanced Programming Methods for Digital Computers*, Seiten 15–27, 1956.
- [Boe81] BOEHM, BARRY W.: *Software Engineering Economics*. Prentice-Hall, 1981. ISBN 0138221227.
- [Boo91] BOOCHE, GRADY: *Object-Oriented Design with Applications*. The Benjamin/Cummings Publishing Company, Redwood City, 1991. ISBN 0-8053-5340-2.
- [Boo94] BOOCHE, GRADY: *Object-Oriented Analysis and Design with Applications*. The Benjamin/Cummings Publishing Company, Redwood City, 2. Auflage, 1994.
- [Boo96] BOOCHE, GRADY: *Object Solutions, Managing the Object-Oriented Project*. Addison-Wesley, Menlo Park, California, 1996.



- [Coc97] COCKBURN, A.: *Structuring Use Cases with Goals*, 1997.
www.members.aol.com/acockburn/papers/usecases.htm (8-4-2001).
- [CY91a] COAD, P. und E. YOURDON: *Object-Oriented Analysis*. Yourdon Press, Prentice Hall, Englewood Cliffs, second Auflage, 1991.
- [CY91b] COAD, P. und E. YOURDON: *Object-Oriented Design*. Yourdon Press, Prentice Hall, Englewood Cliffs, second Auflage, 1991.
- [Fow97] FOWLER, MARTIN: *Analysis Patterns: Reusable Object Models*. Addison Wesley Longman, 1997.
ISBN 0-201-89542-0.
- [Jac95] JACOBSON, IVAR: *The Use-Case Construct in Object-Oriented Software Engineering*. In: CARROLL, JOHN M. (Herausgeber): *Scenario-Based Design: Envisioning Work and Technology in System Development*. John Wiley & Sons, New York, 1995. ISBN 0-471-07659-7.
- [JCJÖ92] JACOBSON, I., M. CHRISTERSON, P. JONSSON und G. ÖVERGAARD: *Object-Oriented Software Engineering – A Use Case Driven Approach*. Addison Wesley, Wokingham, 1992.
ISBN 0-201-54435-0.
- [JRH⁺04] JECKLE, MARIO, CHRIS RUPP, JÜRGEN HAHN, BARBARA ZENGLER und STEFAN QUEINS: *UML 2 glasklar*. Carl Hanser Verlag, 2004. ISBN 3-446-22575-7.
- [Kec06] KECHER, CHRISTOPH: *UML 2.0: Das umfassende Handbuch*. Galileo Press, 2. Auflage, 2006.
ISBN 3-89842-738-8.



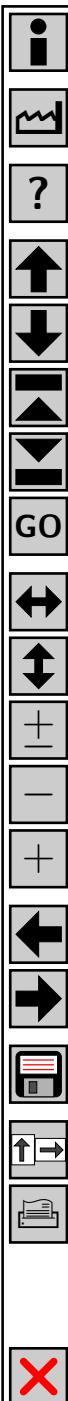
- [LL98] LEWIS, JOHN und WILLIAM LOFTUS: *Java Software Solutions: Foundations and Program Design*. Addison Wesley, 1998. ISBN 0-201-57164-1.
- [Rat97] RATIONAL SOFTWARE CORPORATION, Santa Clara: *Unified Modeling Language 1.1 – UML Summary, Notation Guide, UML Semantics, Object Constraint Specification*, September 1997. www.rational.com/uml.
- [RBP⁺91] RUMBAUGH, JAMES R., MICHAEL R. BLAHA, WILLIAM PREMERLANI, FREDERICK EDDY und WILLIAM LORENSEN: *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, 1991. ISBN 0-13-629841-9.
- [Roy70] ROYCE, W. W.: *Managing the Development of Large Software Systems*. In: *Proc. IEEE WESTCON (Nachdruck in: Proceedings of the 9th International Conference on Software Engineering, Seiten 328-338, Monterey, USA, 1987)*, Seiten 1–9, Los Angeles, USA, 1970.
- [SM88] SHLAER, S. und S. MELLOR: *Object-Oriented System Analysis – Modeling the World in Data*. Yourdon Press, Prentice Hall, Englewood Cliffs, 1988.
- [SM92] SHLAER, S. und S. MELLOR: *Object Lifecycles – Modeling the World in States*. Yourdon Press, Prentice Hall, Englewood Cliffs, 1992.

OBJEKTOIENTIERTE SOFTWARE-ENTWICKLUNG IM ÜBERBLICK

Lernziele

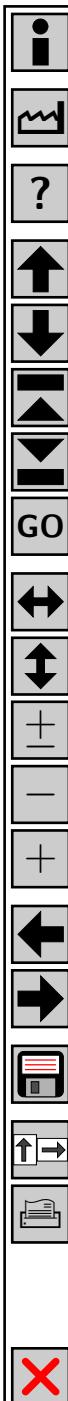
Wissen

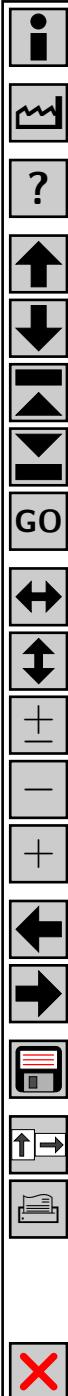
- Die Phasen der Software-Entwicklung
- Historische Entwicklung der Objektorientierung
- Was ist objektorientierte Systementwicklung
- Die Begriffe **Objektorientierte Analyse** (OOA) und **Objektorientierter Entwurf** (OOD)
- Welche Konzepte verwendet die Objektorientierung



Verständnis

- Warum ist die Anwendung der Objektorientierung sinnvoll
- Was ist eine Methode (zur objektorientierten Software-Entwicklung)
- Wie unterscheiden sich die Phasen **Analyse** und **Entwurf**
- Welche Ergebnisse soll die Analysephase liefern
- Welche Ergebnisse soll die Entwurfsphase liefern





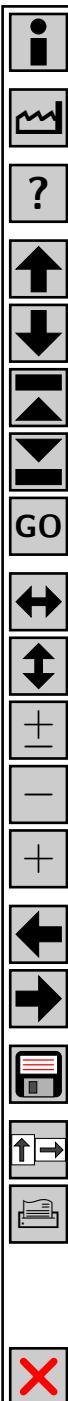
Die Phasen der Software-Entwicklung [LL98]

Der Software-Lebenszyklus

Überblick

- **Definition 1:** **Software-Lebenszyklus** (*Software Life Cycle*)

Der **Software-Lebenszyklus** beschreibt die Phasen eines Programms (bzw. Software-Systems) von der **ersten Konzeption** bis zur **endgültigen Stilllegung**.



- Abb. 1 zeigt den Lebenszyklus eines Programms

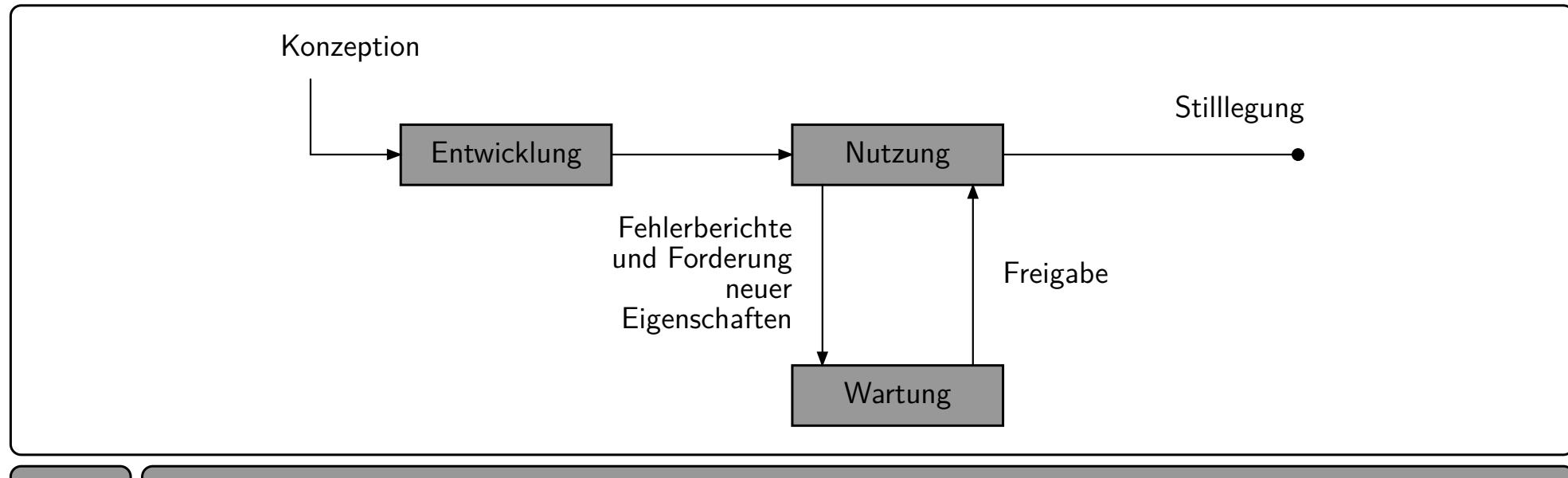


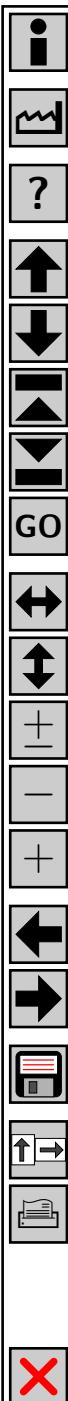
Abb. 1 Der Lebenszyklus eines Programms

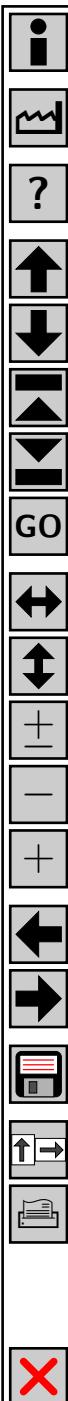
- Fundamentale Stadien eines Programms
 - ***Entwicklung*** (*Development*)
 - ***Nutzung*** (*Use*)
 - ***Wartung*** (*Maintenance*)

- Entwicklung
 - Erzeugung eines ablauffähigen Programms
 - Die Programmversion wird an den Anwender übergeben
→ **Release**-Übergabe
- Nutzung
 - Feststellung von Programmfehlern → *Defect Report*
 - Vorschläge für die Realisierung zusätzlicher Programmeigenschaften → *Feature Request*
- Wartung
 - Modifikation eines Programms zur Fehlerkorrektur und/oder Funktionserweiterung
 - Die Änderungen werden an einer Programmkopie vorgenommen
 - Das Ergebnis ist eine neue Programmversion und -freigabe



- **Nicht für jede Fehlerkorrektur oder Funktionserweiterung wird automatisch eine neue Programmversion freigegeben**
- **Oft werden neben der aktuellen Version auch weiterhin ältere Programmversionen benutzt**





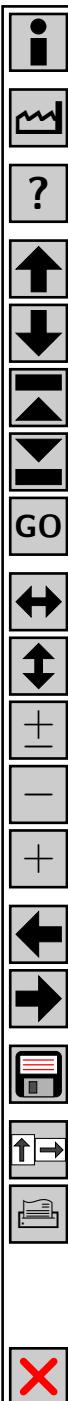
Der Faktor Zeit im Lebenszyklus eines Programms

- Die Länge des Lebenszyklus eines Programms hängt von verschiedenen Faktoren ab
 - Anwendungszweck
 - Nützlichkeit
 - Konstruktionsgüte
- Der Anteil des Entwicklungszeitraums variiert stark
 - Von **wenigen Wochen**
 - Bis hin zu **mehreren Jahren**

Ähnliche Werte gelten für die Nutzungsdauer und die Wartungszeiträume



- **Insbesondere bei langen Nutzungsdauern sind die für die Entwicklung und Wartung verantwortlichen Personen nicht mehr identisch**
- **Folge:** **Der Erfolg der Wartungsaufgabe hängt entscheidend von der Verständlichkeit des Programms ab**
 - **Was ist das Problem?**
 - **Wie und wo muss das Programm korrigiert werden, um das Problem zu lösen?**

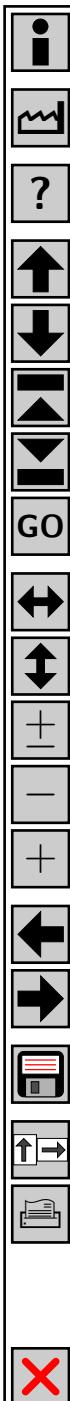


Die Rolle der Entwicklungsphase

- Förderung der Programmverständlichkeit
 - Präzise definierte Anforderungen
 - Sorgfältige Durchführung der Analyse und des Entwurfs
 - Klar strukturierte Implementierung
 - Vollständige Dokumentation



Je *komplexer* ein Programm aufgebaut ist, desto *leichter* schleichen sich während der Entwicklung Fehler ein



- Abb. 2 zeigt die Rolle der Entwicklungsphase
 - Erhöhte Investitionen in die Entwicklung führen oft zu einer deutlichen Reduktion der Wartungskosten
 - Beispiel: Eine sorgfältige Dokumentation erleichtert eine spätere Einarbeitung

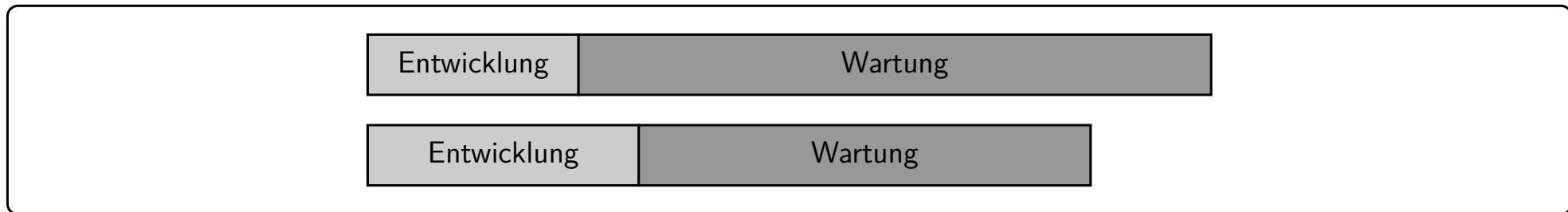


Abb. 2 Die Rolle der Entwurfsphase



Ziel bei der Programmentwicklung **muss** es sein, die **Summe** aus der **Entwicklungs- und Wartungszeit (und damit Kosten)** zu **minimieren**

Modelle für den Software-Entwicklungsprozess

Das Wasserfallmodell

- Eines der ersten Modelle für den Software-Entwicklungsprozess war das **Wasserfallmodell** (*Waterfall Model*) → Abb. 3 zeigt die bereits 1956 entwickelte Urform des Wasserfallmodells (damals noch als **Stagewise Model** bezeichnet [Ben56])

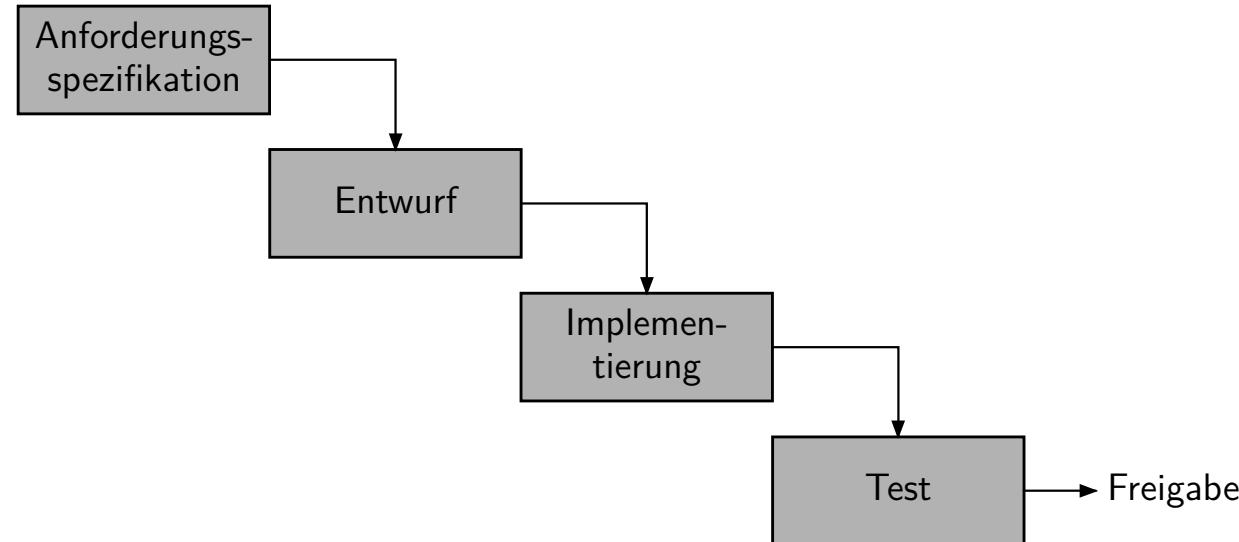
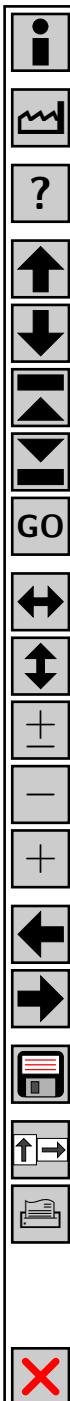


Abb. 3 Das Wasserfallmodell in seiner Urform (Stagewise Model)

- i
- w
- ?
- ↑
- ↓
- █
- ▲
- ▼
- GO
- ↔
- ↕
- ±
- +
- ←
-
- 💾
- ↑→
- 🖨️
- X

- Struktur des Stagewise Modells
 - Die in [Abb. 3](#) beschriebenen Phasen werden ***genau einmal linear*** durchlaufen
 - Zu einer ***früheren*** Phase des Entwicklungsprozesses darf nicht zurückgekehrt werden
- Das von ROYCE in [\[Roy70\]](#) vorgeschlagene Wasserfallmodell erweitert das Stagewise-Modell um ***Rückkopplungsschleifen***
 - Rückkopplungen werden auf angrenzende Phasen eingeschränkt
 - Damit werden teure Überarbeitungen über mehrere Phasen hinweg vermieden



- Abb. 4 zeigt das Wasserfallmodell mit seinen von ROYCE vorgesehenen Phasen

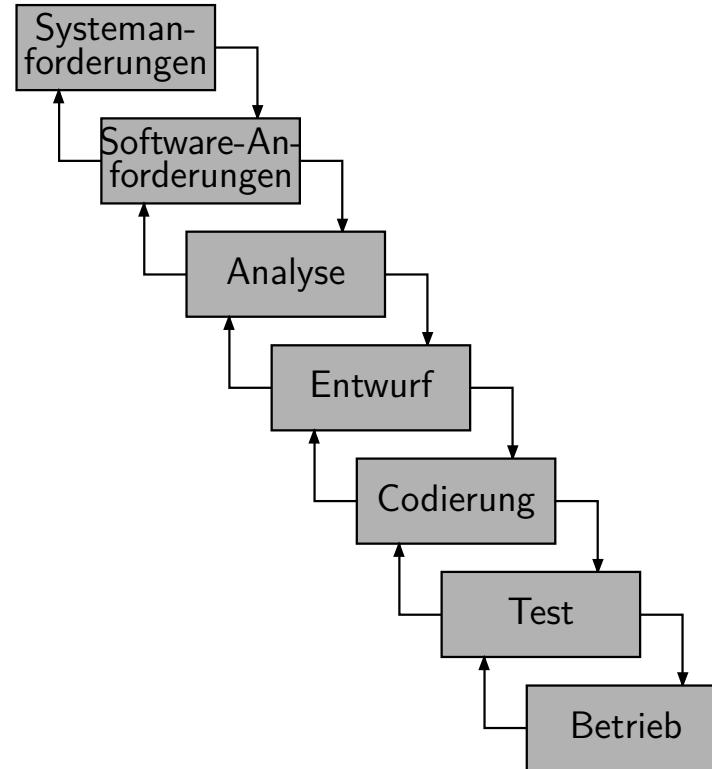


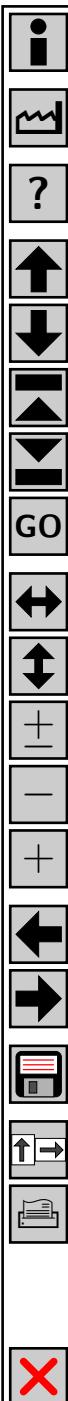
Abb. 4 Das Wasserfallmodell nach ROYCE

 Dieses Modell wird (erst) von BOEHM in [Boe81] als Wasserfallmodell bezeichnet, da die Ergebnisse einer Phase wie bei einem Wasserfall in die nächste Phase fallen



Die folgende Beschreibung bezieht sich auf die Phasen aus Abb. 3

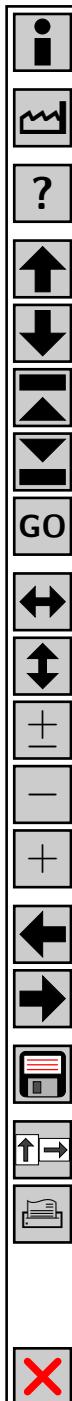
- Anforderungsspezifikation (*Establish requirements*)
 - Zielsetzung
 - ◊ Spezifikation, **was** das Programm leisten soll → **Analyse**
 - ◊ **Wie** das Programm die gewünschten Aufgaben erfüllt, ist in dieser Phase unerheblich
 - Typische Anforderungen
 - ◊ Festlegung der Programmfunktionalität
 - ◊ Struktur der **Benutzungsschnittstelle** (*User Interface*)
 - Spezifikation des nach außen sichtbaren Systemverhaltens
 - ◊ Vorgabe von Restriktionen, z. B.
 - ▷ *Wann muss das Programm fertig sein*
 - ▷ *Welche Programmausführungszeiten müssen eingehalten werden*
 - ▷ *Welche Hardware-Vorgaben sind zu berücksichtigen*



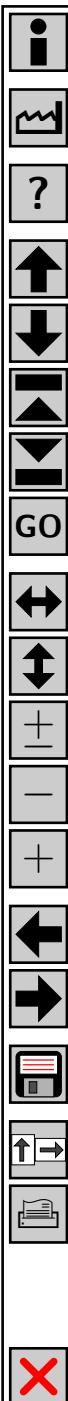
- Entwurf (*Create Design*)
 - Zielsetzung
 - ◊ Spezifikation, **wie** das Programm die Anforderungen erfüllen soll
 - Ergebnisse bei Verwendung einer **objektorientierten Methode**
 - ◊ *Welche Klassen werden benötigt*
 - ◊ *Welche Beziehungen bestehen zwischen den Klassen*
 - ◊ *Welche Operationen können auf den Objekten der einzelnen Klassen ausgeführt werden*
 - ◊ *Wie arbeiten die Objekte zusammen*
- Implementierung (*Implement Code*)
 - Umsetzung des Entwurfs unter Verwendung einer **geeigneten Programmiersprache** in ein ablauffähiges Programm



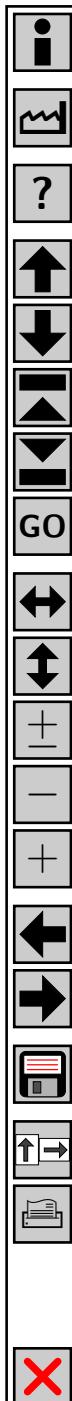
- **Geeignet** soll bedeuten, dass die Entwurfstechnik und das Programmierparadigma der verwendeten Programmiersprache zusammenpassen sollten
- Im Prinzip ließe sich ein **objektorientierter Entwurf** auch mit einer **imperativen Programmiersprache** (z. B. C oder PASCAL) realisieren → das ist aber keine gute Idee!

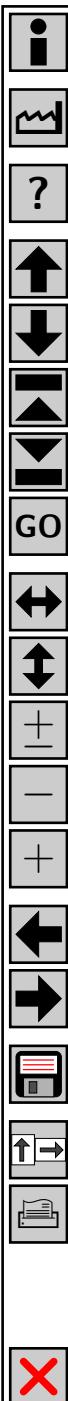


- Test (*Test System*)
 - Zielsetzung
 - ◊ Fehlersuche
 - Vorgehensweise
 - ◊ Ermittlung der **relevanten** Testfälle
 - ▷ Definition der Eingabewerte für das Programm
 - ▷ Ermittlung der für den jeweiligen Testfall erwarteten Ausgabewerte
 - ◊ Durchführung der Testläufe
 - ◊ Auswertung der Testläufe → Vergleich der **tatsächlichen** Ergebnisse mit den **erwarteten** Ergebnissen



- Kritik am Wasserfallmodell
 - Eine abgeschlossene Phase kann (zumindest aus der Sicht des Entwicklungsprozesses) nicht mehr **überarbeitet** werden
 - ⚠ Die Phasen 1 bis n sind abgeschlossen, wenn man sich in der Phase n+2 befindet
 - ◊ Ein extrem hoher Aufwand ist erforderlich, um die **Vollständigkeit** einer Phase sicherzustellen
 - ⚠ Alle relevanten Aspekte müssen berücksichtigt worden sein
 - ◊ In der Praxis ist es (nahezu) unmöglich, die einzelnen Phasen derart auszuführen, dass in einer Phase getroffene Entscheidungen später nicht eventuell doch wieder rückgängig gemacht werden müssen
 - ⚠ Die Ergebnisse der Folgephasen hängen natürlich von Entscheidungen früherer Phasen ab
 - Jede Phase muss immer in ihrer vollen Breite ausgeführt werden, d. h. es ist nicht möglich, sich zunächst auf ein Kernsystem zu konzentrieren und dieses nach und nach zu erweitern → **evolutionäre** Vorgehensweise





Das iterative Modell

- Abb. 5 zeigt eine Weiterentwicklung des Wasserfallmodells, das **iterative Modell**

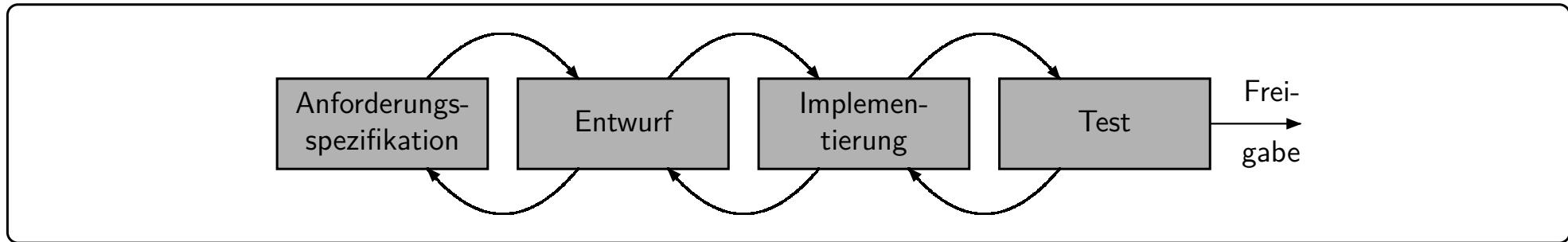
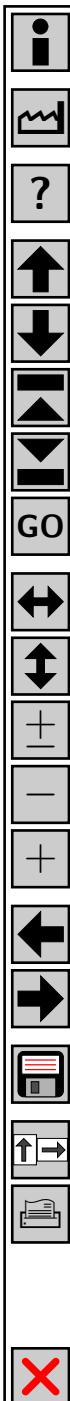
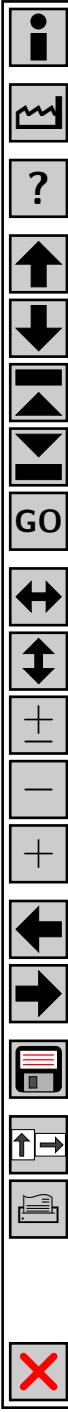


Abb. 5 Das iterative Modell

- Struktur des iterativen Modells
 - Es werden diesselben Phasen wie im Wasserfallmodell durchlaufen
 - Eine Rückkehr in bereits abgeschlossene Phasen ist möglich



- Vorteile
 - Werden in einer späteren Phase neue Aspekte entdeckt, die z. B. die Anforderungs- oder Entwurfsphase betreffen, so erlaubt dieses Modell formal die Rückkehr in die betroffenen Phasen
 - Durch die Möglichkeit zur Änderung früherer Phasen ist das Modell wesentlich realistischer als das Wasserfallmodell
 - Nachteil
 - Es besteht die Gefahr, dass die ersten Phasen nicht mehr **so ernst** genommen werden, da eine Rückkehrsmöglichkeit besteht
-  Je **früher** in einem Entwicklungsprozess Fehlentscheidungen getroffen werden, desto **teurer** wird deren Korrektur



Das iterative Modell mit Prototyperstellung und Bewertung

- **Definition 2: Prototyp**

Ein **Prototyp** ist ein Programm, das geschrieben wurde, um ein bestimmtes **Konzept zu sondieren**.

- Beispiele für die Verwendung von Prototypen

- Generierung einer ersten Version der Benutzungsschnittstelle:

Ist der Kunde mit der generellen Struktur zufrieden?

- Test einer bisher (vom Anwendungsentwickler) selten oder gar nicht verwendeten Klassenbibliothek:

Sind die zur Verfügung gestellten Klassen zur Realisierung einer bestimmten Anforderung geeignet?

- Test, ob eine vorgegebene Anforderung überhaupt erfüllbar ist:

Kann der geforderte Gleichungslöser innerhalb von 100 msec eine quadratische Gleichung lösen und das Ergebnis ausdrucken ?



Prototypen können auch erfolgreich eingesetzt werden, um am Projektende Aussagen des Auftraggebers folgenden Inhalts zu vermeiden:

*Ich weiß, dass das Ergebnis dem entspricht, was ich gesagt habe, aber das ist nicht das Ergebnis, was ich eigentlich im Sinn hatte
(I know that's what I said I wanted, but that's not what I meant)*

- Abb. 6 zeigt eine Weiterentwicklung des iterativen Modells, das **iterative Modell mit Prototyperstellung und Bewertung**

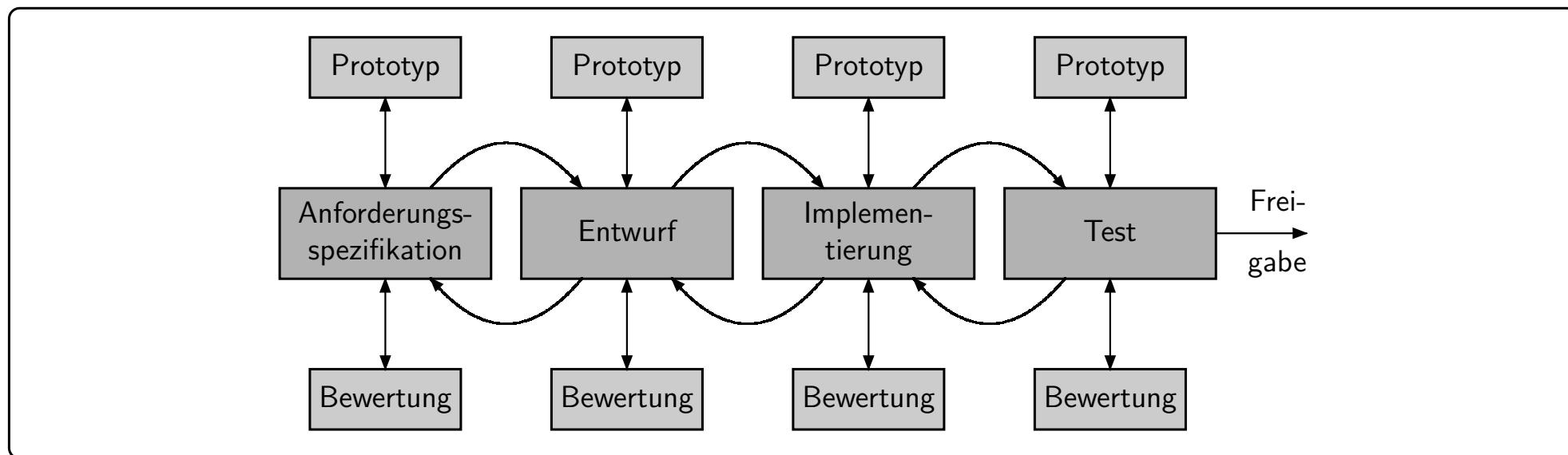
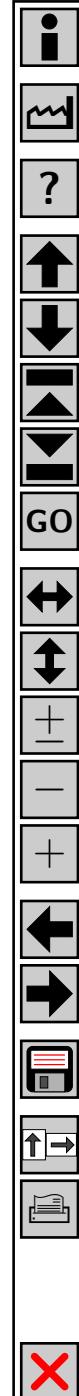
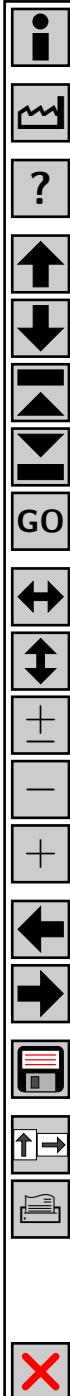


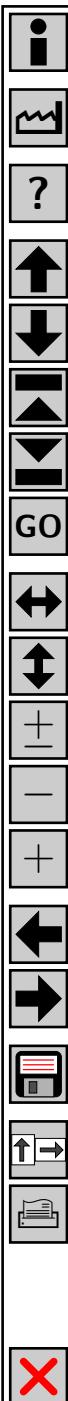
Abb. 6 Das iterative Modell mit Prototyperstellung und Bewertung

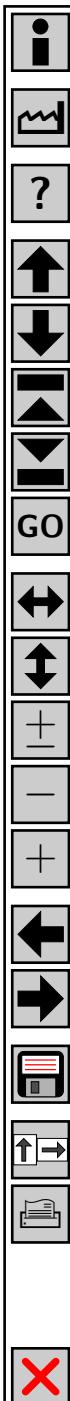




- Struktur des iterativen Modells mit Prototypenstellung und Bewertung
 - Es werden dieselben Phasen wie im iterativen Modell durchlaufen
 - In jeder Phase ist es **optional** möglich, Prototypen zu erstellen
 - Anhand der **Beurteilung** des Prototypen kann entschieden werden, ob die zugehörigen Entwurfs- oder Implementierungsalternativen weiter verwendet oder verworfen werden sollen
 - Jede Phase wird mit einer **Bewertung** abgeschlossen
- Bewertungskriterien
 - Anforderungsphase
 - ◊ Sind die Anforderungen vollständig?
 - ◊ Sind die Anforderungen in sich konsistent?
 - ◊ Sind die Anforderungen präzise formuliert?
 - Entwurfsphase
 - ◊ Wurde jede Anforderung im Entwurf berücksichtigt?
 - Implementierungsphase
 - ◊ Wurde der Entwurf gewissenhaft umgesetzt?
 - Testphase
 - ◊ Enthält die Testphase alle relevanten Testfälle?

- Die **Code-Walkthrough**-Technik zur Entwurfs- und Implementierungsbewertung
 - Im Rahmen einer Besprechung **lesen mehrere** Projektbeteiligte den Entwurf bzw. die Implementierung durch und diskutieren die wesentlichen Sachverhalte
 - Daneben kann auch kontrolliert werden, inwieweit die Implementierung vorgegebenen Codierungs-Konventionen entspricht, z. B.
 - ◊ Kommentierung
 - ◊ Einrückung
 - ◊ Namensgebung





Ein iterativer Gesamtansatz

- Neben einer Iteration zwischen den verschiedenen Phasen kann auch das Gesamtmodell mehrfach iteriert werden → siehe Abb. 7

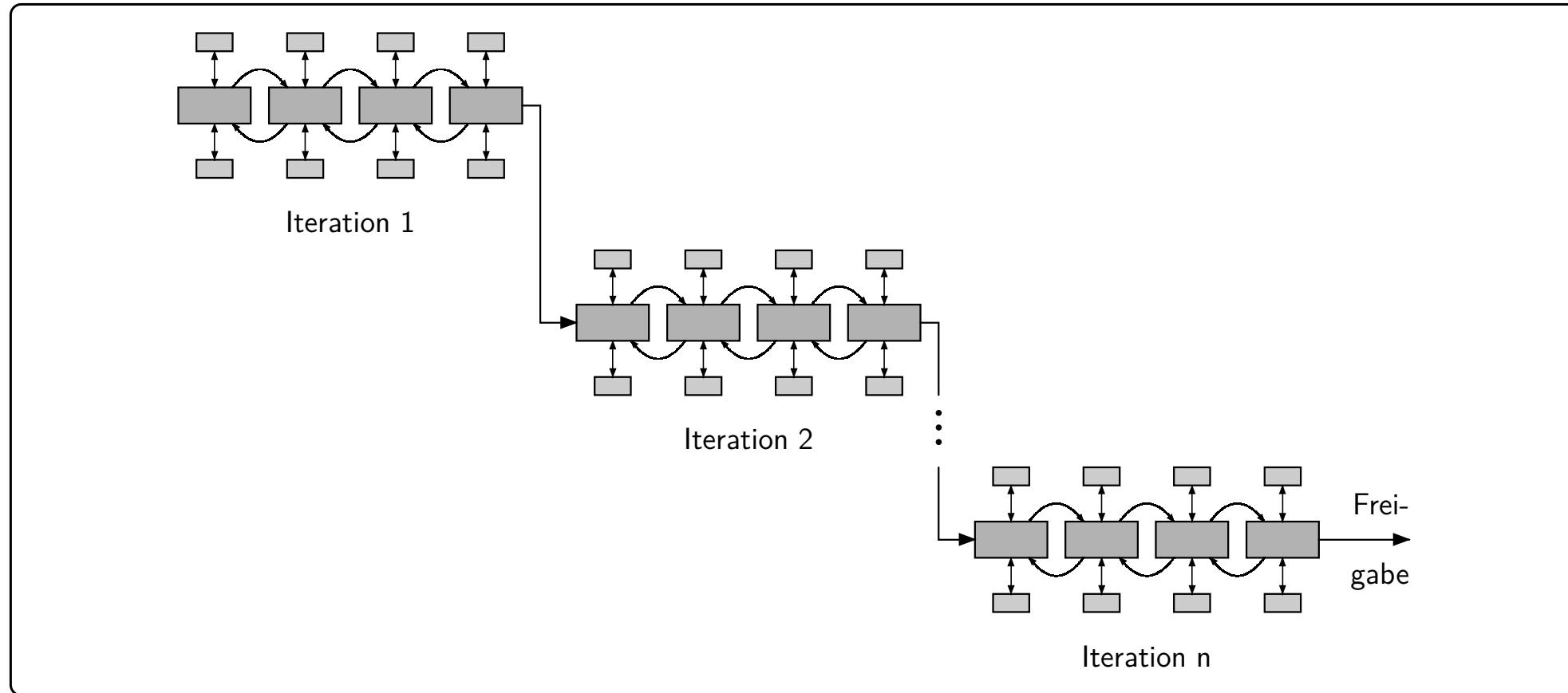
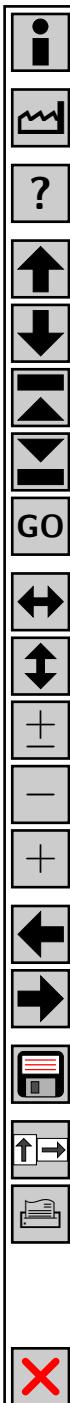


Abb. 7 Der iterative Entwicklungsansatz

- Struktur
 - Eine Modelliteration kann folgende Ergebnisse liefern
 - ◊ Ein System, das eine **Teilmenge** der **Gesamtanforderungen** realisiert → **evolutionäre Vorgehensweise**
 - ◊ Realisierung eines bestimmten Systemteils (z. B. die graphische Benutzeroberfläche) → **schichtenorientierte Vorgehensweise**
 - das Ergebnis der Iteration i kann als Teil der Anforderungs- und/oder Entwurfsphase in der Iteration $i + 1$ verwendet werden
- [Abb. 8](#) zeigt ein Beispiel für eine evolutionäre Vorgehensweise



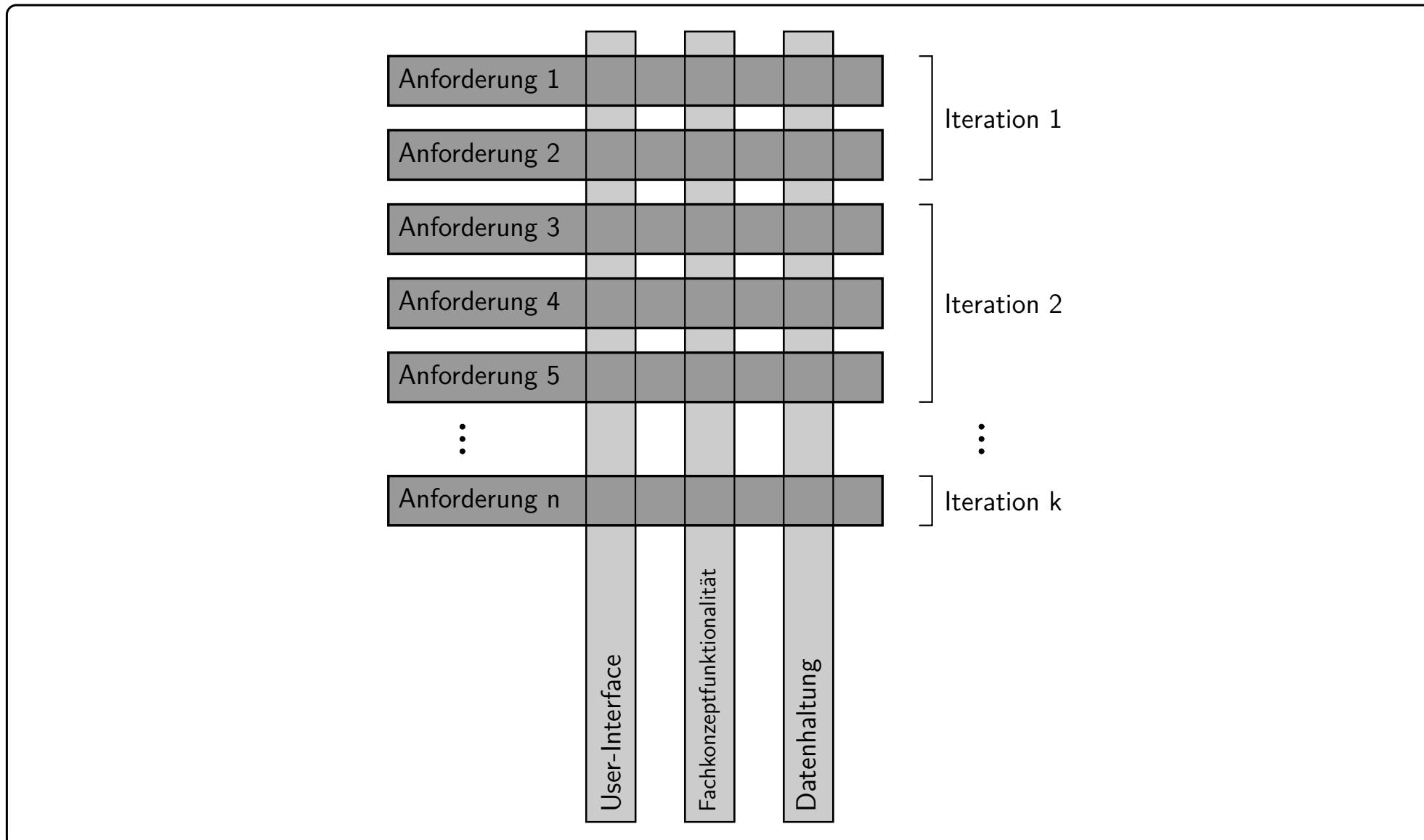
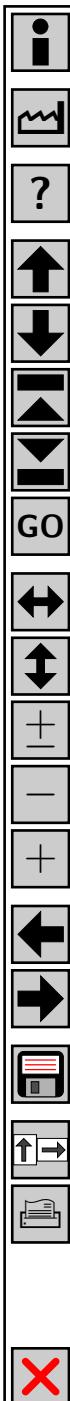
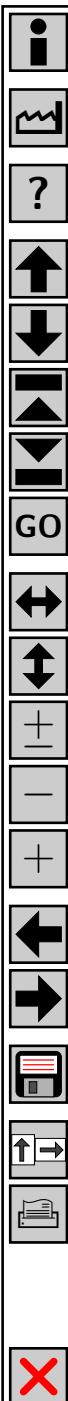


Abb. 8 Beispiel für eine evolutionäre Vorgehensweise



- Abb. 9 zeigt ein Beispiel für eine schichtenorientierte Vorgehensweise

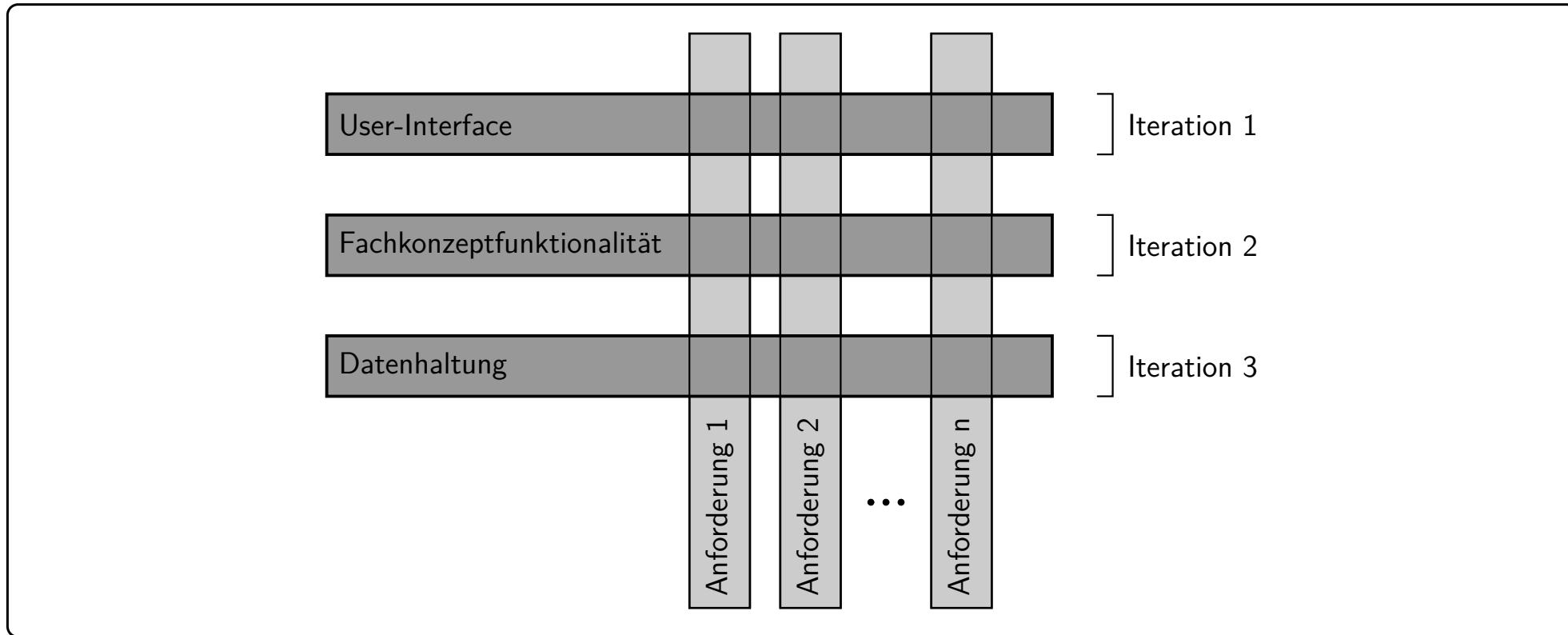
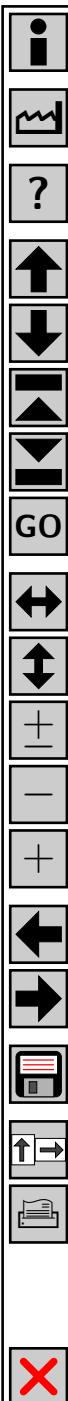


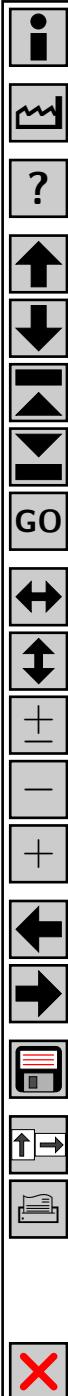
Abb. 9

Beispiel für eine schichtenorientierte Vorgehensweise



Weitere Vorgehensmodelle

- Es existieren sehr viele weitere Vorgehensmodelle
- Beispiele → Vorlesung Projektmanagement
 - V-Modell
 - Nebenläufiges Modell
 - Spiralmodell



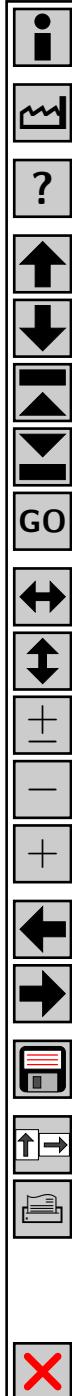
Testtechniken

Definition von Testfällen

- Struktur eines **Testfalls**
 - Festlegung der Randbedingungen und Eingaben
 - ◊ Systemzustand als Ausgangspunkt für den Test
 - ◊ Aktuelle Werte für die Parameter einer Operation
 - ◊ Aktionen eines Benutzers
 - Ermittlung des erwarteten Ergebnisses

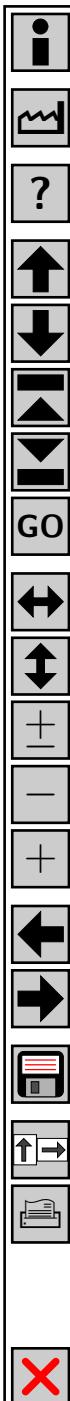


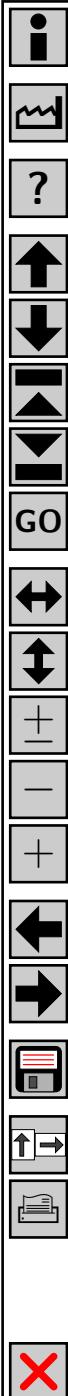
- **Testergebnisse sollen reproduzierbar sein, insbesondere beim Auftreten von Fehlern**
- **Falls der Faktor Zeit Testergebnisse beeinflusst, ist eine Reproduzierbarkeit – wenn überhaupt – nur sehr schwer erreichbar**



- Auswahl der **relevanten** Testfälle
 - Üblicherweise ist es nicht möglich, alle Kombinationen der Eingabewerte eines Programms zu testen → schon ein einzelner 32-Bit-Integer Eingabewert würde zu ca. 4 Milliarden Testfällen führen
 - Da sich jedoch viele Testfälle prinzipiell ähneln, sollten in einem ersten Schritt die Testfälle in entsprechende Gruppen unterteilt werden
 - Für jede identifizierte Gruppe sind dann (einige) geeignete Testfälle zu definieren
 - Definition von Testfällen für die Gruppengrenzen sowie den Bereich nahe der Gruppengrenze
- Beispiel: Quadratwurzelberechnung für einen Integer-Eingabewert
 - Als Gruppen ergeben sich:
 - ◊ Die Menge der positiven Integer-Zahlen (incl. 0) → hier existiert ein Ergebnis
 - ◊ Die Menge der negativen Integer-Zahlen → hier ist die Wurzelfunktion nicht definiert
 - Durchführung der Tests für einige Repräsentanten der beiden Mengen
 - Durchführung des Tests für den Eingabewert 0 (Gruppengrenze)
 - Durchführung des Tests für die Eingabewerte 1 und -1 (Bereich nahe der Gruppengrenze)

- Beispiel: Test eines Programms, das Integer-Zahlen zwischen 0 und 99 als Codierung für die Jahre 1900 bis 1999 verwendet
 - Hier existieren drei Gruppen
 - ◊ Negative Werte
 - ◊ Die Werte zwischen 0 und 99
 - ◊ Werte ab 99
 - Gruppengrenzen
 - ◊ 0 und 99
 - Sinnvolle Testfälle
 - ◊ -1000 , -1 , 0 , 1 , 50, 98 , 99 , 100 , 1000





Blackbox-Test

- Das Programm wird als eine **Blackbox** aufgefasst
 - Die interne Struktur ist für die Tests nicht relevant
 - Getestet werden soll das nach außen sichtbare Verhalten des Programms



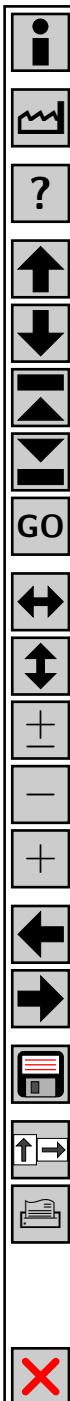
- Diese Testmethode ist sehr stark an die Anforderungsspezifikation gebunden**
- Es sollten diejenigen Fälle getestet werden, die für die spätere Programmbenutzung relevant sind**
- Die Testfälle können oft direkt aus der Anforderungsspezifikation abgeleitet werden und lassen sich in der Regel in Gruppen einteilen**

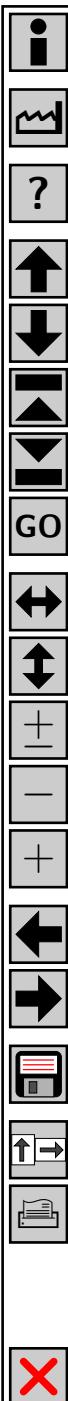
Whitebox-Test

- Hier steht der Test der internen Programmstruktur im Vordergrund
- Zielsetzung
 - Jeder Programmteil sollte mindestens einmal im Rahmen der Testfälle durchlaufen werden → *Statement Coverage*
- Definition der Testfälle
 - Es müssen zunächst die möglichen Pfade durch den Programm-Code analysiert werden
 - Von besonderer Bedeutung sind hier Anweisungen zur Steuerung des Kontrollflusses
 - ◊ Verzweigungen (z.B. if, case, switch)
 - ◊ Schleifen (z.B. for, while, repeat)
 - Die Eingabewerte für die Testfälle sind so zu wählen, dass die **Ausdrücke**, die die Schleifen bzw. Verzweigungen kontrollieren, jeden möglichen Wert mindestens einmal annehmen (z.B. true und false)
→ *Conditional Coverage*



Den einzelnen Pfaden durch ein Programm können meistens wieder Gruppen von Testfällen zugeordnet werden





Was ist eine objektorientierte Methode?

Der Methodenbegriff

- **Definition 3: Methodenbegriff (allgemein)**

*Der Begriff **Methode** beschreibt die systematische Vorgehensweise zur Erreichung eines bestimmten Ziels (griech: *methodos*).*

- **Definition 4: Methodenbegriff (in der Software-Technik)**

*Der Begriff **Methode** wird (auch) als Oberbegriff von **Konzepten**, **Notation** und **methodischer Vorgehensweise** verstanden.*

- Abb. 10 zeigt die Bestandteile einer Methode der Software-Technik

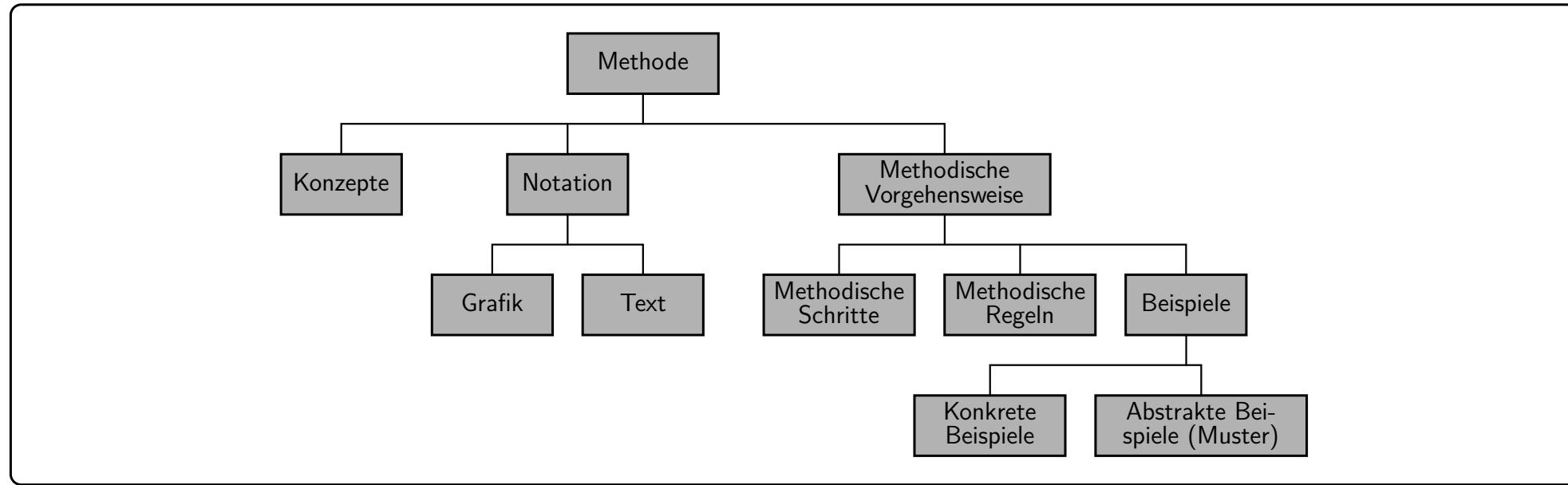
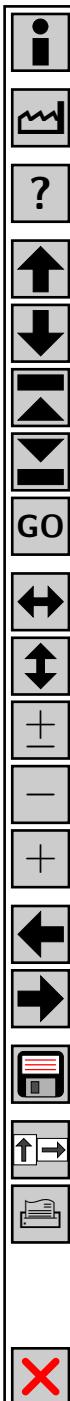
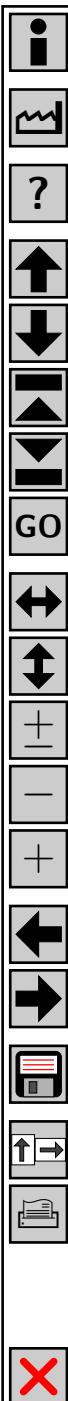


Abb. 10 Die Bestandteile einer Methode in der Software-Technik





Objektorientierte Methoden

Konzepte

- Abb. 11 enthält die Konzepte einer objektorientierten Methode

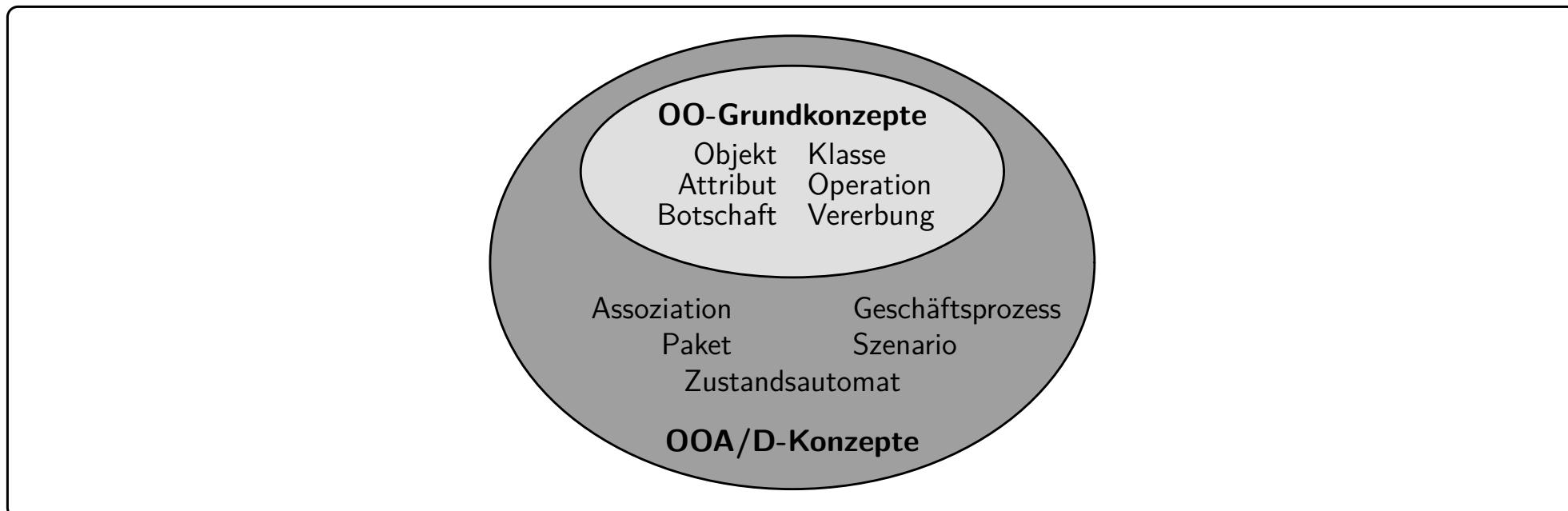
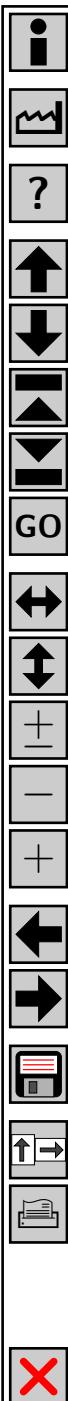
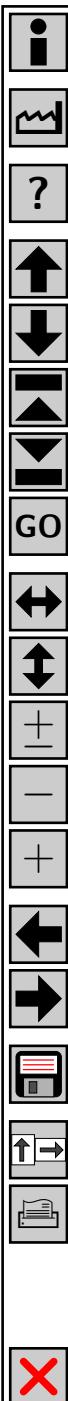


Abb. 11

Die Konzepte einer objektorientierten Methode

- Objektorientierte Grundkonzepte
 - Sie sind in allen Phasen (Analyse, Entwurf, Implementierung) vorhanden
 - Diese Konzepte wurden ursprünglich für objektorientierte Programmiersprachen entwickelt
 - Programmiersprachen, die nur einen Teil dieser Konzepte unterstützen, werden als **objektbasiert** bezeichnet
 - ◊ Üblicherweise fehlt objektbasierten Programmiersprachen das Vererbungskonzept

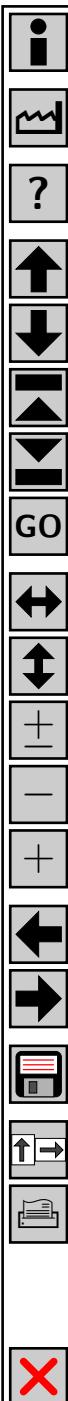


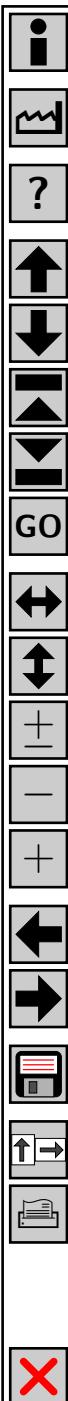


- Modellierung des **Fachkonzepts**
 - Fachkonzept: Umsetzung der Wünsche und Anforderungen eines Auftraggebers an ein neues Software-System in ein **implementierungsunabhängiges** Modell
 - Bei einer objektorientierten Methode stellt das OOA-Modell (OOA: *Object-Oriented Analysis*) das Fachkonzept dar
 - Die objektorientierten Grundkonzepte reichen hierfür nicht aus
 - Es werden zusätzlich Konzepte aus der **semantischen Datenmodellierung** benötigt
 - ◊ Statische Konzepte
 - ▷ **Assoziationen** → Modellierung der Beziehungen zwischen Objekten
 - ▷ **Pakete** → Zusammenfassung von Modellelementen (z. B. Klassen) zur Beschreibung (und Zerlegung) der Systemstruktur
 - ◊ Dynamische Konzepte zur Modellierung des **dynamischen Verhaltens**
 - ▷ **Geschäftsprozesse**
 - ▷ **Szenarios**
 - ▷ **Zustandsautomaten**

Notation

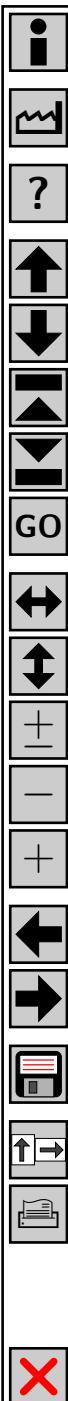
- Bestandteile der Notation
 - **Grafiken**, z. B. Klassendiagramm
 - **Texte**, z. B. Spezifikationen
- Die in [Abb. 11](#) dargestellten Konzepte werden von den meisten Notationen unterstützt
- Die UML-Notation
 - Alle Konzepte aus [Abb. 11](#) sind vorhanden
 - Die meisten **Werkzeuge (Tool)** zur Unterstützung der objektorientierten Software-Entwicklung bieten die UML-Notation an





Methodische Vorgehensweise

- Die meisten objektorientierten Methoden enthalten zwischen 4 und 13 **methodische Schritte**
- Praktischer Einsatz einer Methode
 - Oft reicht ein derart grobes Methodenraster nicht aus
 - Andererseits ist ein sehr detailliertes Vorgehensmodell in der Regel nur noch für bestimmte Anwendungen geeignet
- Vorgehensweise erfahrener Software-Entwickler
 - Anwendung – meist mehr oder weniger **intuitiv** – hunderter von **Regeln**
 - Diese Regeln werden dann **situationsspezifisch** eingesetzt
 - Für die Regelanwendung gibt es **keine** fest vorgegebene **Reihenfolge**
 - Rückgriff auf bereits **gelöste, ähnliche Problemstellungen**
 - Verwendung einer Sammlung von **spezifischen Beispielen (Muster, Pattern)**



Die Methode nach [Bal99]

Zielsetzung

- Präzise Trennung der Phasen (Schritte) Analyse und Entwurf
- Bewahrung der leichten Durchgängigkeit zwischen beiden Phasen

Beteiligte Personengruppen

- Fachexperten
 - Experten aus dem Sachgebiet, für das das neue Software-System entwickelt werden soll
 - Da nur der Fachexperte weiß, **was** das System aus Benutzersicht leisten soll, ist es – insbesondere aus Kostengründen – sinnvoll, wenn diese sich in die OOA-Technik einarbeiten
- OOA-Experten
 - Unterstützung der Fachexperten bei der **methodischen Projektbetreuung**



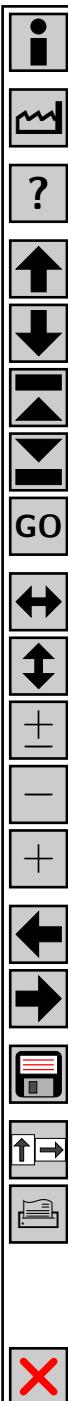
- Die Erstellung eines OOA-Modells ist ein *kreativer Prozess*
- Vorhandene Modelle oder Muster leisten oft nur eine geringe Hilfestellung

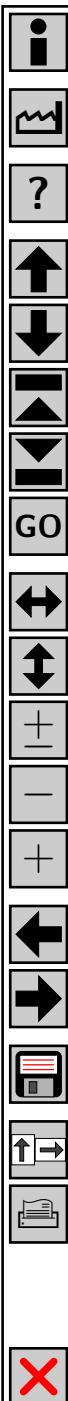
- Software-Konstrukteure bzw. Programmierer
 - Erstellung des Entwurfs
 - Durchführung der Implementierung



- Durch die enge Verzahnung von Entwurf und Implementierung bei der objekt-orientierten Entwicklung ist eine *personelle Trennung* hier nicht sinnvoll
- Im Gegensatz zur Analyse nimmt die *Standardisierung* beim Entwurf durch ein großes Angebot von *Mustern, Frameworks* und *Klassenbibliotheken* ständig zu

- Spezialisierte Software-Experten
 - Durch die *hohe Komplexität* der Software-Entwicklung ist eine weitere Spezialisierung sinnvoll
 - Beispiele
 - ◊ Datenbankanbindung
 - ◊ Gestaltung von Benutzeroberflächen

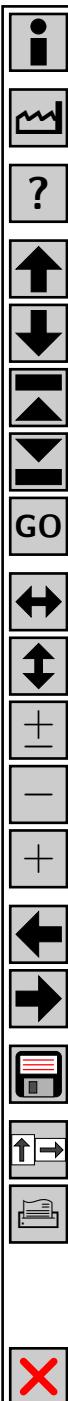


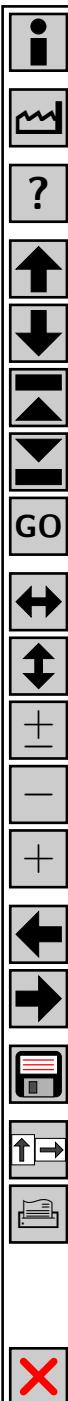


Wichtige Eigenschaften der Methode nach [Bal99]

- Unterstützung aller notwendigen objektorientierten Konzepte für Analyse und Entwurf
- Verwendung der UML-Notation
- Präzise Trennung der UML-Modellelemente in Analyse und Entwurf
- Verwendung von Analysemustern
- Methodische Vorgehensweise für die objektorientierte Analyse
- Abbildung des Analysemodells auf eine objektorientierte Benutzeroberfläche
- Verwendung von Entwurfsmustern
- Realisierung der Drei-Schichten-Architektur im Entwurf
- Eine standardisierte Anbindung der Benutzeroberfläche an Fachkonzept und Datenhaltung
- Die objekt-relationale Abbildung zur Anbindung an relationale Datenbanken

- Die Anbindung objektorientierter Datenbanken
- Die standardisierte Verteilung objektorientierter Systeme im Netz
- Die Transformation des Entwurfs in JAVA oder C⁺⁺
- Die analytische Qualitätssicherung in Analyse und Entwurf

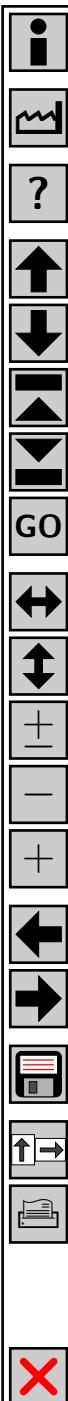




Objektorientierte Analyse

Aufgabenstellung der Systemanalyse

- Die Aufgaben der Systemanalyse werden zunächst unabhängig von der Objektorientierung betrachtet
- Ziele der Systemanalyse
 - **Ermittlung** und **Beschreibung** der **Wünsche** und **Anforderungen** eines Auftraggebers an ein neues Software-System
 - Erstellung eines **Fachkonzeptmodells** → Eigenschaften
 - ◊ Konsistenz
 - ◊ Vollständigkeit
 - ◊ Eindeutigkeit
 - ◊ Realisierbarkeit → Verwendung von Prototypen

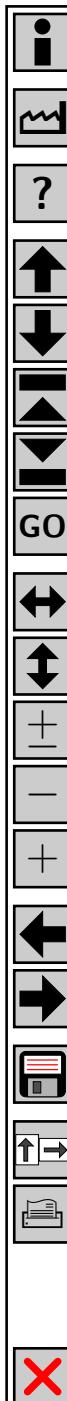


- Randbedingungen
 - Bewusste **Ausklammerung** aller **Implementierungsaspekte** bei der Modellbildung
 - Voraussetzung einer **perfekten Technik**
 - ◊ Der Prozessor kann jede Funktion ohne Verzögerung ausführen
 - ◊ Es treten keine Verarbeitungsfehler auf
 - ◊ Das System fällt nie aus
 - ◊ Der Speicher kann unendlich viele Informationen aufnehmen
 - ◊ Der Prozessor kann ohne Zeitverzögerung auf den Speicher zugreifen
 - ◊ Die Verteilung der Software auf mehrere Rechner spielt keine Rolle
 - ◊ Die Form der Datenspeicherung ist unerheblich



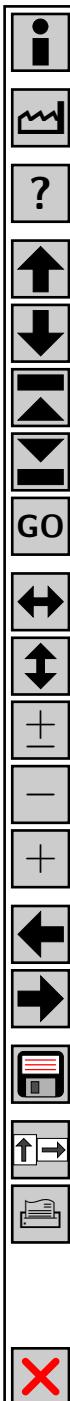
- **Es ist Aufgabe des Systemanalytikers, die *wahren Anforderungen* seines Auftraggebers zu modellieren**
- **Die Modellierung soll durch keine Implementierungstechnik eingeschränkt werden**

- Das Dilemma der Systemanalytiker
 - Der Auftraggeber hat noch kein vollständiges Modell des zukünftigen Systems im Kopf
 - Für die Anforderungen des Auftraggebers gilt daher (meistens):
 - ◊ Sie sind **unklar**
 - ◊ Sie sind in sich **widersprüchlich**
 - ◊ Sie sind **fallorientiert**
 - ◊ Sie sind auf unterschiedlichen **Abstraktionsebenen** definiert
 - Schlussfolgerungen
 - Die Systemanalyse gehört zu den anspruchsvollsten Tätigkeiten der Software-Entwicklung
 - Die **wahren** Anforderungen an ein System sind nicht plötzlich da, sondern die Systemanalyse bildet einen **kontinuierlichen** Prozess, um Informationen zu sammeln, zu filtern und zu dokumentieren
-  ● Es ist **nicht Aufgabe des Auftraggebers, den Systemanalytiker zu verstehen**
● **Der Systemanalytiker wird dafür bezahlt, sich dem Auftraggeber verständlich zu machen**

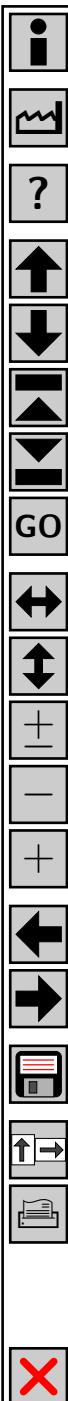


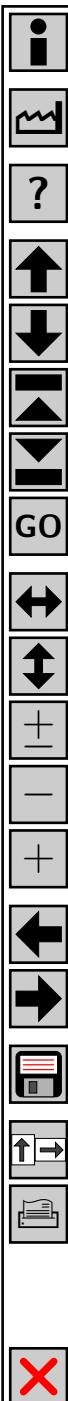
Das OOA-Modell

- Zielsetzung der objektorientierten Analyse
 - Das zu lösende Problem muss verstanden werden
 - Beschreibung des Problems in einem OOA-Modell
 - ◊ Darstellung der prinzipiellen Problemstruktur
 - ◊ Das Modell darf keine Optimierungen für das verwendete Computer-System oder die benutzte Basis-Software enthalten
- Ausgangspunkt: Objekte der **realen** Welt
 - **Anfassbare** Objekte → z. B. Autos, Personen
 - **Begriffe** oder **Ereignisse** aus dem **Anwendungsbereich** → z. B. Vektor, Matrix
- Abbildung der realen Objekte durch Verwendung geeigneter **Abstraktionen** in Objekte des OOA-Modells
 - Die Objekte des OOA-Modells sollen nur die für den Anwendungsbereich notwendigen Eigenschaften besitzen
 - Beispiel: Bei der Modellierung von Mitarbeitern einer Firma sind deren Hobbys üblicherweise irrelevant



- Präsentation des OOA-Modells
 - Da ein **normaler** Auftraggeber (normal im Sinne: er ist kein ausgebildeter Systemanalytiker!) dieses OOA-Modell wahrscheinlich nicht vollständig verstehen kann, sollte ein **Prototyp** der **Benutzeroberfläche** erstellt werden
 - Diskussion der Änderungswünsche unter Anwendung des Prototyps





Produkte der Analysephase

Pflichtenheft

- **Definition 5: Pflichtenheft**

Das **Pflichtenheft** enthält eine textuelle Beschreibung, **was** das System leisten soll.

- Zielsetzungen bei der Erstellung

- Es dient als **Einstiegsdokument** in das Projekt für alle, die später das System **pflegen** und **warten** sollen
 - Es bildet die **Ausgangsbasis** für eine **systematische Modellbildung** durch den Systemanalytiker



- Das Pflichtenheft besitzt ein **niedrigeres Detaillierungsniveau** als das OOA-Modell
 - Es ist **nicht** das Ziel, anhand des Pflichtenheftes das System zu implementieren

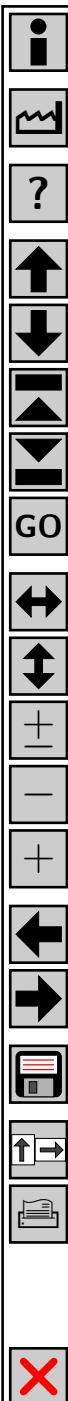
OOA-Modell

- **Definition 6:** OOA-Modell

Das OOA-**Modell** stellt die fachliche Lösung des zu realisierenden Systems dar → Fachkonzept.

- Bestandteile des OOA-Modells (vgl. [Abb. 12](#))

- Statisches Modell
 - ◊ Dieses steht bei typischen **Datenbankanwendungen** im Vordergrund
- Dynamisches Modell
 - ◊ Dieses ist insbesondere bei **interaktiven Anwendungen** von Bedeutung



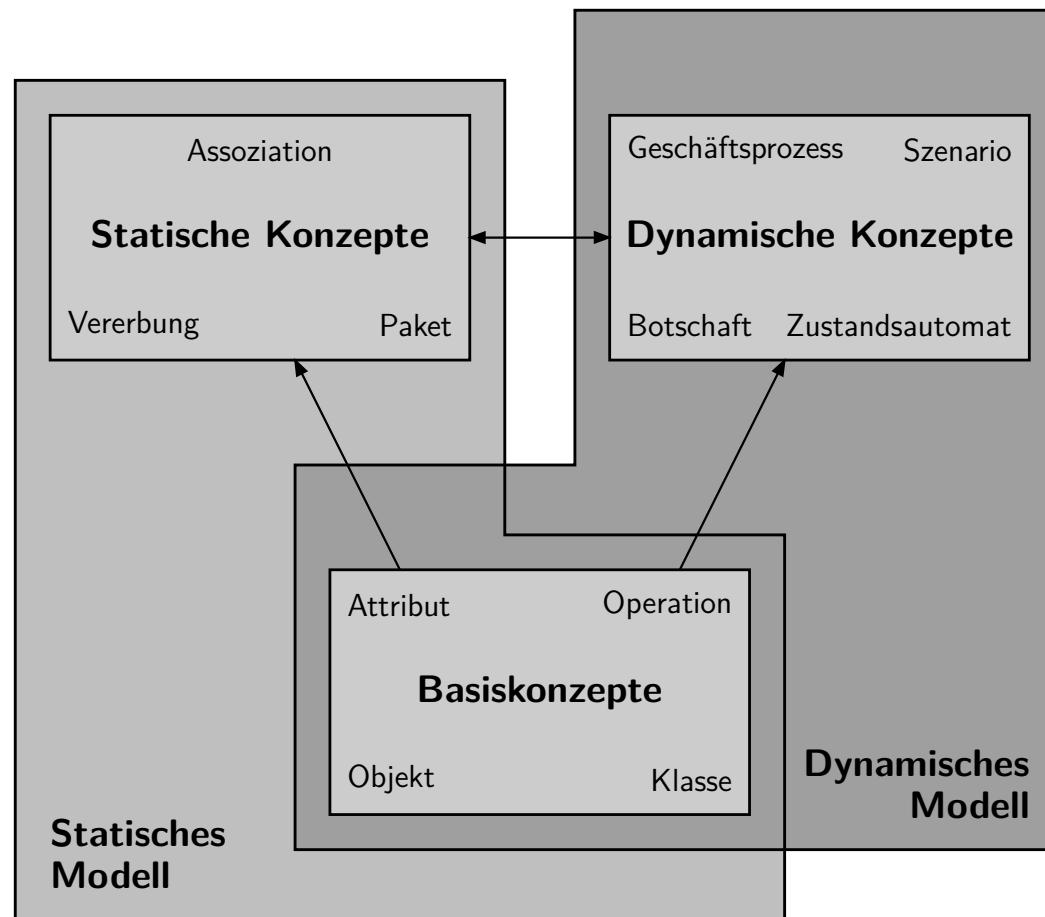
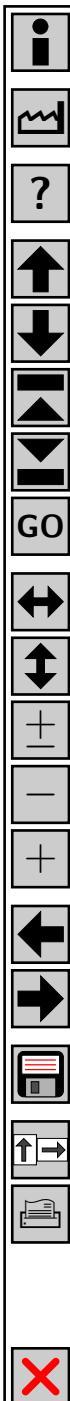


Abb. 12

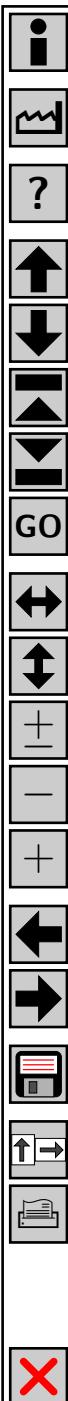
Statisches und dynamisches Modell



Der Begriff der *Basiskonzepte* ist nicht mit dem Begriff der *Grundkonzepte* aus Abb. 11 zu verwechseln

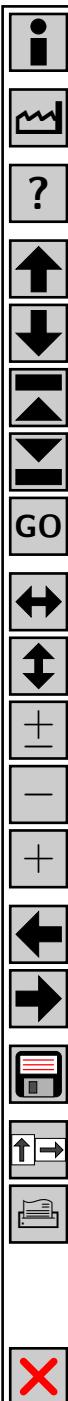
Statisches Modell

- Beschreibung der **Klassen** – und damit der Objekte – des Systems
- Beschreibung der **Assoziationen** (Beziehungen) zwischen den Klassen
- Beschreibung der **Vererbungsstrukturen** zwischen den Klassen
→ *Welche Klasse stellt eine Spezialisierung einer anderen Klasse dar?*
- Beschreibung der **Daten** (*Attribute*) der Objekte (und damit des Systems)
- Bildung von **Teilsystemen** durch die Definition von **Paketen**



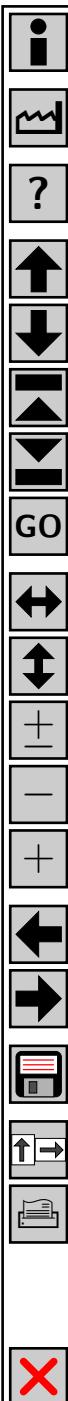
Dynamisches Modell

- Festlegung von **Funktionsabläufen**
- **Geschäftsprozesse** beschreiben die *durchzuführenden Aufgaben* auf einem sehr hohen Abstraktionsniveau
- **Szenarios** zeigen, wie Objekte *miteinander kommunizieren*, um eine bestimmte Aufgabe zu erledigen
- **Zustandsautomaten** beschreiben in der Analyse die *Lebenszyklen* von Objekten, d. h. die Reaktionen eines Objekts auf *Ereignisse* (Botschaften)

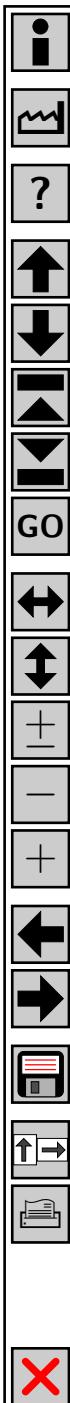


Prototyp der Benutzungsoberfläche

- Anwendung
 - Abklärung mit dem künftigen Benutzer bzw. Auftraggeber, ob das System **wie gewünscht** spezifiziert ist
- Struktur
 - **Ablauffähiges** Programm, das alle Attribute des OOA-Modells visualisiert
 - Bestandteile
 - ◊ Fenster
 - ◊ Dialoge
 - ◊ Menüs, usw.
 - Der Prototyp realisiert weder **Anwendungsfunktionen** noch besitzt er die Fähigkeit, **Daten** zu speichern

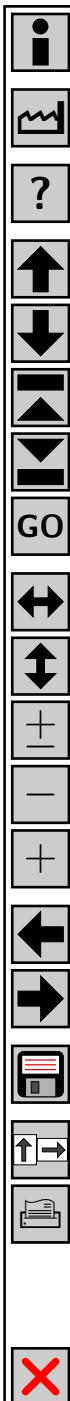


- Realisierung des Prototyps
 - Es sollte möglichst die vollständige Benutzungsoberfläche realisiert werden
 - Verwendung entsprechender Werkzeuge
 - Eine **ergonomische Gestaltung** steht hier zwar noch nicht im Vordergrund, sollte aber **nicht vollkommen ignoriert** werden
 - Ergänzende Dokumentation
 - Falls eine vollständige Realisierung der Benutzungsoberfläche nicht möglich ist, wird eine **ergänzende Dokumentation** erstellt
 - Diese kann auch weitergehende Informationen enthalten
→ z. B. *wer besitzt welche Zugriffsrechte auf welche Daten*
-  • **Die Trennung von Fachkonzept und Benutzungsoberfläche ist ein Grundkonzept der Entwicklung**
 - Begründung: Benutzungsoberflächen ändern sich aufgrund des technischen Fortschritts schneller als die Funktionalität des Fachkonzepts
 - Das Fachkonzept beschreibt, **welche Informationen** auf dem Bildschirm sichtbar sind
 - Die Benutzungsoberfläche legt fest, **in welchem Format** sie dargestellt werden

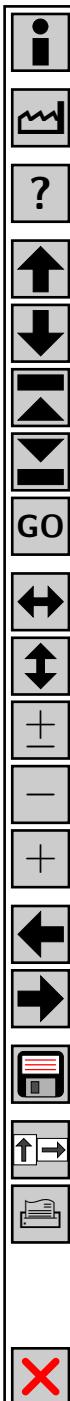


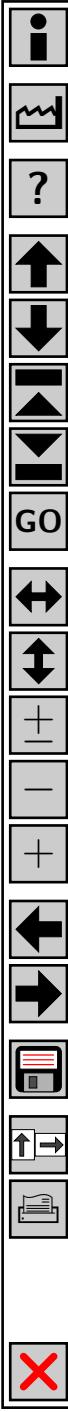
Erstellung des OOA-Modells

- Zusammensetzung des OOA-Entwicklungs-Teams
 - Systemanalytiker
 - Fachexperten
 - Zukünftige Benutzer bzw. Benutzerrepräsentanten
- ⚠
 - Das Entwicklungs-Team darf nicht zu groß werden
 - Es sollten auf jeden Fall weniger als 10 Personen sein
- Beispiel: Entwicklungs-Team für ein Buchhaltungssystem
 - Systemanalytiker
 - Buchhaltungsexperte, der alle einzuhaltenden Vorschriften kennt
 - Angestellte des Steuerberaters, die das System später benutzen sollen



- Durchführung der OOA-Entwicklung
 - Erstellung einer ersten Version des OOA-Modells
 - Ableitung des Prototyps der Benutzungsoberfläche aus dem OOA-Modell
 - Evaluierung dieses Prototyps mit dem zukünftigen Benutzer
 - Umsetzung der Erkenntnisse in der nächsten OOA-Modellversion
 - Iteration dieses **Gesamtprozesses**, bis ein befriedigendes OOA-Modell erreicht ist





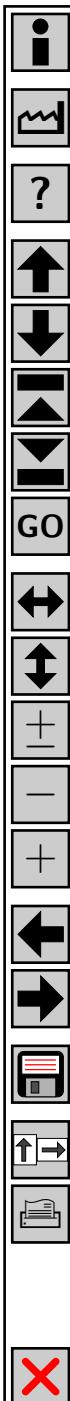
Objektorientierter Entwurf

Aufgabenstellung des Systementwurfs

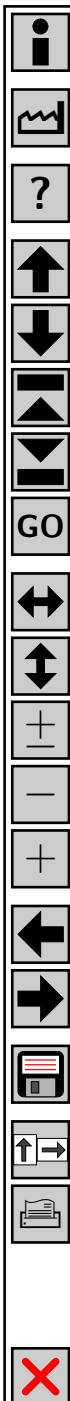
- Die Systemanalyse ist von einer ***idealen Systemumgebung*** ausgegangen
 - Aufgabe der Entwurfsphase
 - Realisierung der spezifizierten Anwendung auf einer Plattform unter den ***geforderten*** technischen ***Randbedingungen***
 - Der Entwurf findet allerdings noch auf einem ***höheren Abstraktionsniveau*** als die Implementierung statt
 - Ergebnis der Entwurfsphase
 - ***Entwurfsmodell***, das unter den Gesichtspunkten ***Effizienz*** und ***Standardisierung*** konzipiert wurde
-  • **Der objektorientierte Entwurf wird dadurch erheblich vereinfacht, dass von der Analyse zum Entwurf *kein Paradigmenwechsel* stattfindet**
- **Entwurfs- und Implementierungsphase sind sehr stark *miteinander verzahnt* → entworfene Klassen können direkt implementiert werden**

Die Drei-Schichten-Architektur

- Eigenschaften **veralteter** Systeme
 - Die anwendungsspezifische **Funktionalität** ist noch relativ **modern**
 - Die **Benutzungsoberfläche** und/oder **Datenhaltung** sind **veraltet**
- Anpassung des Systems
 - Für die Aktualisierung der Benutzungsoberfläche muss das komplette System neu geschrieben werden
 - Ebenso führt der Austausch oder die erstmalige Verwendung einer Datenbank aus Gründen der Leistungsfähigkeit oder des Datentransfers faktisch zu einer Neuimplementierung des Systems



- Entwurfsziel: **Fachkonzept**, **Benutzungsoberfläche** und **Datenhaltung** sind weitgehend zu entkoppeln
 - Wie die Benutzungsoberfläche aussieht, hängt entscheidend vom verwendeten GUI (GRAPHICAL USER INTERFACE) ab
 - Die Datenhaltung wird wesentlich durch den Datenbanktyp bestimmt
 - ◊ Hierarchische, relationale oder objektorientierte Datenbank
 - ◊ Alternativ kann die Datenhaltung in einfachen Fällen durch Dateien realisiert werden
- Das Entwurfsziel führt direkt zur Verwendung einer **Drei-Schichten-Architektur** → vgl. OOD-Modell in [Abb. 13](#)
 - Benutzungsoberfläche
 - Fachkonzept
 - Datenhaltung



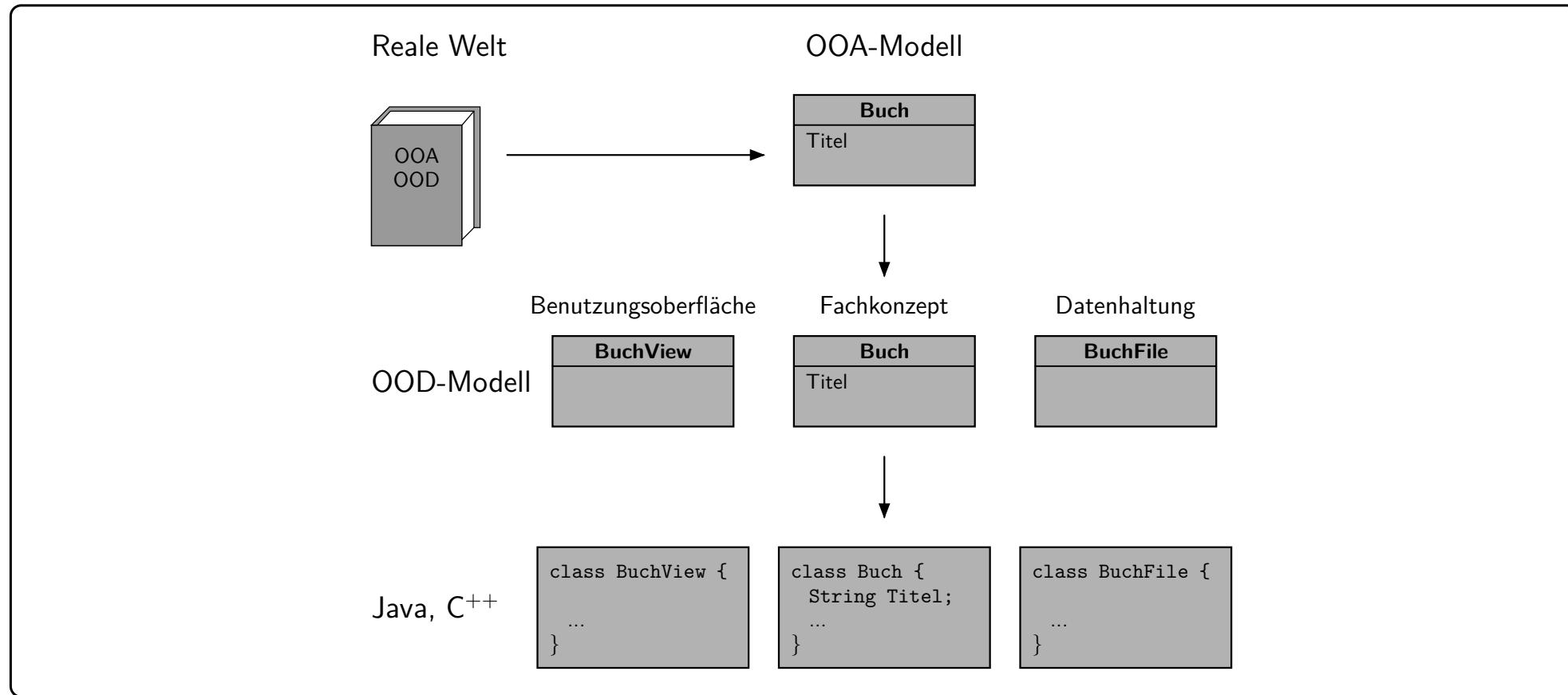
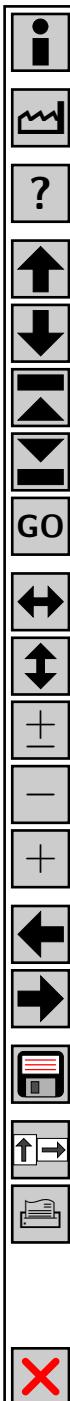
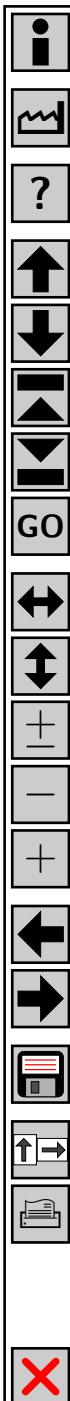


Abb. 13

Die Drei-Schichten-Architektur

- Vorgehensweise → vgl. [Abb. 14](#)
 - Verwendung des OOA-Modells als erste Version der Fachkonzeptschicht
 - Überarbeitung des OOA-Modells unter den Gesichtspunkten **Effizienz** und **Wiederverwendung** → Berücksichtigung vorhandener Klassenbibliotheken und Schnittstellen zu anderer Software
 - Weiterentwicklung des Benutzungsoberflächenprototyps aus der OOA-Phase
 - Auswahl des Datenhaltungskonzepts → Dateien oder Datenbank (plus Datenbanktyp)
 - Verteilung der Software auf mehrere Rechner bei **Client-Server-Anwendungen**



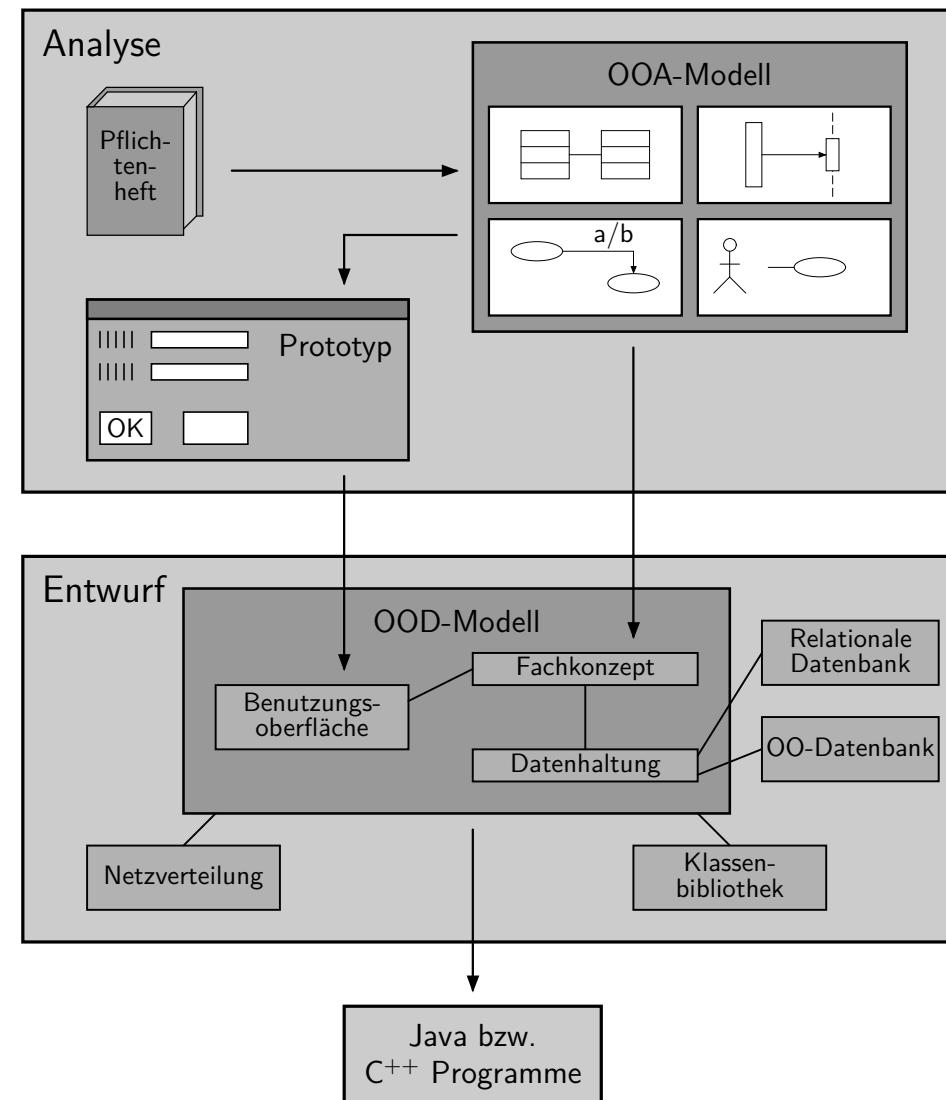
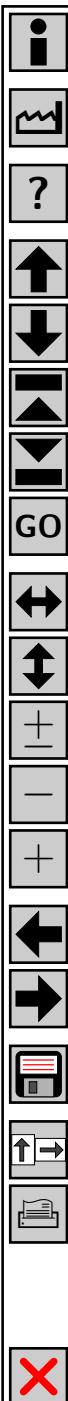
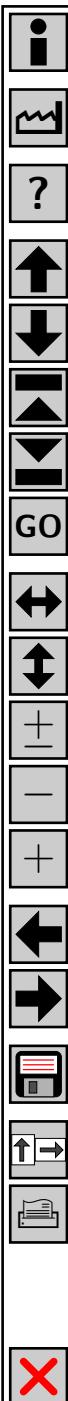


Abb. 14 Die Abgrenzung zwischen Analyse und Entwurf

Die Produkte der Entwurfsphase

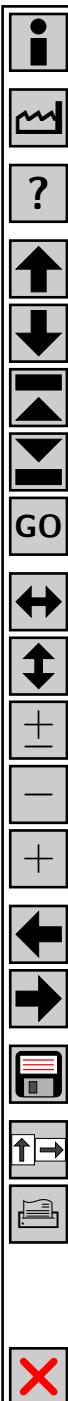
OOD-Modell

- Das OOD-Modell ist ein **Abbild** der späteren Programme
 - Jede **Klasse**, jedes **Attribut** und jede **Operation** findet sich in den Programmen wieder
 - Verwendung **derselben Namen** im OOD-Modell wie in den Programmen
- ⚠ • Im Gegensatz zum objektorientierten Programm-Code zeigt das OOD-Modell das System jedoch auf einer **höheren Abstraktionsebene**
 - Es soll insbesondere das **Zusammenwirken** einzelner Elemente verdeutlichen
- Bestandteile des OOD-Modells
 - Statisches Modell
 - Dynamisches Modell



Statisches Modell

- Im Gegensatz zur Analysephase ist das statische Modell wesentlich **umfangreicher**
- Struktur des statischen Modells
 - Es enthält **alle Klassen** des Programms, welche die **Architektur** des Systems beschreiben
 - Klassen, die nur **Typen** beschreiben (wie z. B. **STRING**) werden nicht eingetragen
- Verwendung von **Paketen**
 - Modellierung von **Teilsystemen** (analog zur Analysephase)
 - Darstellung der verschiedenen **Schichten** (vgl. Drei-Schichten-Architektur)

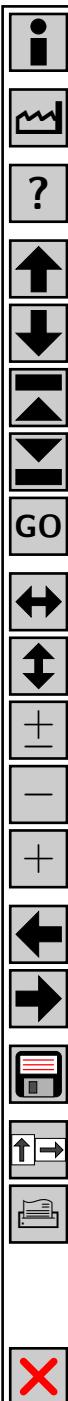


Dynamisches Modell

- Das dynamische Modell ist im Entwurf von besonderer Bedeutung
- Es ermöglicht eine ***übersichtliche Beschreibung*** der ***komplexen Kommunikation*** zwischen den Objekten

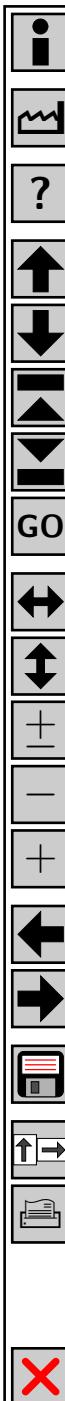


Die Kommunikationsbeziehungen zwischen den Objekten sind anhand des Programm-Codes nur schwer nachvollziehbar



Zusammenfassung

- Eine (objektorientierte) **Methode** setzt sich aus **Konzepten**, einer **Notation** und einer **methodischen Vorgehensweise** zusammen
- Die UML bildet zur Zeit den Standard für eine **objektorientierte Notation**
- In der **Analyse** muss ein **Fachkonzept** des zu realisierenden Systems erstellt werden
- Das OOA-Modell beschreibt die essentielle Struktur und Semantik des Problems, aber noch keine technische Lösung
- Aus dem OOA-Modell wird ein **Prototyp** der Benutzeroberfläche abgeleitet
- Aufgabe des **Entwurfs** ist es, das Fachkonzept auf einer Plattform unter den geforderten technischen Randbedingungen zu realisieren
- Das OOD-Modell soll ein **Abbild** des späteren **objektorientierten Programms** sein



Glossar

Analyse (*analysis*): Aufgabe der Analyse ist die Ermittlung und Beschreibung der Anforderungen eines Auftraggebers an ein Software-System. Das Ergebnis soll die Anforderungen vollständig, widerspruchsfrei, eindeutig, präzise und verständlich beschreiben.

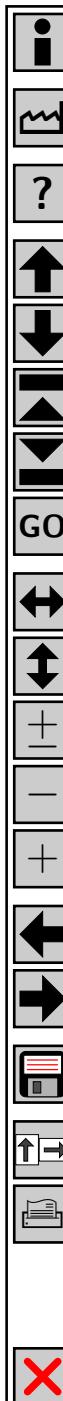
Dynamisches Modell: Das dynamische Modell ist der Teil des OOA-Modells, welches das Verhalten des zu entwickelnden Systems beschreibt. Es realisiert außer den Basiskonzepten (Objekt, Klasse, Operation) die dynamischen Konzepte (Geschäftsprozess, Botschaft, Zustandsautomat).

Entwurf (*design*): Aufgabe des Entwurfs ist – aufbauend auf dem Ergebnis der Analyse – die Erstellung der Software-Architektur und die Spezifikation der Komponenten, d. h. die Festlegung von deren Schnittstellen, Funktions- und Leistungsumfang. Das Ergebnis soll die zu realisierenden Programme auf einem höheren Abstraktionsniveau widerspiegeln.

Konzept (*concept*): Der Begriff des Konzepts wird in der Informatik im Sinne von Leitidee verwendet, z. B. Konzepte der Programmierung, Konzepte der Objektorientierung. Ein Konzept beschreibt einen definierten Sachverhalt (z. B. eine Klasse) unter einem oder mehreren Gesichtspunkten.

Methode (*method*): Der Begriff Methode beschreibt die planmäßig angewandte, begründete Vorgehensweise zur Erreichung von festgelegten Zielen. In der Software-Technik wird der Begriff Methode als Oberbegriff von → Konzepten, → Notation und → methodischer Vorgehensweise verwendet.

Methodische Vorgehensweise (*method*): Eine methodische Vorgehensweise ist eine planmäßig angewandte, begründete Vorgehensweise zur Erreichung von festgelegten Zielen. Sie wird häufig als → Methode bezeichnet.





Notation (*notation*): Darstellung von → *Konzepten* durch eine festgelegte Menge von grafischen und/oder textuellen Symbolen, zu denen eine Syntax und Semantik definiert ist.

Objektorientierte Analyse (*object oriented analysis*): Ermittlung und Beschreibung der Anforderungen an ein Software-System mittels objektorientierter Konzepte und Notationen. Das Ergebnis ist das OOA-Modell.

Objektorientierter Entwurf (*object oriented design*): Aufbauend auf dem OOA-Modell erfolgt die Erstellung der Software-Architektur und die Spezifikation der Klassen aus der Sicht der Realisierung. Das Ergebnis ist das OOD-Modell, das ein Spiegelbild der objektorientierten Programme auf einem höheren Abstraktionsniveau bildet.

Objektorientierte Software-Entwicklung (*object oriented software development*): Bei einer objektorientierten Software-Entwicklung werden die Ergebnisse der Phasen Analyse, Entwurf und Implementierung objektorientiert erstellt. Für letztere werden objektorientierte Programmiersprachen verwendet. Auch die Verteilung auf einem Netz kann objektorientiert erfolgen.

OOA: → *Objektorientierte Analyse*

OOA-Modell: Fachliche Lösung des zu realisierenden Systems, die in einer objektorientierten → *Notation* modelliert wird. Das OOA-Modell besteht aus dem → *statischen* und dem → *dynamischen Modell* und ist das wichtigste Ergebnis der → *Analyse*.

OOD: → *Objektorientierter Entwurf*

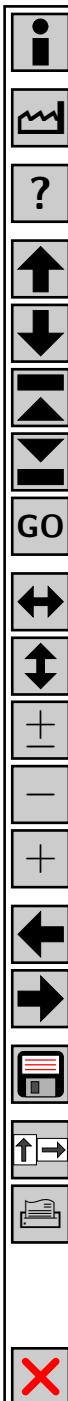
OOD-Modell: Technische Lösung des zu realisierenden Systems, die in einer objektorientierten → *Notation* modelliert wird. Das OOD-Modell ist ein Abbild des späteren (objektorientierten) Programms.

Prototyp (*prototype*): Der Prototyp dient dazu, bestimmte Aspekte vor der Realisierung des Software-Systems zu überprüfen. Der Prototyp der Benutzeroberfläche zeigt die vollständige Oberfläche des zukünftigen Systems, ohne dass bereits Funktionalität realisiert wird.

Statisches Modell: Das statische Modell realisiert außer den Basiskonzepten (Objekt, Klasse, Attribut) die statischen Konzepte (Assoziation, Vererbung, Paket). Es beschreibt die Klassen des Systems, die Assoziationen zwischen den Klassen und die Vererbungsstrukturen. Außerdem enthält es die Daten des Systems (Attribute). Die Pakete dienen dazu, Teilsysteme zu bilden, um bei großen Systemen einen besseren Überblick zu ermöglichen.

Systemanalyse: → Analyse

UML: *Unified Modeling Language*, die von BOOCH, RUMBAUGH und JACOBSON bei der *Rational Software Corporation* entwickelt und 1997 von der OMG (Object Management Group) als Standard akzeptiert wurde.



DER ANALYSEPROZESS

Lernziele

Wissen

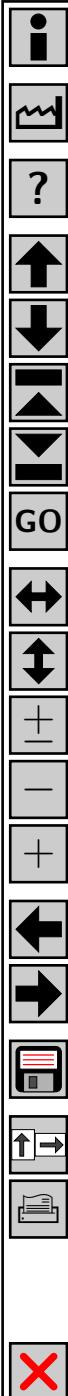
- Der Aufbau eines Pflichtenheftes
- Die Struktur des Analyseprozesses

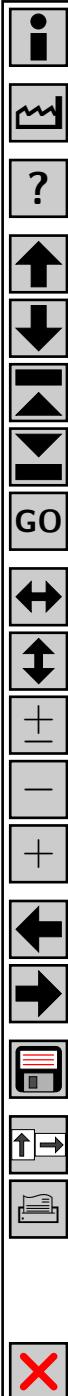
Verständnis

- Der balancierte Makroprozess
- Alternative Makroprozesse

Anwendung

- Auswahl eines geeigneten Makroprozesses





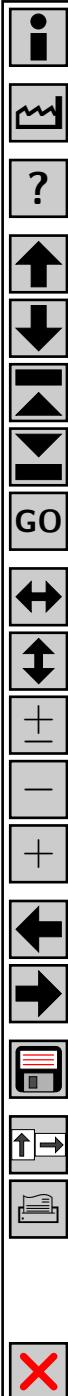
Grundlage: Das Pflichtenheft

Überblick

- Das Pflichtenheft ist die Voraussetzung für die Erstellung eines OOA-Modells
- Alternative Bezeichnungsweisen
 - Fachkonzept
 - Anforderungsspezifikation



- Die Erstellung eines Pflichtenheftes ist *keine* Aufgabe der objektorientierten Analyse
- Das OOA-Modell wird ebenfalls als Fachkonzept bezeichnet → vgl. S. 46



Gliederungsschema für ein Pflichtenheft

Struktur

1 Zielbestimmung

1.1 Muss-Kriterien

1.2 Kann-Kriterien

1.3 Abgrenzungskriterien

2 Einsatz

2.1 Anwendungsbereiche

2.2 Zielgruppen

2.3 Betriebsbedingungen

3 Umgebung

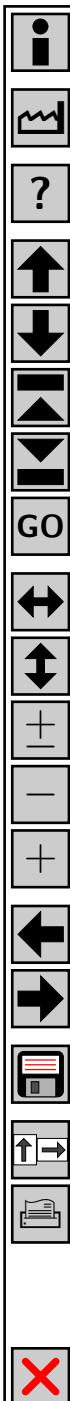
3.1 Software

3.2 Hardware

3.3 Orgware

4 Funktionalität

5 Daten



6 Leistungen

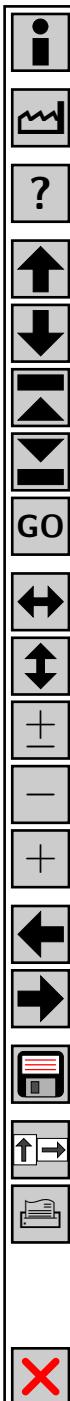
7 Benutzungsoberfläche

8 Qualitätsziele

9 Ergänzungen

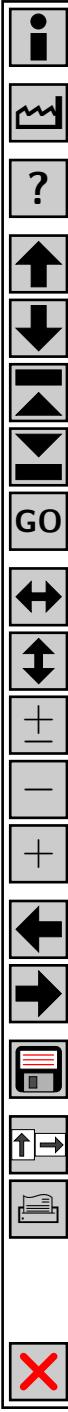


Selbstverständlich gibt es auch andere Gliederungsschemen für Pflichtenhefte



Zielbestimmung

- Allgemeines
 - Es sind Ziele zu formulieren
 - ◊ Beispiel: Mindestbestand von Artikeln automatisch sicherstellen
 - Es sollen nicht die für die Erreichung eines Ziels notwendigen Funktionen beschrieben werden
 - ◊ Beispiel: Erstellung von Bestellvorschlagslisten für Artikel, deren Mindestbestand unterschritten ist
 - Die Abgrenzung zwischen Zielen und Funktionen kann schwierig sein
- **Muss-Kriterien**
 - Ziele, die das Software-System **unbedingt** erfüllen muss
 - Ist die Erfüllung eines derartigen Ziels nicht möglich, dann kann das System für den vorgesehenen Zweck nicht eingesetzt werden
- **Beispiel 1:** Muss-Kriterien für ein Werkzeug zur Erstellung von OO-Modellen
 - Unterstützung der UML-Notation
 - Mehrbenutzerfähigkeit
 - Automatische Erstellung der Dokumentation



- **Kann-Kriterien**

- Ziele, die das Produkt erfüllen sollte, auf die aber ***zunächst verzichtet*** werden kann
- Im Rahmen der Projektplanung kann bei Terminproblemen dann eine Konzentration auf die Muss-Kriterien erfolgen

- **Beispiel 2:** Kann- und Muss-Kriterien bei einem Buchhaltungssystem

- Muss-Kriterium
 - ◊ Automatische Erstellung einer Umsatzsteuervoranmeldung
- Kann-Kriterium
 - ◊ Ausdrucken der Voranmeldung auf einem von den Finanzämtern genehmigten Formular → notfalls kann das Formular auch handschriftlich ausgefüllt werden

- **Abgrenzungskriterien**

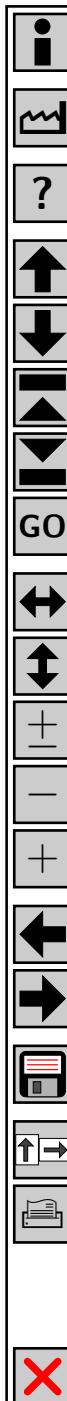
- Ziele, die mit dem Produkt ***bewusst nicht*** erreicht werden sollen
- Diese Ziele können durchaus Bestandteil vergleichbarer Anwendungen sein

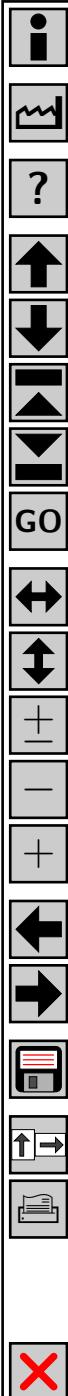
- **Beispiel 3:** Werkzeug für OO-Modelle

- Die automatische Optimierung bei der Darstellung von Diagrammen ist nicht vorgesehen

Einsatz

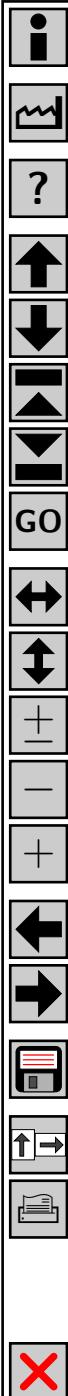
- Allgemeines
 - Die Analyse des Einsatzes liefert wichtige Informationen für die **Benutzungsoberfläche** und die **Qualitätsanforderungen** des zukünftigen Systems
- Anwendungsbereiche
 - **Wo** wird das System eingesetzt → z. B. Buchhaltung in einem Unternehmen
- Zielgruppen
 - **Wer** soll das System anwenden → z. B. Buchhalter
- Betriebsbedingungen
 - Angaben folgender Art:
 - ◊ Welche physikalische Umgebung liegt für das Software-System vor → z. B. Büroumgebung, Werkstatteinsatz
 - ◊ Die tägliche Betriebszeit
 - ◊ Ist eine ständige Beobachtung des Software-Systems durch den Bediener notwendig, oder liegt ein unbeaufsichtigter Betrieb vor





Umgebung

- **Software**
 - Auf welchen Software-Systemen (incl. Versionsnummern) soll das Produkt eingesetzt werden?
 - Welche Schnittstellen zu anderen Software-Produkten sind geplant?
- **Hardware**
 - Welche Hardware-Voraussetzungen sind vorhanden bzw. für den Produkteinsatz vorgesehen?
- **Orgware**
 - Unter welchen organisatorischen Randbedingungen bzw. Voraussetzungen soll das neue Produkt eingesetzt werden
 - Welche organisatorischen Schritte müssen durchgeführt werden, damit das Software-System eingesetzt werden kann?
- **Beispiel 4:** Orgware bei einem Buchhaltungssystem
 - Vor dem Einsatz eines Buchhaltungssystems muss zunächst von einem Buchhalter ein **Kontenplan** erstellt werden

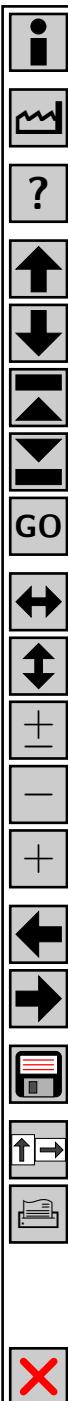


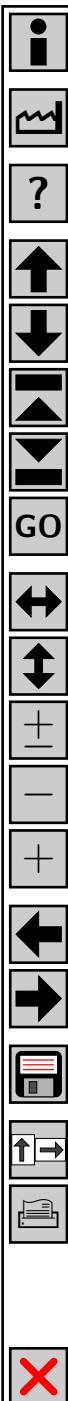
Funktionalität

- Beschreibung der Systemfunktionalität auf der **obersten Abstraktionsebene**
 - Aufzählung der **typischen Arbeitsabläufe**
 - Ein Arbeitsablauf soll immer zu einem **Ergebnis** für den Benutzer führen
 - Folge: Werden mehrere Arbeitsabläufe benötigt, um zu einem Ergebnis zu kommen, dann ist die Beschreibung der Arbeitsabläufe falsch
- !**
- es ist zu diesem Zeitpunkt noch nicht notwendigerweise abzusehen, ob ein Arbeitsablauf vollständig durch Software realisiert werden kann, oder auch organisatorische Schritte enthält
 - die beschriebenen Arbeitsabläufe bilden die **Grundlage** für die spätere Identifikation der **Geschäftsprozesse**
- Auflistung der wichtigsten **Dokumente**, die durch das System zu erstellen sind → Berichte, Reports
- !**
- dieses Kapitel des Pflichtenheftes soll die Basis für das spätere OOA-Modell legen
 - es wird keine vollständige textuelle Beschreibung der funktionalen Anforderungen verlangt

Daten

- Welche Daten sind aus Benutzersicht ***langfristig*** zu speichern?
- Welchen Umfang besitzen diese Daten voraussichtlich?
- **Beispiel 5:** Artikel- und Kundenverwaltung
 - Es sind ca. 50 000 bis 200 000 Artikel zu verwalten
 - Es sind ca. 3 000 Kunden zu verwalten



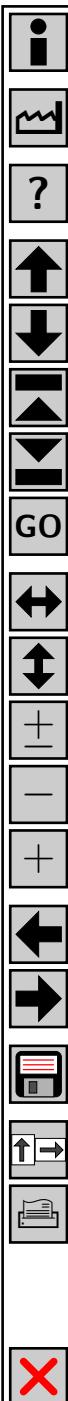


Leistungen

- Angabe aller **zeitlichen** Anforderungen
- **Beispiel 6:** Auffinden eines Artikels
 - Das Finden eines Artikel soll maximal 2 Sekunden dauern

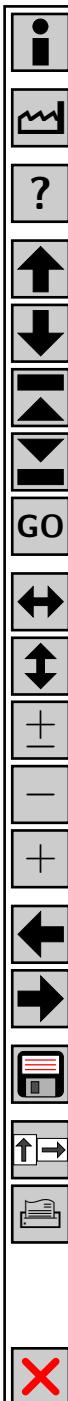


- Die Zeitvorgaben müssen mit den im Kapitel Daten genannten Datenmengen sowie der aufgeführten Hardware erreichbar sein
- Auf keinen Fall sollte sich erst nach der vollständigen Systemimplementierung herausstellen, dass eine Zeitvorgabe nicht erfüllbar ist



Benutzungsoberfläche

- Formulierung der **grundlegenden Anforderungen** an die Benutzungsoberfläche
 - Welche Eigenschaften (z. B. Erfahrung im Umgang mit Software-Systemen) besitzen die zukünftigen Benutzer?
 - In welcher Art (z. B. wie häufig) wird das System genutzt?
- **Beispiel 7:** Benutzung eines Buchhaltungssystems
 - Ein Buchhalter, der **täglich mehrere hundert** Buchungssätze eingibt, benötigt eine andere Benutzungsoberfläche als ein Freiberufler, der **monatlich** seine Umsatzsteuervoranmeldung mit einem Buchhaltungsprogramm erstellt

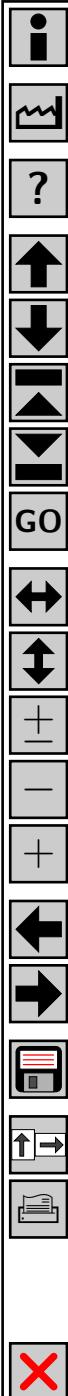


Qualitätsziele

- Welche Qualitätsmerkmale sind für das neue System besonders wichtig?
- **Beispiel 8:** Qualitätsziele
 - Wenn das System auf vielen **verschiedenen Plattformen** laufen soll, dann ist **Portabilität** ein wichtiges Kriterium
 - Einfache Wartbarkeit und Erweiterbarkeit sind oft wichtige Qualitätsziele

Ergänzungen

- Alle sonstigen Anforderungen an das Software-System, die den vorherigen Kapiteln thematisch nicht zuzuordnen sind, werden hier aufgeführt
→ **individuelle Erweiterung** des Pflichtenheftes

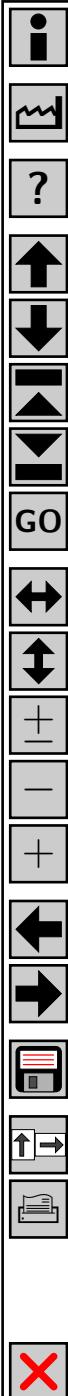


Methodische Vorgehensweise

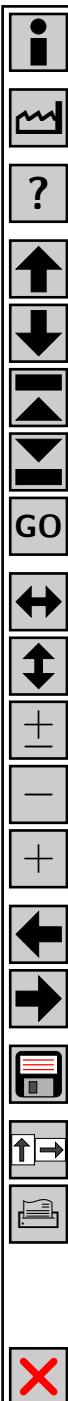
Überblick

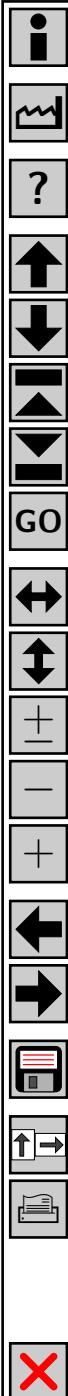
- Wie in vielen anderen Bereichen gilt auch für die Software-Entwicklung:
Eine **Produktverbesserung** kann nur durch eine Verbesserung des **Herstellungsprozesses** erreicht werden.
- Zur Herstellung des Produkts **AnalysemodeLL** dient der **Analyseprozess**
- Auswahl einer **methodischen Vorgehensweise**
 - Gratwanderung zwischen **Formalismus** und **Formlosigkeit**
 - Zuviel Formalismus ⇒ jegliche **Kreativität** wird **erstickt**
 - Formlose Vorgehensweisen sind **chaotisch** ⇒ die **Wahrscheinlichkeit** für einen **Projekterfolg** ist **sehr gering**

- BOOCH [Boo94] sieht fünf – sich etwas überlappende – **Schulen** (Richtungen) bei objektorientierten Methoden
 - Anarchists: Sie ignorieren alle methodischen Vorgehensweisen und verlassen sich nur auf ihre Kreativität
 - Behaviorists: Sie konzentrieren sich auf Rollen und Verantwortlichkeiten
 - Storyboarder: Sie sehen die Welt als Menge von Geschäftsprozessen
 - Information Modeller: Sie betrachten zunächst nur die Daten; das Verhalten ist sekundär
 - Architects: Sie haben ihren Fokus auf Frameworks (Rahmenarchitektur) und Patterns (Muster) gerichtet



- Zu starke Betonung des ***statischen Modells***
 - Es entsteht ein ***semantisches Datenmodell*** in objektorientierter Notation
 - Die ***Dynamik*** des Systems wird außer Acht gelassen
- Zu starke Betonung des ***dynamischen Modells***
 - Es entsteht eine ***funktionale Zerlegung*** des Systems
→ ***Use Case Driven Approach*** und ***Scenario Driven Approach***
 - Es wird sehr schwer, die funktionale Struktur auf eine objektorientierte Architektur abzubilden
- Für eine erfolgreiche Modellierung ist das ***Zusammenwirken*** von ***statischem*** und ***dynamischem Modell*** unabdingbar
- Zur Validierung des statischen Modells wird das dynamische Modell benötigt und umgekehrt





Die Struktur des Analyseprozesses

- **Definition 7:** Analyseprozess

Der **Analyseprozess** beschreibt die methodische Vorgehensweise zur Erstellung eines **objekt-orientierten Analysemodells**.

Er besteht aus einem **Makroprozess**, der die **grundlegenden methodischen Schritte** vorgibt, sowie **methodischen Regeln**, die situations- und anwendungsspezifisch verwendet werden.

- Methodische Schritte

- Beschreibung der Vorgehensweise auf einem sehr hohen Abstraktionsniveau → Makroprozess
- Festlegung der **Anwendungsreihenfolge** der einzelnen Schritte
- Für die praktische Durchführung einer Systemanalyse ist dieses grobe Vorgehensmodell noch nicht ausreichend

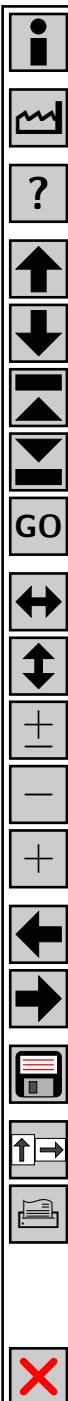


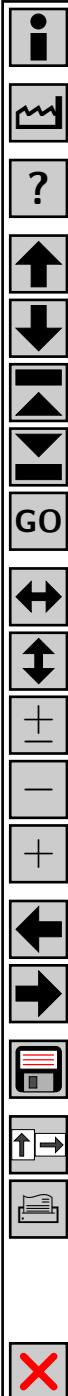
Ein zu detailliertes Vorgehensmodell ist jedoch oft nur für einen speziellen Anwendungsbereich einsetzbar und kann nicht problemlos auf andere Bereiche übertragen werden

- Methodische Regeln
 - Jedem methodischen Schritt werden methodische Regeln zugeordnet
 - Für die Regelanwendung gibt es keine festgelegte Reihenfolge
→ situationsspezifischer Einsatz



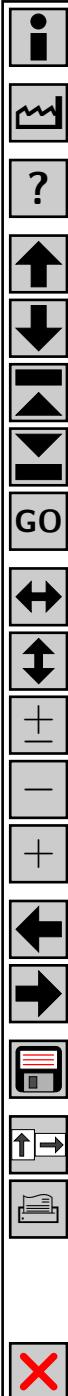
Die Anwendung methodischer Regeln führt zu der notwendigen Verfeinerung des Vorgehensmodells, die die praktische Durchführbarkeit sicherstellt





Zu beachtende Grundsätze

- Die Erstellung eines OOA-Modells ist ein ***kreativer Prozess***
- Ein Analyseprozess bietet üblicherweise Freiheitsgrade in der Anwendungsreihenfolge der Einzelschritte
- Unabhängig von der konkreten Vorgehensweise sollten aber folgende Grundsätze beachtet werden [Fow97]
 - Es gibt keine richtigen oder falschen Modelle; es gibt nur Modelle, die ***mehr oder weniger gut ihren Zweck*** erfüllen
 - Ein gutes Modell ist immer ***verständlich, d. h. es sieht einfach aus***
 - Die Erstellung verständlicher Modelle erfordert ***viel Aufwand***
 - Das Wissen von ***kompetenten Fachexperten*** ist absolut notwendig für ein gutes Modell
 - Das modellierte System sollte ***nicht zu flexibel*** sein und ***zu viele Sonderfälle*** enthalten; diese Modelle sind aufgrund ihrer Komplexität immer schwer verständlich und damit schlechte Modelle
 - Jeder Sonderfall sollte dahingehend überprüft werden, ob er es ***wert ist***, die Komplexität des Modells und des zu realisierenden Systems zu erhöhen



Balancierter Makroprozess

Struktur des verwendeten Analyseprozesses

Methodische Schritte für den balancierten Makroprozess

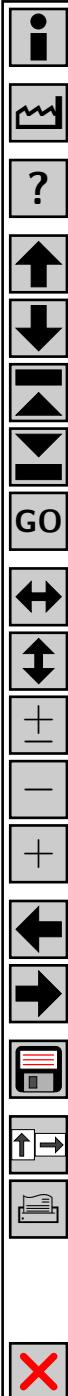
- Identifikation der relevanten Geschäftsprozesse
- Ermittlung der Klassen aus den Geschäftsprozessen
- Aufbau des (ersten) statischen Modells
- Erstellung des (ersten) dynamischen Modells
- Parallelе Weiterentwicklung des statischen und dynamischen Modells, um deren Wechselwirkungen angemessen berücksichtigen zu können

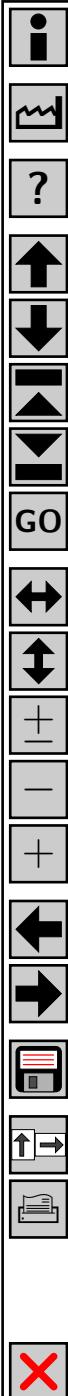


- **Vorteile durch die Konzentration auf das statische Modell vor dem dynamischen Modell**
 - Das Gesamtmodell besitzt eine größere Stabilität
 - Durch die Bildung von Klassen entsteht eine wesentliche Abstraktionsebene
- Der vorgestellte Makroprozess berücksichtigt die *Gleichgewichtigkeit (Balancing)* von statischem und dynamischem Modell → balancierter Makroprozess
- Der Systemanalytiker muss permanent zwischen beiden Modellen wechseln, bis ein akzeptables Analysemodell erstellt ist

Informationsquellen für das statische Modell

- Formulare
- Dateibeschreibungen
- Vorhandene Systemdokumentationen
- Interviews





Verwendung eines evolutionären Gesamtmodells

- Der balancierte Makroprozess wird im Rahmen eines ***evolutionären Gesamtmodells*** eingesetzt
- Struktur des evolutionären Gesamtprozesses → [Abb. 15](#)
 - Objektorientierte Analyse für einen ***Produktkern***
 - Entwurf und Implementierung des Produktkerns
 - Erweiterung (Analyse, Entwurf, Implementierung) des Produktkerns in den folgenden Zyklen, bis ein auslieferbares System vorhanden ist



- Ein evolutionärer Prozess ist immer ***iterativ***, weil er eine ständige Verfeinerung der Systemarchitektur erfordert
- Alle Erfahrungen und Ergebnisse eines Iterationsschrittes fließen in den nächsten Schritt ein

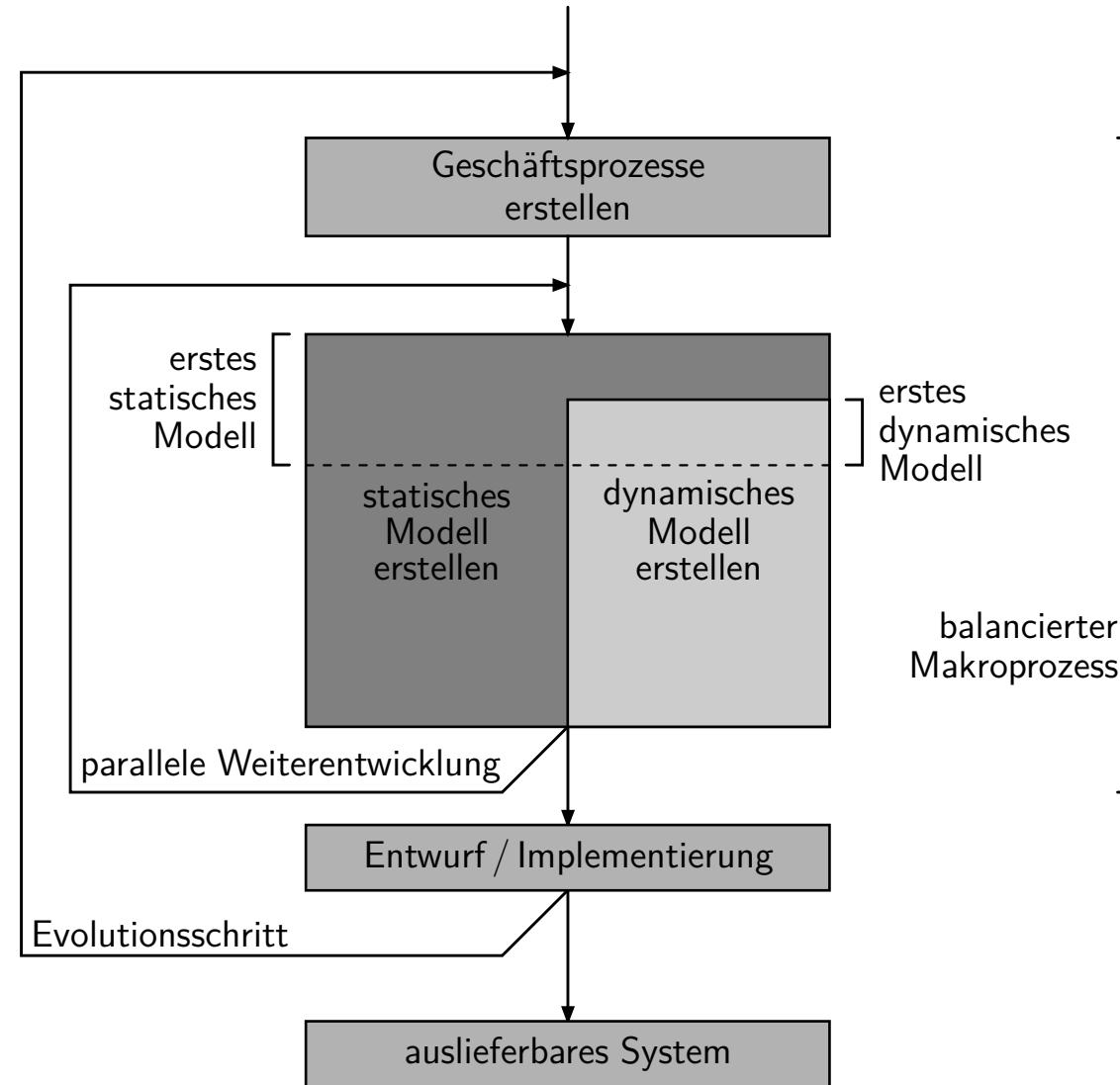
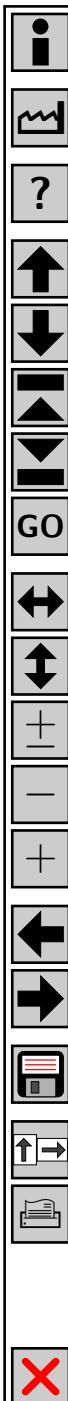
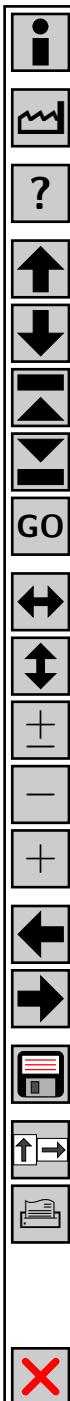


Abb. 15

Evolutionäre Erstellung des Gesamtsystems

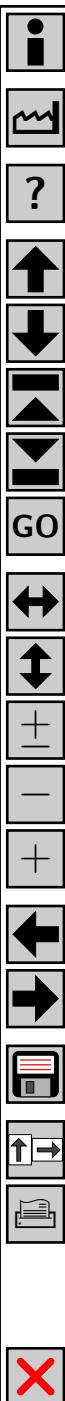
Integrierte Qualitätssicherung

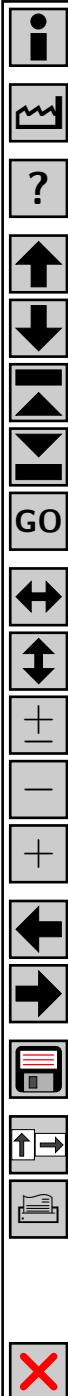
- Eine systematische Software-Entwicklung ist üblicherweise mit dem Einsatz von **Werkzeugen** verbunden
- Wichtige (wünschenswerte) Eigenschaften eines Werkzeuges
 - Durchführung formaler Prüfungen → Konsistenzprüfungen
 - ◊ Beispiel: Sind die verwendeten Klassennamen eindeutig?
 - ◊ Beispiel: Gibt es für die in den Sequenzdiagrammen verwendeten Objekte innerhalb des Klassendiagramms entsprechende Klassendefinitionen?
 - Einfache Durchführung von Änderungen und Erweiterungen
 - Generierung von Dokumenten
 - Verwaltung aller für das Software-System relevanten Dokumente (auch der extern – d. h. außerhalb des Werkzeuges – erzeugten Dokumente)
- Folge: Der Einsatz von Werkzeugen garantiert von vorneherein eine bestimmte Qualität





- Viele Qualitätskriterien sind jedoch *semantischer* (d. h. inhaltlicher) Natur und lassen sich nicht automatisch überprüfen
 - Analog zu einem Compiler finden die meisten Konsistenzprüfungen eines Systementwicklungswerkzeuges auf der *syntaktischen* Ebene statt
-
- Zur Sicherstellung der inhaltlichen Qualität enthält die detaillierte Vorstellung des balancierten Makroprozesses ***analytische Schritte***
 - Zusätzlich zu der integrierten Qualitätssicherung kann es nach der Fertigstellung des OOA-Modells eine ***formale Inspektion*** geben





Aufgabenbereiche des balancierten Makroprozesses

Überblick

- Analyse im Großen
 - 6 Schritte zum statischen Modell
 - 3 Schritte zum dynamischen Modell
- !** • Die bei der Analyse im Großen durchzuführenden Schritte sind *nicht spezifisch* für eine objektorientierte Entwicklung
- Statische und dynamische Modellierung besitzen dagegen einen objektorientierten Charakter

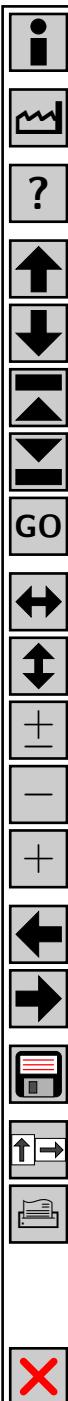
Analyse im Großen

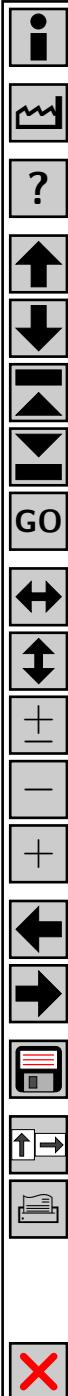
1. Geschäftsprozesse aufstellen

- Erstellung der essentiellen Geschäftsprozesse
 - *Beschreibung der Geschäftsprozesse* (textuell, semiformal, Aktivitätsdiagramm)
 - *Geschäftsprozessdiagramm*

2. Pakete bilden

- Bildung von Teilsystemen, d. h. Zusammenfassung der Modellelemente zu Paketen
- Bei großen Systemen, die im Allgemeinen durch mehrere Teams bearbeitet werden, muss die Bildung von Paketen am Anfang stehen
 - *Paketdiagramm*





6 Schritte zum statischen Modell

1. Klassen identifizieren

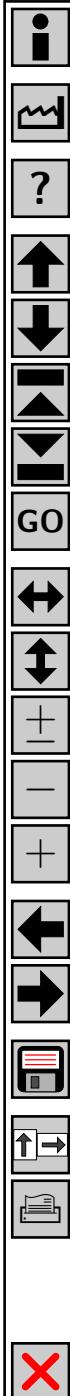
- Für jede Klasse werden nur so viele Attribute und Operationen identifiziert, wie es für das prinzipielle Problemverständnis notwendig ist
 - *Klassendiagramm*
 - *Kurzbeschreibung Klassen*

2. Assoziationen identifizieren

- Eintragen der reinen Verbindungen in das Klassendiagramm
- Weitere Angaben wie Kardinalität, Art der Assoziation etc. werden erst einmal weggelassen
 - *Klassendiagramm*

3. Attribute identifizieren

- Identifikation aller Attribute des Fachkonzepts
 - *Klassendiagramm*



4. Vererbungsstrukturen identifizieren

- Erstellung der Vererbungsstrukturen auf Basis der identifizierten Attribute (und Klassen)
→ *Klassendiagramm*

5. Assoziationen vervollständigen

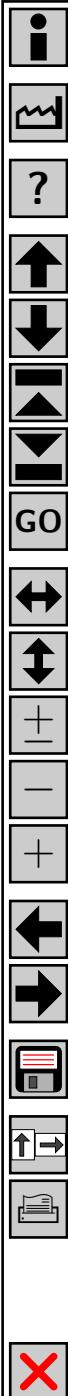
- endgültige Festlegung, ob eine einfache Assoziation, Aggregation oder Komposition vorliegt
- Angabe der Kardinalitäten, Rollen, Namen und Restriktionen
→ *Klassendiagramm*
→ *Objektdiagramm*

6. Attribute spezifizieren

- Erstellung der vollständigen Spezifikation für alle identifizierten Attribute
→ *Attributspezifikation*



- Die angegebenen Schritte müssen nicht notwendigerweise immer sequentiell durchlaufen werden
- Beispielsweise lassen sich oft gleichzeitig mit dem Identifizieren der Klassen auch Assoziationen finden



3 Schritte zum dynamischen Modell

1. Szenarios erstellen

- Präzisierung jedes Geschäftsprozesses durch eine Menge von Szenarios unter Verwendung von Interaktionsdiagrammen
 - *Sequenzdiagramm*
 - *Kommunikationsdiagramm*

2. Zustandsautomat erstellen

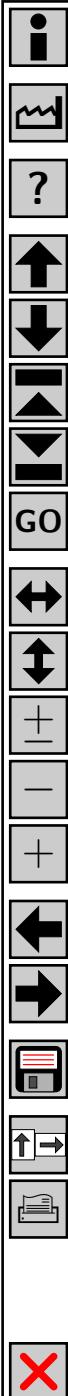
- Prüfung für jede Klasse, ob ein nicht-trivialer Lebenszyklus erstellt werden kann
 - *Zustandsdiagramm (Zustandsautomat)*

3. Operationen beschreiben

- Prüfung, ob eine Beschreibung notwendig ist
- Wenn ja, dann ist je nach Komplexitätsgrad die entsprechende Form zu wählen
 - *Fachliche Beschreibung der Operation*
 - *Zustandsdiagramm*
 - *Aktivitätsdiagramm*



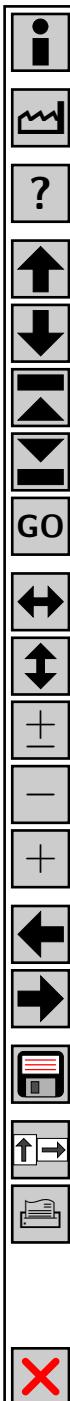
- Operationen kommen nicht nur im dynamischen Modell vor, sie werden auch in das Klassendiagramm eingetragen
- Sie stellen daher *die Verbindung* zwischen dem statischen und dem dynamischen Modell her

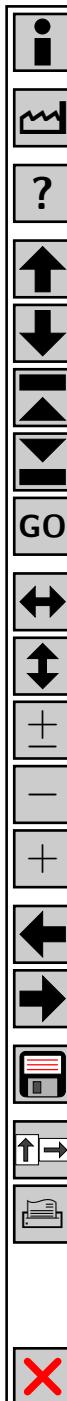


Alternative Makroprozesse

Szenariobasierter Makroprozess

- Einsatzkriterien
 - Umfangreiche funktionale Anforderungen liegen vor
 - Alte Datenbestände existieren nicht
- Struktur
 1. Geschäftsprozesse formulieren
 2. Szenarios aus den Geschäftsprozessen ableiten
 3. Interaktionsdiagramme aus den Szenarios ableiten
 4. Klassendiagramme erstellen
 5. Zustandsdiagramme erstellen



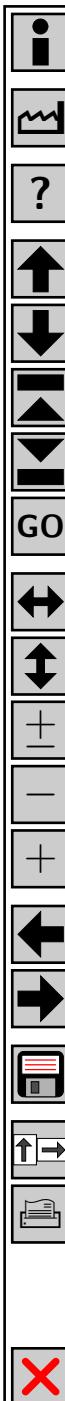


Datenbasierter Makroprozess

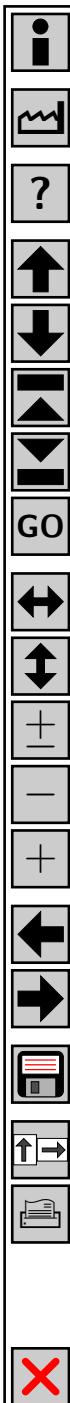
- Einsatzkriterien
 - Ein umfangreiches Datenmodell oder alte Datenbestände existieren
 - Der Umfang der funktionalen Anforderungen ist zunächst unbekannt
- Beispiel: Auskunftssystem mit flexibel gestaltbaren Anfragen
- Struktur
 1. Klassendiagramme erstellen
 2. Geschäftsprozesse formulieren
 3. Szenarios aus den Geschäftsprozessen ableiten
 4. Interaktionsdiagramme aus den Szenarios und dem Klassendiagramm ableiten
 5. Zustandsdiagramme erstellen

Häufige Fehler beim Analyseprozess

- Das 100 %-Syndrom
 - Das Problemverständnis muss nicht 100-prozentig sein, um mit der Entwurfsphase zu beginnen
 - Beispielsweise können fehlende Attribute oder Operationen in einer objektorientierten Architektur in einem späteren Iterationsschritt (Evolutionsschritt) leicht ergänzt werden
- Zu frühe Qualitätsoptimierung
 - Werden sehr früh schon Qualitätsoptimierungen vorgenommen, besteht die Gefahr, dass Änderungen des fachlichen Konzepts diese wieder invalidieren
 - Am Anfang der Modellentwicklung muss die Korrektheit des fachlichen Konzepts im Vordergrund stehen unabhängig von der Qualität des Modells
 - In einer zweiten Phase kann dann das fachlich korrekte Modell im Hinblick auf die Qualität des OOA-Modells verbessert werden

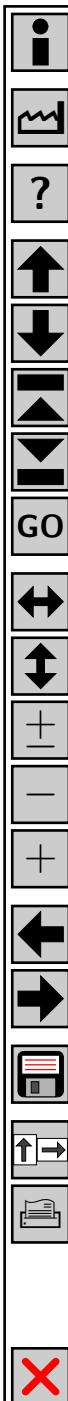


- Bürokratische Auslegung einer Methode
 - Man sollte nicht zu lange darüber diskutieren, wie der Methoden-Guru XY eine Aussage auf Seite z wohl gemeint haben könnte
 - *Follow the spirit, not the letter of a method*
- Entwurfskriterien in der Analyse berücksichtigen
 - Systemanalytikern, die zuvor jahrelang entworfen und implementiert haben, fällt oft die präzise Trennung von Analyse und Entwurf schwer



Zusammenfassung

- Das **Pflichtenheft** ist die Voraussetzung für die Erstellung des OOA-Modells aber kein Bestandteil der objektorientierten Analyse
- Der **Analyseprozess** beschreibt die methodische Vorgehensweise zur Erstellung eines OOA-Modells
- Der Analyseprozess besteht aus einem **Makroprozess** und **methodischen Regeln**
- Der balancierte Makroprozess berücksichtigt die Gleichwertigkeit von statischem und dynamischem Modell
- Der balancierte Makroprozess wird in ein **evolutionäres Vorgehensmodell** eingebunden



Glossar

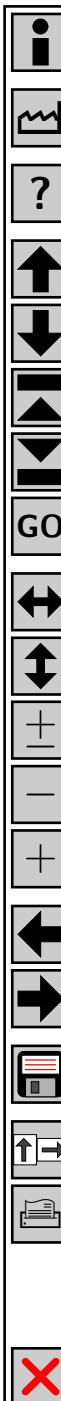
Analyseprozess: Der Analyseprozess beschreibt die methodische Vorgehensweise zur Erstellung eines objektorientierten Analysemodells. Er besteht aus einem → *Makroprozess*, der die grundlegenden Vorgehensschritte vorgibt, und der situations- und anwendungsspezifischen Anwendung von methodischen Regeln.

Balancierter Makroprozess: Der balancierte → *Makroprozess* unterstützt die Gleichgewichtigkeit von statischem und dynamischem Modell. Er beginnt mit dem Erstellen von → *Geschäftsprozessen* und der Identifikation von Klassen. Dann werden statisches und dynamisches Modell parallel erstellt und deren Wechselwirkungen berücksichtigt.

Datenbasierter Makroprozess: Beim datenbasierten → *Makroprozess* wird zunächst das Klassendiagramm erstellt und aufbauend darauf werden die → *Geschäftsprozesse* und die anderen Diagramme des dynamischen Modells entwickelt.

Makroprozess: Der Makroprozess beschreibt auf einem hohen Abstraktionsniveau die einzelnen Schritte, die zur systematischen Erstellung eines OOA-Modells durchzuführen sind. Der Makroprozess kann die Gleichgewichtigkeit von statischem und dynamischem Modell (→ *balancierter Makroprozess*) unterstützen oder → *datenbasiert* bzw. → *szenariobasiert* sein.

Szenariobasierter Makroprozess: Der szenariobasierte → *Makroprozess* beginnt mit dem Erstellen von → *Geschäftsprozessen* und → *Interaktionsdiagrammen* und leitet daraus das Klassendiagramm ab.

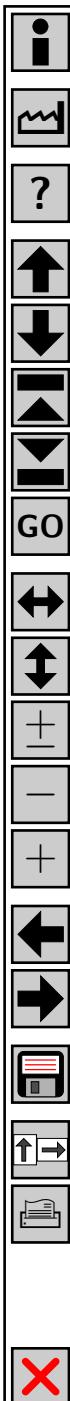


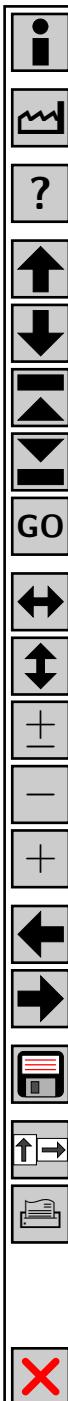
BASISKONZEPTE DER OBJEKTORIENTIERTEN ANALYSE

Lernziele

Verständnis

- Was ist ein ***Objekt***
- Der Unterschied zwischen ***externen*** und ***internen Objekten***
- Was ist eine ***Klasse***
- Was ist die Bedeutung der Objektverwaltung
- Was ist ein ***Attribut***
- Der Unterschied zwischen einem ***Objektattribut*** und einem ***Klassenattribut***
- Was ist eine ***Operation***
- Der Unterschied zwischen ***Objektoperationen***, ***Klassenoperationen*** und ***Konstruktoroperationen***





Anwendung

- Die UML-**Notation** für Objekte, Klassen, Attribute und Operationen
- Die **Identifikation** von **Objekten** und **Verbindungen** und ihre **Modellierung** im **Objektdiagramm**
- Die **Identifikation** von **Klassen**, **Attributen** und **Operationen** in einem **Text** und ihre **Modellierung** im **Klassendiagramm**
- **Spezifikation** von **Attributen**

Objekte

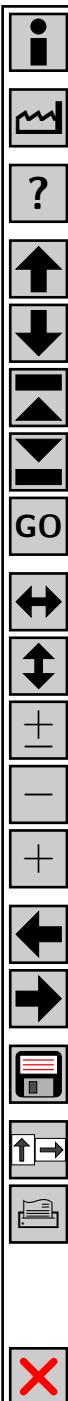
Begriffsdefinitionen

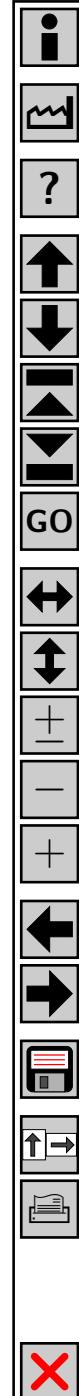
- **Definition 8:** Objekt im allgemeinen Sprachgebrauch

Ein Objekt ist ein **Gegenstand des Interesses**, insbesondere einer **Beobachtung, Untersuchung** oder **Messung**.

- **Beispiel 9:** Objekte im Sinne von Def. 8

- **Dinge** → z. B. Fahrrad, Büro
- **Personen** → z. B. Kunde, Mitarbeiter
- **Begriffe** → z. B. Programmiersprache, Krankheit





- **Definition 9:** **Objekt** innerhalb der **objektorientierten Software-Entwicklung**

Ein **Objekt** (*Object*) besitzt einen bestimmten **Zustand** und reagiert mit einem **definierten Verhalten** auf seine Umgebung.

Jedes Objekt besitzt eine **Identität**, die es von allen anderen Objekten unterscheidet.

Ein Objekt kann ein oder mehrere andere Objekte **kennen** (**Verbindungen** (*Link*)) zwischen Objekten.

- **Definition 10:** **Zustand** (*State*) eines Objekts

Der **Zustand** umfasst die **aktuellen Werte** der **Attribute**.

- Abgrenzung zwischen **Attributen** und **Attributwerten**

- Attribut

- ◊ Ein Attribut ist ein **inhärentes, unveränderliches Merkmal** eines Objekts

- Attributwert

- ◊ Der Attributwert kann **Änderungen** unterliegen

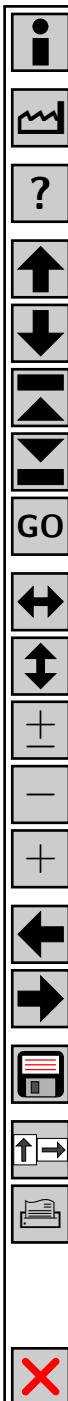
- **Definition 11:** **Verhalten** (*Behavior*) eines Objekts

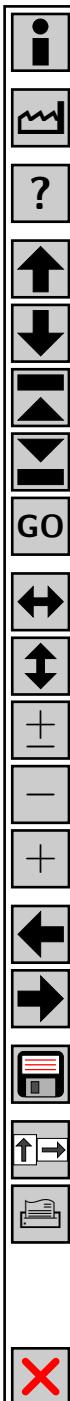
Das **Verhalten** eines Objekts wird durch eine Menge von **Operationen** beschrieben.

Eine **Änderung** oder **Abfrage** des Zustands ist nur mittels der Operationen möglich.

- **Beispiel 10:** Mitarbeiterobjekt

- Ein Mitarbeiter besitzt eine **Personalnummer**, einen **Namen** und erhält ein bestimmtes **Gehalt**
- Neue **Mitarbeiter** werden **eingestellt**
- Das **Gehalt** vorhandener Mitarbeiter kann **erhöht** werden
- Ein **Mitarbeiterausweis** kann **gedruckt** werden





- Abb. 16 zeigt ein **Mitarbeiterobjekt**
 - Die **Attribute** (Personalnummer, etc.) werden durch die **Operationen** vor der Außenwelt **verborgen**

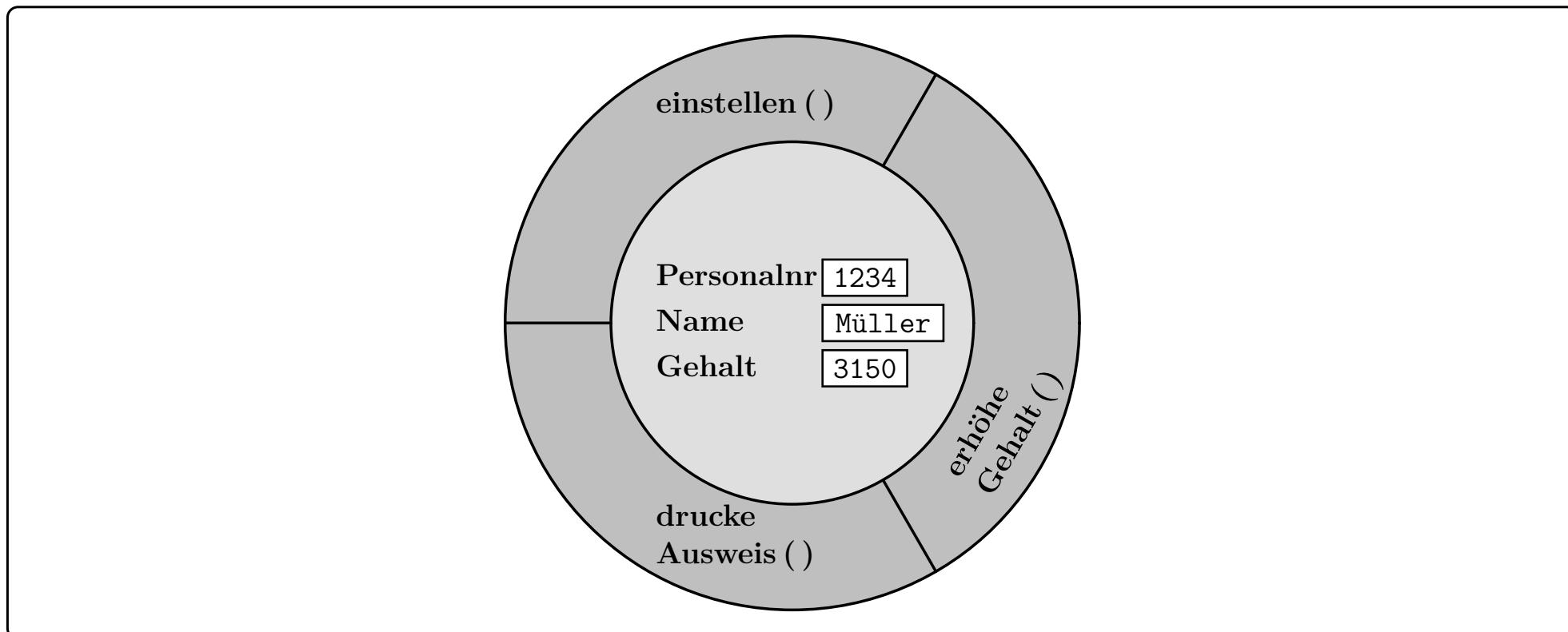


Abb. 16

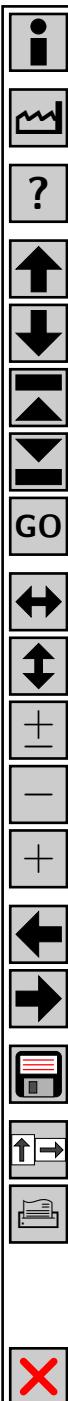
Mitarbeiterobjekt

UML-Notation für Objekte

- UML: UNIFIED MODELLING LANGUAGE



- Bedauerlicherweise ist die UML-Notation in den verschiedenen Lehrbüchern nicht immer einheitlich
- Das Originaldokument „Unified Modeling Language: Superstructure (version 2.0, formal/05-07-04)“ (vgl. www.uml.org) ist als Nachschlagewerk nur bedingt geeignet
- Die hier verwendete UML-Notation orientiert sich an [Bal99], [Kec06] und [JRH⁺⁰⁴]



- UML-Notation für ein Objekt (vgl. [Abb. 17](#))
 - Darstellung als Rechteck
 - Das Rechteck besitzt zwei Felder
 - Oberes Feld: **Objektbenennung**
 - ◊ Ein bestimmtes Objekt einer Klasse erhält für den Systemanalyseprozess einen Namen
 - Unteres Feld: enthält die im jeweiligen Kontext relevanten **Attribute**
 - ◊ Dieses Feld ist optional

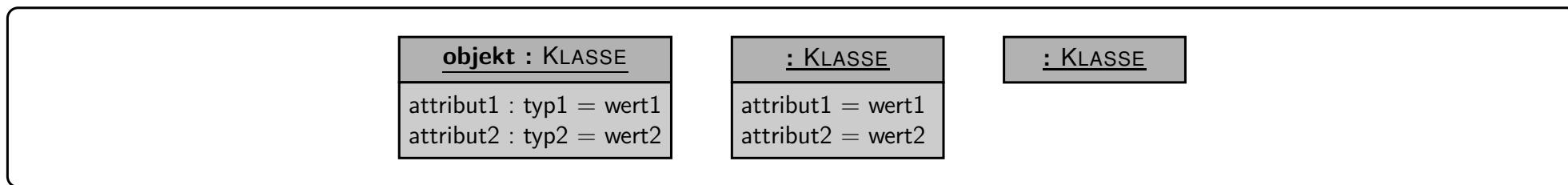
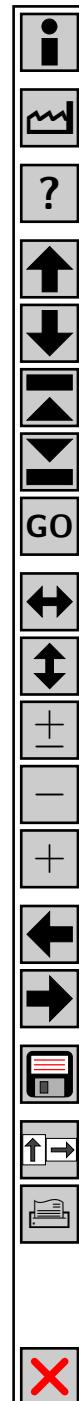


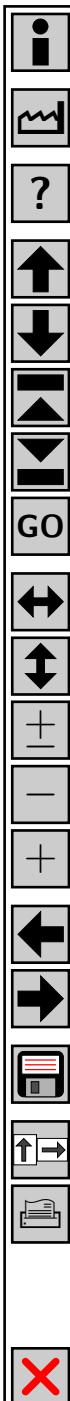
Abb. 17 UML-Notation für Objekte

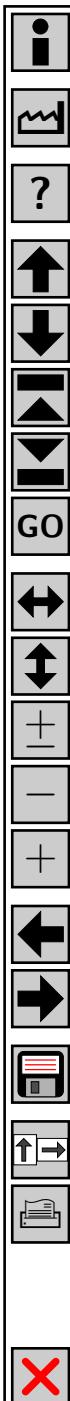


- Varianten der **Objektbenennung** (im Klassen- oder Objektdiagramm)
 - : KLASSE
 - ◊ Es handelt sich um ein ***anonymes*** Objekt der Klasse
 - ◊ Diese Variante wird benutzt, wenn es nur wichtig ist, zu welcher Klasse das Objekt gehört
→ **Standardfall**
 - objekt : KLASSE
 - ◊ Das Objekt soll über einen Namen angesprochen werden



- Der Objektname identifiziert nur ein UML-Element in einem Diagramm (z. B. Objektdiagramm)
- Er hat nichts mit einem Objektattribut „name“ zu tun
- Der Objektname kann beispielsweise verwendet werden, um in anderen Dokumenten auf das entsprechende Objekt Bezug nehmen zu können





- Alternativen bei der **Attributangabe**
 - Attribut : Typ = Wert
 - Attribut = Wert
 - ◊ Diese Form wird empfohlen, da der Attributtyp bereits bei der Klasse definiert ist
 - Attribut
 - ◊ Sinnvoll, wenn der konkrete Attributwert (hier) nicht von Interesse ist
- **Objektdiagramm (Object Diagram)** → siehe [Abb. 18](#)
 - Spezifikation von Objekten zusammen mit ihren **Verbindungen** untereinander
 - Wichtige Bestandteile
 - ◊ Objekte
 - ◊ Attributwerte
 - ◊ Verbindungen zwischen Objekten (symbolisiert durch Linien)

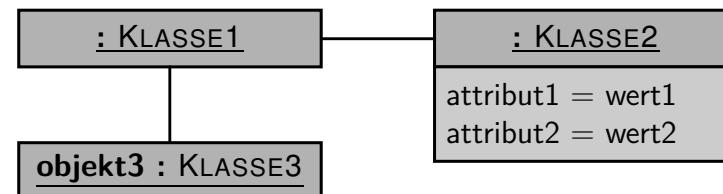
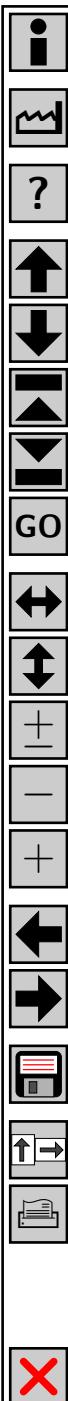
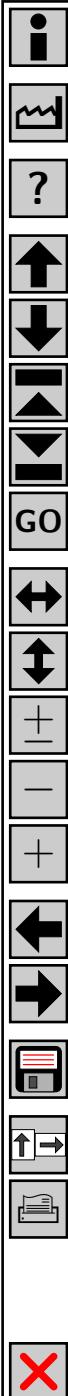


Abb. 18 UML-Notation für Objektdiagramme



- Ein Objektdiagramm kann als eine *Momentaufnahme (Schnappschuss, Snapshot)* des Systems aufgefasst werden
- Meistens werden *anonyme Objekte* verwendet
- Von einem Objektdiagramm spricht man, wenn das Diagramm ausschließlich Objekte (und evtl. Verbindungen) enthält
- Objekte können natürlich auch in Klassendiagrammen enthalten sein



Objektidentität, -gleichheit und -namen

Objektidentität

- **Definition 12: Objektidentität (*Object Identity*)**

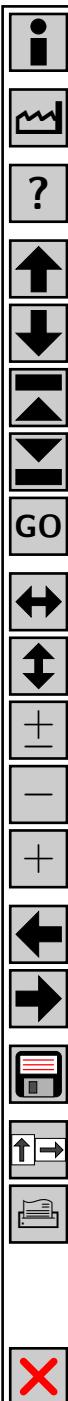
Die Objektidentität ist die Eigenschaft, die ein Objekt von **allen anderen** Objekten unterscheidet.

- **Bedeutung** der Objektidentität

- **Alle** Objekte (unabhängig von ihrer Klassenzugehörigkeit) sind aufgrund ihrer **Existenz** unterscheidbar
- Zwei Objekte können **niemals dieselbe** Identität besitzen
- Die Identität eines Objektes kann sich **nicht ändern**



- **Die Objektidentität ist kein explizit zugreifbares Attribut**
- **Objektidentitäten werden systemintern verwaltet, z. B. vom Laufzeitsystem einer objektorientierten Programmiersprache**



Objektgleichheit

- **Definition 13: Objektgleichheit**

Zwei Objekte – mit unterschiedlicher Identität – sind in der Regel **gleich**, wenn sie **dieselben Attributwerte** besitzen.



- Zwei gleiche Objekte müssen natürlich auch dieselben Attributtypen besitzen
- Der Vergleich von Objekten ist üblicherweise nur für Objekte derselben Klasse sinnvoll
- Oft ist es jedoch erforderlich, für die Objekte einer Klasse explizit eine Vergleichsoperation zu definieren

- **Beispiel 11:** Objektgleichheit und -identität (siehe Abb. 19)
 - Die Personen Michael und Susi haben beide *ein Kind mit dem Namen* Daniel → **Gleichheit**
 - Michael und Janine sind die Eltern **dieselben** Kindes → **Identität**

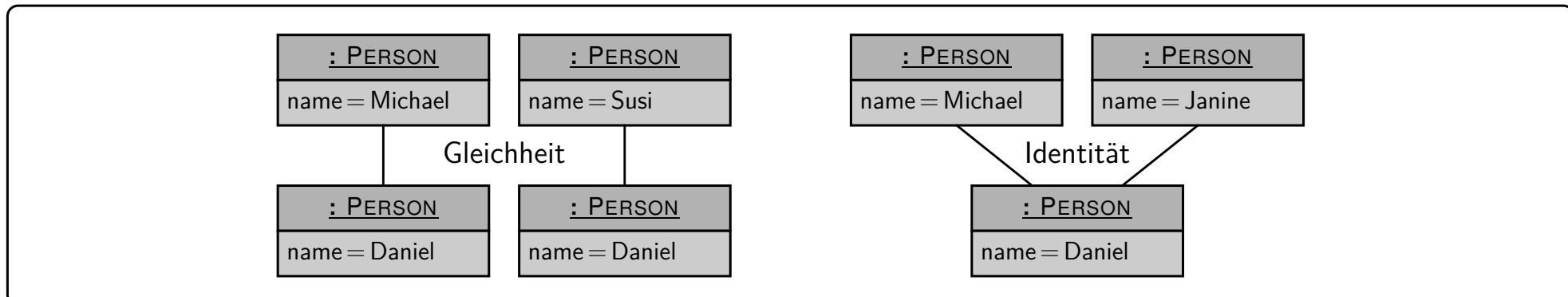
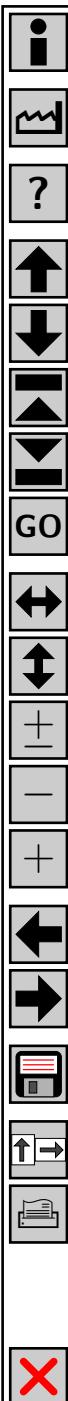
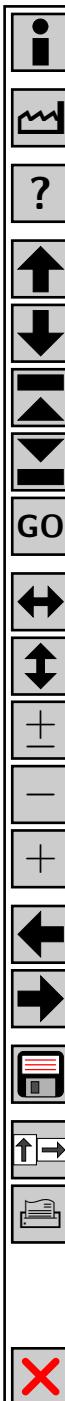


Abb. 19

Objektgleichheit und -identität





Objektnamen

- **Definition 14: Objektname**

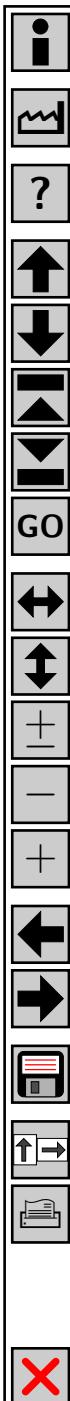
Der **Objektname identifiziert** ein Objekt im **Objektdiagramm**.



- Im Gegensatz zur Objektidentität muss der Objektname nur im betrachteten Kontext eindeutig sein (d. h. im Objektdiagramm)
- Besitzen Objekte in verschiedenen Diagrammen denselben Namen, so kann es sich um unterschiedliche Objekte handeln

Das Geheimnisprinzip

- **Struktur** eines Objekts
 - **Zustand** (Daten) und **Verhalten** (Operationen) bilden eine **Einheit**
 - Das Objekt **kapselt** Zustand und Verhalten
- Ein Objekt realisiert das **Geheimnisprinzip** (*Geheimniskonzept, Information Hiding*), wenn gilt:
 - Die Daten eines Objekts können ausschließlich unter Verwendung der Operationen gelesen und verändert werden → siehe [Abb. 20](#)
 - Ein **direkter Zugriff** von außen auf die Daten ist **verboten** (!!)



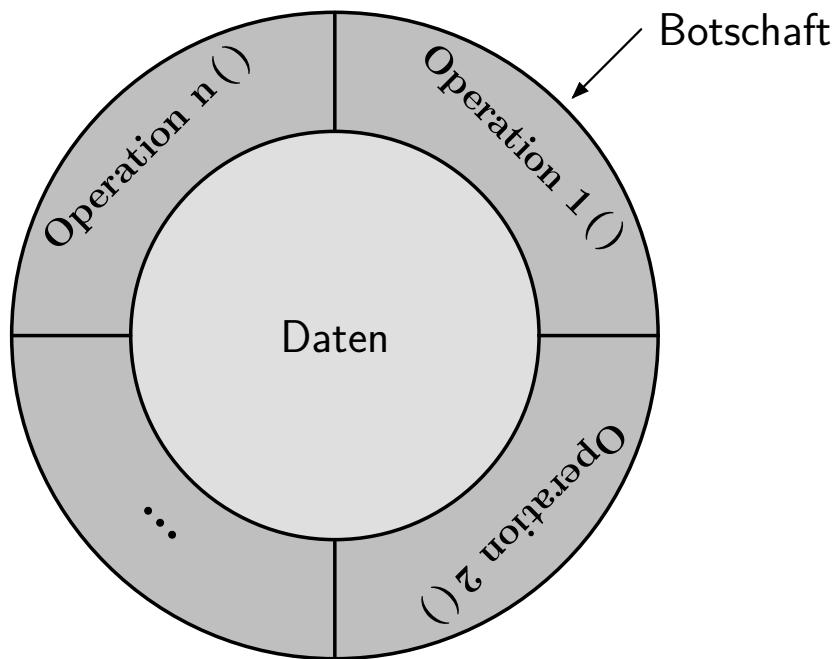
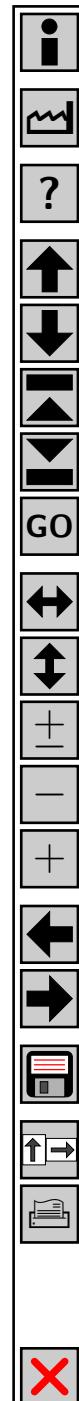


Abb. 20

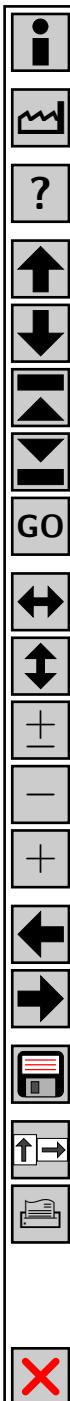
Das Geheimnisprinzip

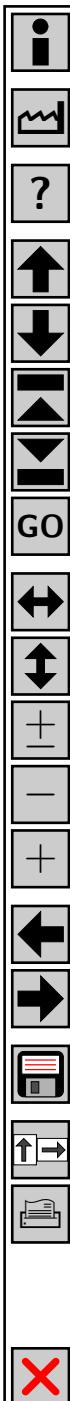


- Durch die Verwendung des Geheimnisprinzips wird die Repräsentation der Daten (d. h. ihre konkrete Implementierung) nach außen verborgen
- Wenn die Operationen semantisch korrekt implementiert sind, dann ist die Konsistenz der Daten sichergestellt

Externe und interne Objekte

- **Externe** Objekte
 - Existenz in der **realen Welt**
- **Internes** Objekt
 - Für ein **Software-System** relevantes Objekt
- Abbildung externer auf (interne) Objekte im OOA-Modell
(OOA: *Object-Oriented Analysis*)
 - Wahl einer für das Modell geeigneten **Objektabstraktion**
 - ◊ Üblicherweise ist nur ein kleiner Teil der Eigenschaften eines externen Objekts für das zu modellierende Software-System relevant
 - In der realen Welt sind Objekte oft **aktiv**, während sie im OOA-Modell **passiv** sind





- **Beispiel 12:** Modellierung einer Person
 - Der reale **Kunde** Müller, der Bankgeschäfte durchführt, soll modelliert werden
 - ◊ Die Eigenschaft, dass Müller in seiner Freizeit ein begeisterter **Golfspieler** ist, ist für eine Bank-Software uninteressant
 - ◊ Bei der Modellierung eines **Golftuniers** sind sicherlich andere Eigenschaften von Müller relevant
 - In der realen Welt ist Müller ein aktives Objekt
 - ◊ Er schickt z. B. Überweisungsaufträge an seine Bank
 - Im OOA-Modell einer Bank ist er ein passives Objekt
 - ◊ Es werden über den Kunden Müller Daten und Vorgänge gespeichert

Klassen

Begriffsdefinitionen

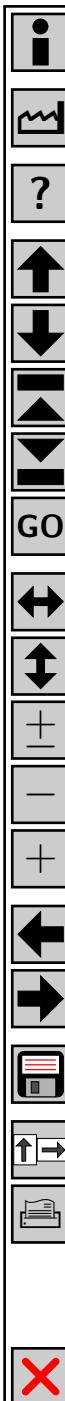
- **Definition 15: Klasse (Class)**

Eine Klasse **definiert** für eine **Menge** von Objekten deren **Struktur** (*Attribute*), **Verhalten** (*Operationen*) und **Beziehungen**.

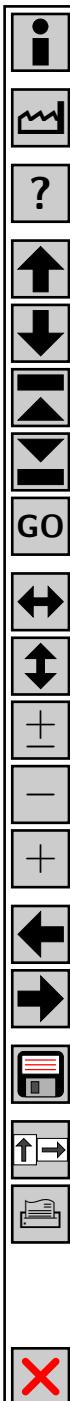
Sie besitzt einen **Mechanismus**, um **neue Objekte** zu erzeugen (*Object Factory*).

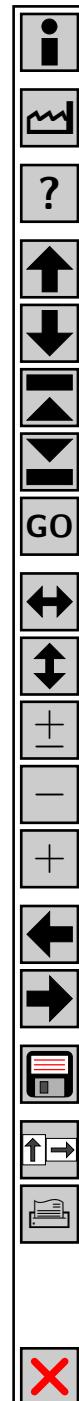


Jedes erzeugte Objekt gehört zu genau einer Klasse



- **Beziehungen** (*Relationship*)
 - Assoziationen
 - Vererbungsstrukturen
- **Verhalten** (*Behaviour*)
 - Beschreibung über die Menge der **Botschaften**, auf die Objekte reagieren können
 - Jede Botschaft aktiviert eine **Operation gleichen Namens**, d. h. Name der Botschaft = Name der Operation
 - die Differenzierung zwischen Botschaften und Operationen ist z. B. in verteilten Systemen wichtig





- **Beispiel 13:** Die Klasse Mitarbeiter
 - Die beiden Objekte aus Abb. 21 gehören zur Klasse Mitarbeiter
 - Daher besitzen sie **dieselben Attribute** und **Operationen**, aber in der Regel nicht diesselben Attributwerte

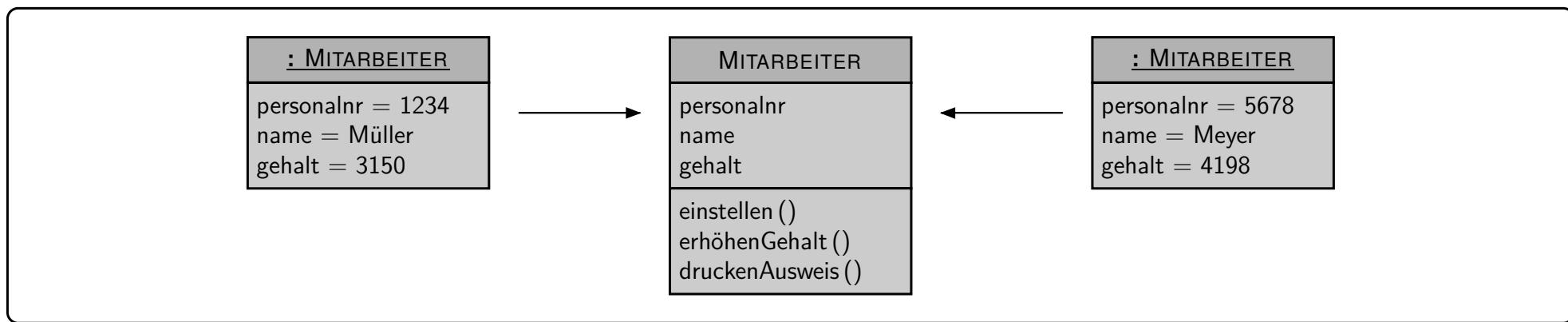


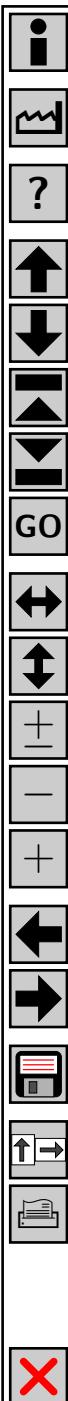
Abb. 21

Die Klasse Mitarbeiter

UML-Notation für Klassen

Notation

- UML-Notation für eine Klasse (vgl. [Abb. 22](#))
 - Namensfeld mit dem **Klassennamen**
 - ◊ Der Name **beginnt immer** mit einem **Großbuchstaben**
 - Attributliste
 - ◊ Neben den Attributnamen können weitere Angaben gemacht werden
→ siehe Abschnitt **Attribute**
 - ◊ Die Attributliste ist optional
 - Operationsliste
 - ◊ Neben dem Operationsnamen können weitere Angaben gemacht werden
→ siehe Abschnitt **Operationen**
 - ◊ die Operationsliste ist optional



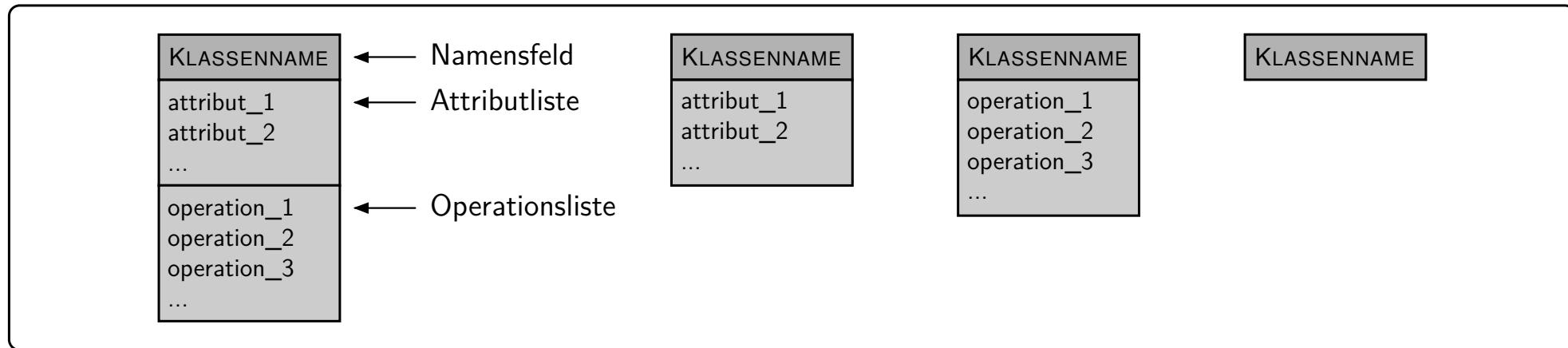
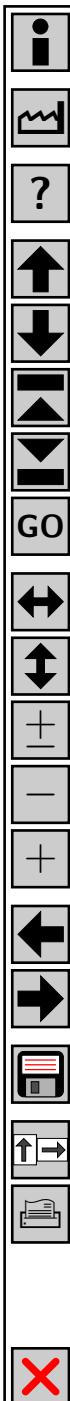


Abb. 22

UML-Notation für Klassen

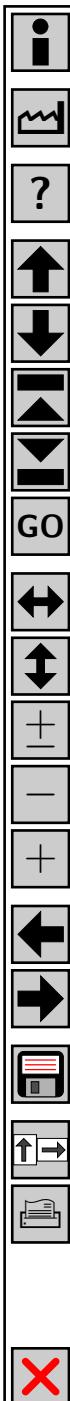


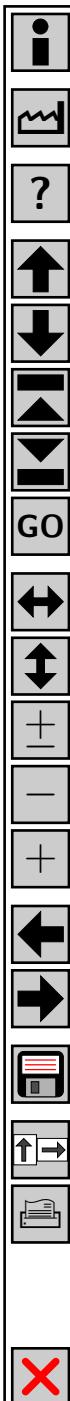
Klassendiagramm

- Das **Klassendiagramm** beschreibt das **statische Modell** des Systems

 Bei großen Systemen werden – schon aus Gründen der Übersichtlichkeit – mehrere Klassendiagramme zur Beschreibung verwendet

- (Wesentliche) Bestandteile des Klassendiagramms
 - Klassensymbole
 - Assoziationen
 - Vererbungsbeziehungen



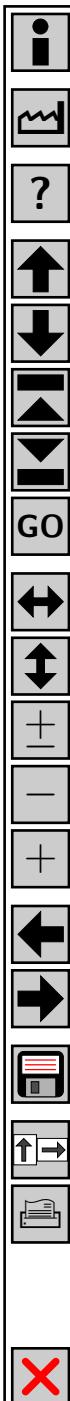


Klassenname

- Eindeutigkeit von Klassennamen
 - Eine Klasse kann einem **Paket** zugeordnet werden → ein **Paket** stellt ein **Teilsystem** des Gesamt-systems dar
 - Der Klassename (nur) **muss** innerhalb eines **Paketes eindeutig** sein
 - Bei Bedarf kann der Klassename in der UML wie folgt erweitert werden: **PAKETNAME::KLASSENNAME**
- (Empfohlener) Aufbau eines Klassennamens
 - **Substantiv** im Singular
 - Optionale Ergänzung durch ein **Adjektiv**

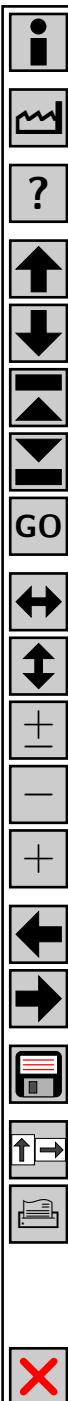
 **Der Klassename soll ein Objekt der Klasse beschreiben**

- **Beispiel 14:** Klassennamen
 - **MITARBEITER**
 - **KUNDE**
 - **KUNDENBESTANDALT**



Stereotypen und Merkmale

- (Optionale) Erweiterungen des Namensfeldes einer Klasse
 - Angabe eines **Stereotypen**
 - Angabe einer Liste von **Merkmale**
- Ein **Stereotyp** klassifiziert ein **Modellelement** (z. B. Klasse, Operation) näher → siehe [Abb. 23](#)
 - UML enthält einige vordefinierte Stereotypen, z. B. «interface»
 - Weitere Stereotypen können definiert werden
 - Stereotypen werden in **französischen Anführungszeichen** (*guillemets*) angegeben
→ z. B. «Stammdaten»



- Ein **Merkmal** (*Property*) beschreibt die Eigenschaften eines bestimmten **Modellelements** → siehe [Abb. 23](#)
 - Mehrere Merkmale können in Listenform zusammengefasst werden
 - Eine Merkmalsliste hat die Form { Schlüsselwort = Wert, ... } oder { Schlüsselwort }

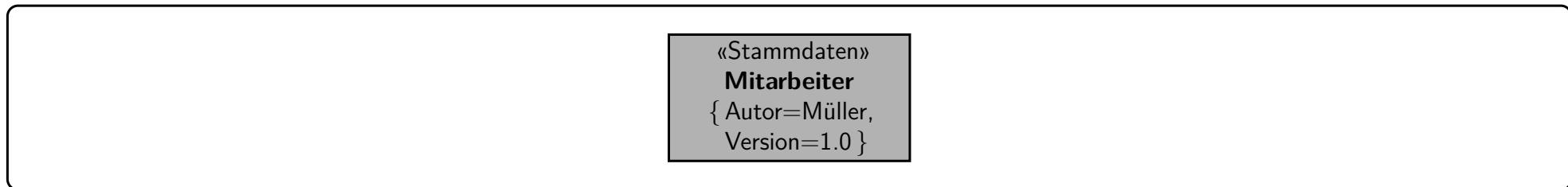
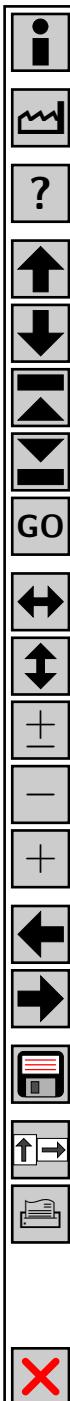


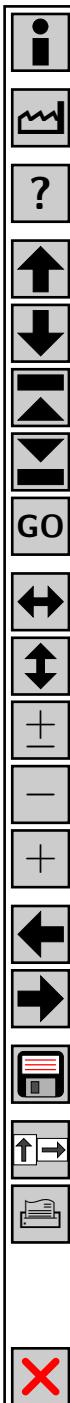
Abb. 23 Stereotypen und Merkmale

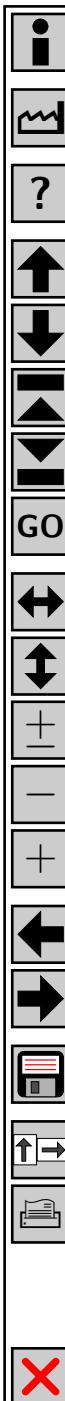
Abstrakte Klassen

- Eigenschaften einer **abstrakten Klasse**
 - **Keine Objekterzeugung** möglich
 - Definition der **Gemeinsamkeiten** einer Gruppe von Unterklassen
- Alternative Konzeptionsmöglichkeiten einer abstrakten Klassen
 - Mindestens eine Operation ist **abstrakt**
 - ◊ Für diese Operation sind nur der **Name** und ihre **Parametertypen** definiert
 - ◊ Der **Operationsrumpf** bleibt leer → keine Spezifikation oder Implementierung der Operationssemantik
 - Alle Operationen sind vollständig spezifiziert oder implementiert
 - ◊ Eine Objekterzeugung ist aber nicht vorgesehen
- Verwendung einer abstrakten Klasse
 - Ableitung von Unterklassen (Vererbungskonzept) → sofern abstrakte Operationen vorliegen, müssen diese spezifiziert bzw. implementiert werden



- UML-Notation
 - Verwendung eines *kursiv* geschriebenen Klassennamens
 - alternativ (oder zusätzlich) kann das ***Merkmal*** { abstract } verwendet werden





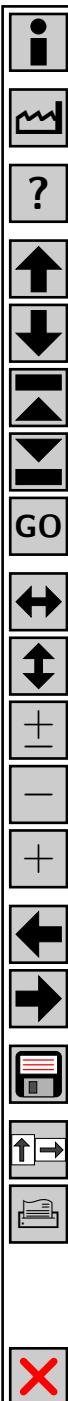
Klassenzugehörigkeit und Objektverwaltung

Klassenzugehörigkeit eines Objekts

- Ein Objekt **kennt** seine Klasse
- Alle Objekte einer Klasse besitzen **dieselben Operationen**
 - Die Operationen und ihre Spezifikationen bzw. Implementierungen werden der Klasse zugeordnet
 - Da ein Objekt seine Klasse kennt, kann es dort seine Operationen finden



- **Die Objekte einer Klasse besitzen natürlich auch dieselben Attributdefinitionen**
- **Da sich der Objektzustand durch die aktuellen Attributwerte definiert, benötigt jedes Objekt seine eigene Implementierung der Attribute**



Objektverwaltung

- Eine Klasse **weiß nicht**, welche Objekte sie **besitzt**
- Für die Phase der **Systemanalyse** ist die Kenntnis aller Objekte einer Klasse aber sehr praktisch
 - daher wird festgelegt, dass eine Klasse **Buch** über die erzeugten und gelöschten Objekte **führt**
→ **Objektverwaltung** (vgl. Abb. 24)
 - damit können z. B. **Anfragen** und **Manipulationen** auf der Menge aller Objekte einer Klasse durchgeführt werden
→ *Object Warehouse*

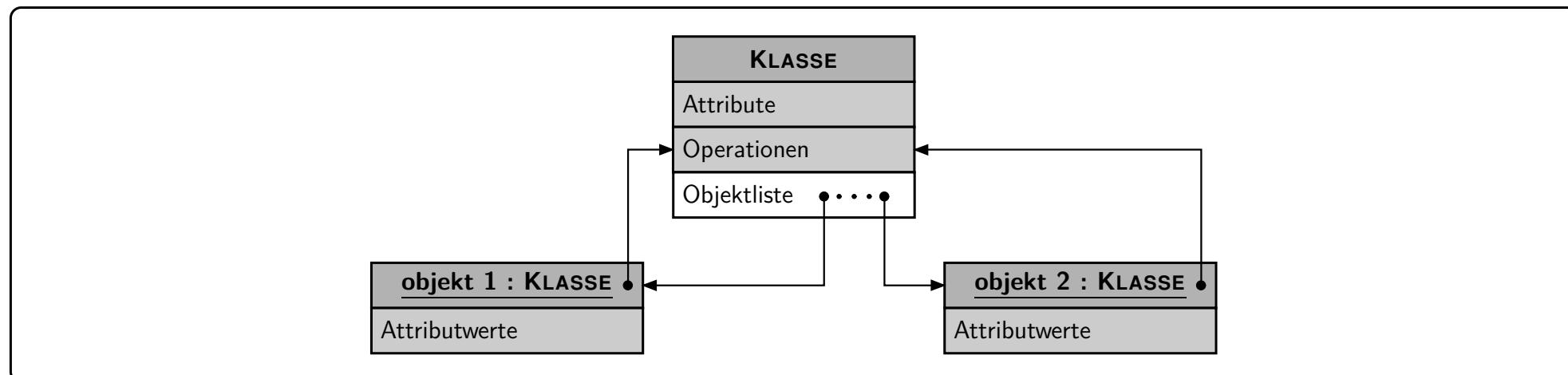
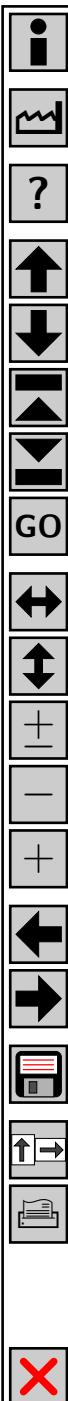


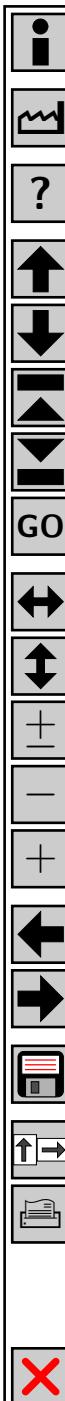
Abb. 24

Die Verwaltung der Objekte einer Klasse



- In den Phasen Entwurf und Implementierung muss die Objektverwaltung vom Programmierer realisiert werden
- Hierfür wird eine entsprechende Datenstruktur verwendet, z. B. eine Liste





Attribute

Begriffsdefinition

- **Definition 16:** *Attribut* (*Attribute*)

Die Attribute beschreiben die **Daten**, die von den Objekten einer Klasse angenommen werden können.

Jedes Attribut ist von einem bestimmten **Typ**.

Alle Objekte einer Klasse besitzen **dieselben Attribute**, jedoch (im Regelfall) **unterschiedliche Attributwerte**.

- **Beispiel 15:** Die Klasse **STUDENT** und ein Objekt

- Attributwerte dürfen *leer* sein → vgl. Abb. 25
- Für ein derartiges *optionales* Attribut kann der (erste) Attributwert zu einem beliebigen Zeitpunkt – oder auch nie – festgelegt werden

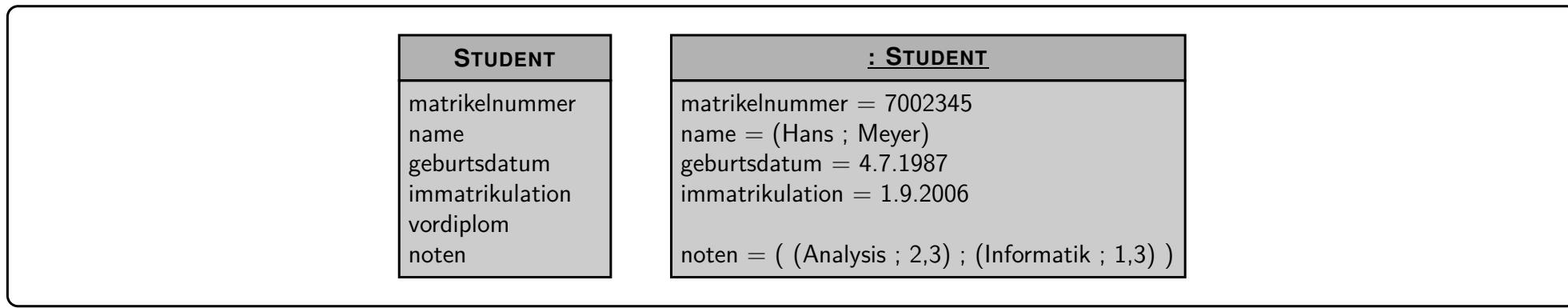
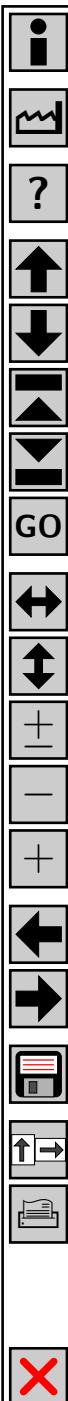
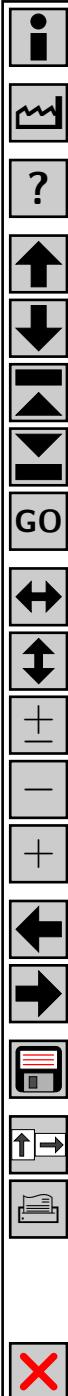


Abb. 25 Die Klasse STUDENT und ein Objekt

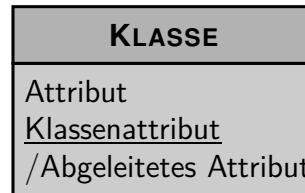




UML-Notation für Attribute

Notation

- UML kennt drei Attributvarianten → vgl. Abb. 26 (linke Seite)
 - (Objekt-) Attribut
 - Klassenattribut → unterstrichen
 - Abgeleitetes Attribut → Präfix / → Näheres siehe Vorlesung **Fallstudie Systemanalyse**
- Ein Attribut wird durch seinen **Namen** und seinen **Typ** beschrieben → vgl. Abb. 26 (rechte Seite)



Attribut : Typ [= Anfangswert] [{ Merkmal_1, Merkmal_2, ... }]
[...] optionales Beschreibungselement

Abb. 26 Die UML-Notation für ein Attribut

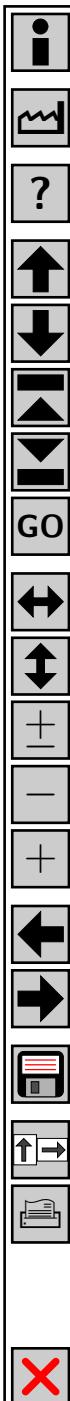
- **Optionale** Elemente einer Attributbeschreibung
 - **Anfangswert** (*Initial Value*)
 - ◊ Definition, welchen Wert das Attribut für ein **neu erzeugtes** Objekt annimmt
 - Liste von **Merkmalen**



- Zur besseren Lesbarkeit des Analysemodells wird in ein Klassendiagramm in der Regel nur der Attributname eingetragen
- Die weiteren Informationen zu einem Attribut werden dann separat beschrieben

Attributname

- UML-Namenskonventionen
 - Verwendung eines **Substantivs**
 - **Erster Buchstabe** wird **klein** geschrieben
- **Eindeutigkeit** von Attributnamen
 - Innerhalb einer **Klasse** muss der Attributname **eindeutig** sein
 - Außerhalb des Klassenkontextes wird bei OOA-Modellen die Form Klasse.attribut benutzt





Klassenattribute

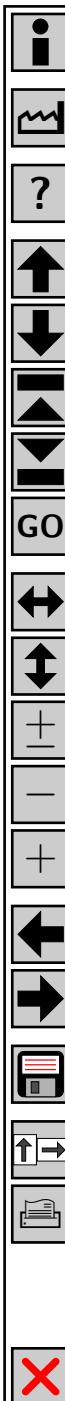
- **Definition 17: Klassenattribut**

Ein Klassenattribut liegt vor, wenn ***nur ein*** Attributwert ***für alle Objekte*** der Klasse existieren soll.



Ein Klassenattribut existiert auch dann, wenn *kein Objekt* der zugehörigen Klasse vorhanden ist

- Klassenattribute werden in der UML-Notation ***unterstrichen***
- **Beispiel 16:** Verwendung eines ***Klassenattributs*** in der ***Entwurfsphase***
 - Jedes Objekt einer Klasse MOTOR soll als Objektattribut eine ***eindeutige*** Motornummer erhalten
 - Diese Motornummer wird bei der Objekterzeugung (Produktion des Motors) festgelegt
 - Das Klassenattribut ***naechsteMotornummer*** enthält die als nächstes zu verwendende Motornummer
 - Nach jeder Objekterzeugung muss der Wert von ***naechsteMotornummer*** entsprechend geändert werden



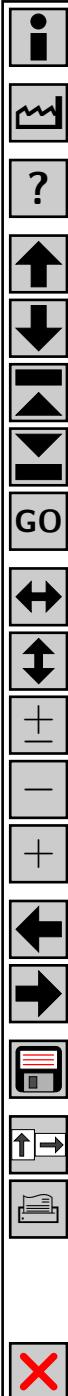
Attributtypen

Überblick

- UML definiert **keine** vorgegebenen Attributtypen
- In der Folge werden innerhalb der Systemanalyse vier Attributtypen unterschieden → Ziel: einheitliches Analysemodell
 - **Standardtypen**
 - **Aufzählungstypen** → Näheres siehe Vorlesung **Fallstudie Systemanalyse**
 - **Strukturtypen** (*strukturierter Typ, elementare Klasse*)



- **in der Analyse dient die Typdefinition dem Zweck, das Attribut aus fachlicher Sicht möglichst präzise zu beschreiben**
- **In den Phasen Entwurf und Implementierung wird in Abhängigkeit der verwendeten Programmiersprache der Attributtyp (eventuell) neu bestimmt**



Standardtypen

- [Tab. 1](#) enthält die Definition der Standardtypen
 - **vk**: Vorkommastellen, **nk**: Nachkommastellen
 - Die Angabe der Länge (**x**) bei einem **String**

Typ	Bedeutung	Wertebereich
String(x)	Zeichenkette	
Int	ganze Zahl	32 Bit
UInt	positive ganze Zahl	32 Bit
Float	Gleitkommazahl	32 Bit
Double	Gleitkommazahl	64 Bit
Fixed(vk, nk)	Festkommazahl	
Boolean	Wahrheitswert	true, false
Date	Datumswert	
Time	Zeit	

Tab. 1

Standardtypen für Attribute

Strukturtypen

- Ein **Strukturtyp** setzt sich aus **mehreren Attributen** zusammen → vgl. z. B. *Records* in PASCAL oder *Structs* in C



- Strukturtypen werden nicht ins Klassendiagramm eingetragen
- Sie werden einmal definiert und stehen dann allgemein zur Verfügung
- Strukturtypen werden mit dem Postfix T versehen

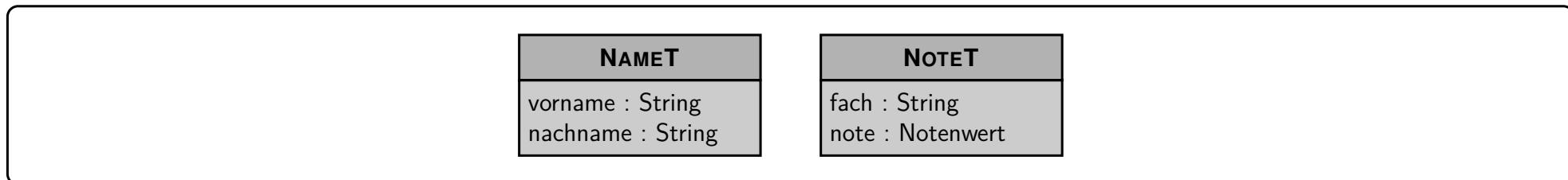
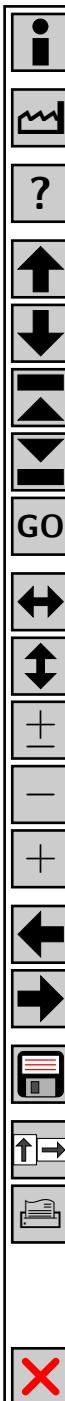
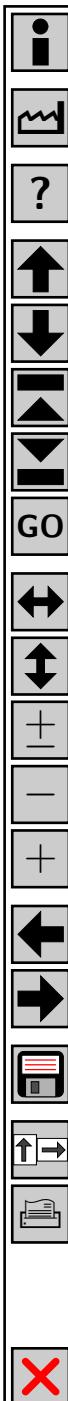


Abb. 27

Die Strukturtypen **NAMET** und **NOTET**



Operationen

Begriffsdefinition

- **Definition 18: Operation**

Eine Operation ist eine **ausführbare Tätigkeit**.

Alle Objekte einer Klasse verwenden **dieselben** Operationen.

Eine Operation kann auf **alle Attribute** eines Objekts dieser Klasse direkt zugreifen (Ausnahme: Klassenoperation).

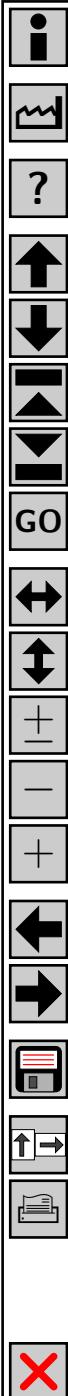
Die **Menge** aller Operationen wird als das **Verhalten** der Klasse oder als die **Schnittstelle** der Klasse bezeichnet.

- **Definition 19: Abstrakte Operation** (vgl. [S. 145](#))

Bei einer abstrakten Operation werden nur ihr *Name* und ihre *Parametertypen* definiert.

Der *Operationsrumpf* bleibt *leer*.

Eine abstrakte Operation kann nicht ausgeführt werden.



Operationstypen

Objektoperationen

- Eine **Objektoperation** (auch: **Operation**) wird stets auf ein einzelnes Objekt angewendet
 - Das Objekt **muss** bereits existieren
 - Auch das **Löschen** eines Objekts gehört zu den Objektoperationen
- **Beispiel 17:** Objektoperationen der Klasse **STUDENT** → siehe [Abb. 28](#)
 - Typische Beispiele für Objektoperationen
 - ◊ `druckenStudienbescheinigung()`
 - ◊ `notierenNote()`
 - ◊ `berechnenDurchschnitt()`
 - Löschen eines Studentenobjekts
 - ◊ `exmatrikulieren()`

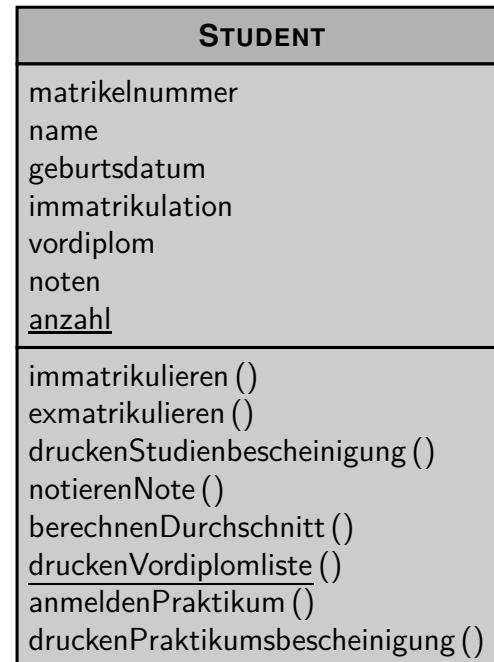
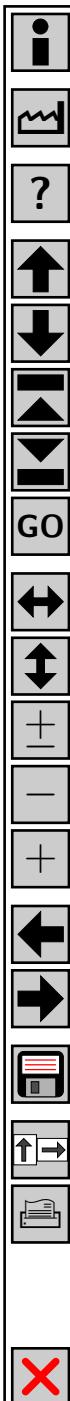


Abb. 28

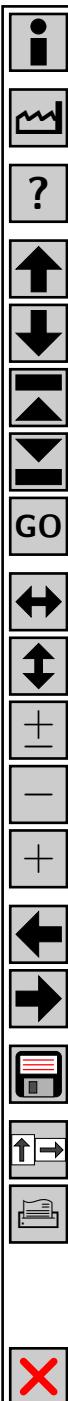
Die Klasse **STUDENT** und ihre Operationen

Klassenoperationen

- Eine **Klassenoperation** ist nicht einzelnen Objekten, sondern der Klasse zugeordnet
- Typische Anwendung → Manipulation eines **Klassenattributs**



- Eine Klassenoperation ist auch anwendbar, wenn noch kein Objekt der Klasse existiert
- Eine Klassenoperation hat *keinen* Zugriff auf *Objektattribute*



- **Beispiel 18:** Klassenoperation der Klasse **HiWi** → [Abb. 29](#)
 - **Stundenlohn** ist ein Klassenattribut → der Stundenlohn ist für alle Aushilfskräfte identisch
 - **erhöhenStundenlohn ()** ist eine Klassenoperation

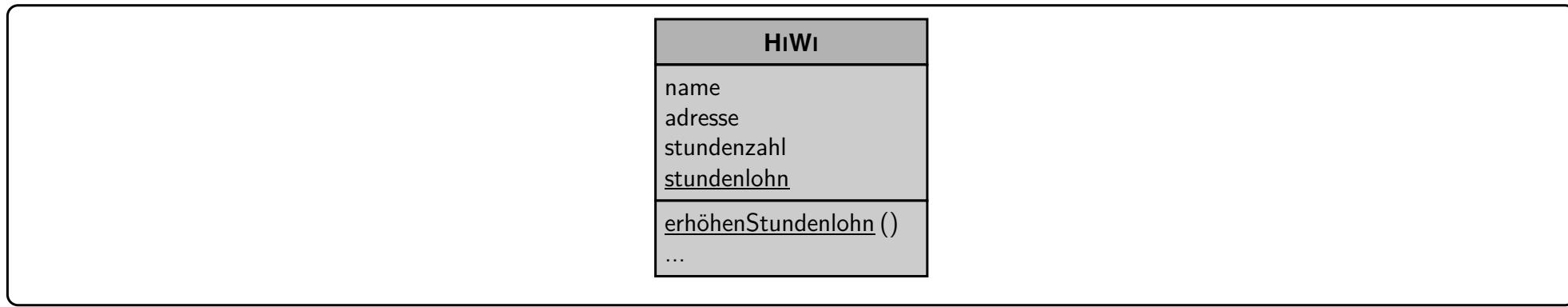
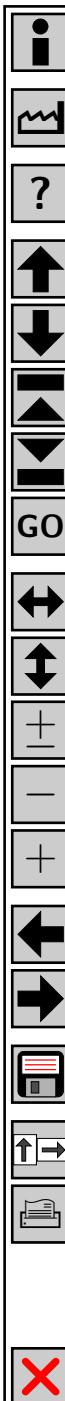


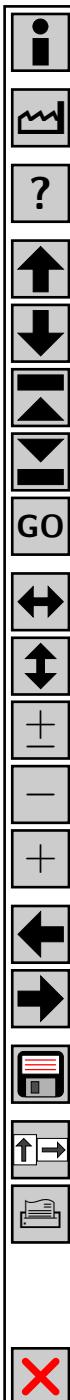
Abb. 29

Die Klasse **AUSHILFE**

Objektoperationen besitzen natürlich auch den Zugriff auf die Klassenattribute



- Anwendung von Klassenoperationen in der **Analysephase**
 - Manipulation der Klassenattribute (siehe oben)
 - Operationen auf **Objektmengen**
- Operationen auf Objektmengen
 - Eine Klassenoperation bezieht sich hier auf **mehrere** oder **alle** Objekte der Klasse
 - Ausnutzung der Objektverwaltungseigenschaft einer Klasse in der Analysephase (vgl. [S. 148](#))
 - Derartige Klassenoperationen werden auch als **Selektionsoperationen** bezeichnet
 - Dieser Typ der Klassenoperationen hat (über die selektierten Objekte) wieder Zugriff auf die Objektattribute (und natürlich auch auf die Klassenattribute)
- **Beispiel 19:** Klassenoperationen der Klasse **STUDENT** → siehe [Abb. 28](#)
 - **druckenVordiplomliste()** wählt unter allen Studenten diejenigen aus, die ein Vordiplom besitzen
 - Ausgabe einer entsprechenden Liste

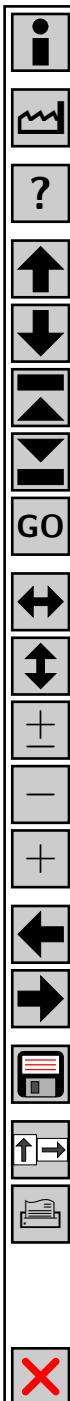


Konstruktoroperationen

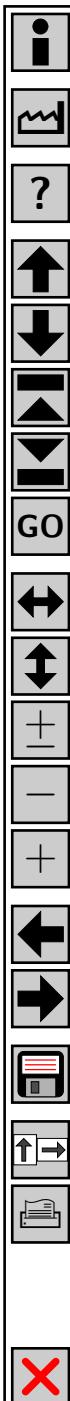
- Eine Konstruktoroperation **baut** – direkt nach der Speicherplatzzuweisung – die **Struktur** eines **neuen Objekts** der Klasse auf
 - Überprüfung, ob die **aktuellen Parameterwerte** des Konstruktoraufrufes **fehlerfrei** sind
→ andernfalls kann ein konsistenter Anfangszustand des Objektes nicht gewährleistet werden
 - Explizite **Berechnung** und **Initialisierung** bestimmter Attributwerte
 - Aufbau von z. B. **Array-** oder **Listenstrukturen**
 - Aufbau von **Objektbeziehungen**

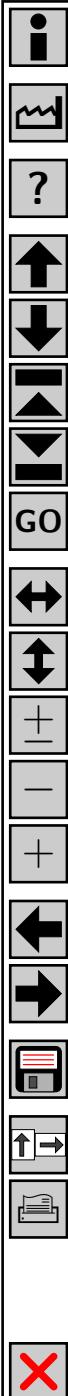


- Die Speicherplatzzuweisung erfolgt typischerweise durch den Aufruf eines **new-Operators**, der als **Klassenoperation** aufgefasst werden kann
- Ein Konstruktor selbst ist dann eine Objektoperation, weil bei seinem Aufruf das Objekt in rudimentärer Form bereits existiert
- Da ein Konstruktor in der Regel nur in Kombination mit dem **new-Operator** aufrufbar ist, wird ein Konstruktorauftrag (ein wenig fälschlicherweise) häufig auch mit einer Objekterzeugung gleichgesetzt



- **Beispiel 20:** Konstruktoroperation der Klasse **STUDENT** → [Abb. 28](#)
 - Die Operation **immatrikulieren()** baut ein neues Objekt der Klasse **STUDENT** auf
 - Vorgehensweise
 - ◊ Kontrolle, dass der Wert für das Attribut **name** den vorgegebenen Regeln entspricht, z. B.
 - ▷ Keine leeren Zeichenketten für Vor- und Nachname
 - ▷ Keine Verwendung von Umlauten, Ziffern, Sonderzeichen
 - ◊ Erzeugung eines eindeutigen Wertes für das Attribut **matrikelnummer**

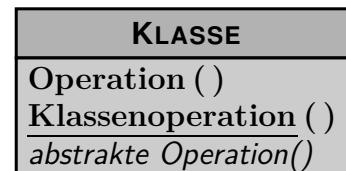




UML-Notation für Operationen

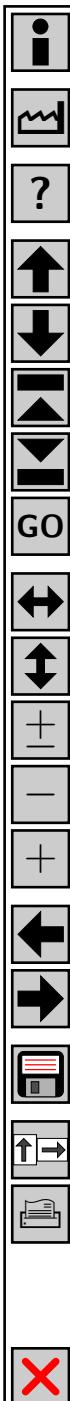
Notation

- Abb. 30 zeigt die UML-Notation für Operationen
 - Eine Operation wird über ihren **Namen** beschrieben
 - **Klassenoperationen** werden **unterstrichen**
 - Zusätzlich kann für jede Operation eine **Merkmalsliste** angegeben werden



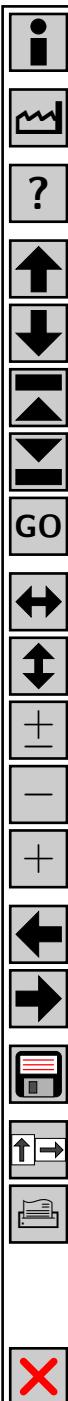
Operation () { Merkmal1, Merkmal2, ... }

Abb. 30 Die UML-Notation für Operationen



Operationsname

- Der Name soll ausdrücken, **was** eine Operation **leistet**
- Namenskonventionen
 - Verwendung eines **Verbs**
 - Zusätzlich können **Substantive** benutzt werden
 - Das Verb ist die erste Namenskomponente
 - Der Operationsname beginnt mit einem **Kleinbuchstaben**
 - Die (optionalen) Substantive beginnen jeweils mit einem **Großbuchstaben**
- **Beispiel 21:** Operationsnamen
 - `verschieben()`
 - `erhöhenGehalt()`
- Eindeutigkeit
 - Der Name muss im Kontext der Klasse eindeutig sein
 - Außerhalb des Klassensymbols (z. B. in einem Begleittext) wird eine Operation mit `Klasse.operation()` bezeichnet



Merkmaliste

- Die Merkmalsliste (vgl. [Abb. 30](#)) kann z. B. verwendet werden, um eine abstrakte Operation – statt durch die kursive Schreibweise – mit dem Merkmal { abstract } zu kennzeichnen
- Sie kann – aus Platzgründen – auch außerhalb des Klassensymbols angegeben werden

Verwendung von Stereotypen

- Operationen können unter Verwendung von **Stereotypen** zur besseren Übersichtlichkeit **gruppiert** werden
→ siehe [Abb. 31](#)

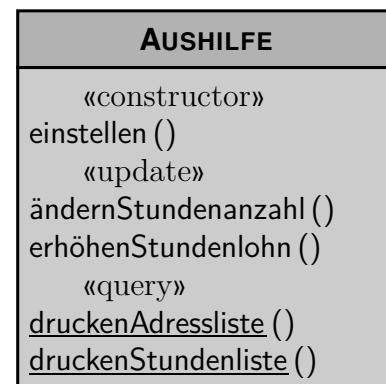


Abb. 31

Verwendung von Stereotypen zur Operationsgruppierung

Externe und interne Operationen

- **Externe** Operation

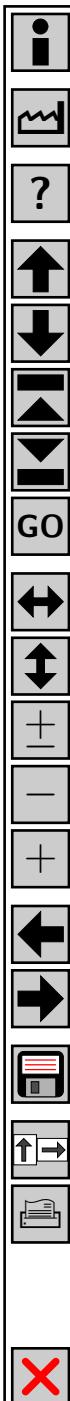
- Die Operation kann von **außerhalb** des Systems aufgerufen werden
- Beispiel: Aufruf einer Operation über die Benutzeroberfläche

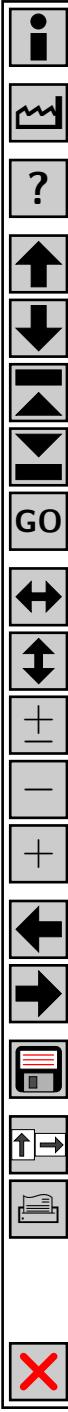
- **Interne** Operation

- Aktivierung durch eine Operation des Systems
- Die aktivierende Operation kann eine externe oder interne Operation sein



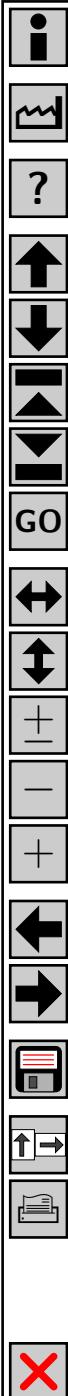
- Das Ziel der **Systemanalyse** ist die Ermittlung der **externen** Operationen
- Eine **interne** Operation wird nur ins **Klassendiagramm** aufgenommen, wenn sie für das Verständnis erforderlich ist





Klassifikation von Operationen nach ihrem Verwendungszweck

- Operationen mit **lesendem** Zugriff auf Attribute derselben Klasse (*Accessor Operation*)
 - Beispiel: `druckenStudienbescheinigung()` → vgl. [Abb. 28](#)
- Operationen mit **schreibendem** Zugriff auf Attribute derselben Klasse (*Update Operation*)
 - Beispiel: `notierenNote()` → vgl. [Abb. 28](#)
- Operationen zur Durchführung von **Berechnungen**
 - Beispiel: `berechnenDurchschnitt()` → vgl. [Abb. 28](#)
- Operationen zum **Konstruieren** (Erzeugen) von Objekten (*Constructor Operation*)
 - Beispiel: `immatrikulieren()` → vgl. [Abb. 28](#)
- Operationen zum **Löschen** von Objekten (*Destructor Operation*)
 - Beispiel: `exmatrikulieren()` → vgl. [Abb. 28](#)



- Operationen, die Objekte einer Klasse nach bestimmten Kriterien **selektieren** (*Query Operation, Select Operation*)
 - Beispiel: `druckenVordiplomliste()` → vgl. [Abb. 28](#)
- Operationen zum **Herstellen** von **Verbindungen** zwischen Objekten (*Connect Operation*)
 - Beispiel: siehe [Abb. 32](#)
 - ◊ Der Student **s1** will ein Praktikum bei einer Firma absolvieren
 - ◊ Von **s1** wird zum Firmenobjekt eine Verbindung aufgebaut: `anmeldenPraktikum()`
→ vgl. [Abb. 28](#)

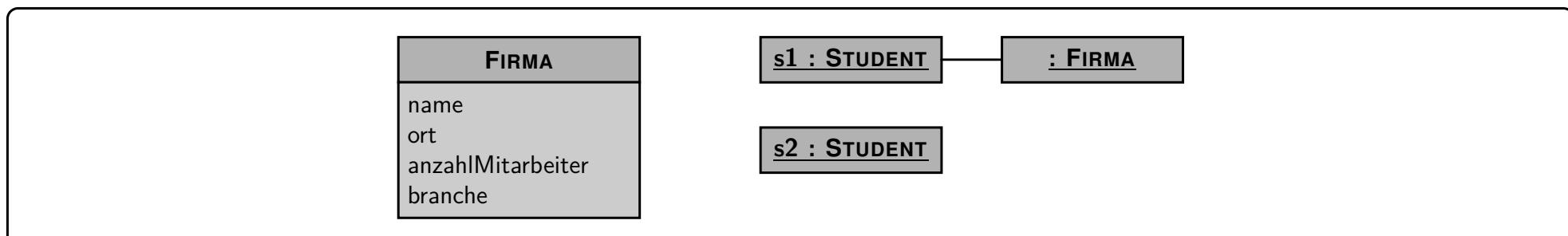
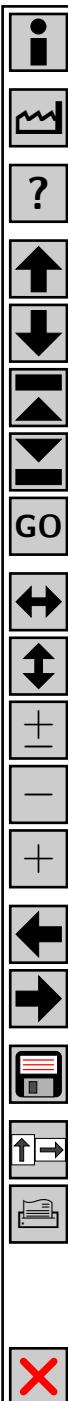


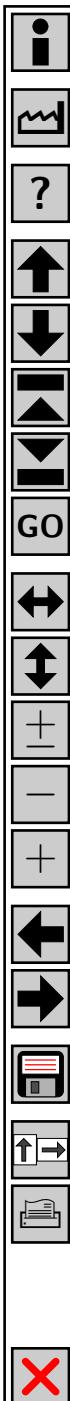
Abb. 32 Aufbau einer Verbindung zwischen Objekten

- Operationen zum **Abbau** von **Verbindungen** (*Disconnect Operation*)
 - Beispiel: Beendigung eines Praktikums
- Operationen, die Operationen auf einem oder mehreren Objekten **anderer Klassen** aktivieren
 - Beispiel: Ausdrucken einer Praktikumsbescheinigung
 - ◊ Aufruf der Operation **druckenPraktikumsbescheinigung()** für das Objekt **s1**
→ vgl. [Abb. 28](#))
 - ◊ Diese Operation nutzt dann die Objektverbindung zu **:Firma**, um auf deren Operationen zum Auslesen der benötigten Attributwerte zuzugreifen → vgl. [Abb. 32](#)

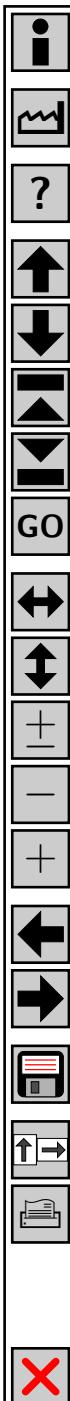


Basis- und Verwaltungsoperationen

- Basisoperationen sind **grundlegende** Operationen, die fast jede Klasse benötigt
 - Viele dieser Operationen sind **interne** Operationen
 - Sie werden insbesondere bei einer detaillierten Beschreibung der **Objektinteraktionen** benötigt
→ Spezifikation von Interaktionsdiagrammen
 - Sie werden üblicherweise **nicht** in das Klassendiagramm eingetragen
- Interne, elementare Basisfunktionen
 - **new ()**: Erzeugen eines neuen Objekts
 - **delete ()**: Löschen eines Objekts
 - **setAttribute ()**: Schreiben eines Attributwertes, z. B. **setGehalt ()**
 - **getAttribute ()**: Lesen eines Attributwertes, z. B. **getGehalt ()**
 - **link ()**: Aufbau einer Verbindung zwischen Objekten
 - **unlink ()**: Entfernen einer Verbindung zwischen Objekten
 - **getlink ()**: Lesen einer Verbindung zwischen Objekten



- Externe Verwaltungsoperationen werden oft zur Modellierung von Interaktions- und Zustandsdiagrammen benötigt → Beschreibung dynamischer Abläufe
 - Sie werden in der Regel nicht in das Klassendiagramm eingetragen
- Externe Verwaltungsoperationen
 - **erfassen ()**: Erfassen eines neuen Objekts, wobei im Unterschied zur Basisoperation **new ()** weitere Aufgaben, z. B. das Senden von Botschaften an andere Objekte, damit verbunden sein können
 - **ändern ()**: Ändern eines vorhandenen Objekts
 - **löschen ()**: Löschen eines Objekts
 - **erstelleListe ()**: alle Objekte der Klasse anzeigen



Spezifikation von Operationen

- Beschreibung der **Funktionsweise** einer Operation aus **Benutzersicht**
 - Dies ist notwendig, wenn der Operationsname nicht aussagekräftig ist
 - Die Beschreibung erfolgt (üblicherweise) **umgangssprachlich**
- Beschreibung einer Operation (UML-Erweiterung)

Funktion: **tueEtwas()**

Eingabe: Eingabedaten

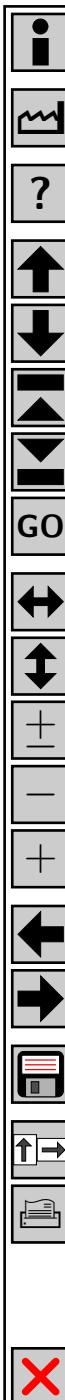
Ausgabe: Ausgabedaten

Wirkung: Beschreibung der Wirkung aus Benutzersicht, wobei der Fokus auf dem Normalverhalten liegt.

Sonderfälle sind anschließend zu beschreiben.

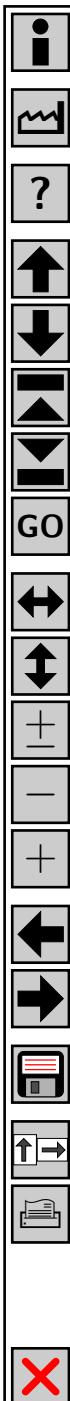


Das Ziel ist eine *leicht lesbare* Beschreibung und keine detaillierte Beschreibung einer Implementierung



Zusammenfassung

- Ein **Objekt** besitzt einen **Zustand**, reagiert mit einem definierten **Verhalten** und hat eine **Identität**
- Objekte und ihre Verbindungen werden im **Objektdiagramm** dargestellt
- Eine **Klasse** beschreibt eine **Kollektion** von Objekten mit gleicher Struktur, gleichem Verhalten und gleichen Beziehungen
- Klassen werden im **Klassendiagramm** dargestellt
- Die **Attribute** beschreiben die **Daten**, die die Objekte einer Klasse besitzen
- Jedes Attribut hat einen bestimmten **Typ**
- Es wird zwischen (**Objekt-**) **Attributen** und **Klassenattributen** unterschieden
- Die **Operationen** beschreiben das **Verhalten** bzw. die **Schnittstelle** der Klasse
- Es wird zwischen (**Objekt-**) **Operationen**, **Klassenoperationen** und **Konstruktoroperationen** unterschieden



Glossar

Abgeleitetes Attribut (*derived attribute*): Abgeleitete Attribute lassen sich aus anderen Attributen berechnen. Sie dürfen nicht direkt geändert werden.

Attribut (*attribute*): Attribute beschreiben Daten, die von den → Objekten der → Klasse angenommen werden können. Alle Objekte einer Klasse besitzen dieselben Attribute, jedoch im Allgemeinen unterschiedliche Attributwerte. Jedes Attribut ist von einem bestimmten → Typ und kann einen Anfangswert (*default*) besitzen. Bei der Implementierung muss jedes Objekt Speicherplatz für alle seine Attribute reservieren. Der Attributname ist innerhalb der Klasse eindeutig.

Attributspezifikation (*attribute specification*): Ein → Attribut wird durch folgende Angaben spezifiziert:

Name: Typ = Anfangswert

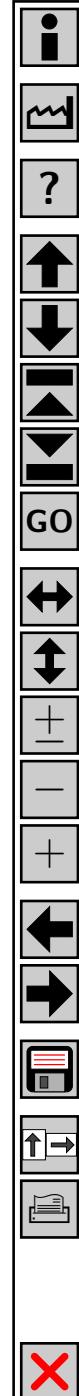
{ mandatory, unique, readOnly, Einheit: ..., Beschreibung: ... }

Dabei gilt: mandatory=Muss-Attribut, unique=Schlüsselattribut, readOnly=Attributwert nicht änderbar.

Elementare Klasse (*support class*): → Strukturtyp

Geheimnisprinzip (*information hiding*): Die Einhaltung des Geheimnisprinzips bedeutet, dass die Attribute und die Realisierung der Operationen außerhalb der Klasse nicht sichtbar sind.

Klasse (*class*): Eine Klasse definiert für eine Kollektion von → Objekten deren Struktur (Attribute), → Verhalten (Operationen) und Beziehungen (Assoziationen, Vererbungsstrukturen). Klassen besitzen – mit Ausnahme von abstrakten Klassen – einen Mechanismus, um neue Objekte zu erzeugen. Der Klassename muss mindestens im Paket, besser im gesamten System eindeutig sein.





Klassenattribut (*class scope attribute*): Ein Klassenattribut liegt vor, wenn nur ein Attributwert für alle → *Objekte* der → *Klasse* existiert. Klassenattribute sind von der Existenz der Objekte unabhängig.

Klassenoperation (*class scope operation*): Eine Klassenoperation ist eine Operation, die für eine → *Klasse* statt für ein → *Objekt* der Klasse ausgeführt wird.

Objekt (*object*): Ein Objekt besitzt einen → *Zustand* (Attributwerte und Verbindungen zu anderen Objekten), reagiert mit einem definierten → *Verhalten* (Operationen) auf seine Umgebung und besitzt eine → *Objektidentität*, die es von allen anderen Objekten unterscheidet. Jedes Objekt ist Exemplar einer → *Klasse*.

Objektdiagramm (*object diagram*): Das Objektdiagramm stellt → *Objekte* und ihre Verbindungen untereinander dar. Objektdiagramme werden im Allgemeinen verwendet, um einen Ausschnitt des Systems zu einem bestimmten Zeitpunkt zu modellieren. Objekte können einen – im jeweiligen Objektdiagramm – eindeutigen Namen besitzen oder es können anonyme Objekte sein. In verschiedenen Objektdiagrammen kann der gleiche Name unterschiedliche Objekte kennzeichnen.

Objektidentität (*object identity*): Jedes → *Objekt* besitzt eine Identität, die es von allen anderen Objekten unterscheidet. Selbst wenn zwei Objekte zufällig dieselben Attributwerte besitzen, haben sie eine unterschiedliche Identität. Im Speicher wird die Identität durch unterschiedliche Adressen realisiert.

Objektverwaltung (*class extension, object warehouse*): In der Systemanalyse besitzen Klassen implizit die Eigenschaft der Objektverwaltung. Das bedeutet, dass die Klasse weiß, welche → *Objekte* von ihr erzeugt wurden. Damit erhält die Klasse die Möglichkeit, Anfragen und Manipulationen auf der Menge der Objekte einer → *Klasse* durchzuführen.

Operation (operation): Eine Operation ist eine Funktion, die auf interne Daten (Attributwerte) eines → *Objekts* Zugriff hat. Auf alle Objekte einer → *Klasse* sind dieselben Operationen anwendbar. Für Operationen gibt es in der Analyse im Allgemeinen eine fachliche Beschreibung. Abstrakte Operationen besitzen nur einen Operationsrumpf. Externe Operationen werden vom späteren Bediener des Systems aktiviert. Interne Operationen werden dagegen immer von anderen Operationen aufgerufen.

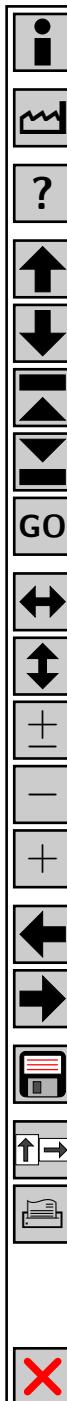
Strukturtyp: Wird der Typ eines → *Attributs* wieder durch eine → *Klasse* realisiert, dann spricht man von einem Strukturtyp oder einer elementaren Klasse. Sie wird nicht in das → *Klassendiagramm* eingetragen.

Typ (type): Jedes → *Attribut* ist von einem bestimmten Typ. Er kann ein Standardtyp (z. B. Int), ein Aufzählungstyp, eine → *Strukturtyp* oder eine Liste (list of Typ) sein.

Der Typbegriff wird auch im Sinne von Klassenspezifikationen verwendet. Er legt fest, auf welche Operationsaufrufe die → *Objekte* einer → *Klasse* reagieren können, d. h. der Typ definiert die Schnittstelle der Objekte. Ein Typ wird implementiert durch eine oder mehrere Klassen.

Verhalten (behavior): Unter dem Verhalten eines → *Objekts* sind die beobachtbaren Effekte aller → *Operationen* zu verstehen, die auf das Objekt angewendet werden können. Das Verhalten einer → *Klasse* wird bestimmt durch die Operationsaufrufe, auf die diese Klasse bzw. deren Objekte reagieren.

Zustand (state): Der Zustand eines → *Objekts* wird bestimmt durch seine Attributwerte und seine Verbindungen (*link*) zu anderen Objekten, die zu einem bestimmten Zeitpunkt existieren.

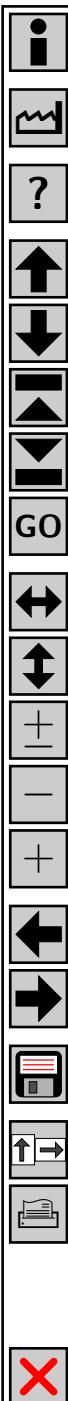


STATISCHE KONZEPTE DER OBJEKTORIENTIERTEN ANALYSE

Lernziele

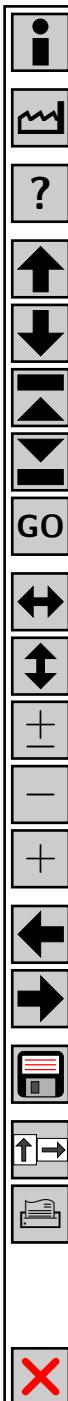
Verständnis

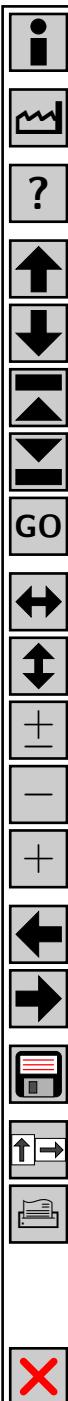
- Was ist eine Assoziation
- Was ist eine assoziative Klasse
- Was sind Aggregations- und Kompositionsbeziehungen
- Was ist das Vererbungskonzept



Anwendung

- Die UML-Notation für Assoziationen und Vererbung
- Identifikation von Assoziationen in einem Text und ihre graphische Darstellung
- Identifikation von Vererbungsstrukturen und ihre graphische Darstellung





Assoziationen

Begriffsdefinitionen

- **Definition 20: Assoziation (Association)**

Eine **Assoziation** modelliert **Beziehungen** (*Verbindungen*) zwischen **Objekten** einer oder mehrerer **Klassen**.

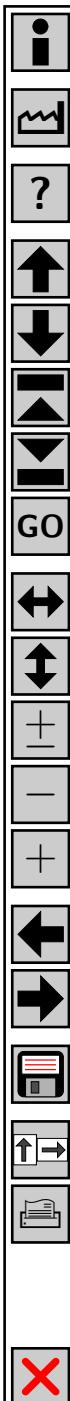
- **Definition 21: Reflexive Assoziation**

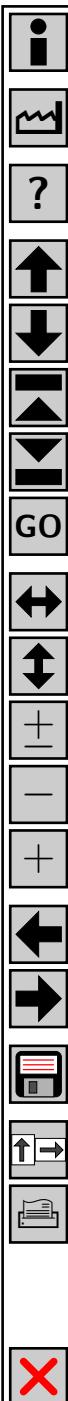
Eine **reflexive Assoziation** besteht zwischen **Objekten derselben Klasse**.



- Eine Assoziation modelliert stets Beziehungen zwischen Objekten, nicht zwischen Klassen
- Es ist aber üblich, von einer Assoziation zwischen Klassen zu sprechen, obwohl streng genommen die Objekte der Klasse gemeint sind

- **Beispiel 22:** Kunden und Konten einer Bank (vgl. [Abb. 33](#) und [Abb. 34](#))
 - Hans Meyer eröffnet am **4.7.1993** ein Geschäftskonto mit der Kontonummer **4711**
 - Er wird dadurch zum Kunden der Bank
 - Zwei Monate später eröffnet er bei derselben Bank noch ein privates Konto mit der Kontonummer **1234**
 - Damit existieren Verbindungen zwischen **Hans Meyer** und den Konten **4711** und **1234**





- Objekt- und Klassendiagramm für [Bsp. 22](#)
 - [Abb. 33](#) zeigt das Objektdiagramm
 - [Abb. 34](#) zeigt das Klassendiagramm

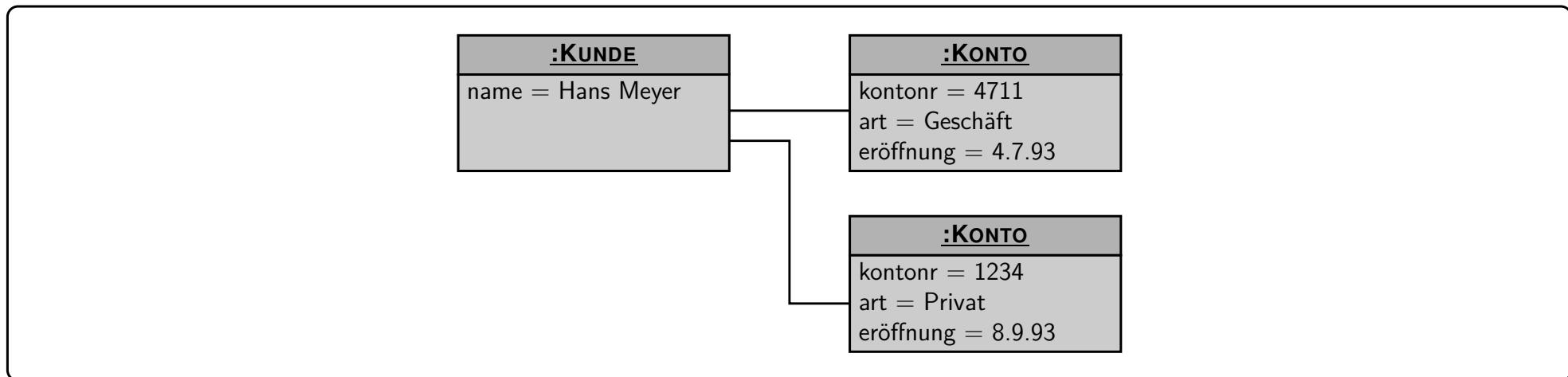


Abb. 33 Objektdiagramm für [Bsp. 22](#)

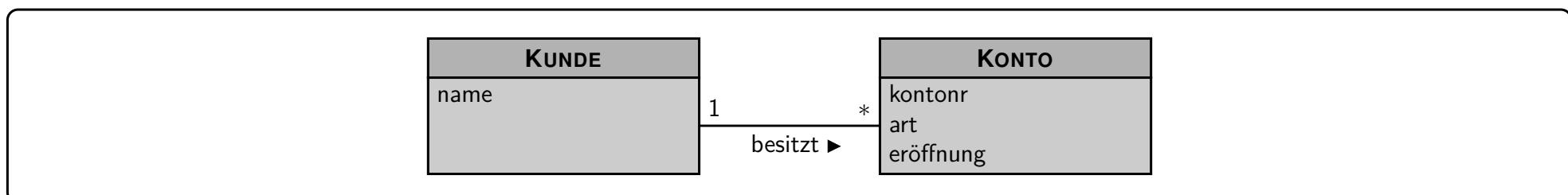
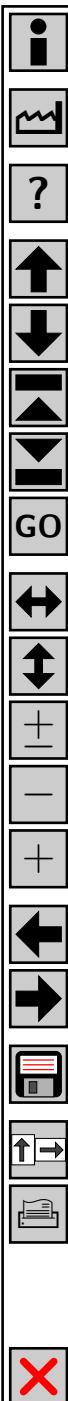


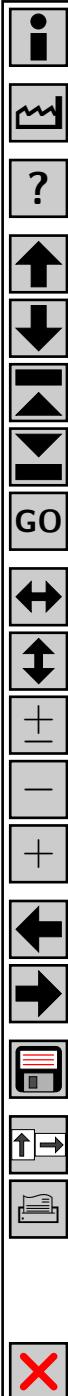
Abb. 34 Klassendiagramm für [Bsp. 22](#)



- Wichtige Eigenschaften für die Objekte der Klassen **KUNDE** und **KONTO**
 - Jeder Kunde kann mehrere Konten besitzen
 - Es kann Kunden geben, die kein Konto besitzen
 - Jedes Konto gehört zu genau einem Kunden
 - Durch die Verwendung dieses Assoziationsstyps ist ein Attribut **Kontoinhaber** in der Klasse **KONTO** nicht mehr erforderlich
 - Die **Menge** aller **Verbindungen** zwischen den Objekten von **KUNDE** und **KONTO** stellt die **Assoziation** zwischen beiden Klassen dar



- **Assoziationen sind in der Systemanalyse inhärent bidirektional**
- **Für Bsp. 22 gilt:**
 - Ein Kundenobjekt kennt alle seine Kontoobjekte
 - Ein Kontoobjekt kennt den zugehörigen Kunden



UML-Notation für binäre und reflexive Assoziationen

Notation

- Assoziationen/Beziehungen werden im Klassen-/Objektdiagramm als **Linien** zwischen den betroffenen Klassen/Objekten dargestellt
- Abb. 35 zeigt die Notation für binäre und reflexive Assoziationen

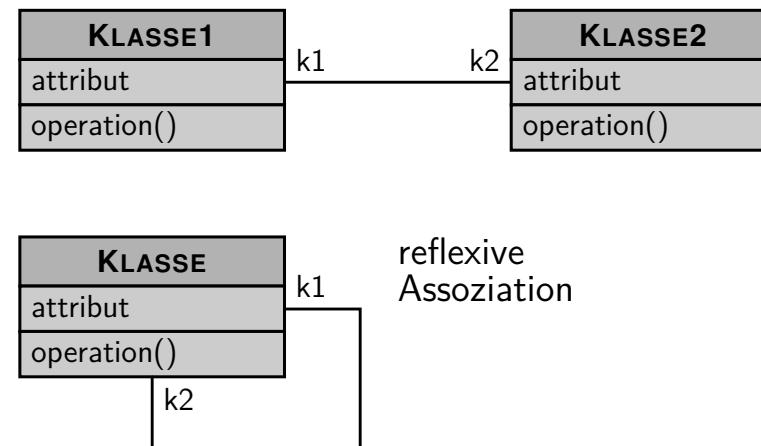
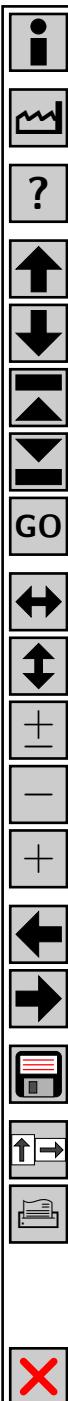


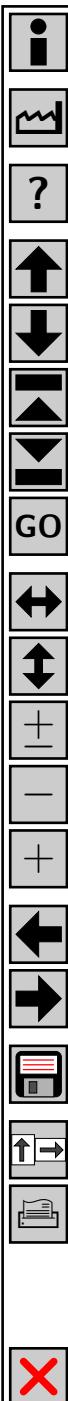
Abb. 35

Notation für Assoziationen

Assoziationsname

- Assoziationen können benannt werden
- Der Name beschreibt in der Regel ***nur*** eine ***Richtung*** der Assoziation, wobei eine Pfeilspitze die ***Leserichtung*** angibt (vgl. [Abb. 34](#))
- Der Assoziationsname ***kann fehlen***, wenn die Bedeutung der Assoziation ***offensichtlich*** ist





Kardinalitäten einer Assoziation

- **Definition 22: Kardinalität (Wertigkeit, Multiplicity, Multiplizität)**

Die **Kardinalität** spezifiziert die *Anzahl* der an der Assoziation *beteiligten Objekte*.



- Im Klassendiagramm müssen die Kardinalitäten angegeben werden
 - Für jede der beiden Richtungen einer Assoziation ist die Kardinalität zu spezifizieren
-
- Kardinalitäten in [Abb. 34](#)
 - Ein Kunde kann beliebig viele Konten besitzen, insbesondere ist auch erlaubt, dass er kein Konto hat
→ Verwendung von *
 - Zu jedem Konto muss es genau einen Kunden geben
→ Verwendung der 1

- Abb. 36 zeigt die Notation für Kardinalitäten

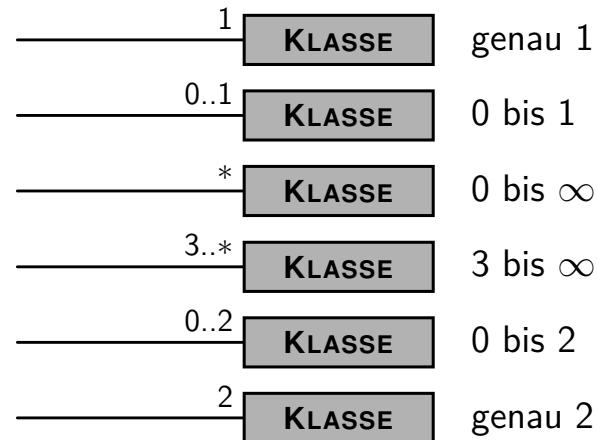
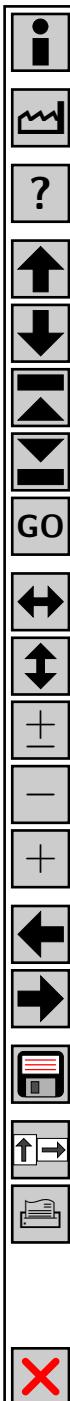


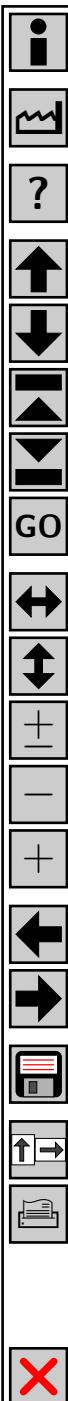
Abb. 36

Notation für Kardinalitäten



Ab UML 2.0 darf maximal nur noch ein Intervall spezifiziert werden





Kann- und Muss-Assoziationen

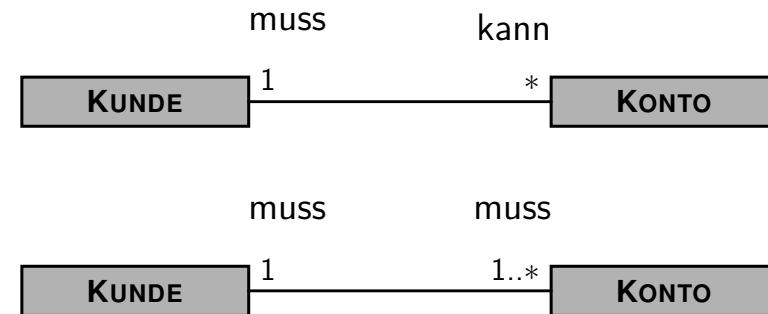
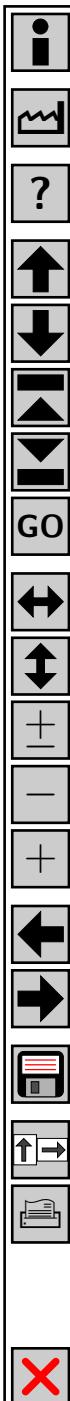
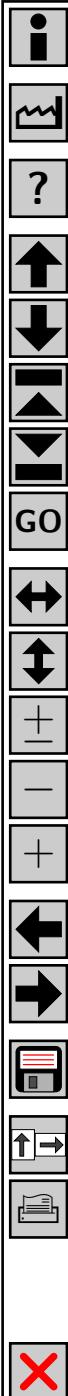


Abb. 37 Kann- und Muss-Assoziationen

- Kann-Assoziation
 - Sie besitzt als Untergrenze den Wert 0
- Die Kann-Assoziation in Abb. 34 bedeutet, dass es Bankkunden geben kann, die kein Konto besitzen
- Muss-Assoziation
 - Sie besitzt als Untergrenze einen Wert von ≥ 1



- Die Muss-Assoziation in [Abb. 34](#)
 - Ein Konto darf nicht auf mehrere Namen laufen, d. h. mehreren Bankkunden zugeordnet sein
 - Ein neues Konto darf nur für einen existierenden Kunden eingerichtet werden
 - Wird ein Kunde im System gelöscht, so müssen auch alle seine Konten gelöscht werden, sofern sie nicht einem anderen Kunden zugeordnet werden



Rollen

- **Definition 23: Rolle (Role, Rollename, Role Name)**
Die **Rolle** beschreibt, welche Bedeutung ein Objekt in einer Assoziation wahrnimmt.
- Eine binäre Assoziation kann maximal zwei Rollen besitzen
- Notation
 - Rollennamen können in Klassen- und Objektdiagrammen verwendet werden
 - Der Rollename wird jeweils an ein Ende der Assoziation geschrieben, und zwar bei der Klasse (bzw. Objekt), deren Bedeutung er näher beschreibt



Die geschickte Wahl der Rollennamen kann oft mehr zur Verständlichkeit eines Modells beitragen als ein Assoziationsname

- **Beispiel 23:** Rollen eines Bankkunden (vgl. Abb. 38)
 - Ein Kunde kann der **Kontoinhaber** sein
 - Ein Kunde kann als **Kontoberechtigter** auftreten

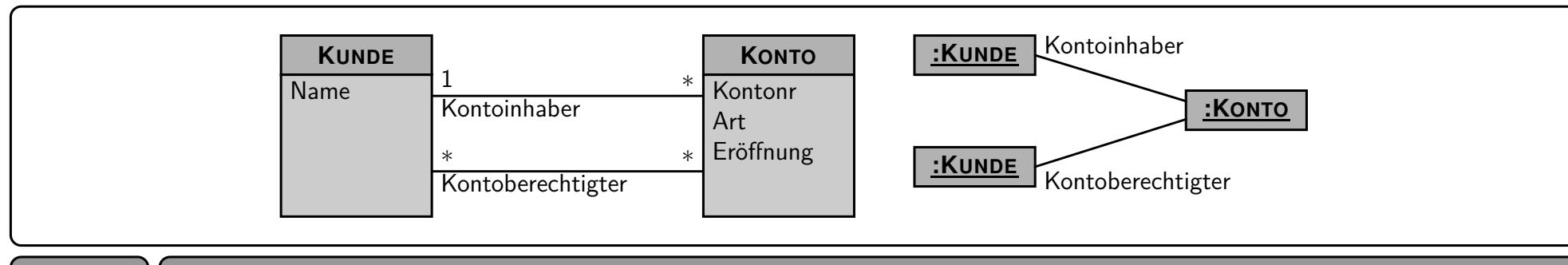
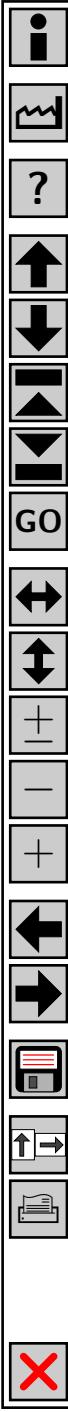


Abb. 38 Rollen von Bankkunden bezogen auf Bankkonten



- **Beispiel 24:** Rollen eines Angestellten in einer reflexiven Assoziation (vgl. Abb. 39)
 - Ein Angestellter kann **Chef** eines anderen Angestellten sein
 - Ein Angestellter kann **Mitarbeiter** eines anderen Angestellten sein

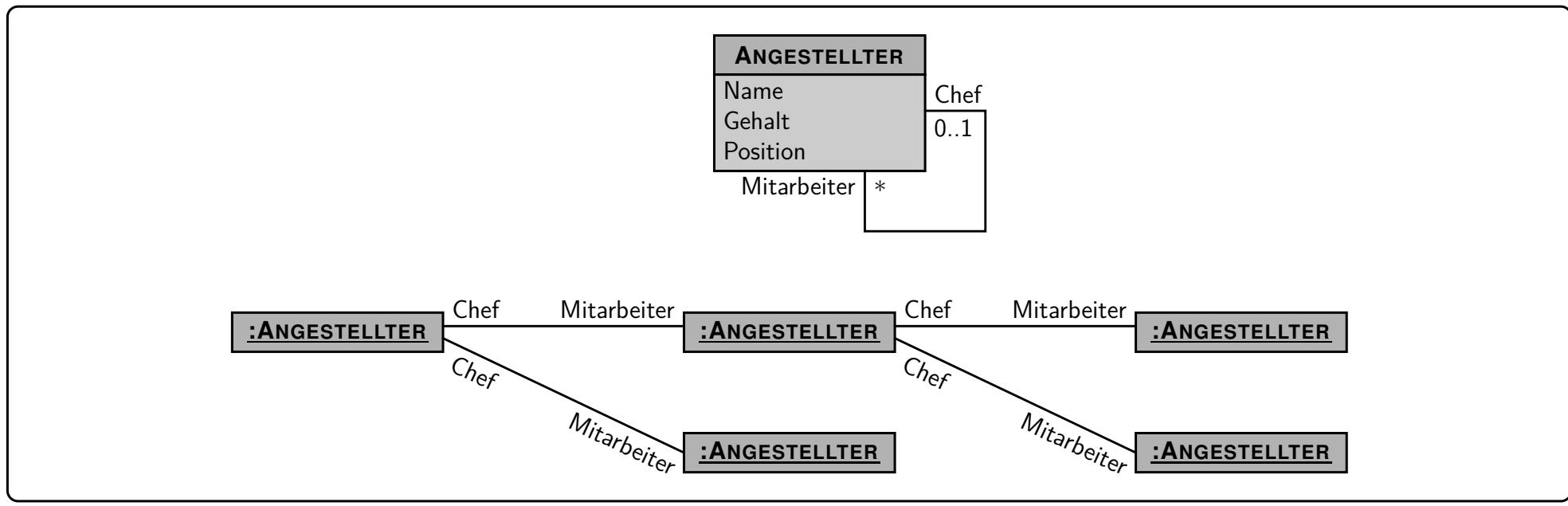


Abb. 39

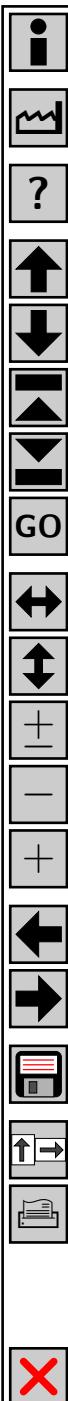
Rollen eines Angestellten in einer reflexiven Assoziation

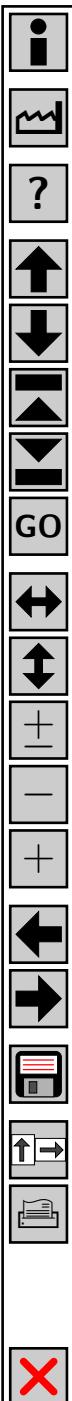


- **Rollen- oder Assoziationsnamen müssen angegeben werden, wenn zwischen zwei Klassen mehrere Assoziationen bestehen**
- **Auch bei einer reflexiven Assoziation müssen Rollennamen verwendet werden**

Assoziative Klassen

- **Definition 24: Assoziative Klasse (Association Class), Assoziationsklasse**
Eine **assoziative Klasse** besitzt sowohl die Eigenschaften der Assoziation als auch die der Klasse.
- Potentielle Eigenschaften
 - Attribute
 - Operationen
 - Assoziationen zu anderen Klassen
- Notation
 - Verwendung eines **Klassensymbols**, das über eine **gestrichelte Linie** mit der **Assoziation** verbunden wird





- **Beispiel 25:** Ausleihe von Büchern (vgl. Abb. 40)
 - Die Assoziation zwischen Lesern und Büchern definiert, welcher Leser welches Buch ausgeliehen hat
 - **AUSLEIHE** selbst besitzt weitere Attribute, z. B. das **Ausleihdatum**
 - Eine Operation könnte das **Verlängern** der Ausleihfrist darstellen
 - Das Attribut **Verlängerung** spiegelt dann den Sachverhalt wieder, ob eine Verlängerung bereits stattgefunden hat

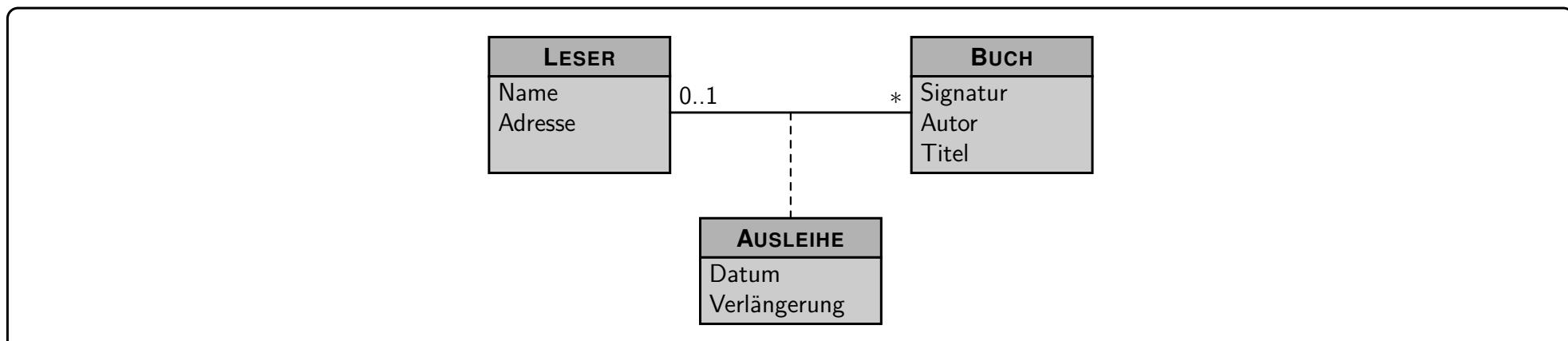


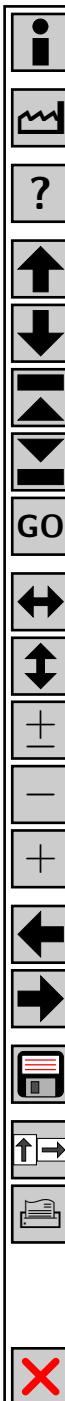
Abb. 40 Modellierung der Ausleihe als assoziative Klasse

Aggregation und Komposition

- UML kennt außer der **einfachen Assoziation** (auch: *Ordinary Association*) noch zwei weitere Arten
 - Aggregation
 - Komposition
- **Definition 25: Aggregation** (engl. *Aggregation*)
Eine **Aggregation** ist ein Sonderfall der Assoziation.
Sie liegt dann vor, wenn zwischen den Objekten der beteiligten Klassen eine Beziehung besteht, die sich als **ist Teil von** oder **besteht aus** beschreiben lässt.



Bei einer Aggregation ist es jedoch erlaubt, dass ein Objekt als Teil mehrerer übergeordneter Objekte Verwendung findet (vgl. Abb. 41, linkes Diagramm)



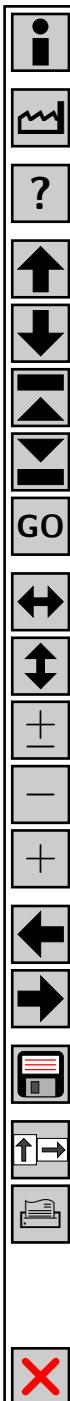
- **Definition 26: Komposition (Composition, Composite Aggregation)**

Eine **Komposition** ist eine *starke Form* der Aggregation mit folgenden Eigenschaften:

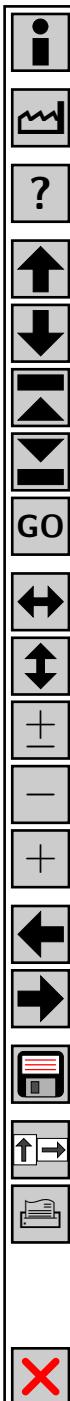
- Jedes Objekt der **Teilkasse** darf – zu einem Zeitpunkt – nur Bestandteil eines **einzigem** Objekts der Aggregatkasse sein
- Die **dynamische Semantik** (*Propagation Semantics*) für Objekte der Aggregatkasse gilt auch für die Objekte der **Teilkasse**
 - wird z. B. das Gesamtobjekt kopiert, so werden auch seine Teile kopiert
- Wird ein Objekt der Aggregatkasse **gelöscht**, so werden auch automatisch die Objekte der Teilkasse gelöscht
 - *they live and die with it*



- Aus der ersten Anforderung folgt, dass die Kardinalität bei der Aggregatkasse nicht größer als 1 sein kann
- Es ist erlaubt, dass ein Objekt der Teilkasse aus dem Objekt der Aggregatkasse entfernt und in ein anderes Objekt der Aggregatkasse integriert wird
 - Autoradio ausbauen und in ein anderes Auto einbauen
- Vor dem Löschen eines Aggregatobjekts darf ein Teilobjekt explizit entfernt werden
 - Autoradio vor dem Verschrotten des Autos ausbauen



- Notation
 - Aggregation: Weiße (bzw. transparente) Raute auf der Seite der Aggregatklasse
 - Komposition: Schwarze (bzw. gefüllte) Raute auf der Seite der Aggregatklasse
- **Beispiel 26:** Aggregation im Vergleich zur Komposition → vgl. [Abb. 41](#)
 - Linkes Diagramm: Aggregation
 - ◊ Ein Hypertext-Buch **besteht aus** mehreren Kapiteln
 - ◊ Jedes Kapitel kann in mehreren Hypertext-Büchern referenziert werden
 - Rechtes Diagramm: Komposition
 - ◊ Ein Dateiverzeichnis **enthält** mehrere Dateien
 - ◊ Jede Datei ist genau in einem Verzeichnis enthalten
 - ◊ Wird das Dateiverzeichnis kopiert, so werden auch alle enthaltenen Dateien kopiert



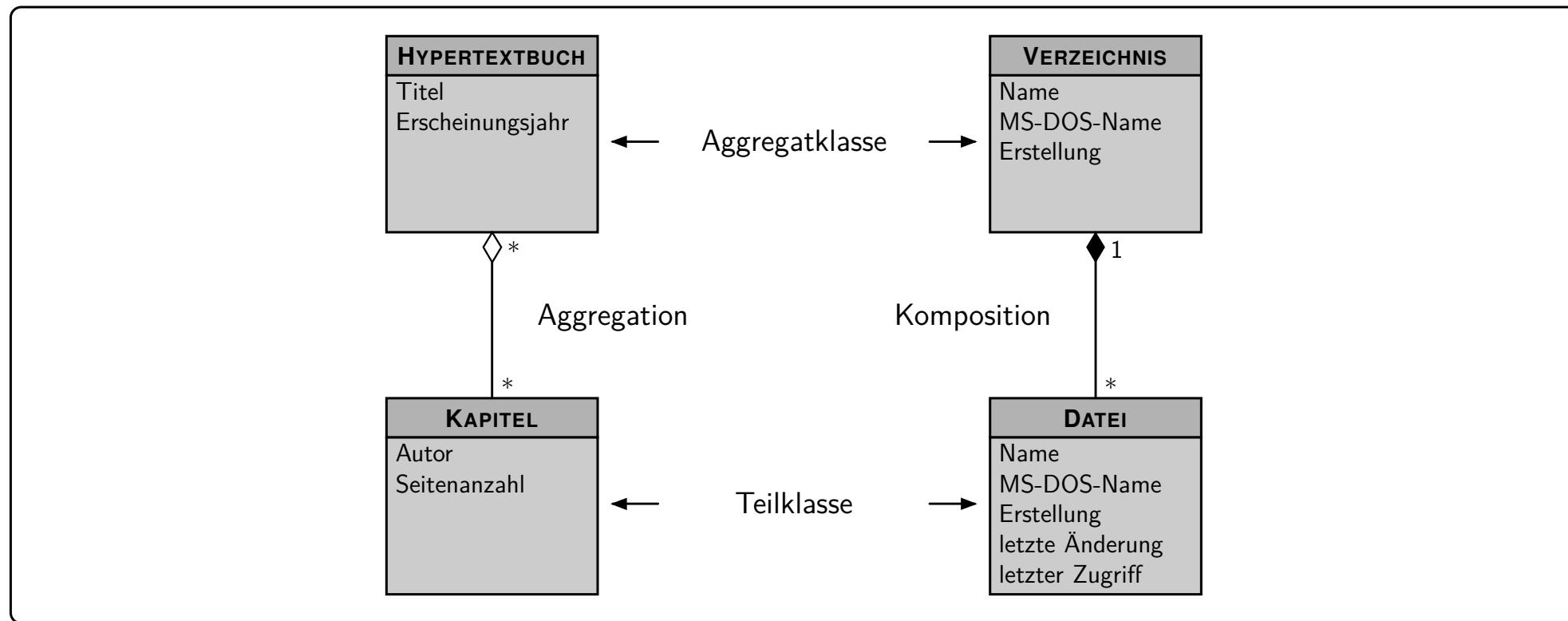
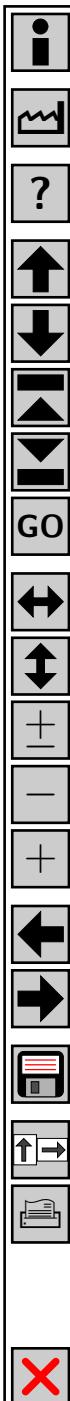
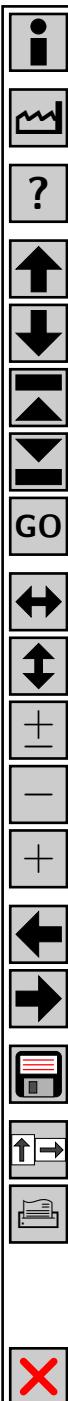


Abb. 41

Aggregation im Vergleich zur Komposition



- Die Abgrenzung zwischen *einfachen Assoziationen, Aggregationen und Kompositionen* ist nicht immer einfach
- Neben der hier vorgestellten Komposition als Sonderfall der Aggregation gibt es in der Literatur noch diverse weitere Spezialfälle von Aggregationsassoziationen



Assoziationen in Objektdiagrammen

- Die meisten Notationselemente der Assoziation können auch in Objektdiagrammen verwendet werden
 - Assoziationsname
 - Rollename
 - Qualifikationsangabe
 - Symbole für Aggregation bzw. Komposition
- Wird ein Assoziationsname verwendet, so muss er unterstrichen werden
- [Abb. 42](#) zeigt ein Objektdiagramm mit der Angabe von Rollen

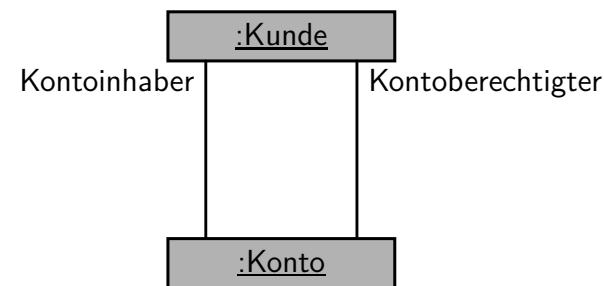
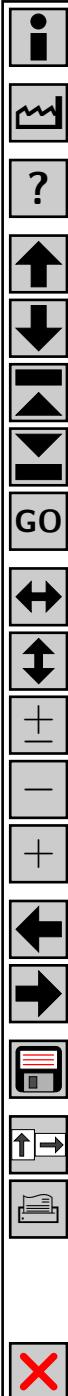


Abb. 42

Assoziationen in Objektdiagrammen



Das Vererbungskonzept

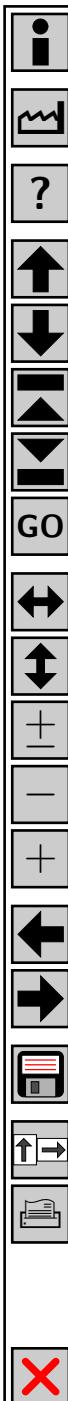
Definition einer Klassenhierarchie

Der Vererbungsbegriff

- **Definition 27: Vererbung (Inheritance)**

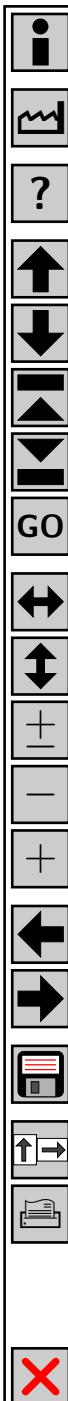
Die **Vererbung** beschreibt eine **Beziehung** zwischen einer **allgemeinen Klasse** (**Basisklasse**) und einer **spezialisierten Klasse**.

Die spezialisierte Klasse ist **vollständig konsistent** mit der Basisklasse, enthält aber (eventuell) **zusätzliche Informationen** (Attribute, Operationen, Assoziationen).



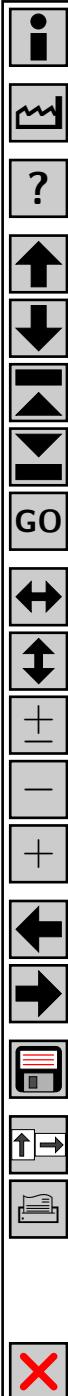
Bezeichnungsweisen

- Für den **Vererbungsprozess**
 - Eine Klasse wird **erweitert**
 - Von einer Klasse wird eine **UnterkLASSE** gebildet
 - Eine Klasse wird **spezialisiert**
 - Von einer Klasse wird eine UnterkLASSE **abgeleitet**
- Für die Klasse, die **erweitert** wird, d. h. die **allgemeinere** Klasse
 - Basisklasse
 - Oberklasse (Super Class)
 - Superklasse (Super Class)
- Für die **neu entstandene** Klasse, d. h. die **spezialisierte** Klasse
 - UnterkLASSE (Sub Class)
 - Abgeleitete Klasse
 - Erweiterte Klasse



Einfach- und Mehrfachvererbung

- Einfachvererbung
 - Eine Klasse besitzt (maximal) **eine direkte** Oberklasse
 - Die Vererbungsstruktur **muss** eine **Baumstruktur** sein
- Mehrfachvererbung
 - Eine Klasse **kann mehrere direkte** Oberklassen haben
 - Die Klassenstruktur **muss** ein **azyklischer Graph** sein



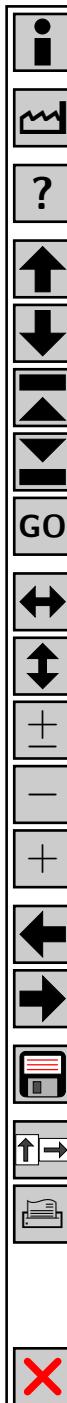
Wichtige Eigenschaften

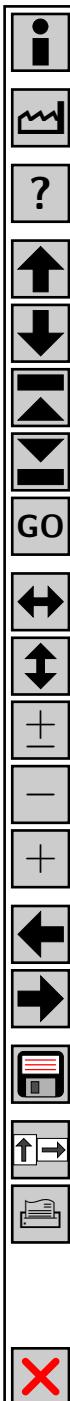
- Die Vererbungsbeziehung entspricht einer ***ist-ein***-Beziehung
- Ein Objekt der ***spezialisierten*** Klasse kann überall dort verwendet werden, wo ein Objekt der ***Basisklasse*** ***erlaubt*** ist
- Durch die Verwendung von Vererbungsbeziehungen entsteht eine ***Klassenhierarchie*** (*Vererbungsstruktur*)



- Das Konzept der Vererbung ist nicht nur gedacht, um gemeinsame Eigenschaften und Verhaltensweisen zusammenfassen
- Die Vererbung muss immer auch eine ***Generalisierung*** bzw. ***Spezialisierung*** darstellen
 - ***ist-ein***-Beziehung zwischen den Objekten der Unter- und der Oberklasse

- Eine abgeleitete Klasse besitzt zunächst einmal **automatisch alle Eigenschaften** der zugehörigen **Basisklasse** (bzw. Basisklassen bei Mehrfachvererbung)
 - Attribute
 - Operationen
 - Assoziationen
 - Mechanismen für eine **problemspezifische Anpassung** abgeleiteter Klassen
 - Definition **zusätzlicher Attribute**
 - Definition **zusätzlicher Assoziationen**
 - Definition **zusätzlicher Operationen**
 - **Redefinition** von Operationen
 - ◊ Die Operation hat denselben Namen (und dieselbe Signatur) jedoch eine **andere Funktionalität** als die entsprechende Operation der Oberklasse → sie ersetzt die Oberklassenoperationen
 - ◊ Die **Redefinition** einer Operation ist nicht mit dem **Überladen** von Operationen zu verwechseln
-  • **Zusätzlich ist es möglich, Attribute der Oberklasse zu Verdecken, indem in der Unterklasse Attribute gleichen Namens (wie in der Oberklasse) definiert werden**
• **Diese Möglichkeit trägt jedoch nicht zur problemspezifischen Anpassung bei**





UML-Notation

- Die Vererbungsbeziehung wird durch ein weißes (bzw. transparentes) Dreieck bei der Basisklasse gekennzeichnet → siehe Abb. 43

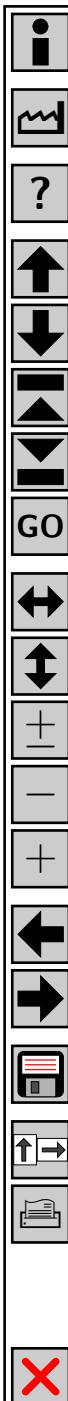


Abb. 43

Alternative Notationen für Vererbungsbeziehungen



- Die Basisklasse muss nicht notwendigerweise eine abstrakte Klasse sein
- Von einer abstrakten Klasse können keine Objekte erzeugt werden



Entwurf einer Klassenhierarchie

- **Beispiel 27:** Entwurf einer Klassenhierarchie (vgl. [Abb. 44](#))
 - Die Klassen **ANGESTELLTER**, **STUDENT** und (studentische) **HILFSKRAFT** sind zunächst unabhängig voneinander spezifiziert (oberer Teil der Abbildung)
 - Der untere Teil der Abbildung enthält eine bezüglich der Informationsmenge gleichwertige Klassenhierarchie
 - ◊ **PERSON** ist als **abstrakte Klasse** modelliert ⇒ Objekte der Klasse **PERSON** können nicht erzeugt werden
 - ◊ **ANGESTELLTER** und **STUDENT spezialisieren PERSON**
 - ◊ **HILFSKRAFT** spezialisiert **STUDENT**

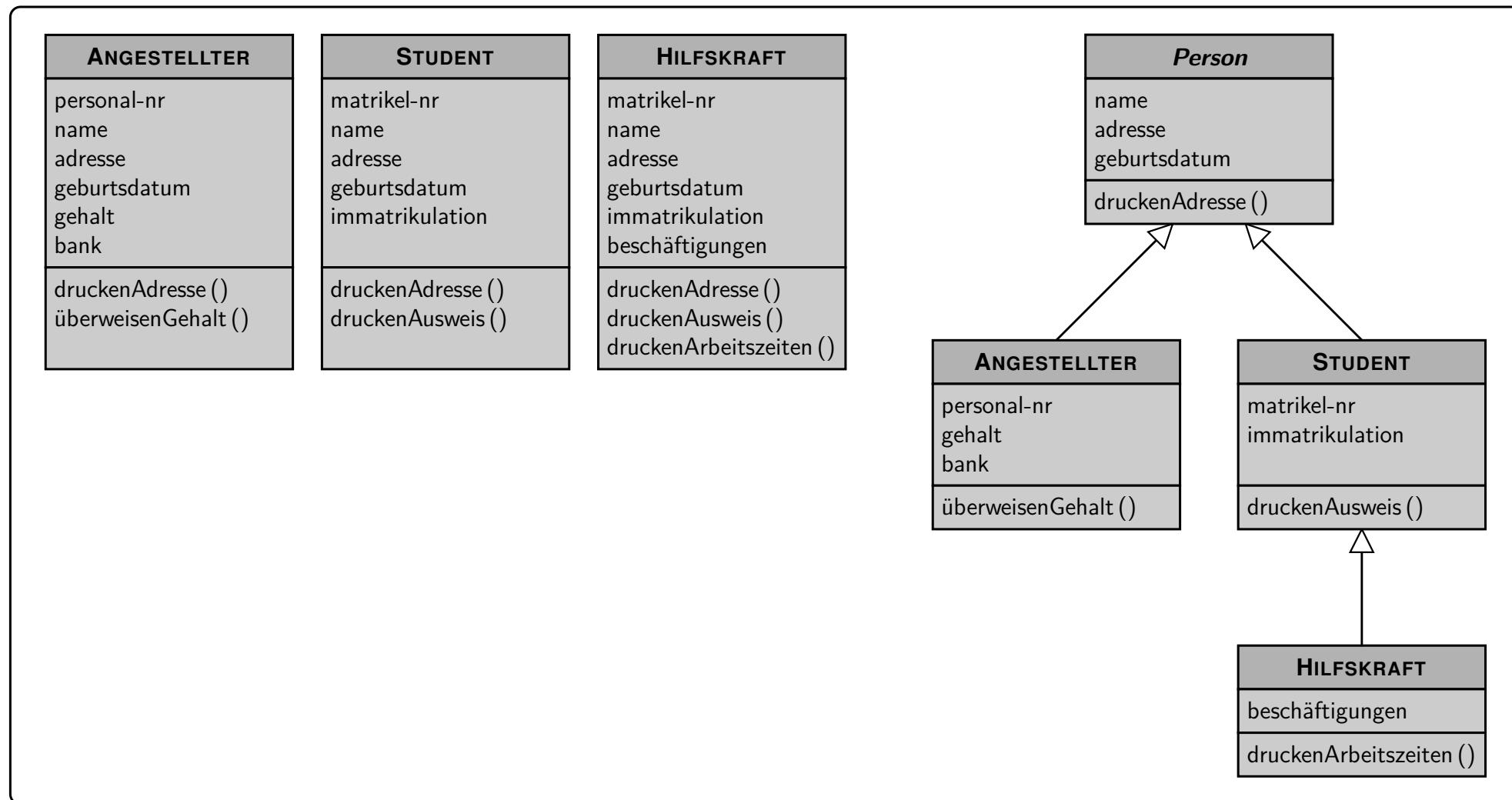
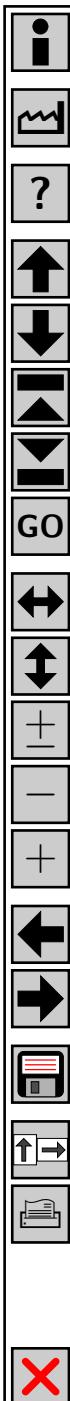
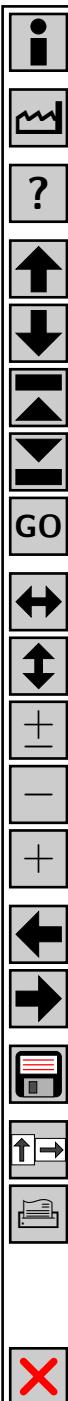


Abb. 44

Entwurf einer Klassenhierarchie



Der Mechanismus der Vererbung

Was wird vererbt?

- Abb. 45 stellt den prinzipiellen Mechanismus der Vererbung dar

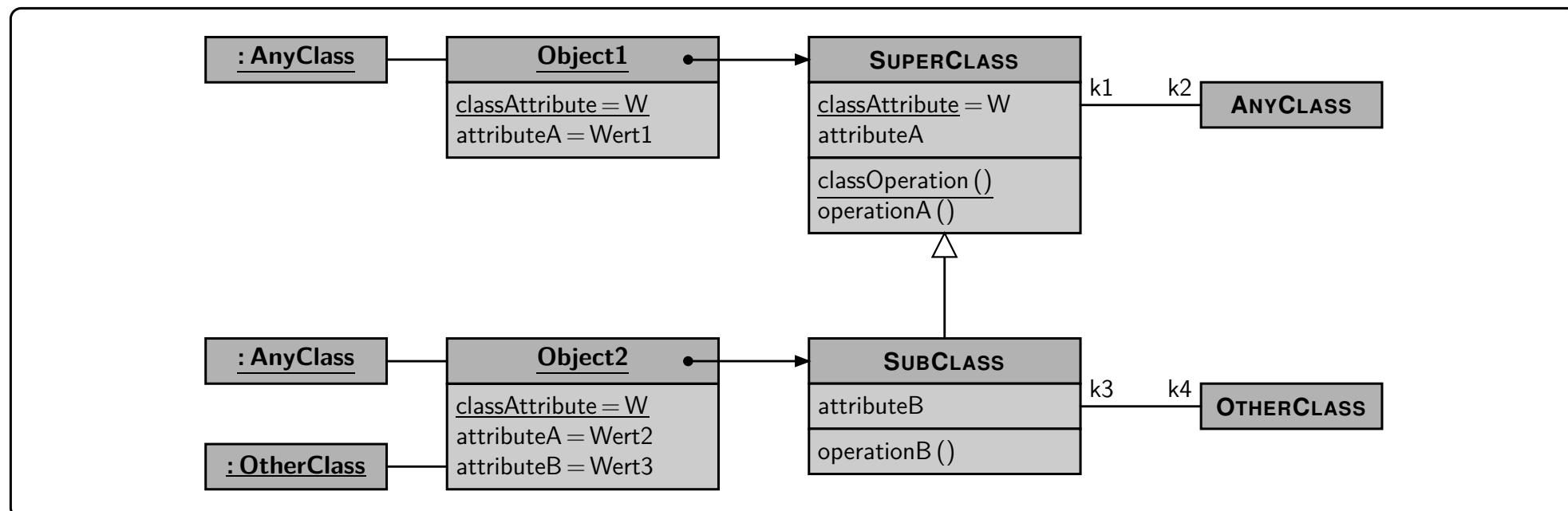
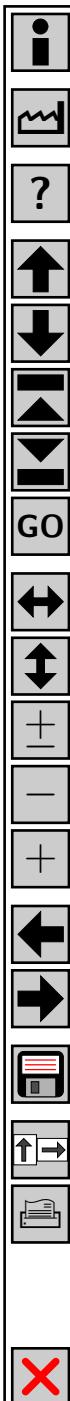
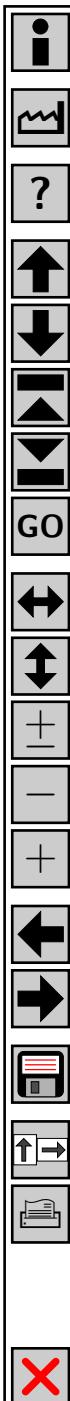


Abb. 45

Der prinzipielle Mechanismus der Vererbung

- Objekt- und Klassenattribute
 - Besitzt ein Objekt von **SUPERCLASS** ein Attribut **A**, dann besitzen auch die Objekte aller **direkten** oder **indirekten Unterklassen** (hier: **SUBCLASS**) dieses Attribut
 - Der **Wert** eines Objektattributs wird **nicht vererbt**
 - Der Wert eines **Klassenattributs** ist sowohl für Objekte von **SUPERCLASS** wie auch von **SUBCLASS** sichtbar (und natürlich identisch)
- Operationen
 - **Alle Operationen** (Objekt- und Klassenoperationen), die auf Objekte von **SUPERCLASS** angewendet werden können, können auf allen Objekten **direkter** und **indirekter Unterklassen** ausgeführt werden

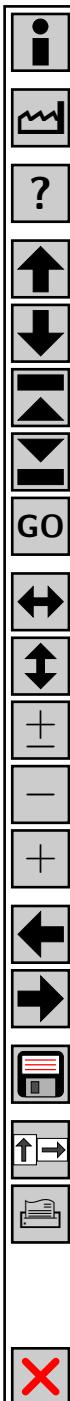




- Assoziationen
 - Existiert eine Assoziation zwischen **SUPERCLASS** und einer Klasse **ANYCLASS**, dann wird diese **Assoziation** an alle **direkten** und **indirekten Unterklassen vererbt**
 - Die an den Assoziationen **beteiligten Objekte** sind (in der Regel) aber **unterschiedlich** → **Object2** steht in Verbindung zu einem anderen Objekt aus **ANYCLASS** als **Object1**



- Die Vererbungsbeziehung zwischen **SUPERCLASS** und **SUBCLASS** spezifiziert, dass ein Objekt aus **SUBCLASS** die **Struktur** der Objekte aus **SUPERCLASS übernimmt und erweitert**
- Die Objekte der Klassen **SUPERCLASS** und **SUBCLASS** sind aber **unabhängig voneinander**, insbesondere was ihre Attributwerte betrifft



Redefinition einer Operation

- **Definition 28: Redefinition einer Operation**

Eine Unterklasse kann das Verhalten einer ihrer Oberklassen **redefinieren** (*überschreiben, redefine, override*).

Das wird erreicht, indem die Unterklasse eine **Operation gleichen Namens** (und gleicher Signatur) wie in der Oberklasse enthält, die eine **andere Funktionalität** besitzt.

- **Beispiel 28:** Redefinition (vgl. [Abb. 46](#))

- Auf ein Objekt der Klasse **KONTO** wird die Operation **buchen()** aus der Klasse **KONTO** angewendet
- Auf ein Objekt der Klasse **SPARKONTO** wird die Operation **buchen()** aus der Klasse **SPARKONTO** angewendet

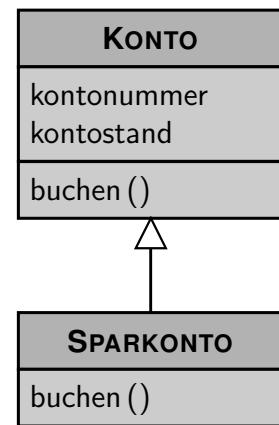
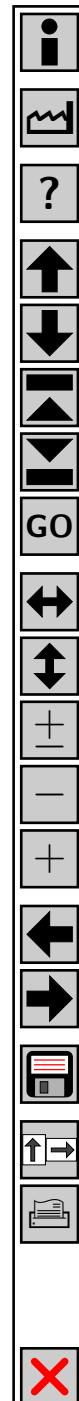
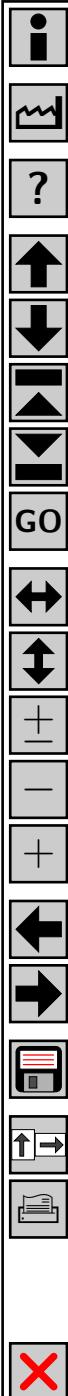


Abb. 46 Die Redefinition einer Operation



Eine redefinierte Operation wird innerhalb ihrer Beschreibung (bzw. Implementierung) üblicherweise die Operation der Oberklasse verwenden



Bewertung des Vererbungskonzepts

Vorteile

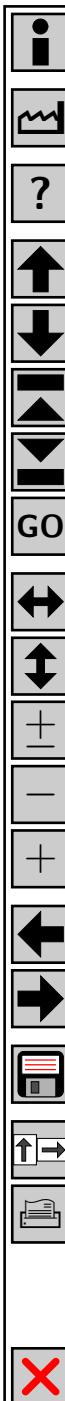
- Unterstützung der **Wiederverwendbarkeit**
 - Aufbauend auf existierenden Klassen können mit wenig Aufwand neue Klassen erstellt werden
- Unterstützung der **Änderbarkeit**
 - Z. B. steht ein in einer Oberklasse hinzugefügtes Attribut automatisch allen Unterklassen zur Verfügung

Nachteile

- Verletzung des **Geheimnisprinzips**
 - Eine abgeleitete Klasse (Unterklasse) sieht die Attribute ihrer Oberklassen
- Die Vorteile der **Lokalität**, die durch die Anwendung des Geheimnisprinzips erreicht wird, gehen verloren
 - Um eine Unterklasse zu verstehen, muss man (in der Regel) auch alle direkten und indirekten Oberklassen analysieren
 - Eine Neuimplementierung einer Oberklasse kann (eventuell) zu einer Neuimplementierung aller Unterklassen führen → hier ist die **leichte Änderbarkeit** dann ein Nachteil

Zusammenfassung

- Die **Assoziation** modelliert **Verbindungen** (*Beziehungen*) zwischen Objekten einer oder mehrerer Klassen
- Meistens werden **binäre** Assoziationen verwendet (Objekte aus zwei Klassen stehen in einer Beziehung)
- Sonderfälle der Assoziation sind die **Aggregation** und die **Komposition**
- Die **Vererbung** beschreibt eine Beziehung zwischen einer allgemeinen Klasse (Basisklasse) und einer spezialisierten Klasse (abgeleitete Klasse) → *ist-ein*-Beziehung
- Aufbauend auf den Basiskonzepten ermöglichen die **statischen Konzepte** die Erstellung des **statischen Modells** eines Systems
- Das statische Modell wird in der UML-Notation innerhalb des **Klassendiagramms** spezifiziert



Glossar

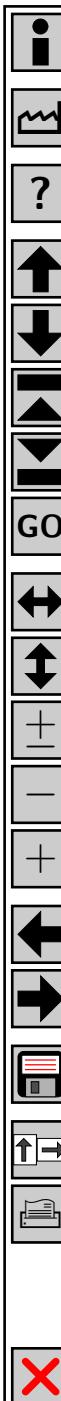
Abstrakte Klasse (*abstract class*): Von einer abstrakten Klasse können keine Objekte erzeugt werden. Die abstrakte Klasse spielt eine wichtige Rolle in Vererbungsstrukturen, wo sie die Gemeinsamkeiten einer Gruppe von → *Unterklassen* definiert. Damit eine abstrakte Klasse verwendet werden kann, muss von ihr zunächst eine Unterklasse abgeleitet werden.

Aggregation (*aggregation*): Eine Aggregation ist ein Sonderfall der → *Assoziation*. Sie liegt dann vor, wenn zwischen Objekten der beteiligten Klassen eine Beziehung besteht, die sich als *ist Teil von* oder *besteht aus* beschreiben lässt.

Assoziation (*association*): Eine Assoziation modelliert Verbindungen zwischen Objekten einer oder mehrerer Klassen. Binäre Assoziationen verbinden zwei Objekte. Eine Assoziation zwischen Objekten einer Klasse heißt reflexiv. Jede Assoziation wird beschrieben durch → *Kardinalitäten* und einen optionalen Assoziationsnamen oder Rollennamen. Sie kann um Restriktionen ergänzt werden. Besitzt eine Assoziation selbst wieder Attribute und gegebenenfalls Operationen und Assoziationen zu anderen Klassen, dann wird sie zu einer → *assoziativen Klasse*. Die Qualifikationsangabe (*qualifier*) zerlegt die Menge der Objekte am anderen Ende der Assoziation in Teilmengen. Eine abgeleitete Assoziation liegt vor, wenn die gleichen Abhängigkeiten bereits durch andere Assoziationen beschrieben werden. Sonderfälle der Assoziation sind die → *Aggregation* und die → *Komposition*. In der Analyse ist jede Assoziation inhärent bidirektional.

Assoziative Klasse (*association class*): Eine assoziative Klasse besitzt sowohl die Eigenschaften der → *Assoziation* als auch die der → *Klasse*.

Einfachvererbung: Bei der Einfachvererbung besitzt jede Unterklassie genau eine direkte Oberklasse. Es entsteht eine Baumstruktur.





Kardinalität (multiplicity): Die Kardinalität bezeichnet die Wertigkeit einer → *Assoziation*, d. h. sie spezifiziert die Anzahl der an der Assoziation beteiligten Objekte.

Klassendiagramm (class diagram): Das Klassendiagramm stellt die Klassen, die → *Vererbung* und die → *Assoziationen* zwischen Klassen dar. Zusätzlich können → *Pakete* modelliert werden.

Komposition (composition): Die Komposition ist eine besondere Form der → *Aggregation*. Beim Löschen des Ganzen müssen auch alle Teile gelöscht werden. Jedes Teil kann – zu einem Zeitpunkt – nur zu einem Ganzen gehören. Es kann jedoch anderen Ganzen zugeordnet werden. Die dynamische Semantik des Ganzen gilt auch für seine Teile.

Oberklasse (super class): In einer Vererbungsstruktur heißt jede Klasse, von der eine Klasse Eigenschaften und Verhalten erbt, Oberklasse dieser Klasse. Mit anderen Worten: Eine Oberklasse ist eine Klasse, die mindestens eine Unterklasse besitzt.

Rolle (role name): Die Rolle beschreibt, welche Bedeutung ein Objekt in einer → *Assoziation* wahrnimmt. Eine binäre Assoziation besitzt maximal zwei Rollen.

Unterklasse (sub class): Jede Klasse, die in einer Vererbungshierarchie Eigenschaften und Verhalten von anderen Klassen erbt, ist eine Unterkelasse dieser Klasse. Mit anderen Worten: Eine Unterkelasse besitzt immer (mindestens) eine direkte Oberklasse.

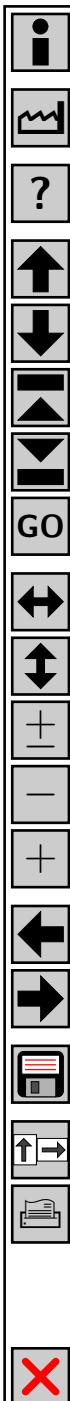
Vererbung (inheritance): Die Vererbung beschreibt die Beziehung zwischen einer allgemeineren Klasse (Basisklasse) und einer spezialisierten Klasse. Die spezialisierte Klasse erweitert die Liste der Attribute, Operationen und → *Assoziationen* der Basisklasse. Operationen der Basisklasse dürfen redefiniert werden. Es entsteht eine Klassenhierarchie.

DYNAMISCHE KONZEpte DER OBJEKTORIENTIERTEN ANALYSE

Lernziele

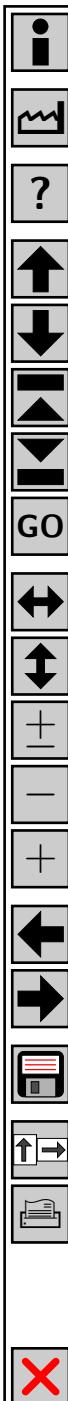
Verständnis

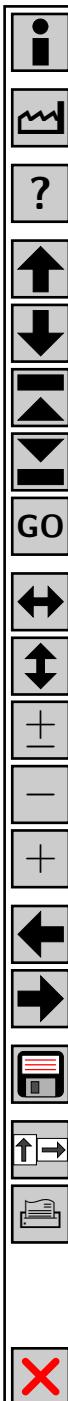
- Was ist ein Geschäftsprozess
- Was ist eine Botschaft
- Was ist ein Szenario
- Was sind Sequenz- und Kommunikationsdiagramme
- Was ist ein Zustandsautomat und welche Rolle spielt er im dynamischen Modell
- Was ist ein Aktivitätsdiagramm



Anwendung

- Die Identifikation von Geschäftsprozessen
- Die Spezifikation von Geschäftsprozessen
- Erstellen von Sequenz- und Kommunikationsdiagrammen
- Erstellen von Zustandsdiagrammen



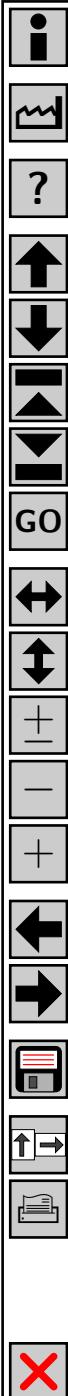


Geschäftsprozess

Der Begriff des Geschäftsprozesses

Einführung des Begriffes *Use Case*

- IVAR JACOBSON hat den Begriff des ***Use Case*** im Zusammenhang mit einer ***objektorientierten Methode*** 1987 vorgestellt
- Obwohl das Use Case Konzept prinzipiell völlig unabhängig von der ***objektorientierten Modellierung*** ist, verwenden es nahezu alle objektorientierten Methoden
- Der Use Case Begriff kann auf zwei unterschiedlichen Abstraktionsebenen betrachtet werden
 - Use Case in einem ***Informationssystem***
 - Use Case in einem ***Unternehmen***

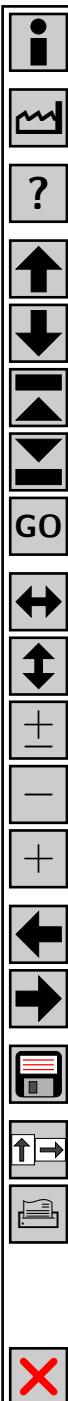


Use Case in einem Informationssystem

- Ein Use Case ist eine Sequenz von **zusammengehörenden Transaktionen**
- Diese Transaktionen werden von einem **Akteur** im Dialog mit einem Informationssystem ausgeführt
- Eine Transaktion ist eine **Menge von Verarbeitungsschritten**, von denen entweder **alle** oder **keiner** durchgeführt wird



- Ein Use Case beschreibt hier eine *spezielle Benutzung* des Informationssystems
- Alle Use Cases zusammen dokumentieren alle Möglichkeiten der Systemnutzung
→ *Use Case Model*
- Statt eines Informationssystems kann natürlich auch ein allgemeines Software-System betrachtet werden



Use Case in einem Unternehmen

- Ein Use Case in einem Unternehmen (*Business System*) besteht aus der Ausführung **unternehmensinterner Aktivitäten**, um die Wünsche eines **Kunden** zu erfüllen
- Die **Aktivitäten** lassen sich als verallgemeinerte Transaktionen auffassen
 - Organisatorische Schritte, z. B. das Treffen einer Entscheidung
 - Transaktion innerhalb eines Software-Systems



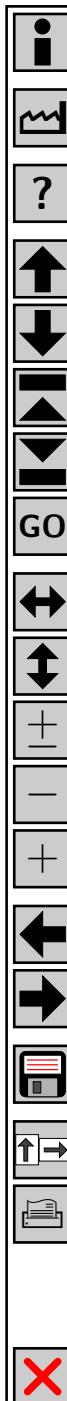
In diesem Kontext wird ein Use Case mit einem **Geschäftsprozess** (*Business Process*) gleichgesetzt

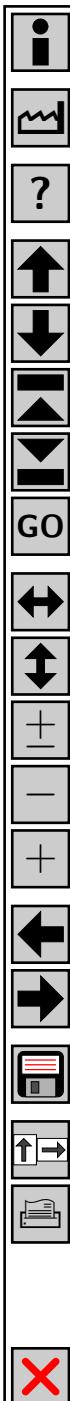
Geschäftsprozesse in der Systemanalyse

- Die **Identifikation** der Use Cases ist der erste Schritt innerhalb der **Analysephase**
- Zielsetzung: Ermittlung, welche Aufgaben mit dem neuen Software-System zu bewältigen sind, um die gewünschten Ergebnisse zu erzielen



- Zu diesem frühen Zeitpunkt ist noch nicht abzusehen, ob ein Use Case ausschließlich durch Software realisierbar ist, oder auch organisatorische Schritte enthält
 - Daher wird in der Folge der Begriff **Geschäftsprozess** verwendet
-
- **Definition 29: Geschäftsprozess (Use Case)**
Ein **Geschäftsprozess** besteht aus mehreren zusammenhängenden Aufgaben, die von einem oder mehreren **Akteuren** durchgeführt werden, um ein Ziel zu erreichen bzw. ein gewünschtes Ergebnis zu erstellen.
 - **Definition 30: Akteur (Actor)**
Ein **Akteur** ist eine **Rolle**, die ein Benutzer eines Systems spielt.
Jeder Akteur hat einen gewissen Einfluss auf das System.





- Wer kann die Rolle eines Akteurs spielen?

- Person
- Organisationseinheit
- Externes System



Akteure befinden sich stets *außerhalb* des Systems

- **Beispiel 29:** Identifikation von Akteuren (vgl. [Abb. 47](#))

- Das betrachtete System ist ein **Handelshaus**
 - ◊ Kunde und Lieferant sind **Akteure**
 - ◊ Die Buchhaltung ist kein Akteur, da sie sich **innerhalb** des Systems befindet
- Das betrachtete System ist ein **Software-System** zur Unterstützung des Auftrags- und Bestellwesens
 - ◊ Hier ist die **Buchhaltung** ein Akteur, da sie von außerhalb mit dem Software-System kommunizieren muss

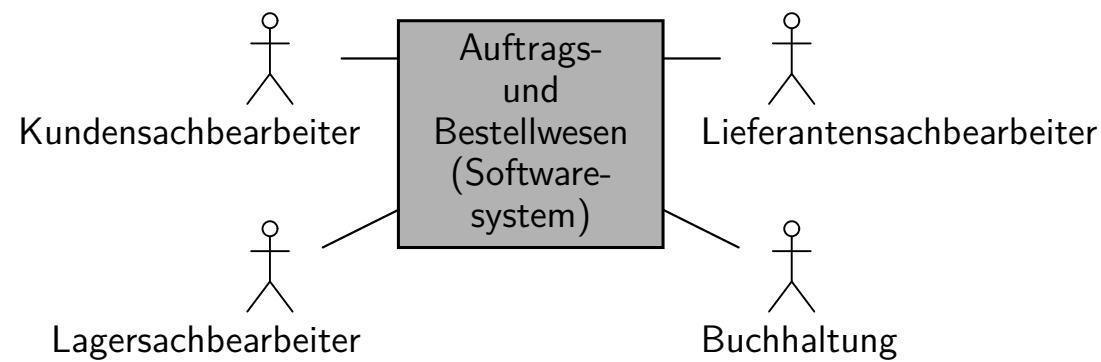
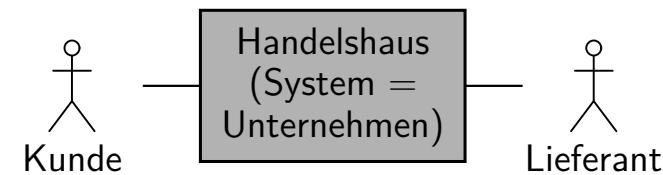
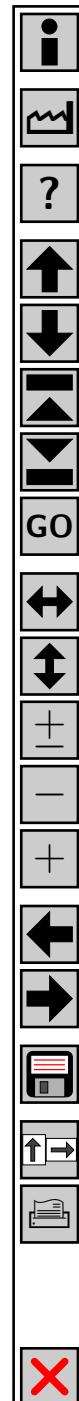
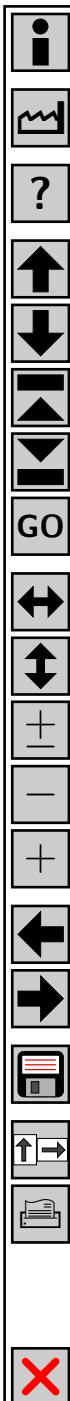


Abb. 47

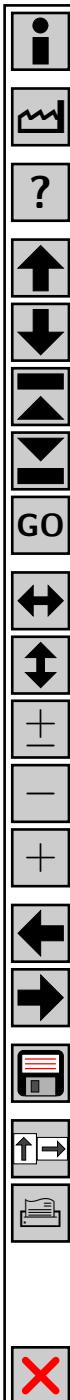
Wer ist der Akteur?



- Abgrenzung zwischen den tatsächlichen Systembenutzern und Akteuren
 - In einer kleinen Firma können die **Aufgaben** des Kunden- und Lieferantensachbearbeiters durchaus von **derselben Person** wahrgenommen werden
 - Trotzdem werden **zwei Akteure** identifiziert
 - Die Akteure **charakterisieren** die **Rollen**, die diese Person bezogen auf das Software-System spielt



Bei der Modellierung eines Software-Systems sind die Akteure diejenigen, die das System später **bedienen** bzw. **Ergebnisse des Systems erhalten**



Alternative Bezeichnungen für Geschäftsprozesse

- Anwendungsfall
- Geschäftsfall
- Geschäftsvorfall
- Workflow
- Business Process
- Use Case

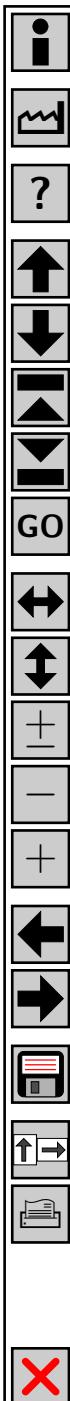
Spezifikation von Geschäftsprozessen

Überblick

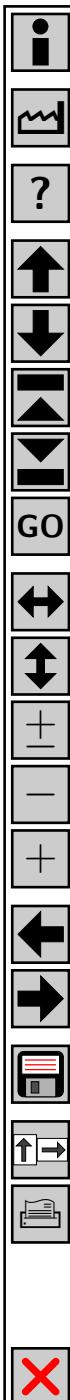
- Grundprinzip der Geschäftsprozessmodellierung
 - **Trennung** von **Funktionalität** und **Benutzungsoberfläche**
 - Begründung
 - ◊ Benutzungsoberflächen ändern sich aufgrund neuer Darstellungstechniken relativ schnell
 - ◊ Portierbarkeit der Software auf verschiedene Plattformen:
Auch hier liegen die wesentlichen Unterschiede oft in der plattformspezifischen Benutzungsoberflächentechnologie → Betriebssystemabhängigkeit

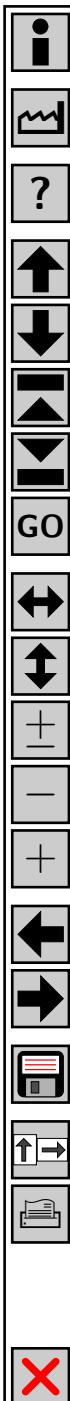


Ergebnis: Die Beschreibung der Funktionalität eines Geschäftsprozesses erfolgt ohne Bezüge zur Benutzungsoberfläche



- Spezifikationstypen
 - **Informale** Spezifikation
 - **Semiformale** Spezifikation
- **Informale** Spezifikation
 - Verwendung einer **umgangssprachlichen** Beschreibung
 - Geeignet für einfache Geschäftsprozesse
- **Semiformale** Spezifikation
 - Verwendung einer **Geschäftsprozessschablone** (*Use Case Template*)
 - Anwendung bei komplexeren Geschäftsprozessen





Informale Spezifikation

- **Beispiel 30:** Auftragsbearbeitung in einem Versandhaus
 - Auftragsbearbeitung hängt von vielen Faktoren ab
 - ◊ Handelt es sich um einen Neukunden?
 - ◊ Sind alle gewünschten Artikel lieferbar?
- Umgangssprachliche Formulierung des Geschäftsprozesses **Auftrag ausführen**:

Eine **Kundenbestellung** kommt in der Versandabteilung an.
Neukunden werden im System registriert und der **Versand** an diese Kunden erfolgt ausschließlich per Nachnahme oder Bankeinzug.
Für alle lieferbaren **Artikel** wird die **Rechnung** erstellt und als **Auftrag** an das **Lager** weitergegeben.
Sind einige der gewünschten Artikel **nicht lieferbar**, so wird der Kunde informiert.
Alle erstellten Rechnungen werden an die **Buchhaltung** weitergegeben.
- Am Geschäftsprozess **Auftrag ausführen** beteiligte **Akteure**
 - Kundensachbearbeiter
 - Lagersachbearbeiter
 - Buchhaltung

Geschäftsprozessschablone

- Verwendung bei einer **umfangreicherem** Spezifikation



- Die Schablone wird als **Checkliste** aufgefasst
- Sie ist daher nicht für jeden Geschäftsprozess **vollständig auszufüllen**

- Struktur der Geschäftsprozessschablone

Geschäftsprozess

Name, bestehend aus zwei oder drei Wörtern → Was wird getan ?

Ziel

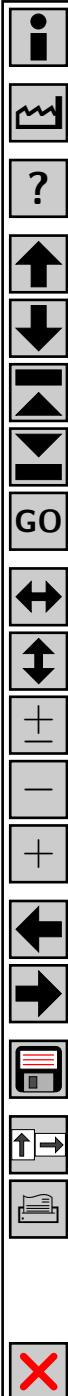
Globale Zielsetzung bei erfolgreicher Ausführung des Geschäftsprozesses

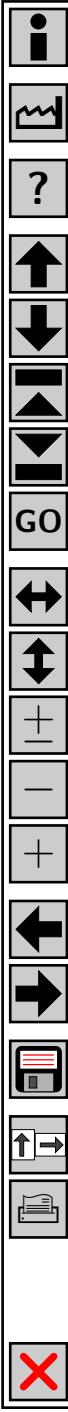
Kategorie

Primär, sekundär oder optional

Vorbedingung

Erwarteter Zustand, bevor der Geschäftsprozess beginnt





Nachbedingung: Erfolg

Erwarteter Zustand nach erfolgreicher Ausführung des Geschäftsprozesses
→ Ergebnis des Geschäftsprozesses

Nachbedingung: Fehlschlag

Erwarteter Zustand, wenn das Ziel nicht erreicht werden kann

Akteure

Rollen von Personen oder andere Systeme, die den Geschäftsprozess auslösen oder daran beteiligt sind

Auslösendes Ereignis

Wenn dieses Ereignis eintritt, dann wird der Geschäftsprozess gestartet

Beschreibung

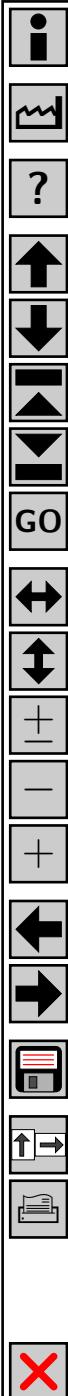
- 1** Erste Aktion
- 2** Zweite Aktion

Erweiterungen

- 1a** Erweiterung des Funktionsumfangs der ersten Aktion

Alternativen

- 1a** Alternative Ausführung der ersten Aktion
- 1b** Weitere Alternative zur ersten Aktion

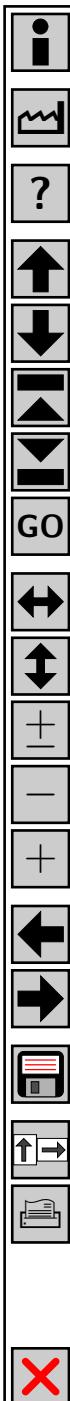


- Kategorie eines Geschäftsprozesses
 - Primär: Ein notwendiges Verhalten wird beschrieben, das **häufig** benötigt wird
 - Sekundär: Ein notwendiges Verhalten wird beschrieben, das **selten** benötigt wird
 - Optional: Ein Verhalten wird beschrieben, das für den Einsatz des Systems zwar nützlich, aber nicht unbedingt notwendig ist
- Vorbedingung
 - Der betrachtete Geschäftsprozess kann nur ausgeführt werden, wenn die genannte Vorbedingung erfüllt ist
- Nachbedingung
 - Die Nachbedingung eines Geschäftsprozesses A kann eine Vorbedingung für den Geschäftsprozess B sein



Vor- und Nachbedingungen legen fest, in welcher Reihenfolge Geschäftsprozesse ausgeführt werden können

- Beschreibung
 - Umgangssprachliche Spezifikation des Geschäftsprozesses
 - Die hier aufgeführten Aktionen beschreiben den Standardfall
- Erweiterungen
 - Hier werden seltener auszuführende Aktionen beschrieben, die **zusätzlich** zu den Aktionen im Standardfall auszuführen sind
- Alternativen
 - Hier werden seltener auszuführende Aktionen beschrieben, die die Aktionen aus dem Standardfall **ersetzen**



- **Beispiel 31:** Geschäftsprozessschablone für **Auftrag ausführen** aus [Bsp. 30](#)

Geschäftsprozess

Auftrag ausführen

Ziel

Ware an Kunden geliefert

Vorbedingung

Keine

Nachbedingung: Erfolg

Ware ausgeliefert (auch Teillieferungen), Rechnungskopie bei Buchhaltung

Nachbedingung: Fehlschlag

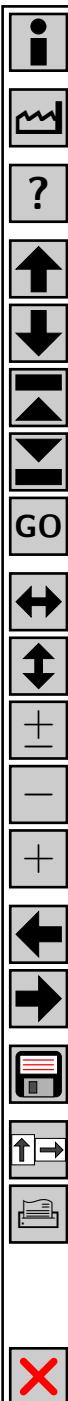
Mitteilung an Kunden, dass nichts lieferbar ist

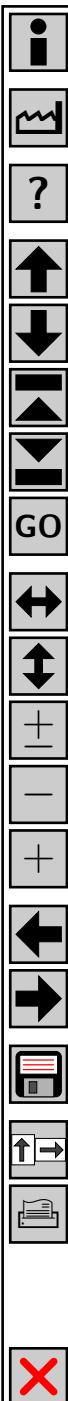
Akteure

Kundensachbearbeiter, Lagersachbearbeiter, Buchhaltung

Auslösendes Ereignis

Bestellung des Kunden liegt vor





Beschreibung

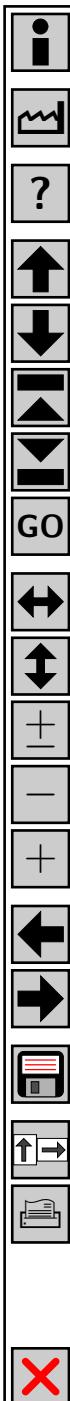
- 1 Kundendaten abrufen
- 2 Lieferbarkeit prüfen
- 3 Rechnung erstellen
- 4 Auftrag vom Lager ausführen lassen
- 5 Rechnungskopie an Buchhaltung geben

Erweiterungen

- 1a Kundendaten aktualisieren

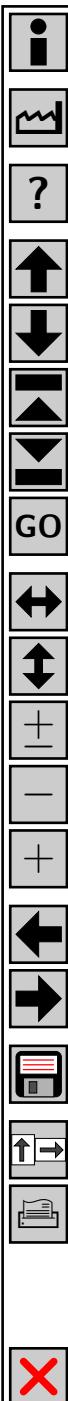
Alternativen

- 1a Neukunden erfassen
- 3a Rechnung mit Nachnahme erstellen
- 3b Rechnung mit Bankeinzug erstellen



Nebenläufigkeit in Geschäftsprozessen

- Die Geschäftsprozessschablone ist ein **einfaches**, aber **effektives** Hilfsmittel zur Beschreibung von Geschäftsprozessen
- Nachteilig ist, dass die sequentielle Reihenfolge der Einzelschritte festgelegt werden muss
- Alternativ können zur Beschreibung der Arbeitsschritte eines Geschäftsprozesses **Aktivitätsdiagramme** verwendet werden, mit denen sich **nebenläufig** ausführbare Arbeitsschritte spezifizieren lassen



Das Geschäftsprozessdiagramm

UML-Notation

- Abb. 48 zeigt die Struktur eines **Geschäftsprozessdiagramms** (*Use Case Diagram*)

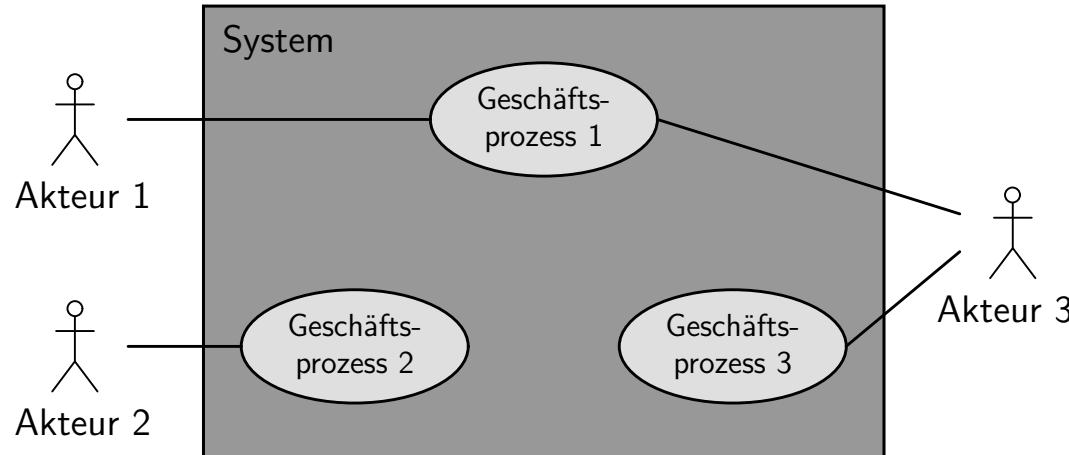
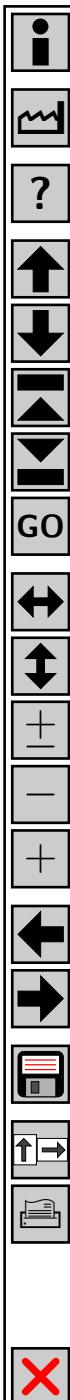


Abb. 48

Notation für Geschäftsprozessdiagramme

- Einsatzgebiete
 - Systembeschreibung auf einem sehr hohen Abstraktionsniveau
 - Definition der Schnittstellen zur Außenwelt
- Bemerkungen zur Notation
 - Akteure werden immer als **Strichmännchen** gezeichnet, auch wenn es sich um ein **externes System** handelt
 - Eine **Linie** zwischen einem Akteur und einem Geschäftsprozess bedeutet, dass eine **Kommunikation** stattfindet



- **Beispiel 32:** Geschäftsprozessdiagramm für einen Ausschnitt eines Warenwirtschaftssystems (vgl. Abb. 49)

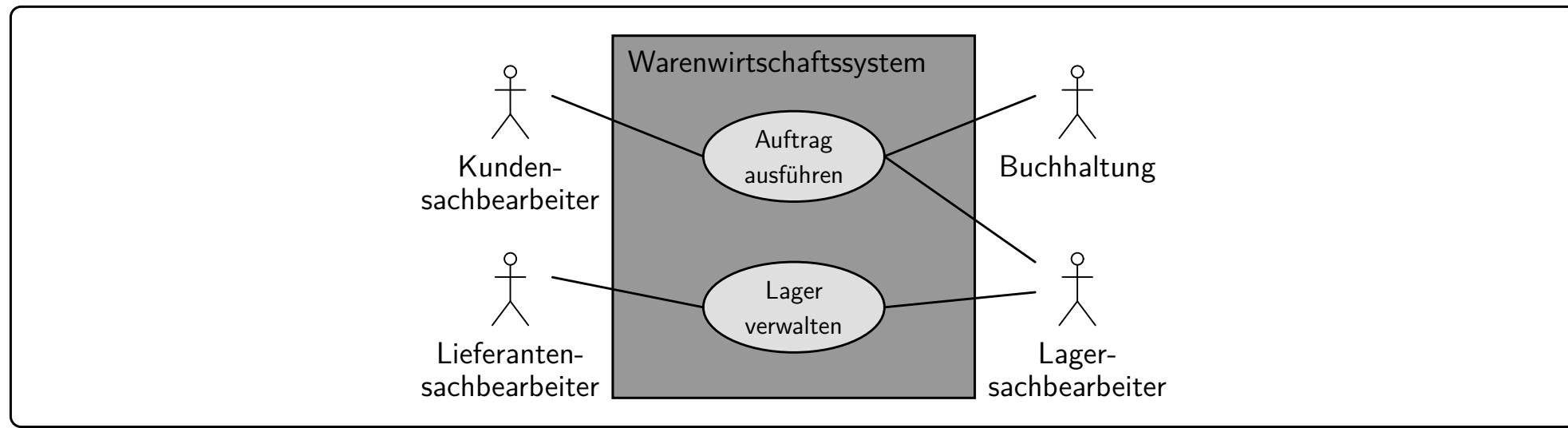
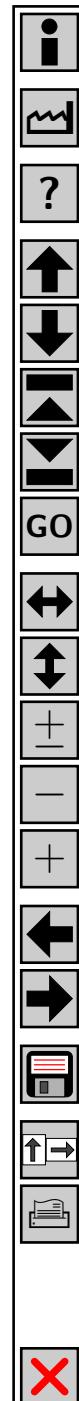
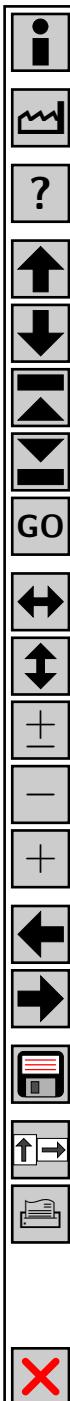


Abb. 49

Geschäftsprozessdiagramm für ein Warenwirtschaftssystem

- Beziehungen zwischen Geschäftsprozessen
 - **extend**-Beziehung
 - **include**-Beziehung
 - **Vererbungsbeziehung** (*Generalisierung*)





Extend-Beziehung

- Struktur einer **extend**-Beziehung
 - Ein Geschäftsprozess B stellt eine Ergänzung eines Geschäftsprozesses A dar, die von bestimmten **Bedingungen** abhängt → A besitzt eine **optionale Verzweigung** nach B
- Anwendung
 - Ein komplexer Geschäftsprozess kann zunächst in einer vereinfachten Form spezifiziert werden
 - Komplexe Sonderfälle können dann in die Erweiterungen verlagert werden
- **Beispiel 33:** Auftragsausführung (siehe [Abb. 50](#))
 - Geschäftsprozess **Auftrag ausführen**
 - ◊ Der Kunde wird lediglich darüber informiert, welche Artikel nicht lieferbar sind
 - Geschäftsprozess **Nachlieferung ausführen**
 - ◊ Der Kunde erhält zusätzlich die Information, wann die fehlenden Artikel nachgeliefert werden
 - ◊ Falls auch dieser Termin nicht einzuhalten ist, wird der Kunde erneut informiert

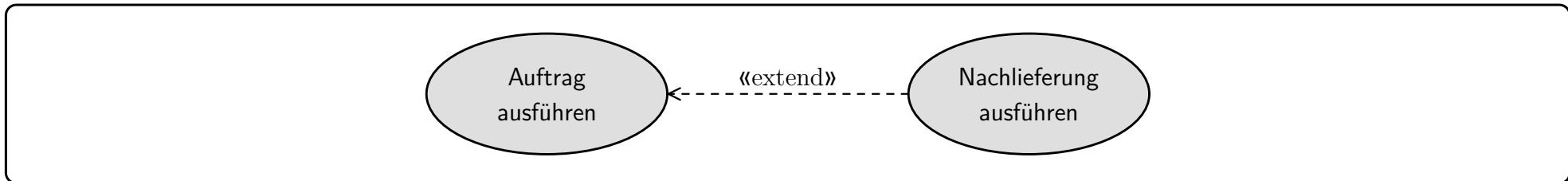
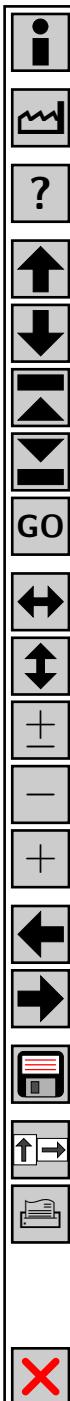
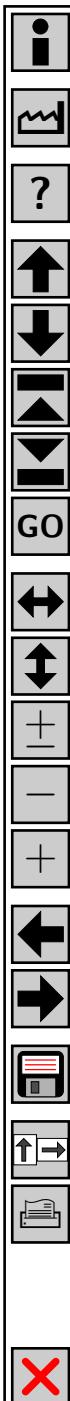


Abb. 50

Extend-Beziehung zwischen Geschäftsprozessen





Include-Beziehung

- Struktur einer **include**-Beziehung
 - Zwei Geschäftsprozesse **B1** und **B2** besitzen ein **gemeinsames Verhalten**
 - Dieses gemeinsame Verhalten kann dann in einem **eigenen** Geschäftsprozess **A** spezifiziert werden
 - **A** wird analog zu einem **Unterprogramm** benutzt
- **Beispiel 34:** Wareneingang (siehe [Abb. 51](#))
 - Gegeben sind die beiden Geschäftsprozesse **Wareneingang aus Einkauf bearbeiten** und **Wareneingang aus Produktion bearbeiten**
 - Beide besitzen ein gemeinsames Verhalten, welches innerhalb des neuen Geschäftsprozesses **Ware eingelagern** spezifiziert wird

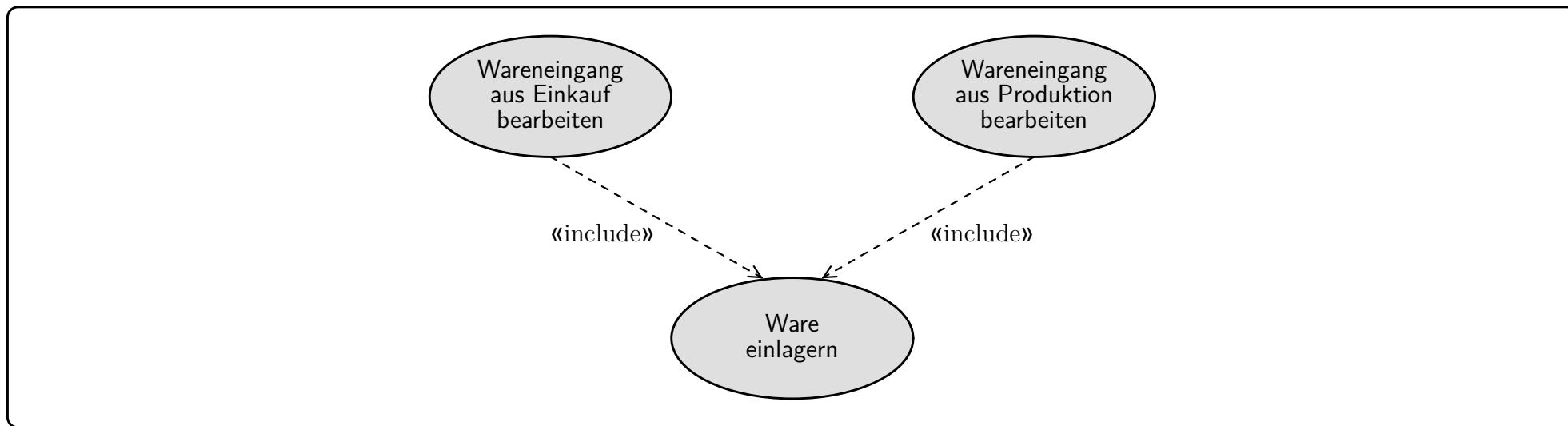
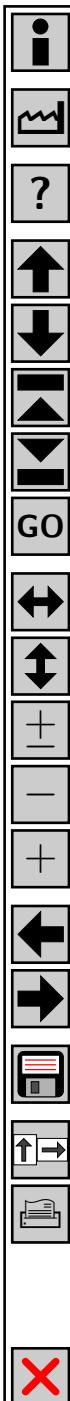


Abb. 51

Include-Beziehung zwischen Geschäftsprozessen



Der Geschäftsprozess Ware einlagern ist ein *künstliches Gebilde*, das an der Systemschnittstelle nicht auftaucht, d. h. niemals von einem Akteur aufgerufen wird

Botschaft

Begriffsdefinitionen

- **Definition 31: Botschaft (Message)**

Eine **Botschaft** ist die Aufforderung eines **Senders** (*Client*) an einen **Empfänger** (*Server, Supplier*), eine **Dienstleistung** zu erbringen.

Der Empfänger interpretiert diese Botschaft und führt eine Operation aus.

- Verarbeitung von Botschaften

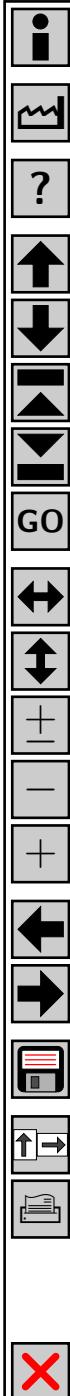
- Eine Botschaft löst (beim Empfänger) eine **Operation gleichen Namens** aus
- Falls ein Objekt in seiner Klasse die Operation nicht findet, werden die Oberklassen durchsucht

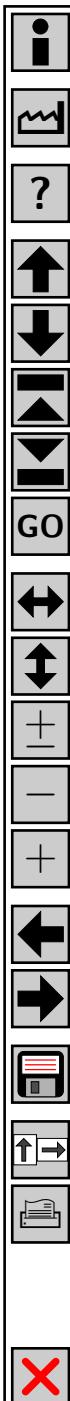


- **Der Sender einer Botschaft weiß nicht, wie die aufgerufene Operation implementiert ist**
- **Das Verhalten eines objektorientierten Systems wird durch die Botschaften beschrieben, mit denen Objekte untereinander kommunizieren**

- **Definition 32: Protokoll (Protocol)**

Die **Menge der Botschaften**, auf die Objekte einer Klasse reagieren, wird als **Protokoll** der Klasse bezeichnet.





UML-Notation für Botschaften

- **Beispiel 35:** Durchschnittsumsatz der Filialen einer Versicherung (vgl. [Abb. 52](#))
 - Der Jahresbeitrag für jeden Vertrag ist zu berechnen
 - Für die Darstellung der zu versendenden Botschaften wird hier ein **Kommunikationsdiagramm** benutzt (siehe Kapitel Szenario, [S. 264](#))

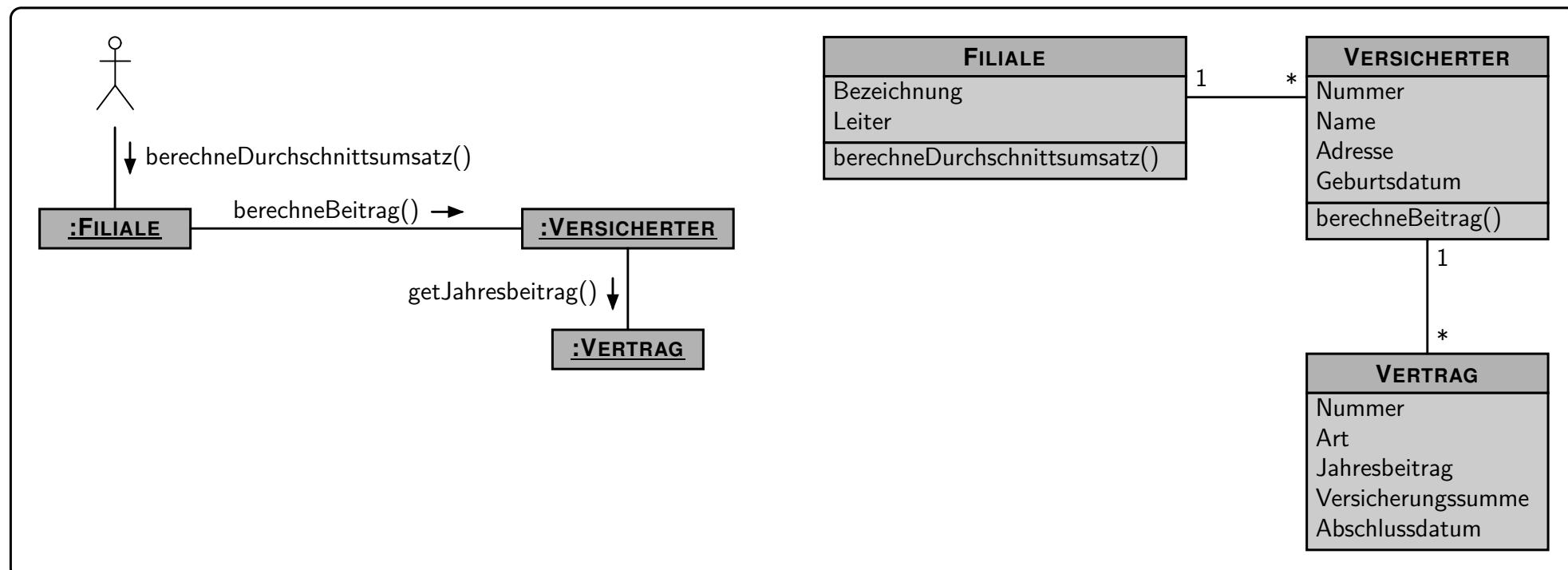


Abb. 52

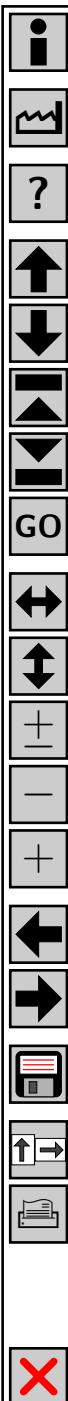
Senden von Botschaften

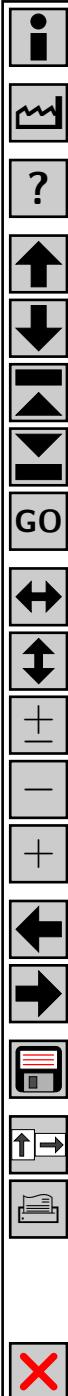


- Die zwischen den Objekten bestehenden *Beziehungen* definieren, an welche Objekte die jeweilige Botschaft zu versenden ist
- In Abb. 52 wird z. B. berechneBeitrag() an alle Versicherten geschickt, die zu einer Filiale gehören

Alternativ verwendete Begriffe

- Nachricht
- Operationsaufruf
- Methodenaufruf





Szenario

Begriffsdefinition

- **Definition 33: Szenario (Scenario)**

Ein **Szenario** ist eine **Sequenz** von **Verarbeitungsschritten**, die unter bestimmten Bedingungen auszuführen ist.

Die Verarbeitungsschritte beginnen mit dem **auslösenden Ereignis** und werden fortgesetzt, bis das (vorgegebene) Ziel erreicht ist oder aufgegeben wird.

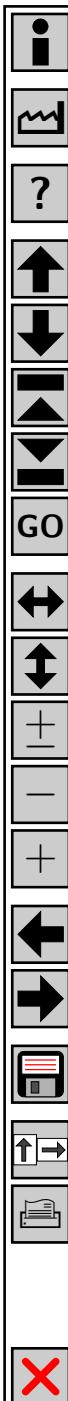
- Geschäftsprozesse und Szenarios

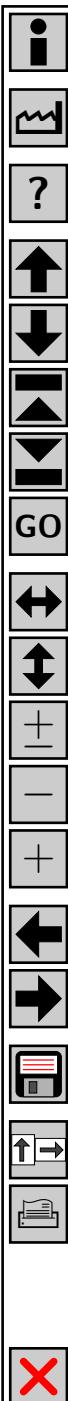
- Ein Geschäftsprozess wird durch eine Kollektion von Szenarios dokumentiert
 - Jedes Szenario wird durch eine oder mehrere **Bedingungen** definiert, die zu einem **speziellen Ablauf** des jeweiligen Geschäftsprozesses führen

- Szenario-Kategorien

- Szenarios, die eine **erfolgreiche Bearbeitung** des Geschäftsprozesses beschreiben
 - Szenarios, die zu einem **Fehlschlag** führen

- **Beispiel 36:** Szenarios für den Geschäftsprozess **Auftrag ausführen** → vgl. [Bsp. 31](#)
 - Auftrag für einen **Neukunden** bearbeiten, wenn mindestens ein **Artikel lieferbar** ist
 - Auftrag bearbeiten, wenn der **Kunde** bereits **existiert** und mindestens ein **Artikel lieferbar** ist
 - Auftrag bearbeiten, wenn der **Kunde** bereits **existiert**, sich seine **Daten geändert** haben und mindestens ein **Artikel lieferbar** ist
- UML-Modellierung von Szenarios
 - Verwendung von **Interaktionsdiagrammen** (*Interaction Diagram*)
 - UML-Diagrammtypen
 - ◊ **Sequenzdiagramm** (*Sequence Diagram*)
 - ◊ **Kommunikationsdiagramm** (*Communication Diagram*)





Sequenzdiagramm

UML-Notation

- Struktur eines Sequenzdiagramms (siehe [Abb. 53](#))
 - Die **vertikale Dimension** definiert (von oben nach unten) die **Zeit**
 - In der **horizontalen Dimension** werden die **Objekte** eingetragen

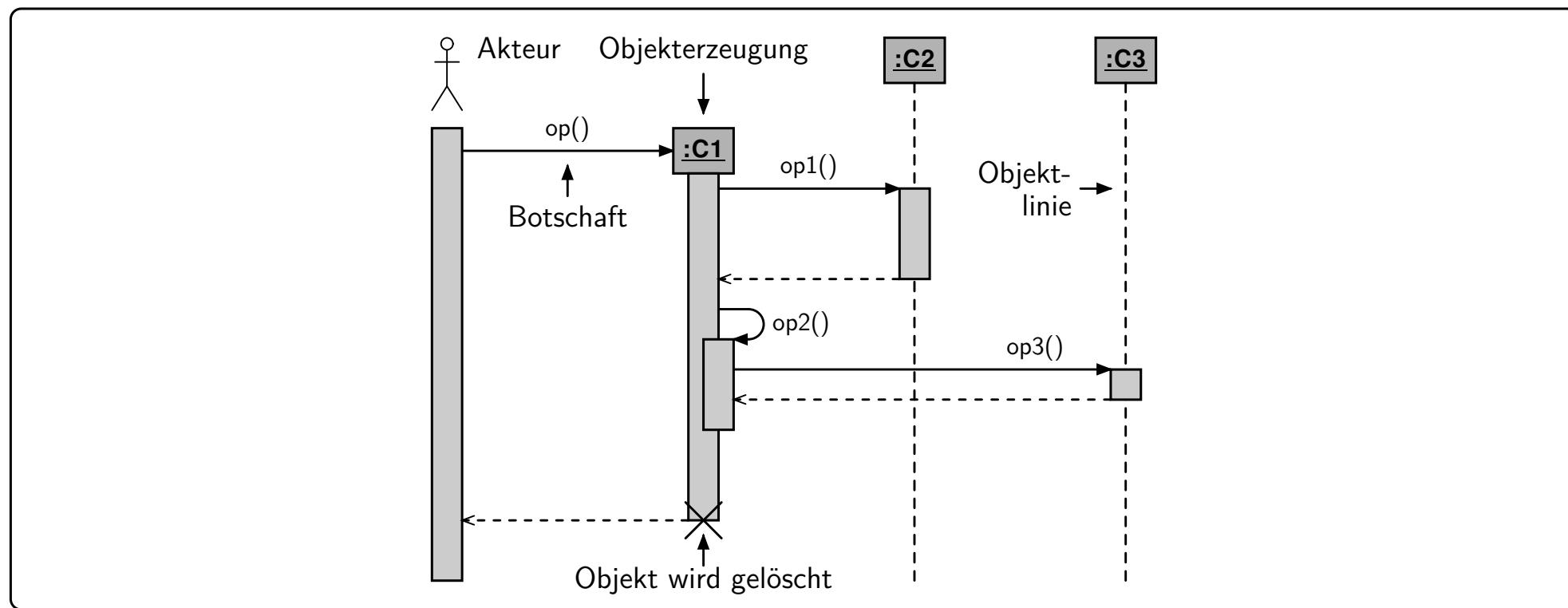
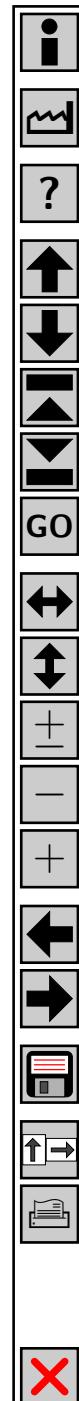
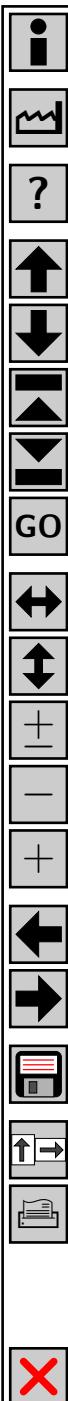


Abb. 53

Basiselemente eines Sequenzdiagramms



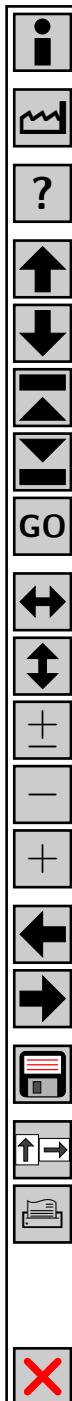
Basiselemente eines Sequenzdiagramms

- Objekt
 - Repräsentation durch ein **Objektsymbol**
 - Üblicherweise werden **anonyme** Objekte benutzt
- Objektlinie
 - Jedes Objekt wird durch eine **Objektlinie** dargestellt → Verwendung einer gestrichelten Linie
 - Am oberen Ende der Linie steht ein **Objektsymbol**
 - Die Objektlinie repräsentiert die **Existenz** des Objekts während einer bestimmten Zeit
 - ◊ Die Linie beginnt mit der Objekterzeugung
 - ◊ Sie endet mit dem Löschen des Objekts
- Objekterzeugung
 - Die Botschaft, die zur Objekterzeugung führt, zeigt auf das Objektsymbol
- Löschen eines Objekts
 - Das Ende der Objektlinie wird durch ein großes "X" markiert

- Operationsausführung
 - Eine Operation wird durch den Empfang einer **gleichnamigen Botschaft** aktiviert
 - Die Ausführung wird durch ein **schmales Rechteck** auf der Objektlinie angezeigt
 - Operationsende bei **sequentiellen Systemen**
 - ◊ Die mit dem Operationsende verbundene Rückkehr des Kontrollflusses an die rufende Operation **kann** durch einen **gestrichelten Pfeil** markiert werden → **Rückgabepfeil**
 - Operationsende bei **nebenläufigen** bzw. asynchronen Systemen
 - ◊ Da hier eine Rückkehr zur rufenden Operation nicht notwendigerweise stattfindet, **muss** der Rückgabepfeil bei **synchronen Operationsaufrufen** verwendet werden

Beispiel: Auftragsbearbeitung

- **Beispiel 37:** Auftrag für einen Neukunden bearbeiten → vgl. [Abb. 54](#)
 - Modellierung des ersten Szenarios aus dem Geschäftsprozess **Auftrag ausführen** (siehe [Bsp. 36](#))
→ Bedingung: mindestens ein Artikel ist lieferbar
 - Auf die Darstellung der Auftragspositionen wird hier verzichtet → vgl. [Abb. 55](#)



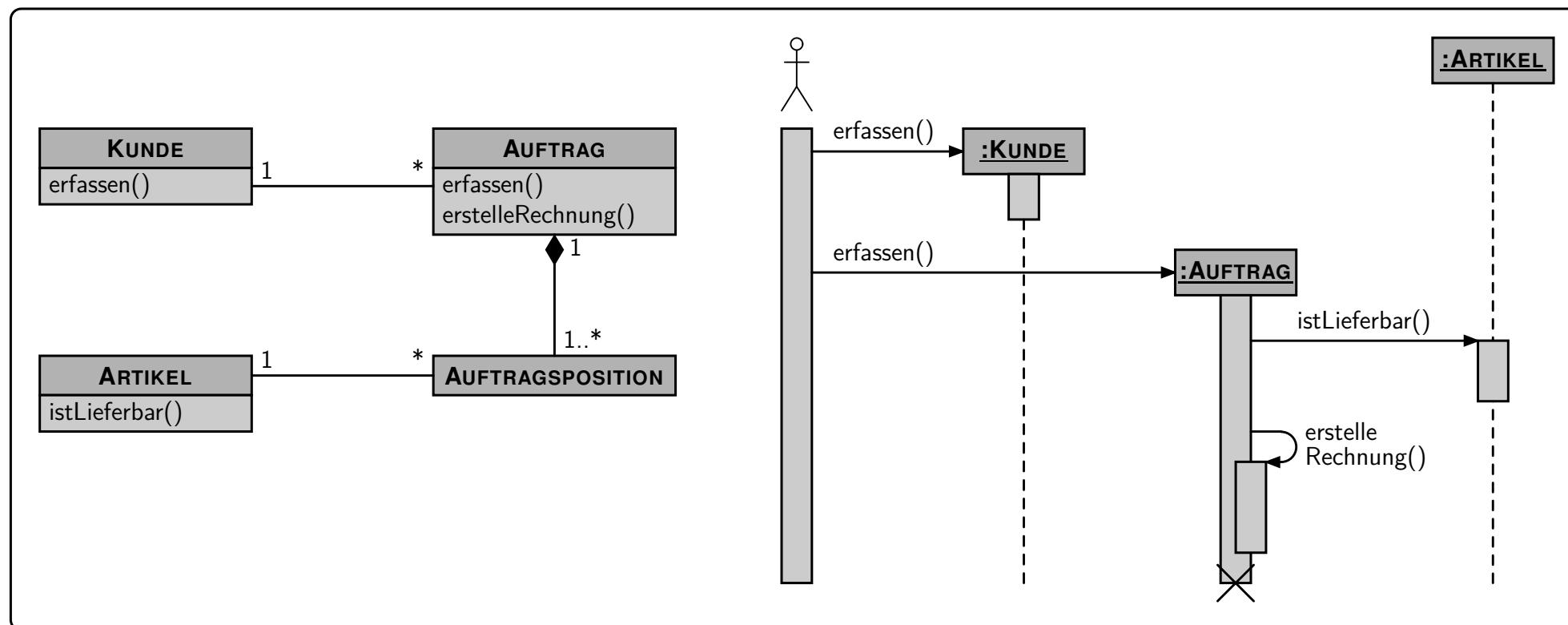
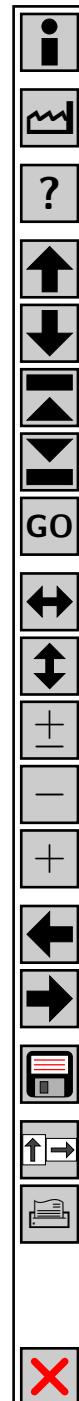
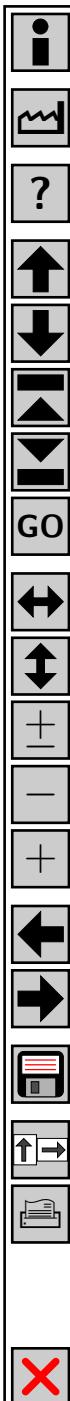


Abb. 54

Auftrag für einen Neukunden bearbeiten





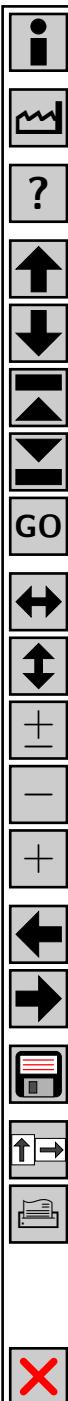
Bedingungen und Wiederholungen

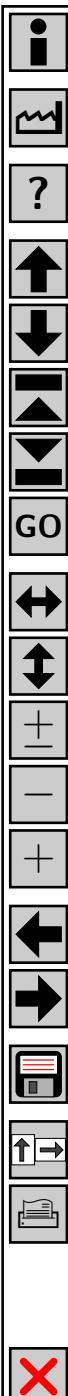
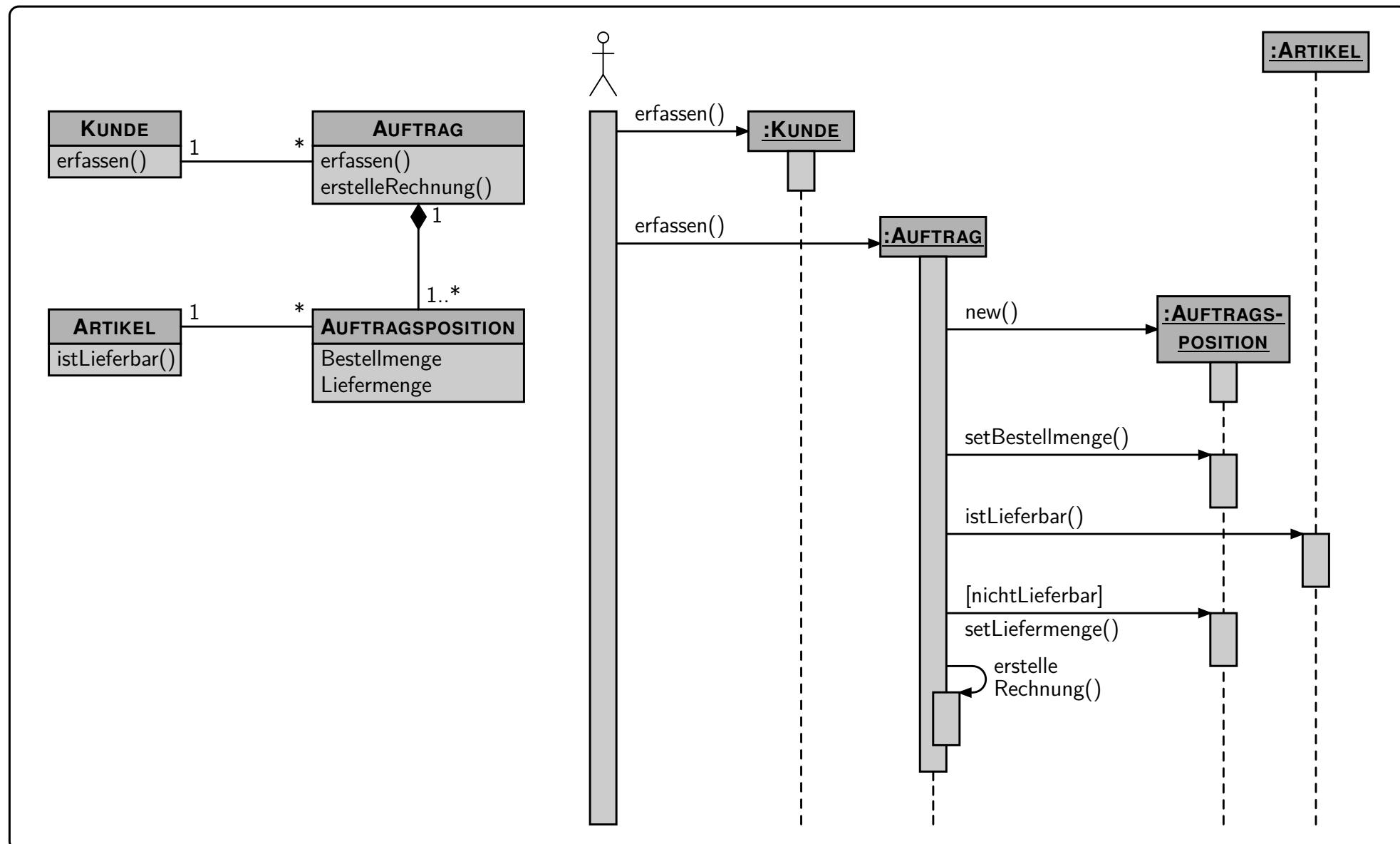
- **Bedingung (Condition)**
 - Eine Bedingung wird in **eckigen Klammern** angegeben
[Bedingung] operation()
 - Die Botschaft **operation ()** wird nur gesendet, wenn die Bedingung erfüllt ist
- **Wiederholung (Iteration)**
 - Eine Wiederholung wird durch einen Stern * gekennzeichnet
 - * operation()
 - ◊ Die Botschaft **operation ()** kann mehrfach gesendet werden
 - Zusätzlich kann eine Bedingung angegeben werden
 - * [Bedingung] operation()
 - ◊ Die Botschaft **operation ()** wird wiederholt gesendet, solange die Bedingung gilt

!

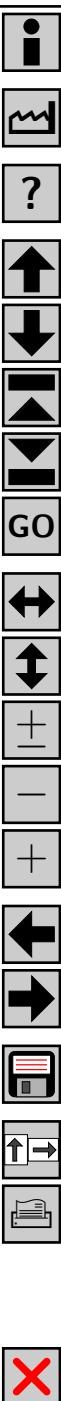
 - **Werden in einem Sequenzdiagramm keine Iterationen eingetragen, so definiert UML, dass die Anzahl der Iterationen unspezifiziert ist → es können also mehrere sein**
 - **Bedingungen werden in der Analyse oft umgangssprachlich formuliert**

- Abb. 55 zeigt das detaillierte Sequenzdiagramm für die Auftragsbearbeitung → vgl. Bsp. 37
 - Auf die Spezifikation der Iterationen wurde verzichtet
 - Sie ergeben sich implizit aus den Kardinalitäten der Assoziationen im Klassendiagramm





- **Beispiel 38:** Auftrag für einen existierenden Kunden bearbeiten → vgl. Abb. 56
 - Modellierung des zweiten und dritten Szenarios aus dem Geschäftsprozess **Auftrag ausführen** → siehe [Bsp. 36](#)
 - Durch die Verwendung einer Bedingung können die beiden Szenarios in einem Sequenzdiagramm zusammengefasst werden



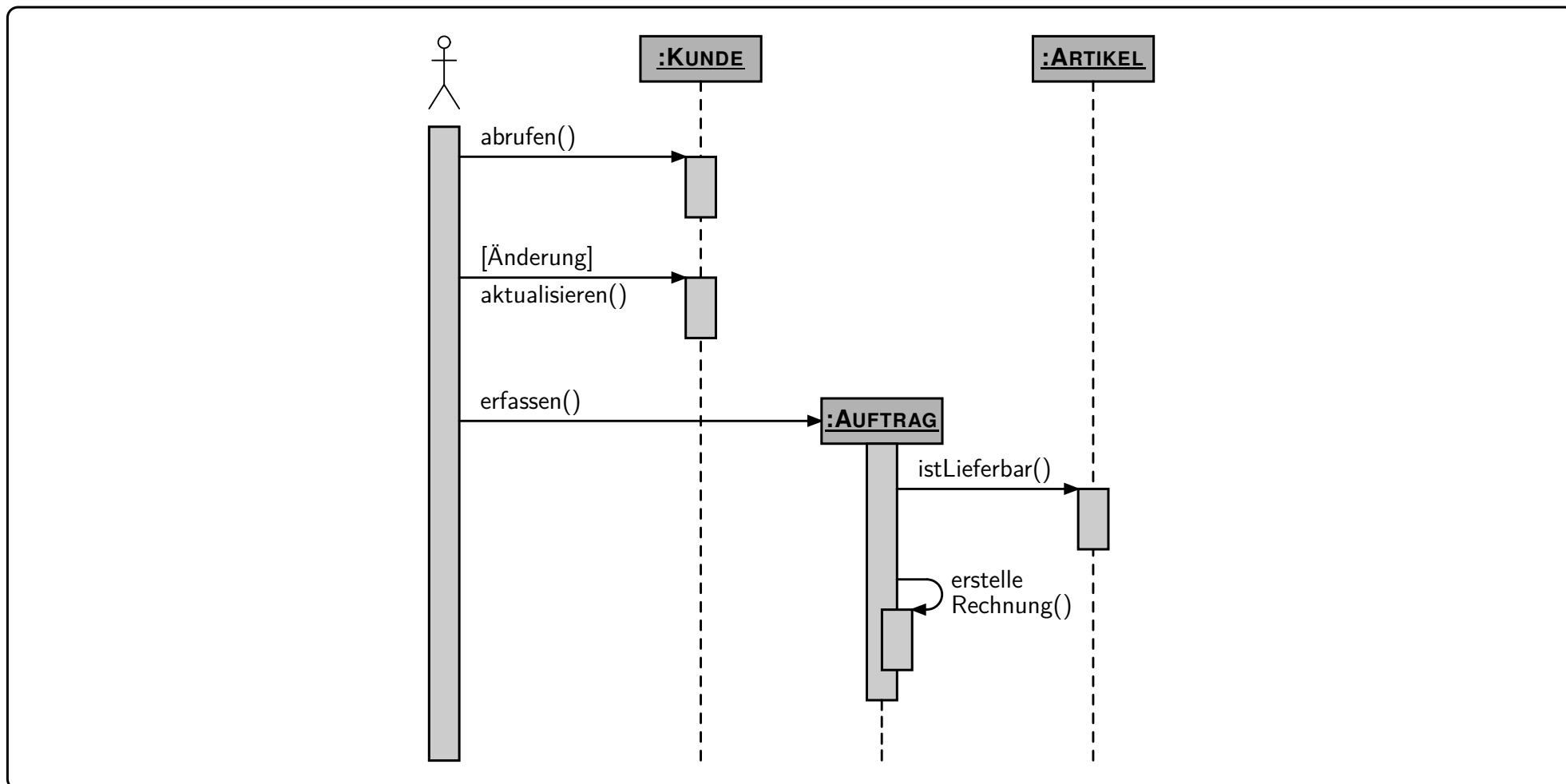
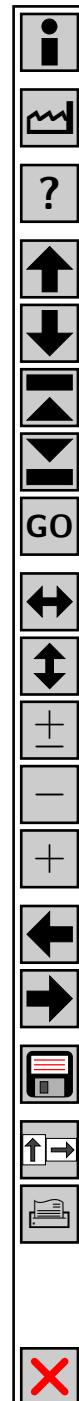
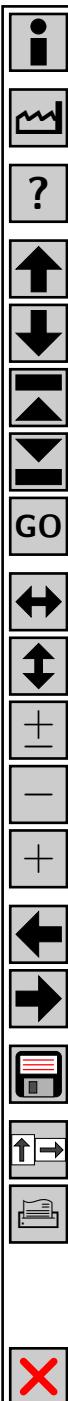


Abb. 56

Auftrag für einen existierenden Kunden bearbeiten

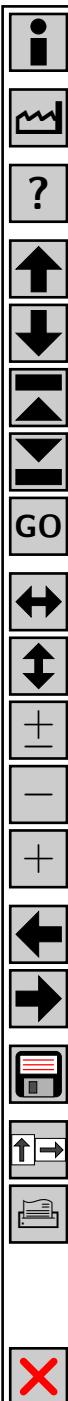


Konsistenzanforderungen

- Sequenz- und Klassendiagramm müssen konsistent sein
 - Alle **Botschaften**, die an ein Objekt gesendet werden, müssen im Klassendiagramm in der **Operationsliste** enthalten sein bzw. **Verwaltungsoperationen** der Klasse sein



- **Verwaltungsoperationen** werden im Sequenzdiagramm eingetragen, um die Kommunikation der Objekte vollständig zu beschreiben
- Zur Erinnerung: In Klassendiagrammen werden Verwaltungsoperationen nicht modelliert

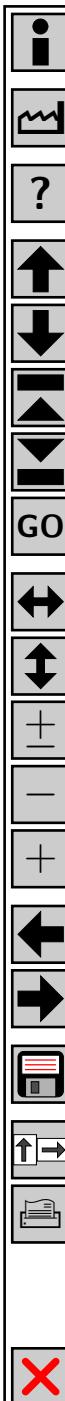


Klassen in Sequenzdiagrammen

- Bei der Verwendung einer **Klassenoperation** sollte statt eines Objektsymbols das **Klassensymbol** im Sequenzdiagramm verwendet werden → Erweiterung der UML-Notation



- Auf der Ebene der Systemanalyse wurde davon ausgegangen, dass eine Klasse alle ihre Objekte kennt**
- Operationen auf dieser Objektmenge wurden als Klassenoperationen modelliert**
- Alternative Notation für die Gesamtmenge der Objekte einer Klasse
 - Verwendung eines Objektsymbols mit dem Namen **ALL: KLASSE**
 - ALL** steht dabei für die Menge aller Objekte



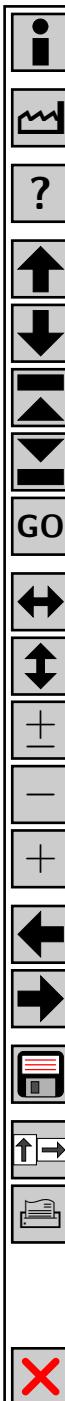
Einsatz von Sequenzdiagrammen

- Analysephase
 - Es werden diejenigen Teile des Kontrollflusses zwischen Objekten beschrieben, die für eine **Diskussion der fachlichen Korrektheit** notwendig sind
 - Die Sequenzdiagramme sind dann eine **Vorgabe** für die **Entwurfs-** und **Implementierungsphase**



- **Es gibt keinen absoluten Maßstab für den Detaillierungsgrad**
- **Er sollte im Einzelfall auf die Zielgruppe abgestimmt werden**

- Entwurfsphase
 - Die Sequenzdiagramme sollen **alle Operationsaufrufe** zwischen den beteiligten Objekten spezifizieren
 - Damit ist dann eine direkte Umsetzung in eine Implementierung möglich



Kommunikationsdiagramm

UML-Notation

- Struktur eines Kommunikationsdiagramms → siehe [Abb. 57](#)
 - Das Diagramm beschreibt **Objekte** und **Verbindungen** zwischen Objekten
 - An die Verbindungen können **Botschaften** eingezeichnet werden
 - Die **Reihenfolge** der Operationsausführungen kann durch eine **hierarchische Nummerierung** der Botschaften spezifiziert werden
 - ◊ op() erzeugt das Objekt der Klasse **C1**
 - ◊ Das Objekt aktiviert die Operation op1()
 - ◊ Danach wird op2() ausgeführt
 - ◊ op2() ruft die Operation op3() auf

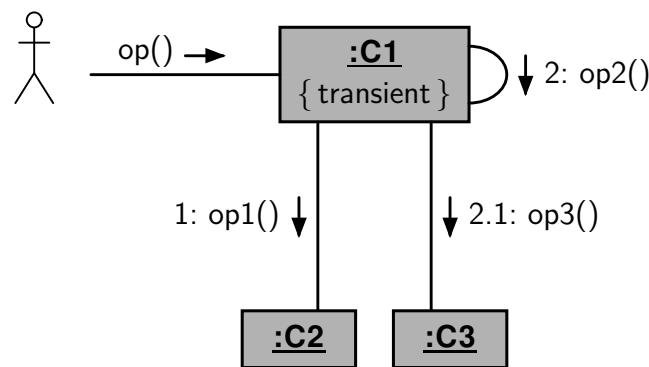
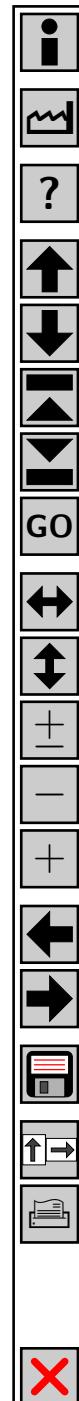
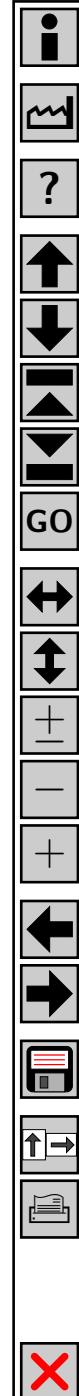


Abb. 57

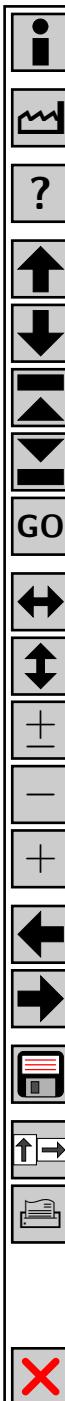
UML-Notation für Kommunikationsdiagramme



Das Kommunikationsdiagramm aus Abb. 57 beschreibt denselben Ablauf wie das Sequenzdiagramm aus Abb. 53



- Spezifikation der Lebensdauer von Objekten
 - Ein Objekt wird **erzeugt** und existiert nach Ausführung der Operationen des Kommunikationsdiagramms weiter
→ Verwendung des Merkmals { new }
 - Das Objekt existiert bereits und wird im Rahmen der Ausführung der Operationen des Kommunikationsdiagramms **gelöscht**
→ Verwendung des Merkmals { destroyed }
 - Das Objekt wird während der Operationsausführungen **erzeugt** und wieder **gelöscht** (siehe Abb. 57)
→ Verwendung des Merkmals { transient }
- Spezifikation der Lebensdauer von Verbindungen
 - Es werden (wie bei den Objekten) die Merkmale { new }, { destroyed } und { transient } verwendet
 - Die Bedeutung ist diesselbe wie bei Objekten



Klassen im Kommunikationsdiagramm

- Analog zum Sequenzdiagramm muss die Möglichkeit bestehen, eine **Klassenoperation** zu aktivieren
- Anstelle des Objekts wird dann das **Klassensymbol** in das Diagramm eingetragen → siehe Abb. 59
- Alternativ kann auch ein Objektsymbol mit dem Namen ALL: KLASSE verwendet werden

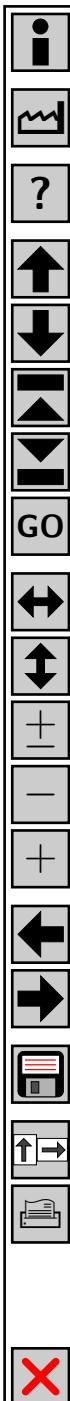
Permanente und temporäre Objektverbindungen

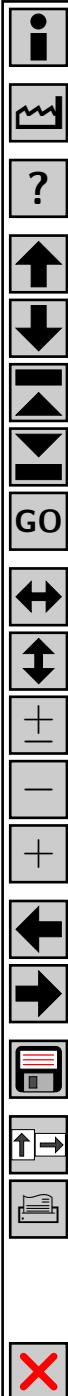
- **Permanente** Verbindungen entsprechen den **Assoziationen**
- **Temporäre** Verbindungen
 - Sie bestehen nur für die **Dauer einer Kommunikation**
 - Das angesprochene Empfängerobjekt kann hierbei auch ohne Vorliegen einer Assoziation vom Sender eindeutig identifiziert werden

- **Beispiel 39:** Aufbau einer temporären Verbindung
 - Alle Objekte werden an der Benutzungsschnittstelle (z. B.) über eine **Auswahlliste** (*List Box*) angeboten
 - Der Benutzer wählt das entsprechende Objekt aus
 - Eine Operation wird für dieses Objekt aufgerufen
- Notation für temporäre Verbindungen
 - Objekte in Kommunikationsdiagrammen können nur kommunizieren, wenn eine Verbindung zwischen ihnen eingetragen ist
 - Temporäre Verbindungen werden mit dem Stereotyp «temp» gekennzeichnet
 - Damit lassen sie sich von Assoziationen unterscheiden



- Ein Objekt kann (in einem Kommunikationsdiagramm) jederzeit eine Botschaft an sich selbst schicken
- Die benötigte *implizite Verbindung* (*Self Link*) wird bei Bedarf eingezeichnet und muss nicht als temporär gekennzeichnet werden, da ein Objekt sich selbst immer kennt (vgl. Abb. 57)





Kommunikationsdiagramm im Vergleich zum Objektdiagramm

- Objektdiagramm
 - Modellierung eines **Schnappschusses** der Systemstruktur
- Kommunikationsdiagramm
 - Spezifikation der Zusammenarbeit von Objekten bei der Ausführung einer bestimmten Operation
- **Beispiel 40:** Kommunikations- und Objektdiagramm für eine Versicherung → siehe [Abb. 58](#)
 - Objektdiagramm
 - ◊ Exemplarische Modellierung der Darmstädter und der Mannheimer Filiale einer Versicherung
 - Kommunikationsdiagramm
 - ◊ Modellierung der Ausführung der Operation `berechneDurchschnittsumsatz()`
 - ◊ Jedes verwendete Objekt stellt einen Platzhalter für ein beliebiges Objekt der entsprechenden Klasse dar

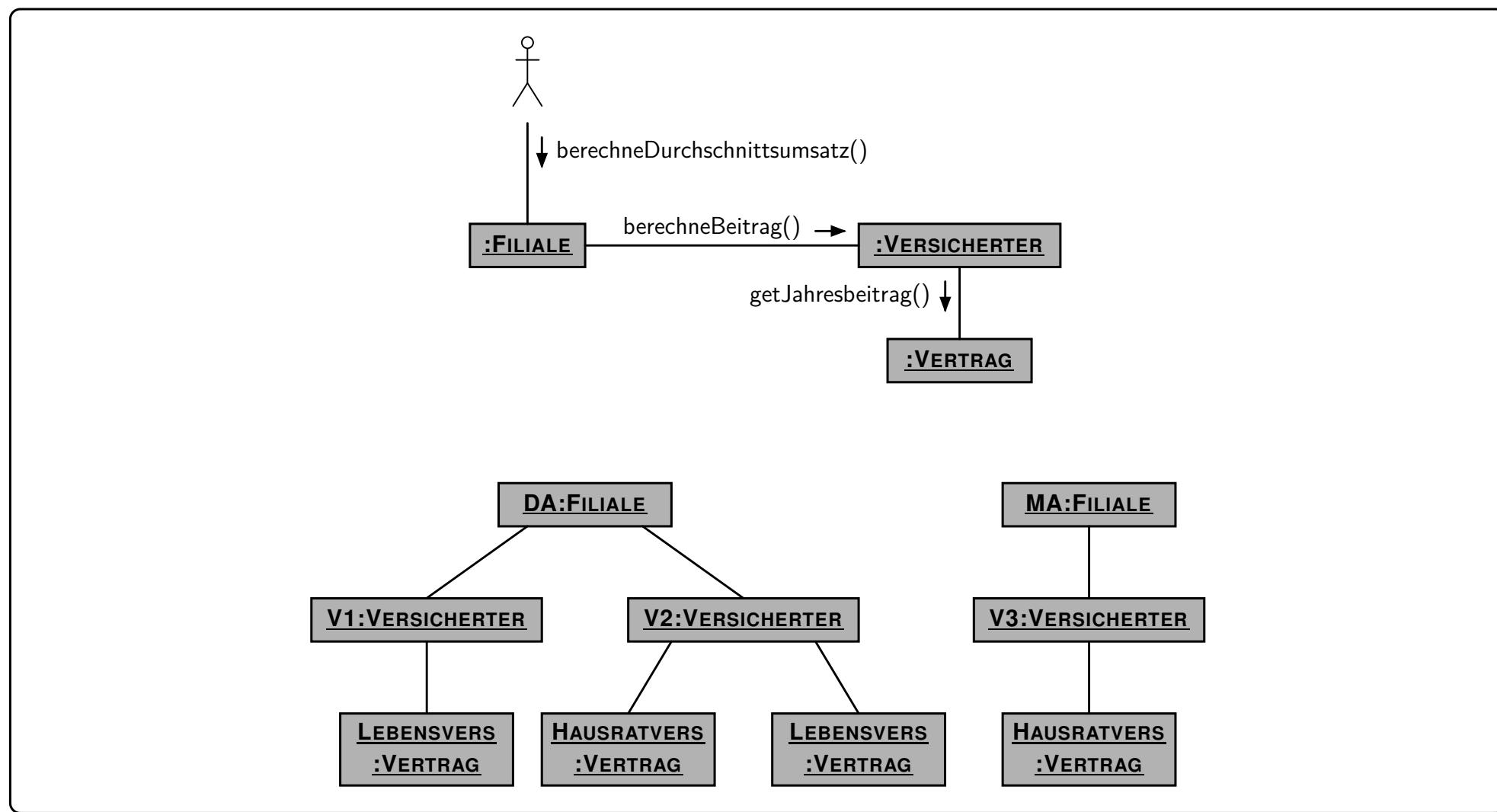
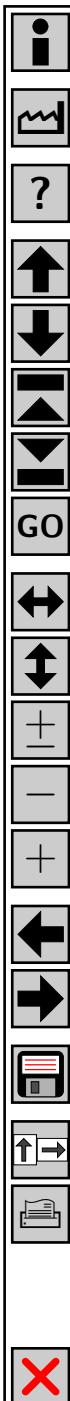
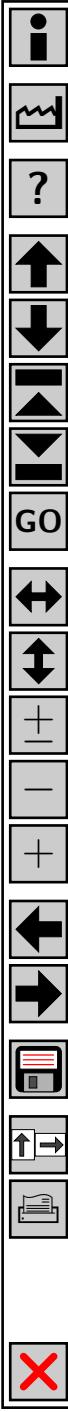


Abb. 58

Kommunikations- und Objektdiagramm



Sequenzdiagramm im Vergleich zum Kommunikationsdiagramm

- Sequenzdiagramm
 - Der **zeitliche Aspekt** des dynamischen Verhaltens wird hervorgehoben
 - Die **Reihenfolge** und die **Verschachtelung** der Operationen ist leicht zu erkennen
 - Es kann **mehrere externe Operationen** enthalten, die nacheinander von einem Akteur aktiviert werden

 **Sequenzdiagramme eignen sich sehr gut für die Modellierung komplexer Szenarios**

- Kommunikationsdiagramm
 - Die **Verbindungen** zwischen Objekten werden betont
 - Die Reihenfolge und Verschachtelung der Operationen wird über eine **hierarchische Nummerierung** angegeben
 - Die Operationsreihenfolge ist (in der Regel) weniger deutlich sichtbar
 - Für **jede externe Operation** ist ein eigenes Kommunikationsdiagramm zu erstellen



- Der Systemanalytiker kann sich bei einem Kommunikationsdiagramm zunächst auf die Objekte und ihre prinzipielle Kommunikation konzentrieren
- Die Spezifikation der konkreten Ausführungsreihenfolge kann in einem zweiten Schritt durch die hierarchische Nummerierung hinzugefügt werden
- Abb. 59 zeigt einen Vergleich der beiden Interaktionsdiagrammtypen

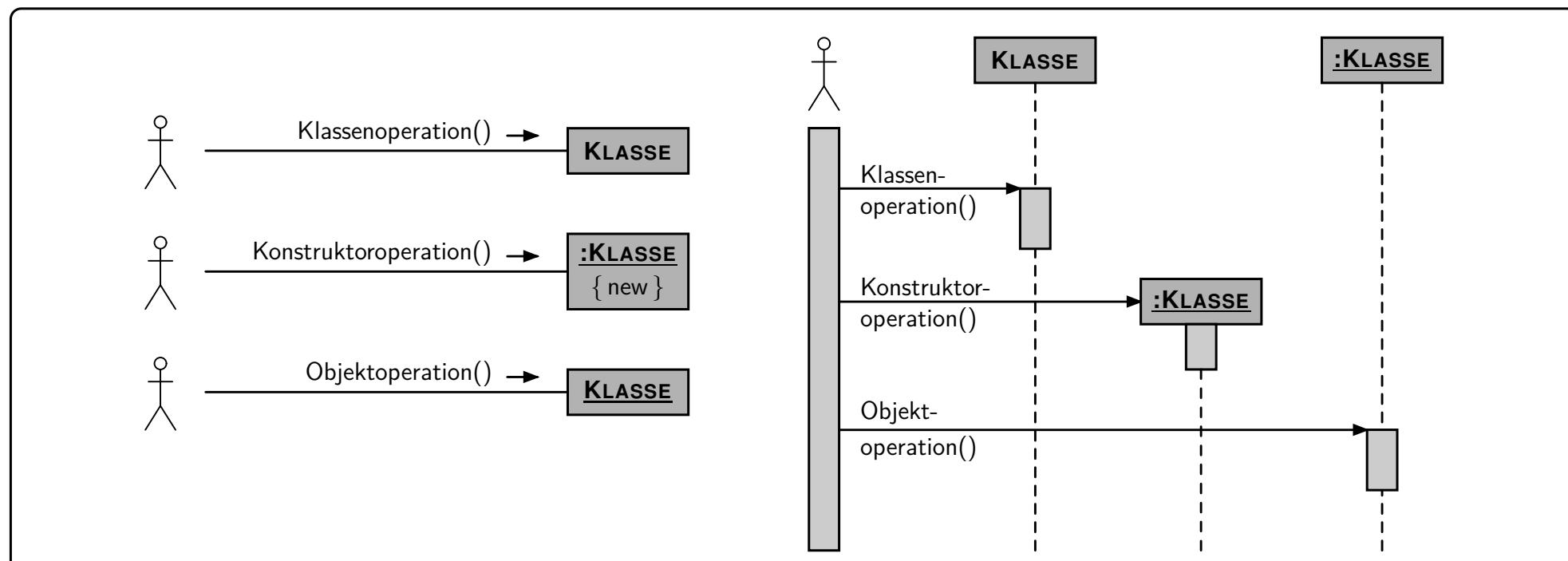
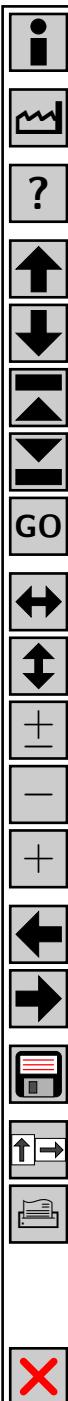
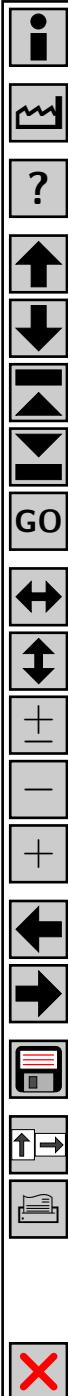


Abb. 59

Kommunikations- und Sequenzdiagramm





Zustandsdiagramm

Zustandsautomat

Begriffsdefinition

- **Definition 34:** **Zustandsautomat** (*Finite State Machine*)

Ein **Zustandsautomat** besteht aus **Zuständen** und **Zustandsübergängen**.

- **Definition 35:** **Zustand** (*State*)

Ein **Zustand** ist eine Zeitspanne, in der (z. B.) ein Objekt auf ein **Ereignis** wartet, d. h. das Objekt verweilt eine bestimmte Zeit in diesem Zustand.

- **Definition 36:** **Ereignis** (*Event*)

Ein **Ereignis** ist ein für den aktuellen Kontext (z. B. einen Zustand) **relevantes Vorkommnis**, das **keine Dauer** besitzt.

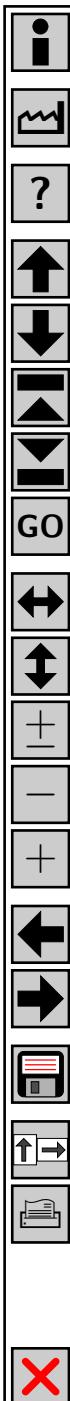
- **Definition 37: Zustandsübergang (Transition)**

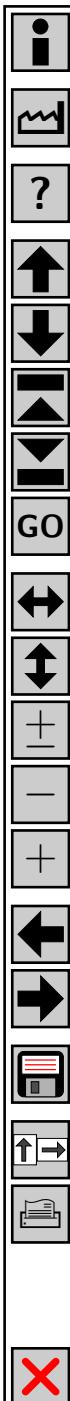
Ein **Zustandsübergang** verbindet einen **Ausgangszustand** mit einem **Folgezustand**, wobei Ausgangs- und Folgezustand identisch sein können.

Ein Zustandsübergang wird immer durch ein **Ereignis** ausgelöst.

Wichtige Eigenschaften

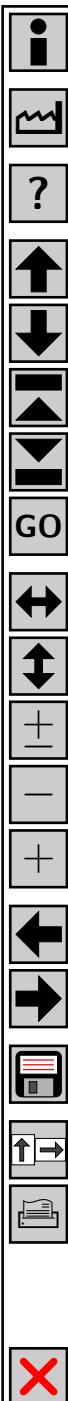
- Das Systemelement, das durch einen Zustandsautomaten beschrieben wird (z. B. ein Objekt), befindet sich **immer in genau einem Zustand**
- Der **Lebenszyklus** des Systemelements wird als eine **Folge** von **Zuständen** modelliert
- Tritt ein Ereignis ein, so hängt der Folgezustand vom aktuellen Zustand und dem Ereignis selbst ab
- Wird im aktuellen Zustand das eingetretene Ereignis **nicht erwartet**, so wird es **vergessen**





Einsatzgebiete in der Systemanalyse

- Modellierung des **Lebenszyklus** von **Objekten**
 - Welche Operationen dürfen in welchen Zuständen ausgeführt werden → Zustandsautomat für eine Klasse **BUCH** (in einer Bibliothek)
- Modellierung von **Geschäftsprozessen**
 - Die Verwendung einer Geschäftsprozessschablone erlaubt keine Spezifikation von **nebenläufigen Aktionen**
 - Zustandsautomaten bieten diese Möglichkeit
- Modellierung **komplexer Operationen**
 - Beispiel: Die Operation `bezahlenParkgebühr()` eines Parkscheinautomaten im Parkhaus



Darstellung von Zustandsautomaten durch Zustandsdiagramme

Struktur eines Zustandsdiagramms

- In der Objektorientierung wird das **Zustandsdiagramm** (*Statechart Diagram*) zur graphischen Darstellung des Zustandsautomaten verwendet

- Abb. 60 zeigt die prinzipielle Struktur eines Zustandsdiagramms

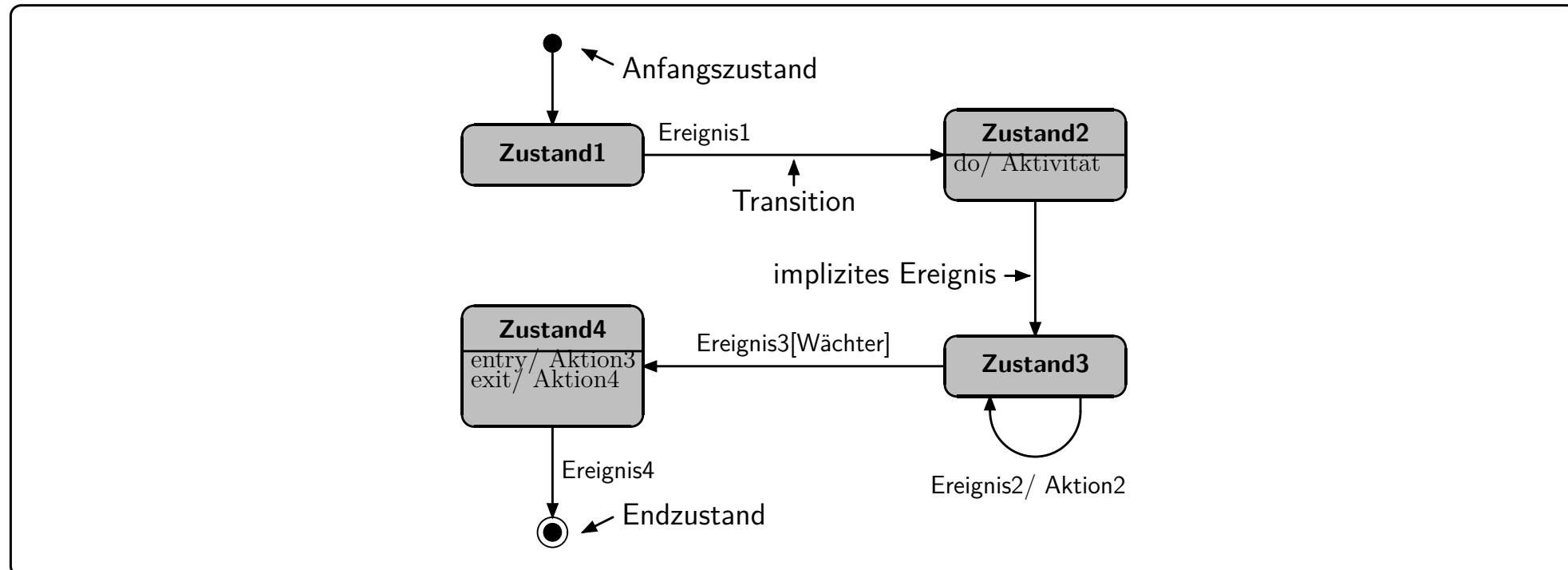
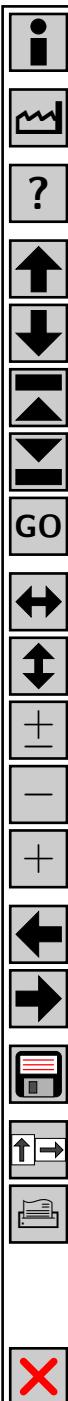
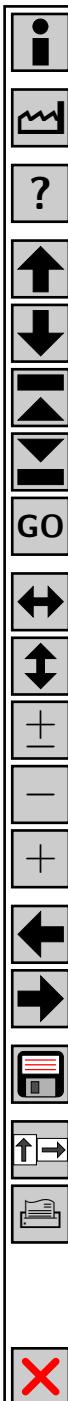


Abb. 60

Notation des Zustandsdiagramms



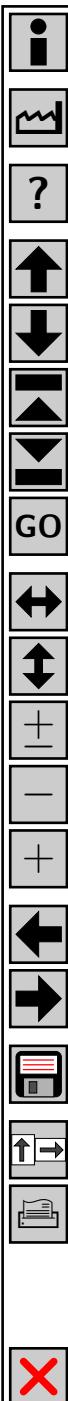


- **Zustandsname**

- Der Zustandsname ist **optional**
 - ◊ Zustandsnamen müssen innerhalb des Zustandsdiagramms **eindeutig** sein
 - ◊ Es sollen **Adjektive** oder **Partizipien** aber **keine Verben** verwendet werden
 - ◊ Beispiel: **ausgeliehen** statt **ausleihen**
- Zustände ohne Namen heißen **anonyme** Zustände
- Alle anonymen Zustände eines Zustandsdiagramms sind (per Definition) **voneinander verschieden**



- Zur besseren Darstellbarkeit/Lesbarkeit eines Zustandsdiagramms kann derselbe benannte Zustand **mehrmals eingezeichnet** werden
- Von dieser Möglichkeit sollte nur sehr restriktiv Gebrauch gemacht werden

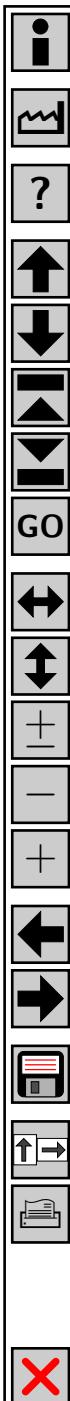


- **Anfangszustand** (*Initial State*)

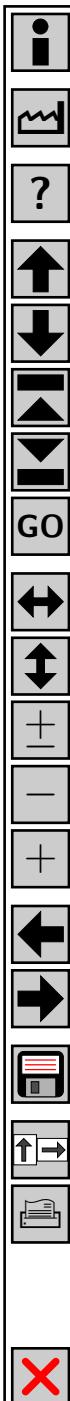
- Jeder Zustandsautomat besitzt **genau einen** Anfangszustand
- Es handelt sich um einen **Pseudozustand** → graphisches Hilfsmittel
- Ein Systemelement kann sich nicht in diesem Zustand befinden



- Die Transition aus dem Anfangszustand in einen Folgezustand kann z.B. die Objekterzeugung darstellen
- Ein neu erzeugtes Objekt befindet sich dann in dem entsprechenden Folgezustand, d.h. dieser ist der Anfangszustand des Objekts



- **Endzustand (Final State)**
 - Ein Zustandsautomat **kann** einen Endzustand besitzen
 - Aus diesem Zustand führen keine Transitionen hinaus
 - Der Endzustand ist ebenfalls ein **Pseudozustand**
 - Durch den Übergang in den Endzustand hört das zugehörige Systemelement auf zu existieren, z. B.
 - ◊ Löschen des Objektes
 - ◊ Beendigung der Geschäftsprozessausführung
- **Transition**
 - Die Ausführung des Zustandsübergangs wird auch als **feuern** bezeichnet → *die Transition feuert*
 - Beim Zustandsübergang kann eine **atomare Aktion** ausgeführt werden → Aufruf einer Operation (z. B. Objekterzeugung)



Beispiel: Zustandsautomat der Klasse BUCH

- **Beispiel 41:** Verwaltung von Büchern in einer Bibliothek → siehe [Abb. 61](#)
 - Das Ereignis neues Buch liegt vor führt zur Ausführung der Operation **erfassen()**, die den Zustandsautomaten aus dem Anfangszustand in den ersten echten Zustand überführt
→ Erzeugung des Buch-Objekts
 - Zur Vereinfachung des Beispiels soll es von jedem Buch nur ein Exemplar geben
 - Die Ausführung der Operation **entfernen()** führt in den Endzustand des Zustandsautomaten
→ Löschen des Buch-Objekts

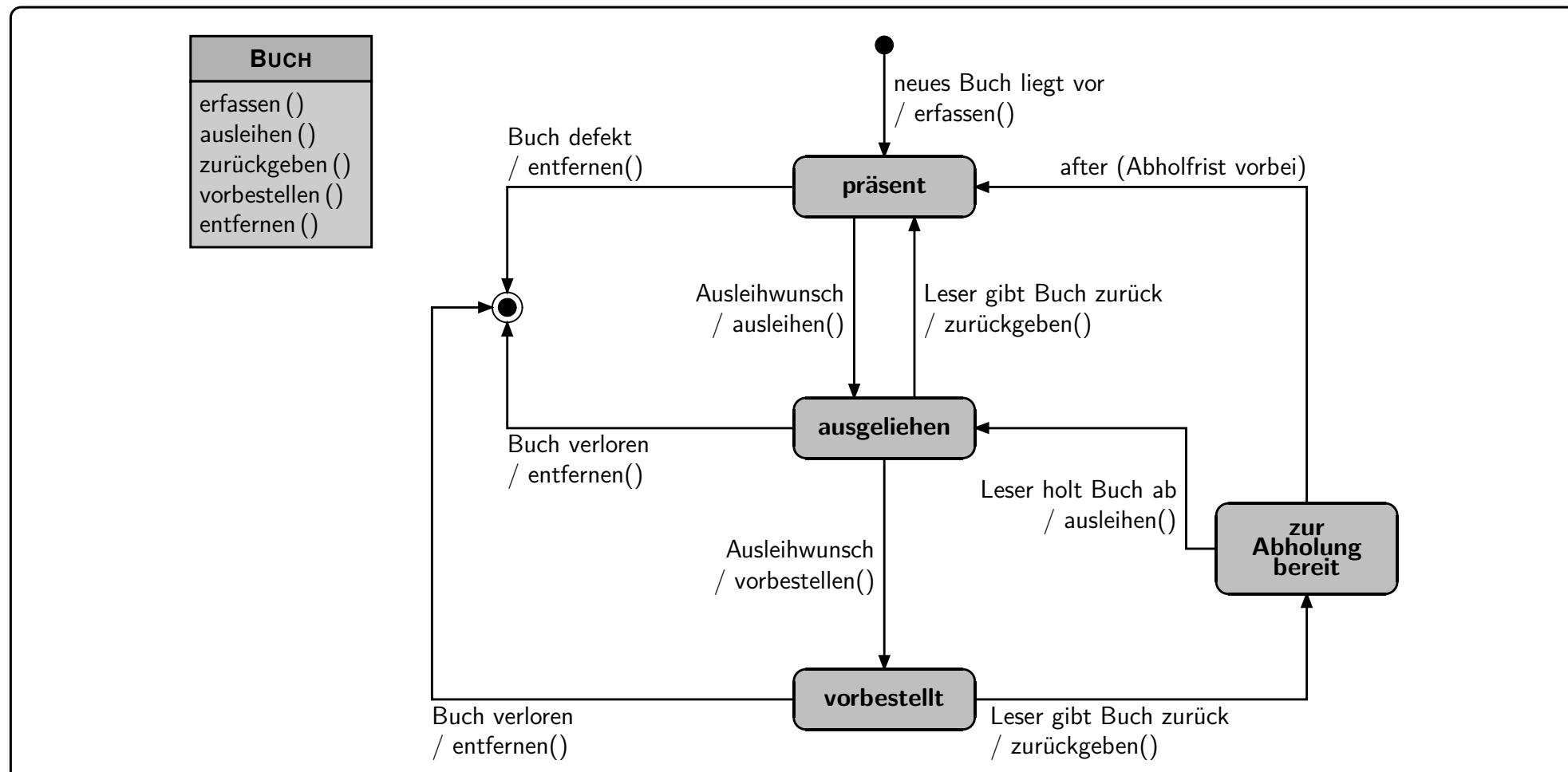
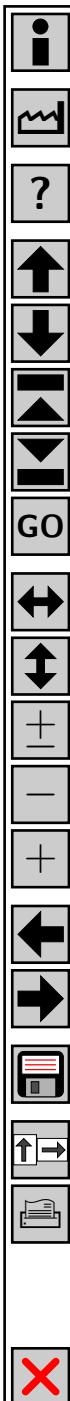
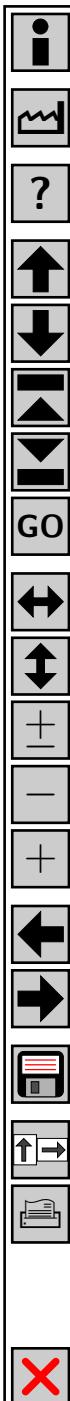


Abb. 61

Zustandsautomat der Klasse **BUCH**

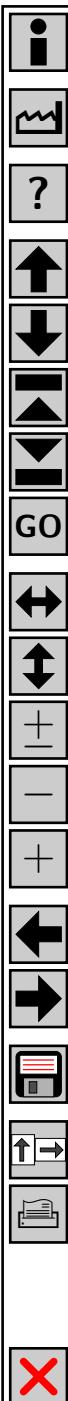
Ereignistypen

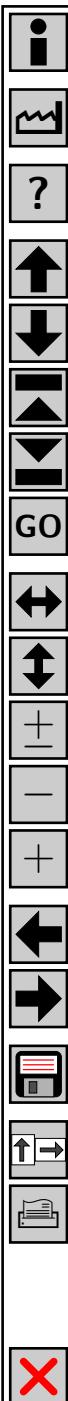
- Eine **Bedingung**, die **wahr** wird
 - Beispiel: when (Temperatur > 100 Grad)
 - Immer wenn die Bedingung wahr wird (und sich der Zustandsautomat im zugehörigen Ausgangszustand befindet), feuert die Transition, d. h. der Zustandsübergang findet statt
- **Signal**
 - Ein anderes Objekt hat ein Signal an das betroffene Objekt geschickt
 - Ein Signal kann Parameter besitzen
 - Beispiel: rechte Maustaste gedrückt (Mausposition)
 - Die Parameterwerte können dann im Folgezustand verarbeitet werden
- **Botschaft**
 - Ein anderes Objekt hat eine Botschaft geschickt, die dann zu einem Operationsaufruf führt





- Signale und Botschaften können bezüglich ihrer Notation im Zustandsdiagramm nicht unterschieden werden
- Für die Transition spielt eine Unterscheidung auch keine Rolle
- Der globale Kontrollfluss unterscheidet sich jedoch
 - Botschaft: Der Kontrollfluss kehrt nach der Operationsausführung zum rufenden Objekt zurück
 - Signal: Ein Signal wird asynchron gesendet, d. h. die sendende Operation kann direkt fortgesetzt werden
- Verstrichene Zeit (*Elapsed Time Event*)
 - Die Transition feuert, nachdem eine vorgegebene Zeitspanne seit einem definierten Zeitpunkt verstrichen ist
 - Beispiel: Der Zeitpunkt ist oft der Eintrittszeitpunkt in den Ausgangszustand der Transition
→ after (5 sec)
- Eintritt eines **Zeitpunkts**
 - Beispiel: when (Datum = 29.2.2016)



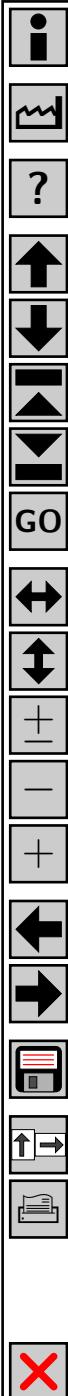


Wächter für ein Ereignis

- Ein Ereignis kann mit einem **Wächter** (*Guard Condition*) kombiniert werden
- Bei Eintritt des Ereignisses wird diese Bedingung ausgewertet
 - Ist die Bedingung erfüllt, feuert die Transition
- Die Transition wird auch als **Guarded Transition** bezeichnet

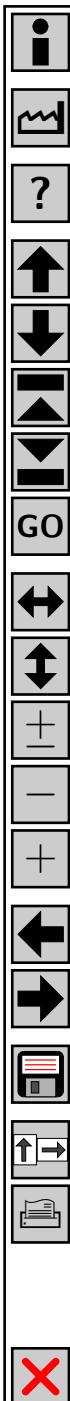


Ein Wächter ist nicht mit einem Ereignis zu verwechseln, das selbst über eine Bedingung definiert wird

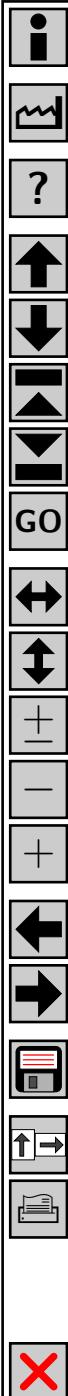


Aktionen und Aktivitäten in einem Zustand

- Innerhalb eines Zustands können ebenfalls Verarbeitungsschritte definiert werden
 - Aktion → **Action**
 - Aktivität → **Activity**
- Aktion
 - Eine Aktion wird immer **vollständig** und **atomar** ausgeführt
 - Sie kann nicht durch eine Transition unterbrochen oder abgebrochen werden
- Aktivität
 - Eine Aktivität modelliert eine **fortdauernde Verarbeitung** → *ongoing activity*
 - Sie kann durch ein Ereignis unterbrochen und beendet werden, das dann (z. B.) eine Transition auslöst
 - Es ist auch möglich, dass eine Aktivität von sich aus aufhört, weil die zugehörige Verarbeitung beendet ist



- Spezifikation der internen Verarbeitung eines Zustands
 - Verwendung einer Liste von Aktionen bzw. Aktivitäten
 - Jedes Listenelement hat die Form
Auslöser / Aktion bzw. Aktivität
 - Statt des Begriffs **Auslöser** wird auch Aktionsmarke (*Action Label*) verwendet
- Vordefinierte Aktionsmarken
 - entry
 - ◊ Die zugehörige **Aktion** wird immer beim **Betreten** des Zustands ausgeführt
 - exit
 - ◊ Die zugehörige **Aktion** wird immer beim **Verlassen** des Zustands ausgeführt
 - do
 - ◊ Es wird eine **Aktivität** gestartet
 - include
 - ◊ Es wird ein **Zustandsautomat** aufgerufen → *Submachine Invocation*



- Ereignisse als Aktionsmarken
 - Notation

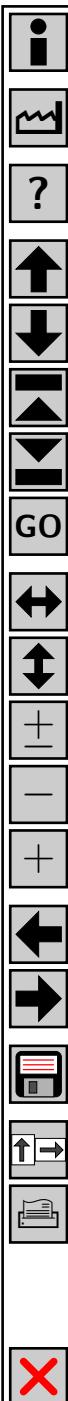
Ereignis (Parameterliste) [Wächter] / Aktion bzw. Aktivität
 - (Parameterliste) und [Wächter] sind optional
 - Das Auslösen eines derartigen Ereignisses hat einen ***internen Zustandsübergang*** zur Folge



- Ein interner Zustandsübergang führt ***nicht*** zu einer Ausführung der entry- bzw. exit-Aktion
- Sind dagegen bei einer Transition Ausgangs- und Folgezustand identisch, so werden bei jedem dieser Übergänge die (eventuell) definierten entry- und exit-Aktionen durchgeführt

Implizite Ereignisse

- Jeder Zustand darf (maximal) eine ausgehende Transition ***ohne explizite Ereignisangabe*** besitzen → ***implizites Ereignis***
- Die Transition wird ausgeführt, wenn die mit dem Zustand verbundene ***Verarbeitung beendet*** ist



Modellierung von Objektlebenszyklen

- Beschreibung des **Lebenszyklus** (*Life Cycle*) eines Objektes über einen Zustandsautomaten
- Alle Objekte der Klasse besitzen denselben Zustandsautomaten
- **Kontinuierlicher Lebenszyklus** (*Circular Lifecycle*)
 - Der Zustandsautomat besitzt **keinen Endzustand** → siehe [Bsp. 42](#)
- **Lebenszyklus mit Endzustand** (*Born-And-Die Lifecycle*)
 - Die Transition in den Endzustand führt zum Löschen des Objekts → siehe [Abb. 61](#)



- **Prinzipiell besitzt jede Klasse einen Lebenszyklus**
- **Oft ist dieser so einfach, dass sich eine Modellierung über einen Zustandsautomaten nicht lohnt**

- **Beispiel 42:** Lebenszyklus einer Klasse **TANK** → siehe Abb. 62

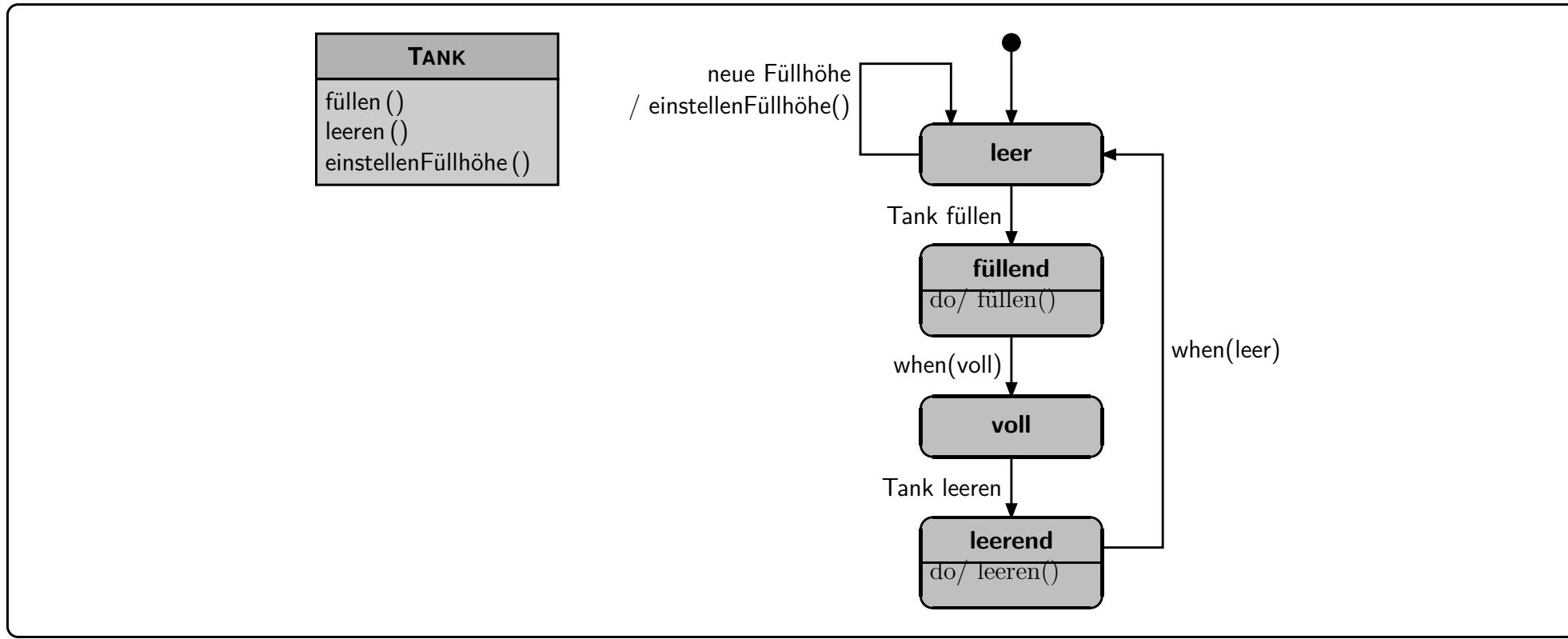
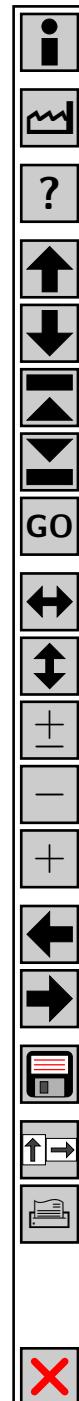
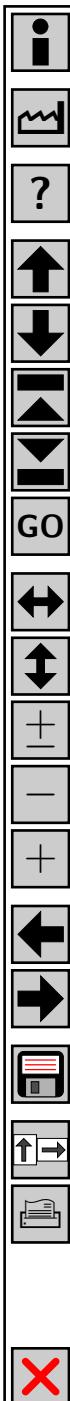


Abb. 62

Lebenszyklus der Klasse **TANK**



Modellierung komplexer Operationen

- Die Wirkungsweise komplexer Operationen lässt sich oft sehr anschaulich über Zustandsautomaten beschreiben
- Anwendungsbeispiel: Das Verhalten der Operation wird durch ***Benutzereingaben gesteuert***
- **Beispiel 43:** bezahlenParkgebühr () → siehe [Abb. 63](#)

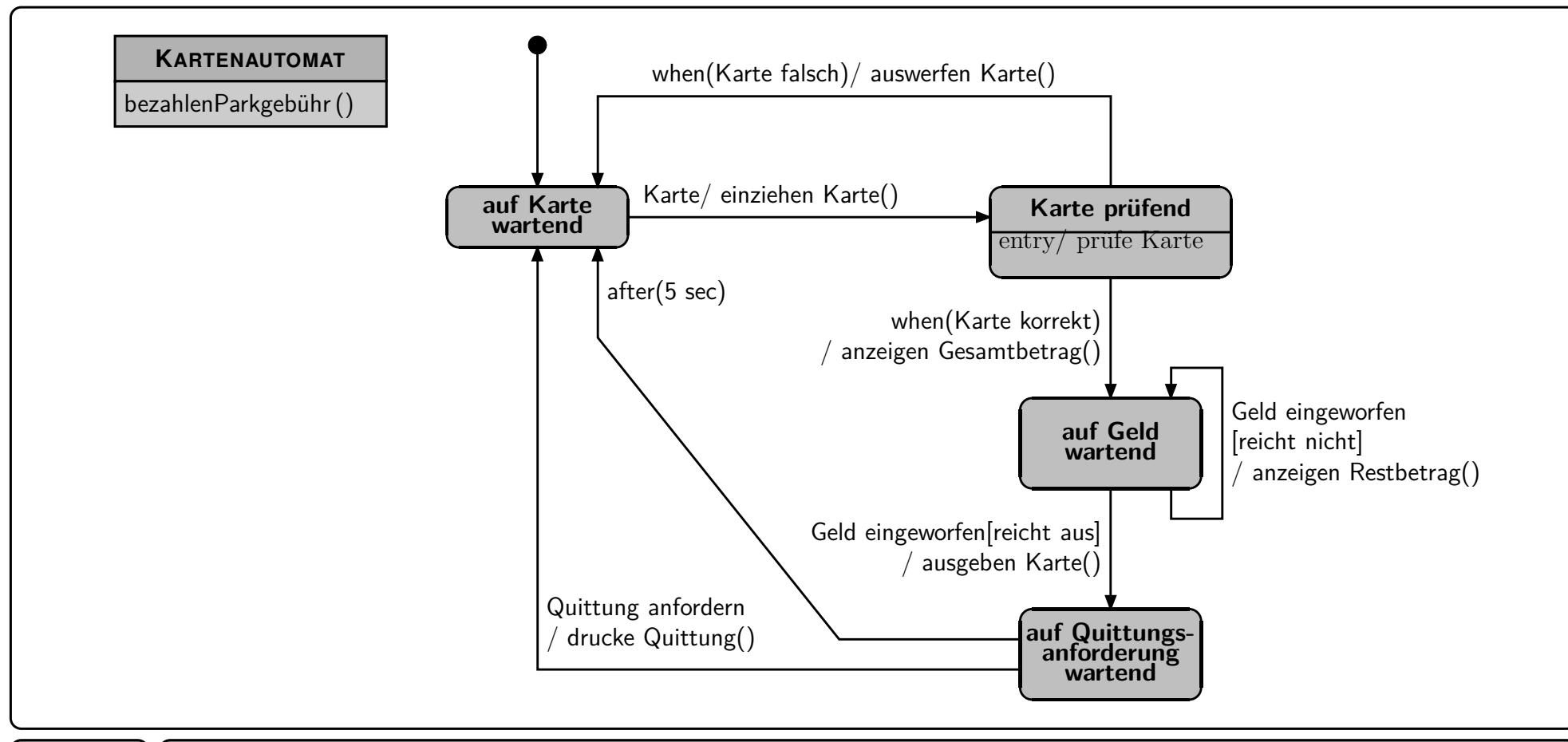
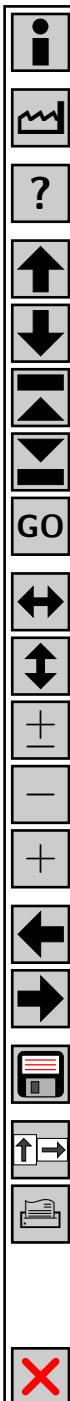
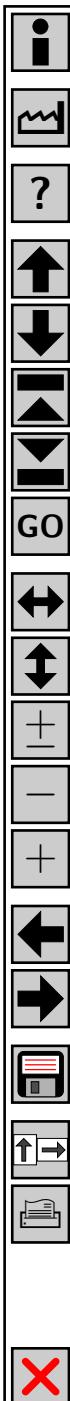


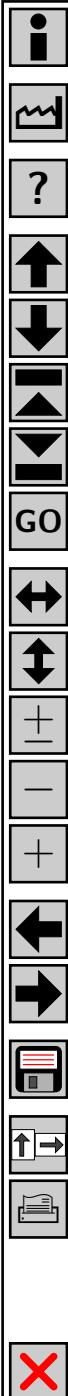
Abb. 63

Die Operation `bezahlenParkgebühr()`

Konsistenzanforderungen

- Modellierung von Objektlebenszyklen
 - Zustandsautomat und Klassendiagramm müssen **konsistent** sein
 - Als **Aktionen** und **Aktivitäten** sind nur Operationen der Klasse erlaubt
- Modellierung von komplexen Operationen
 - Die modellierte Operation ist im Klassendiagramm enthalten
 - **Aktionen** und **Aktivitäten** sind (üblicherweise) **private Operationen** der Klasse
 - Private Operationen werden in der Analysephase (üblicherweise) nicht im Klassendiagramm aufgeführt
 - Die innerhalb der **Entwurfsphase** verwendeten Klassendiagramme enthalten dann auch die privaten Operationen



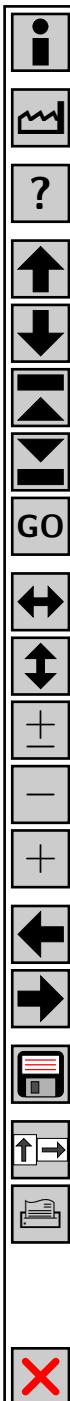


Verfeinerung von Zuständen

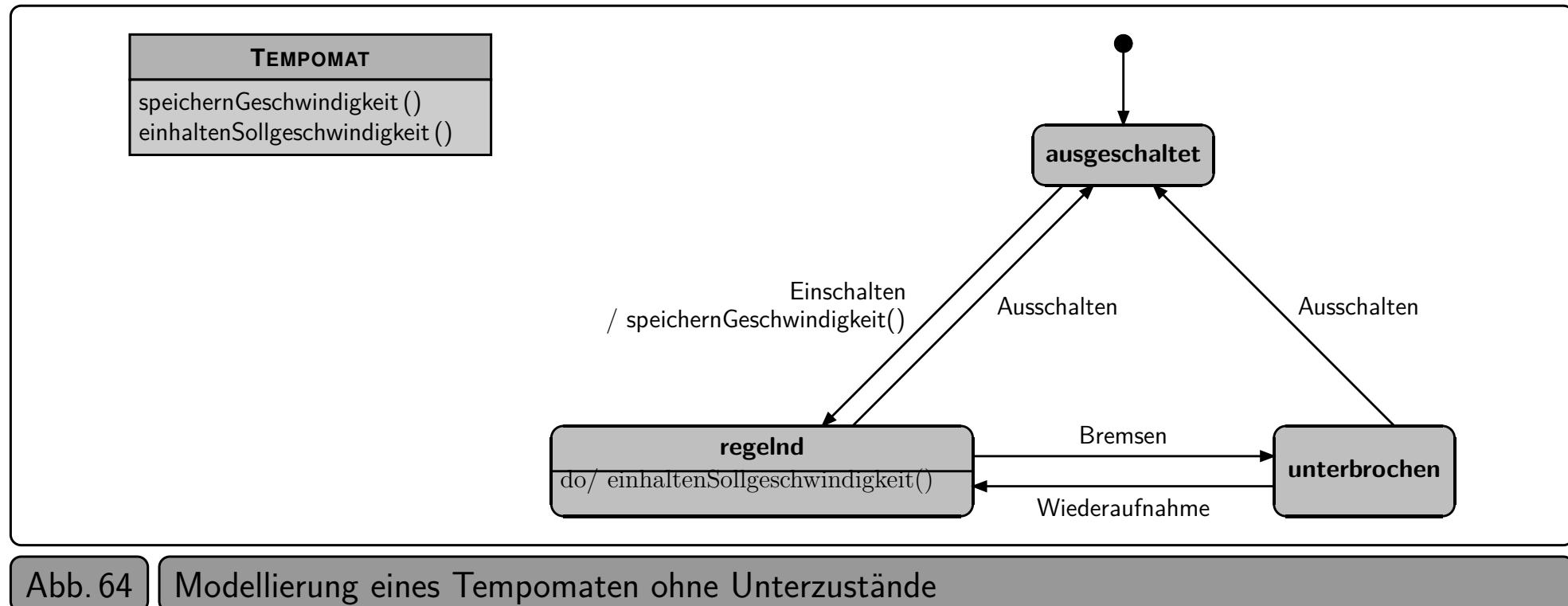
- Ein Zustand kann durch **Unterzüände verfeinert** werden
- Die Unterzüände bilden selbst wieder einen Zustandsautomaten (in der Folge UZ genannt)
- Eine Transition in einen verfeinerten Zustand entspricht der Transition in den **Anfangszustand** von UZ
- UZ kann auch einen **Endzustand** besitzen
 - Ein Übergang in den Endzustand bedeutet, dass die Verarbeitung beendet ist und auf ein Ereignis gewartet wird, das zu einer Transition aus dem verfeinerten Zustand in einen anderen Zustand führt
 - Besitzt der verfeinerte Zustand ein **implizites Ereignis**, so wird beim Übergang in den Endzustand von UZ die dem impliziten Ereignis zugeordnete Transition vorgenommen
- Transition aus einem verfeinerten Zustand, falls UZ keinen Endzustand besitzt
 - Der aktuelle Unterzustand wird verlassen und (falls vorhanden) die entsprechende exit-Aktion des Unterzustands ausgeführt



- **Unterzüände können natürlich auch wieder verfeinert werden**
- **Eine Transition aus einem verfeinerten Zustand führt dann zum Verlassen aller Unterzüände der vorhandenen Verfeinerungsstufen**



- **Beispiel 44:** Modellierung eines Tempomaten
 - Abb. 64 zeigt den Zustandsautomaten ohne Verfeinerung
 - Abb. 65 verwendet das Konzept der Verfeinerung



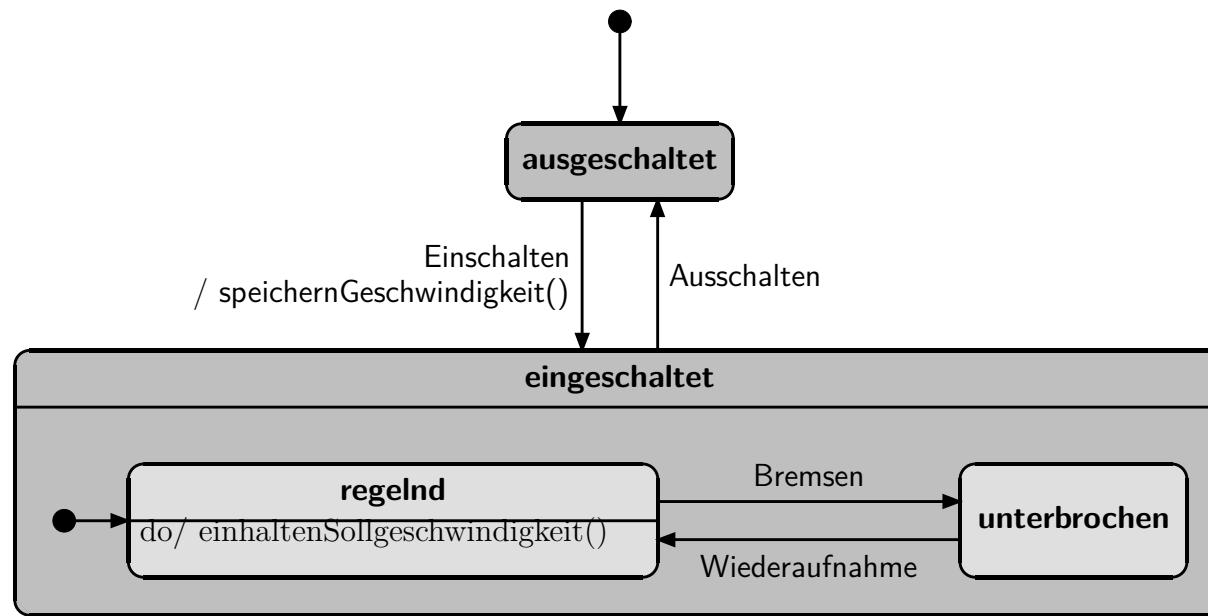
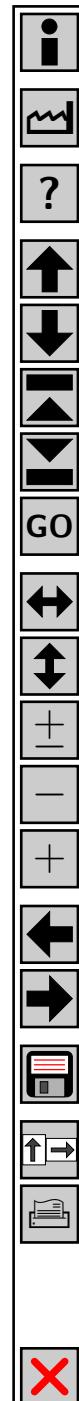
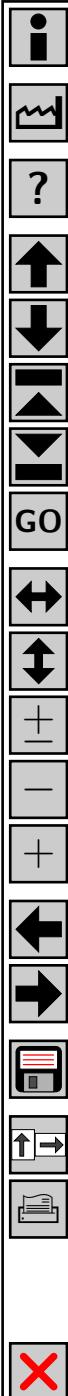


Abb. 65

Modellierung eines Tempomaten mit Unterzuständen

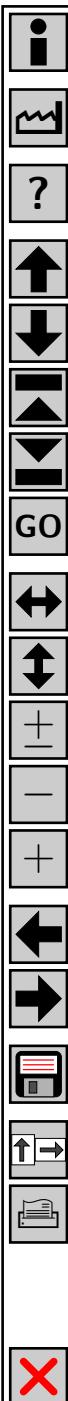


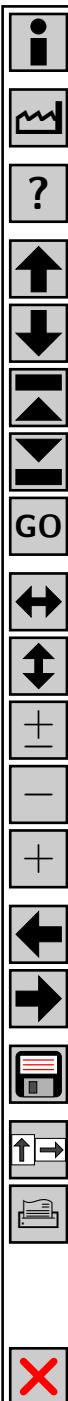
Aktivitätsdiagramm

Eigenschaften

- Das **Aktivitätsdiagramm** (*Activity Diagram*) stellt einen Spezialfall des Zustandsdiagramms dar
 - Alle (oder fast alle) Zustände sind mit einer **Verarbeitung** verbunden → *Action State*
 - Ein Zustand wird verlassen, wenn die zugehörige Verarbeitung beendet ist → Verwendung **impliziter Ereignisse**
 - **Explizite Ereignisse** sollten nicht verwendet werden
 - Transitionen können **Wächter** (*Guard Condition*) zur Steuerung des **Kontrollflusses** besitzen
 - Zustände können **nebenläufig** verarbeitet werden
- ⚠️ • Im Gegensatz zum *herkömmlichen Zustandsdiagramm* beschreibt das Aktivitätsdiagramm *nicht* die Reaktion auf *Ereignisse*
• Stattdessen steht die Spezifikation von *Arbeitsabläufen* im Vordergrund

- Anwendungsgebiete
 - Spezifikation komplexer Operationen
 - Spezifikation von Geschäftsprozessen





UML-Notation für Aktivitätsdiagramme

- Abb. 66 zeigt die Notation für ein Aktivitätsdiagramm

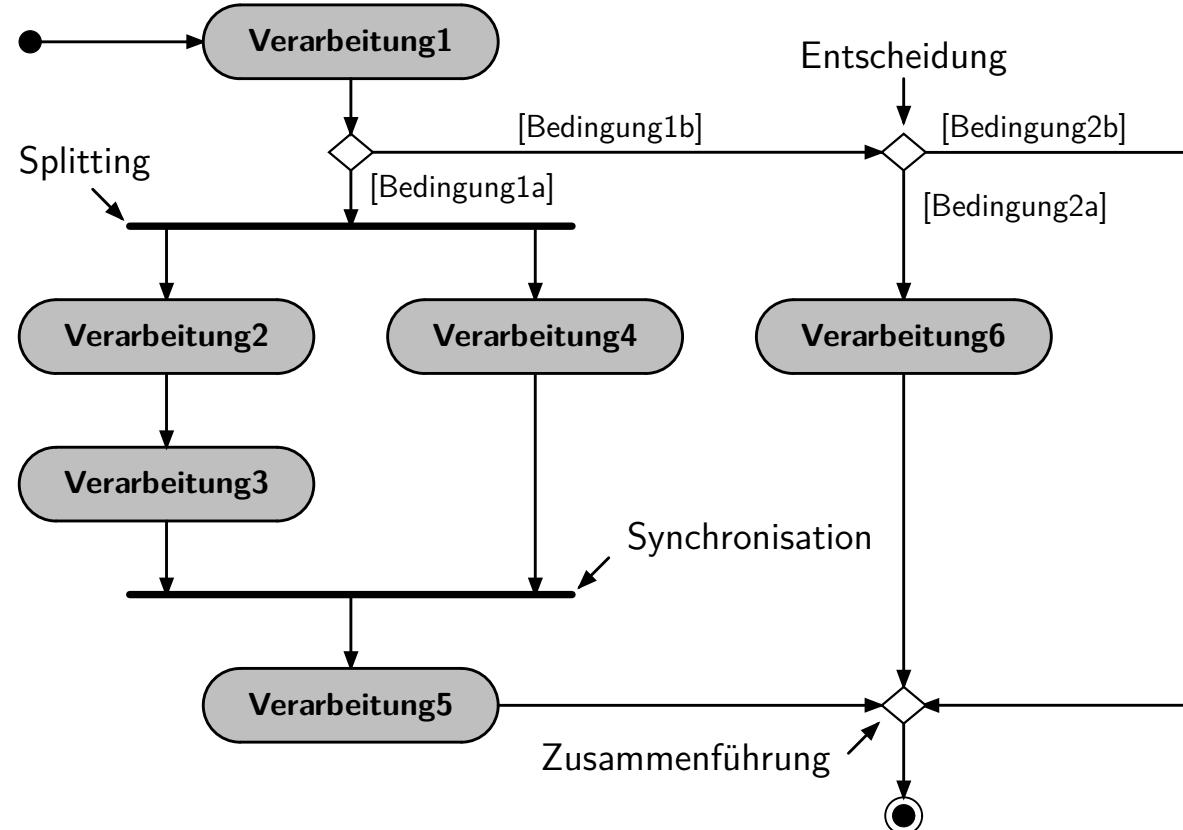
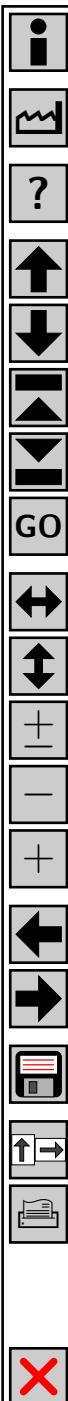


Abb. 66

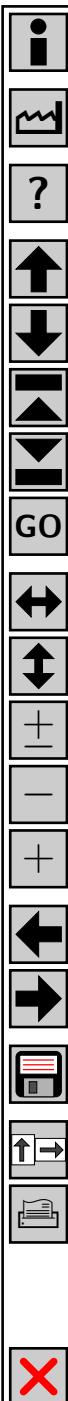
Notation des Aktivitätsdiagramms

-  • Bei Aktivitätsdiagrammen wird ein Zustand durch eine *gerade Ober- und Unterlinie mit konvex geformten Seiten* dargestellt
 - Zustandsdiagramme verwenden dagegen *Rechtecke mit abgerundeten Ecken*
-
- Splitting
 - Der Zustand **Verarbeitung4** kann nebenläufig zu den sequentiell auszuführenden Zuständen **Verarbeitung2** und **Verarbeitung3** ausgeführt werden
 - Synchronisation
 - Erst wenn alle Verarbeitungsschritte des nebenläufigen Diagrammteils (hier **Verarbeitung2** bis **Verarbeitung4**) vollständig ausgeführt sind, kann mit dem Zustand **Verarbeitung5** fortgefahrene werden
-
-  • Nebenläufige Verarbeitungsschritte dürfen natürlich auch sequentiell ausgeführt werden
 - Die Reihenfolge ist in diesem Fall beliebig



Beispiel: Spezifikation eines Geschäftsprozesses

- Abb. 67 zeigt den Standardfall des Geschäftsprozesses Auftrag ausführen als Aktivitätsdiagramm → vgl. Bsp. 31



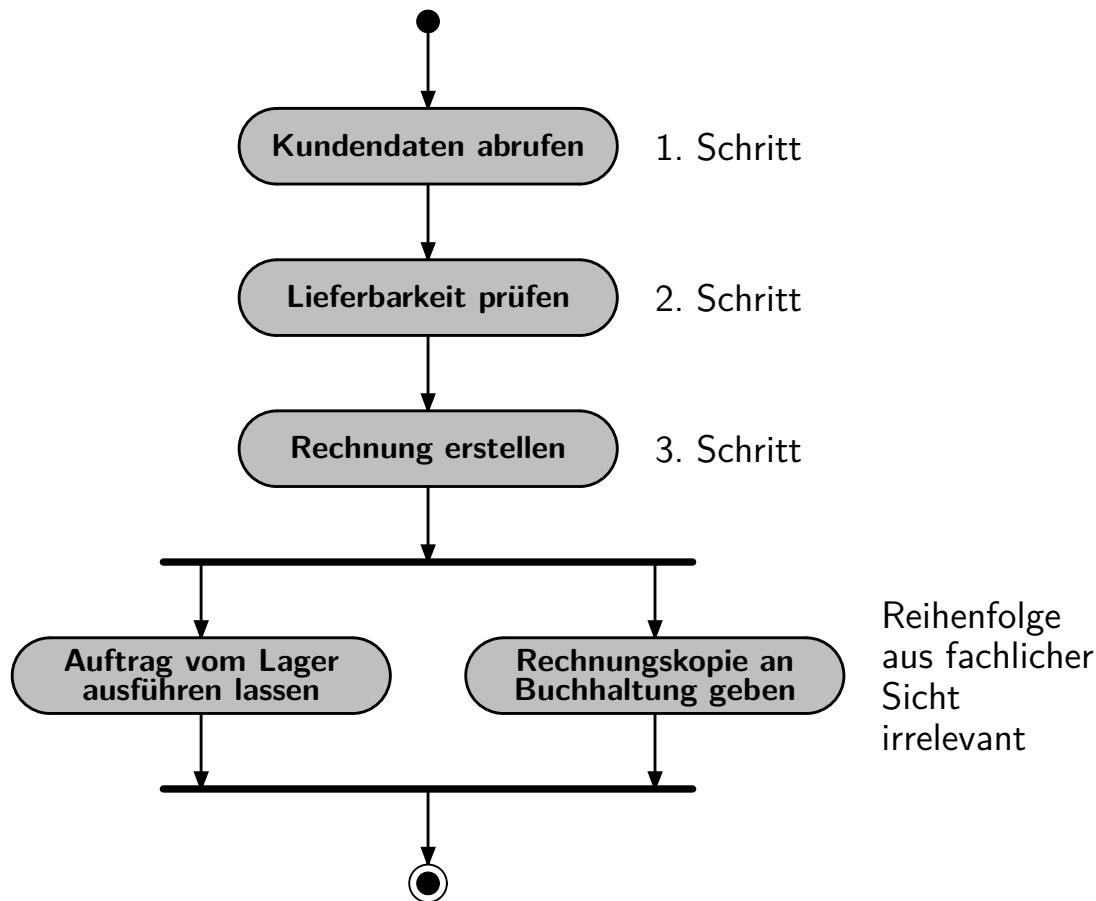
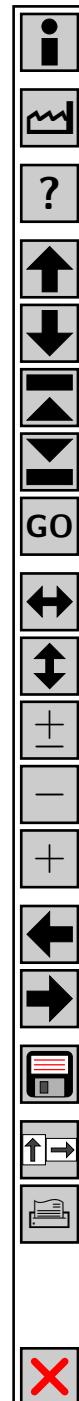
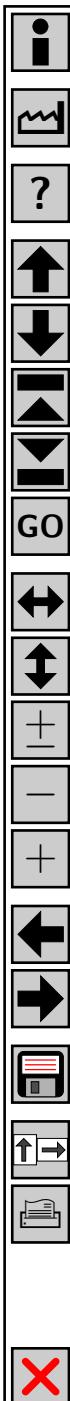


Abb. 67

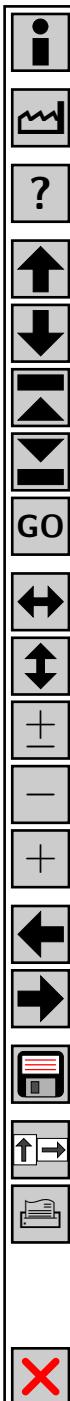
Der Geschäftsprozess Auftrag ausführen als Aktivitätsdiagramm

Zusammenfassung

- **Geschäftsprozesse** beschreiben die Arbeitsabläufe der **Akteure** mit dem System auf einer sehr hohen Abstraktionsebene
- **Geschäftsprozessdiagramme** dienen zur Dokumentation von Geschäftsprozessen
- Die Spezifikation eines Geschäftsprozesses kann *umgangssprachlich*, über eine **Geschäftsprozessschablone** oder durch **Aktivitätsdiagramme** erfolgen
- Eine **Botschaft** ist die Aufforderung eines Senders an einen Empfänger, eine bestimmte Dienstleistung zu erbringen
- Ein Geschäftsprozess wird durch mehrere **Szenarios** verfeinert
- Szenarios können in Form von **Sequenz-** oder **Kollaborationsdiagrammen** dokumentiert werden
- **Zustandsdiagramme** werden für die Beschreibung von **Objektlebenszyklen** und **komplexen Operationen** verwendet



- Ein **Aktivitätsdiagramm** stellt einen Sonderfall des **Zustandsdiagramms** dar
- Aktivitätsdiagramme dienen zur Beschreibung von **Geschäftsprozessen** und **Operationen**



Glossar

Akteur (actor): Ein Akteur ist eine Rolle, die ein Benutzer des Systems spielt. Akteure befinden sich außerhalb des Systems. Akteure können Personen oder externe Systeme sein.

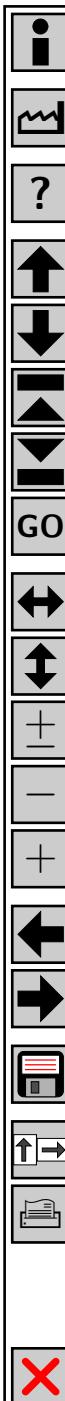
Aktion (action): Eine Aktion ist eine atomare Operation, die durch ein Ereignis ausgelöst wird und sich selbst beendet. Sie kann mit einer → *Transition* verbunden sein. Eine *entry*-Aktion wird bei Eintritt und eine *exit*-Aktionen bei Verlassen des → *Zustands* ausgeführt.

Aktivität (activity): Eine Aktivität ist eine Operation, die mit einem → *Zustand* eines → *Zustandsautomaten* verbunden ist. Sie beginnt bei Eintritt und endet bei Verlassen des Zustands. Sie kann alternativ durch ein Paar von Aktionen, eine zum Starten und eine zum Beenden der Aktivität, beschrieben oder durch ein weiteres → *Zustandsdiagramm* verfeinert werden.

Aktivitätsdiagramm (activity diagram): Ein Aktivitätsdiagramm ist ein Sonderfall eines → *Zustandsdiagramms*, bei dem (fast) alle Zustände mit einer Verarbeitung verbunden sind. Ein → *Zustand* wird verlassen, wenn die Verarbeitung beendet ist. Außerdem ist es möglich, eine Verzweigung des Kontrollflusses zu spezifizieren und zu beschreiben, ob die Verarbeitungsschritte in festgelegter oder beliebiger Reihenfolge ausgeführt werden können.

Botschaft (message): Eine Botschaft ist eine Aufforderung eines Senders (*client*) an einen Empfänger (*server*, *supplier*), eine Dienstleistung zu erbringen. Der Empfänger interpretiert diese Botschaft und führt eine Operation aus.

Ereignis (event): Ein Ereignis tritt immer zu einem Zeitpunkt auf und besitzt keine Dauer. Es kann sein: eine wahr werdende Bedingung, ein Signal, eine Botschaft (Aufruf einer Operation), eine verstrichene Zeitspanne oder das Eintreten eines bestimmten Zeitpunkts. In den beiden letzten Fällen spricht man von zeitlichen Ereignissen.





Geschäftsprozess (*use case*): Ein Geschäftsprozess besteht aus mehreren zusammenhängenden Aufgaben, die von einem → Akteur durchgeführt werden, um ein Ziel zu erreichen bzw. ein gewünschtes Ergebnis zu erstellen.

Geschäftsprozessdiagramm (*use case diagram*): Ein Geschäftsprozessdiagramm beschreibt die Beziehungen zwischen → Akteuren und → Geschäftsprozessen in einem System. Auch Beziehungen zwischen Geschäftsprozessen (*extends* und *uses*) können eingetragen werden. Es gibt auf einem hohen Abstraktionsniveau einen guten Überblick über das System und seine Schnittstellen zur Umgebung.

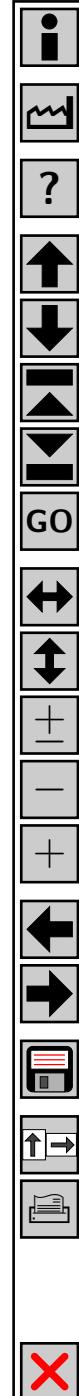
Geschäftsprozessschablone (*use case template*): Die Geschäftsprozessschablone ermöglicht eine semiformale Spezifikation von → Geschäftsprozessen. Sie enthält folgende Informationen: Name, Ziel, Kategorie, Vorbedingung, Nachbedingung Erfolg, Nachbedingung Fehlschlag, Akteure, auslösendes Ereignis, Beschreibung des Standardfalls sowie Erweiterungen und Alternativen zum Standardfall.

Interaktionsdiagramm (*interaction diagram*): In der UML ist Interaktionsdiagramm der Oberbegriff von → Sequenz- und → Kommunikationsdiagramm. Bei anderen Methoden wird (oft) der Begriff Interaktionsdiagramm für das Sequenzdiagramm verwendet.

Kommunikationsdiagramm (*communication diagram*): Ein Kommunikationsdiagramm beschreibt die Objekte und die Verbindungen zwischen diesen Objekten. An jede Verbindung (*link*) kann eine → Botschaft in Form eines Pfeiles angetragen werden. Die Reihenfolge und Verschachtelung der Operationen wird durch eine hierarchische Numerierung angegeben.

Nachbedingung (*postcondition*): Die Nachbedingung beschreibt die Änderung, die durch eine Verarbeitung bewirkt wird, unter der Voraussetzung, dass vor ihrer Ausführung die → Vorbedingung erfüllt war.

Nachricht (*message*): → Botschaft



Protokoll (protocol): Die Menge der → *Botschaften*, auf die Objekte einer Klasse reagieren, wird als Protokoll der Klasse bezeichnet.

Sequenzdiagramm (sequence diagram): Ein Sequenzdiagramm besitzt zwei Dimensionen. Die vertikale repräsentiert die Zeit, auf der horizontalen werden die Objekte angetragen. In das Diagramm werden die → *Botschaften* eingetragen, die zum Aktivieren der Operationen dienen.

Szenario (scenario): Ein Szenario ist eine Sequenz von Verarbeitungsschritten, die unter bestimmten Bedingungen auszuführen sind. Diese Schritte sollen das Hauptziel des → *Akteurs* realisieren und ein entsprechendes Ergebnis liefern. Ein → *Geschäftsprozess* wird durch eine Kollektion von Szenarios dokumentiert.

Transition (transition): Eine Transition (*Zustandsübergang*) verbindet einen Ausgangs- und einen Folgezustand. Sie kann nicht unterbrochen werden und wird stets durch ein → *Ereignis* ausgelöst. Ausgangs- und Folgezustand können identisch sein.

Vorbedingung (precondition): Die Vorbedingung beschreibt, welche Bedingungen vor dem Ausführen einer Verarbeitung erfüllt sein müssen, damit die Verarbeitung definiert ausgeführt werden kann.

Zustand (state): Ein Zustand eines → *Zustandsautomaten* ist eine Zeitspanne, in der ein Objekt auf ein Ereignis wartet. Ein Zustand besteht so lange, bis ein → *Ereignis* eintritt, das eine → *Transition* auslöst.

Zustandsautomat (finite state machine): Ein Zustandsautomat besteht aus → *Zuständen* und → *Transitionen*. Er hat einen Anfangszustand und kann einen Endzustand besitzen.

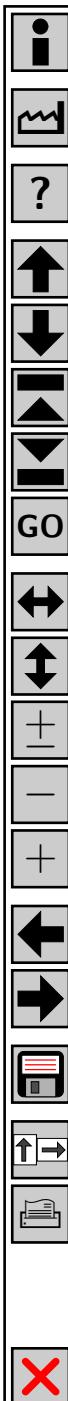
Zustandsdiagramm (statechart diagram): Das Zustandsdiagramm ist eine grafische Repräsentation des → *Zustandsautomaten*.

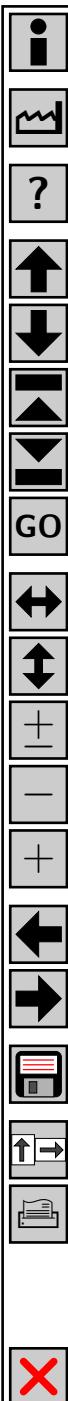
ANALYSESCHRITTE FÜR GESCHÄFTSPROZESSE

Lernziele

Anwendung

- Systematische Identifikation von Geschäftsprozessen
- Dokumentation von Geschäftsprozessen



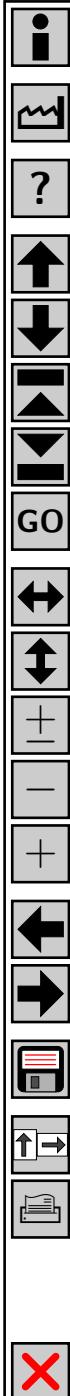


Identifikation von Geschäftsprozessen

Konstruktive Schritte

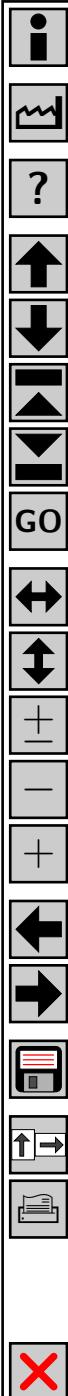
Arbeitsschritte im Überblick

- Akteure ermitteln
- Geschäftsprozesse für die Standardverarbeitung ermitteln
- Geschäftsprozesse für die Sonderfälle formulieren
- Aufsplitten komplexer Geschäftsprozesse
- Gemeinsamkeiten von Geschäftsprozessen ermitteln



Akteure ermitteln

- Akteure befinden sich immer **außerhalb** des betrachteten Systems
- Akteure kommunizieren mit den Geschäftsprozessen des Systems
- **Beispiel 45:** Wer ist der Akteur?
 - Betrachtet wird der Geschäftsprozess **kaufen einer Fahrkarte**
 - ◊ Die Fahrkarte wird am **Schalter** erworben ⇒ der betreffende **Sachbearbeiter** ist der Akteur
 - ◊ Eine Fahrkarte wird an einem **Fahrkartenautomaten** gekauft ⇒ der **Fahrgast** ist der Akteur
 - Ergebnis: Hier lässt sich der Akteur nur dann ermitteln, wenn Informationen über den Einsatz des Gesamtsystems bekannt sind
- Gesichtspunkte bei der Identifikation von Akteuren
 - Welche Personen führen eine Aufgabe zur Zeit durch?
 - ◊ Diese Personen besitzen (üblicherweise) wichtige Kenntnisse über die durchzuführenden Arbeitsabläufe
 - Welche Personen werden zukünftig diese Aufgaben durchführen?
 - Wo befindet sich die Schnittstelle des betrachteten Systems bzw. was gehört nicht mehr zu dem System?



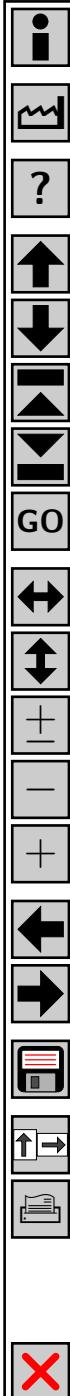
- Als Akteure kommen in Frage
 - Personen
 - Externe Systeme, die Ereignisse auslösen, die zum Start eines Geschäftsprozesses führen
 - eine Organisationseinheit



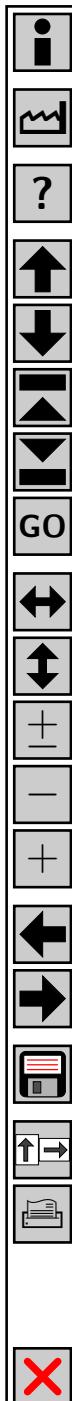
Da ein Akteur eine *Rolle* modelliert, die ein Systembenutzer spielt, kann (z. B.) eine Person mehrfach (d. h. in unterschiedlichen Rollen) als Akteur auftreten

Geschäftsprozesse für die Standardverarbeitung ermitteln

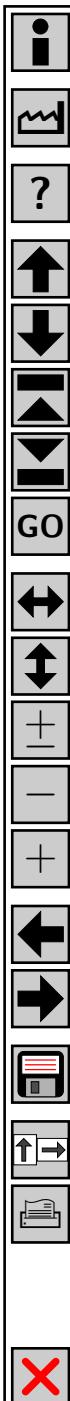
- Mögliche Ansatzpunkte
 - Analyse der typischen Arbeitsabläufe von Personen, die als **Akteure** auftreten
 - Analyse von Ereignissen, die Geschäftsprozesse auslösen
 - Analyse der notwendigen Aufgaben, die das Gesamtsystem erfüllen soll



- Analyse von Arbeitsabläufen (JACOBSON in [JCJÖ92])
 - Typische Fragen für Interviews
 - ◊ Welches Ereignis löst den Arbeitsablauf aus?
 - ◊ Welche Eingabedaten werden benötigt?
 - ◊ Welche Arbeitsschritte sind auszuführen?
 - ◊ Ist eine Reihenfolge der Arbeitsschritte festgelegt?
 - ◊ Welche Zwischenergebnisse werden erstellt?
 - ◊ Welche Endergebnisse werden erstellt?
 - ◊ Welche Vorbedingungen müssen erfüllt sein?
 - ◊ Welche Nachbedingungen (Vorbedingungen anderer Geschäftsprozesse) werden sichergestellt?
 - ◊ Wie wichtig ist diese Arbeit?
 - ◊ Warum wird diese Arbeit durchgeführt?
 - ◊ Kann die Durchführung verbessert werden?
- **Beispiel 46:** Seminarorganisation
 - Innerhalb einer Seminarorganisation ist der Akteur der Kundensachbearbeiter
 - Seine Aufgabe ist es, **Anmeldungen** von Kunden entgegenzunehmen und zu **bearbeiten**
 - Daraus lässt sich der Geschäftsprozess **bearbeite Anmeldung** ableiten



- Analyse von Ereignissen
 - Bei den Akteuren handelt es sich um **technische Schnittstellen** oder **organisatorische Einheiten**
 - Erstellung einer **Ereignisliste**
 - ◊ Welche Ereignisse aus der Umgebung des Systems sind relevant?
 - ◊ Für jedes dieser Ereignisse muss dann ein Geschäftsprozess existieren
 - ▷ Der Geschäftsprozess wird explizit durch das Ereignis angestoßen
 - ▷ Alternativ kann es Aufgabe des Geschäftsprozesses sein, das Ereignis zu entdecken und es dann zu verarbeiten
 - Ereignistypen
 - ◊ **Externe Ereignisse** → sie werden außerhalb des betrachteten Systems erzeugt
 - ◊ **Zeitgesteuerte Ereignisse** → sie werden üblicherweise innerhalb des Systems produziert



- **Beispiel 47:** Ereignisliste für eine Seminarorganisation
 - Eine Seminaranmeldung trifft (über das Internet) ein
→ **Geschäftsprozess bearbeite Anmeldung**
 - Dozent sagt (über das Internet) wegen Krankheit ab
→ **Geschäftsprozess suche Ersatz oder storniere Seminar**
 - Seminar durchgeführt (zeitliches Ereignis)
→ **Geschäftsprozess erstelle Rechnung**

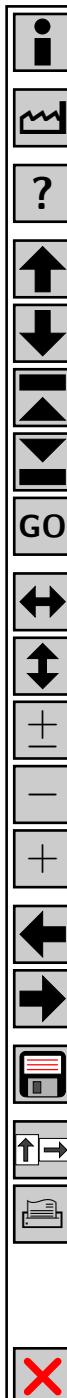


Natürlich kann es auch sein, dass ein *Ereignis eintritt*, welches eine *Person* (d. h. keine technische Schnittstelle oder organisatorische Einheit) veranlasst, einen **Geschäftsprozess zu starten** → vgl. Bsp. 46

- Analyse der notwendigen Aufgaben, die das Gesamtsystem erfüllen soll
 - Annahme: Weder über Akteure noch über Ereignisse lassen sich Geschäftsprozesse identifizieren
 - Vorgehensweise
 - ◊ Ermittlung des Einsatzzwecks bzw. der Ziele des Systems
 - ◊ Ableitung der notwendigen Aufgaben, die zur Erreichung der Ziele auszuführen sind
 - ◊ Welches sind die 10 wichtigsten Aufgaben?
 - ◊ Umgangssprachliche Beschreibung jeder Aufgabe mit 25 (oder weniger) Wörtern
 - ▷ Was ist das Ziel der Aufgabe?
 - ▷ Was ist der Nutzen für das Gesamtsystem, wenn die Aufgabe erfolgreich durchgeführt wurde?

Geschäftsprozesse für die Sonderfälle formulieren

- Modellierung der **Erweiterungen**, die zur Abdeckung von **Sonderfällen notwendig sind**
- Sonderfälle können in der Geschäftsprozessschablone unter **Erweiterungen** bzw. **Alternativen** behandelt werden
- Umfangreiche Sonderfälle sollten als **eigenständiger** Geschäftsprozess spezifiziert und mit «extend» an die Standardverarbeitung angebunden werden



- **Beispiel 48:** Schadensfallbearbeitung bei einer Versicherungsgesellschaft

Geschäftsprozess

Bearbeite Schadensfall

Ziel

Bezahlung des Schadens durch die Versicherung

Kategorie

Primär

Vorbedingung

Keine

Nachbedingung: Erfolg

Schaden ganz oder teilweise bezahlt

Nachbedingung: Fehlschlag

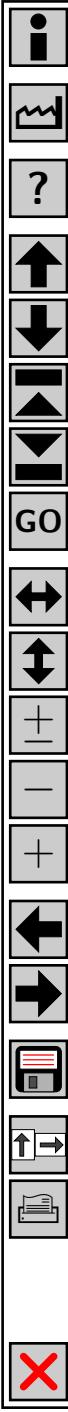
Forderung abgewiesen

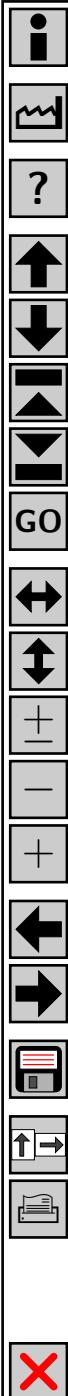
Akteure

Schadenssachbearbeiter

Auslösendes Ereignis

Schadensersatzforderung des Antragstellers, d. h. der versicherten Person





Beschreibung

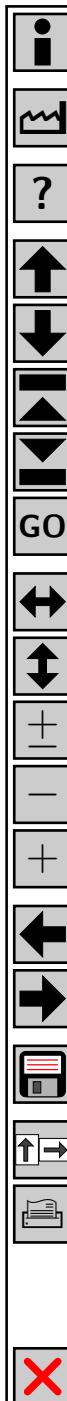
- 1 Prüfung der Forderung auf Vollständigkeit
- 2 Prüfung, ob eine gültige Police vorliegt
- 3 Prüfung aller Details der Police
- 4 Errechnung des Betrags und Überweisung an den Antragsteller

Erweiterungen

- 1a Die vorliegenden Daten vom Antragsteller sind nicht vollständig:
Nachforderung der benötigten Informationen
- 2a Der Antragsteller besitzt keine gültige Police:
Mitteilung an den Antragsteller, dass keine Ansprüche bestehen und Abschluss des Falls
- 4a Der Schaden wird durch die Police nicht abgedeckt:
Mitteilung an den Antragsteller, dass keine Ansprüche bestehen und Abschluss des Falls
- 4b Der Schaden wird durch die Police nur unvollständig abgedeckt:
Verhandlung mit dem Antragsteller, bis zu welchem Grad der Schaden bezahlt wird

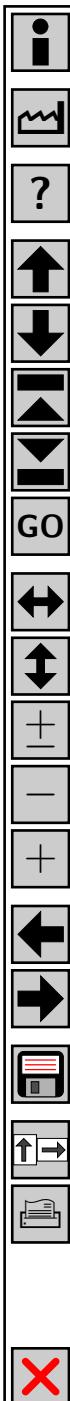
Alternativen

Keine



Aufsplitten komplexer Geschäftsprozesse

- Ein Geschäftsprozess besteht in der Regel aus mehreren Teilaufgaben
 - Sehr komplexe Teilaufgaben sollten als eigenständige Geschäftsprozesse modelliert werden
 - Verwendung der «include»-Beziehung im Geschäftsprozessdiagramm
- In einem Geschäftsprozess treten sehr viele Sonderfälle auf
 - Es sollte geprüft werden, ob das Gesamtverhalten nicht besser durch mehrere Geschäftsprozesse zu beschreiben ist
 - Gemeinsames Verhalten wird dann über die «include»-Beziehung modelliert
 - Umfangreiche Erweiterungen werden ebenfalls in eigenen Geschäftsprozessen modelliert und durch die «extend»-Beziehung mit dem Standard-Geschäftsprozess verbunden

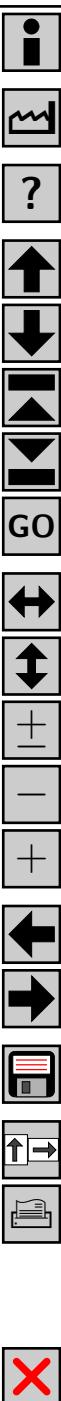


Gemeinsamkeiten von Geschäftsprozessen ermitteln

- Besitzen mehrere Geschäftsprozesse ein gemeinsames Verhalten?
 - Dieses wird als eigenständiger Geschäftsprozess modelliert
 - Verwendung der «include»-Beziehung im Geschäftsprozessdiagramm



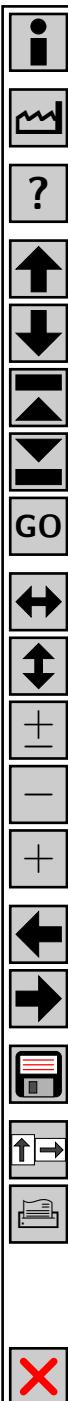
- Die «include»-Beziehung dient in erster Linie einer **redundanzfreien Beschreibung von Geschäftsprozessen**
- Die «include»-Beziehung kann als **Funktionsaufruf** aufgefasst werden
- Geschäftsprozesse sollten nicht zu stark verfeinert werden, so dass eine Art **Funktionsbaum** entsteht



Analytische Schritte zur Validierung von Geschäftsprozessen

Beurteilungskriterien

- Qualität der Beschreibung
- Konsistenz mit dem Klassendiagramm
- Fehlerquellen



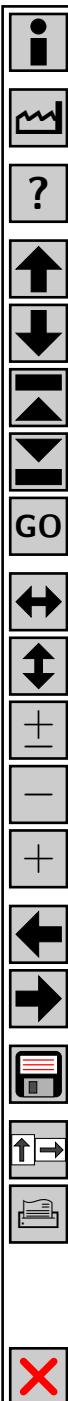
Qualität der Beschreibung

- Liegt eine **gute** Beschreibung der Geschäftsprozesse vor?
 - Der Auftraggeber muss die Beschreibung verstehen können
 - Bei der Beschreibung sollte man sich auf die Kommunikation zwischen den Akteuren und dem System konzentrieren → Beschreibung des **extern wahrnehmbaren** Verhaltens



Es ist nicht das Ziel, die *interne Systemstruktur* oder *Algorithmen* auf der Abstraktionsebene der Geschäftsprozesse zu beschreiben → eine *fachliche* Beschreibung des Arbeitsablaufes ist das Ziel

- Die Standardfälle sollten immer komplett spezifiziert werden
- Die Beschreibung eines Geschäftsprozesses sollte nicht zu lang werden
→ Faustregel: Maximal eine Seite



Konsistenz mit dem Klassendiagramm

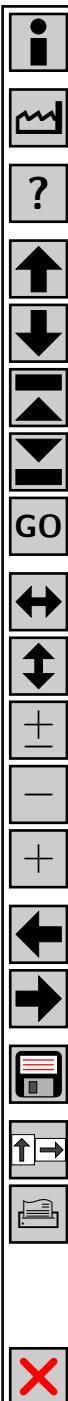
- Annahme: Der Analyseprozess ist soweit fortgeschritten, dass die Klassendiagramme vorliegen
- Konsistenzprüfung
 - Für jeden Geschäftsprozess werden zwei **Objektdiagramme** erstellt und zwar unabhängig von den vorhandenen Klassendiagrammen
 - ◊ Das erste Objektdiagramm **OD1** beschreibt die Objekte und Beziehungen, die vor der Ausführung des Geschäftsprozesses vorhanden sind
 - ◊ Das zweite Objektdiagramm **OD2** stellt die Situation dar, die nach der Ausführung des Geschäftsprozesses vorliegen soll
 - Überprüfung
 - ◊ Können alle Strukturänderungen zwischen **OD1** und **OD2** durch die Ausführung des Geschäftsprozesses erklärt werden?
 - ▷ Welche Arbeitsschritte führen zum Erzeugen bzw. Löschen der Objekte?
 - ▷ Welche Arbeitsschritte führen zum Aufbau bzw. Abbau der Objektbeziehungen?
 - ▷ Wurden vorhandene Restriktionen für Assoziationen durch den Geschäftsprozess berücksichtigt?
 - ◊ Passen Objekt- und Klassendiagramme zusammen?



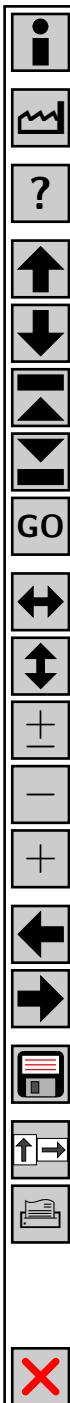
Bei komplexen Geschäftsprozessen kann es natürlich mehrere Varianten von OD2 geben

Fehlerquellen

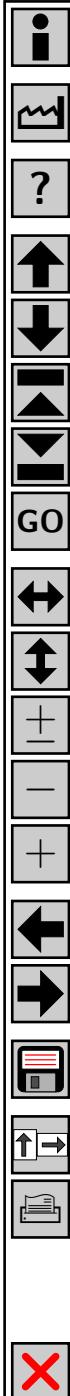
- Sonderfälle werden zu früh betrachtet
- Zu detaillierte Beschreibung der Geschäftsprozesse
- Verwechslung der «include»- mit der «extend»-Beziehung
- Geschäftsprozesse beschreiben (u. a.) Dialogabläufe → dadurch geht die Trennung zwischen Fachkonzept und Benutzeroberfläche verloren



- Zu viele und damit zu kleine Geschäftsprozesse
 - Die Zahl der benötigten Geschäftsprozesse hängt vom **gewählten Abstraktionsniveau** und dem **Anwendungsbereich** ab
 - Schätzwerte von JACOBSON [[Jac95](#)]
 - ◊ Kleineres System (2 bis 5 Mannjahre)
→ 3 bis 50 Geschäftsprozesse (use cases)
 - ◊ Mittleres System (10 bis 100 Mannjahre)
→ 10 bis 60 Geschäftsprozesse
 - ◊ Größere Systeme, z. B. Anwendungen für Banken, Versicherungen, Verteidigung und Telekommunikation
→ Mehrere hundert Geschäftsprozesse
 - BOOCH [[Boo96](#)]
 - ◊ Für ein Projekt mittlerer Komplexität erwartet er etwa ein Dutzend Geschäftsprozesse
 - COCKBURN [[Coc97](#)]
 - ◊ Ein Projekt mit einer Dauer von 50 Mitarbeiterjahren
→ 50 Geschäftsprozesse
 - ◊ Ein Projekt mit einer Dauer von 30 Mitarbeiterjahren (18 Monate Entwicklungsdauer)
→ 200 Geschäftsprozesse



GESAMTGLOSSAR



Abgeleitetes Attribut (*derived attribute*): Abgeleitete Attribute lassen sich aus anderen Attributen berechnen. Sie dürfen nicht direkt geändert werden.

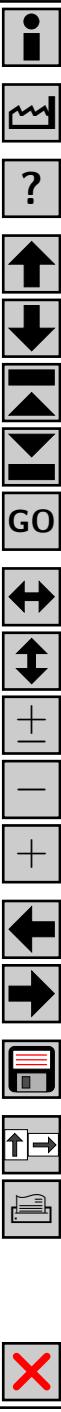
Abstrakte Klasse (*abstract class*): Von einer abstrakten Klasse können keine Objekte erzeugt werden. Die abstrakte Klasse spielt eine wichtige Rolle in Vererbungsstrukturen, wo sie die Gemeinsamkeiten einer Gruppe von → *Unterklassen* definiert. Damit eine abstrakte Klasse verwendet werden kann, muss von ihr zunächst eine Unterklasse abgeleitet werden.

Abstrakter Datentyp (*abstract data type*): Der abstrakte Datentyp (ADT) ist ursprünglich ein Konzept des Entwurfs. Ein abstrakter Datentyp wird ausschließlich über seine (Zugriffs-) Operationen definiert, die auf Exemplare dieses → *Typs* angewendet werden. Die Repräsentation der Daten und die Wahl der Algorithmen zur Realisierung der → *Operationen* sind nach außen nicht sichtbar, d. h. der ADT realisiert das → *Geheimnisprinzip*. Von einem abstrakten Datentyp können beliebig viele Exemplare erzeugt werden. Die → *Klasse* stellt eine Form des abstrakten Datentyps dar.

Aggregation (*aggregation*): Eine Aggregation ist ein Sonderfall der → *Assoziation*. Sie liegt dann vor, wenn zwischen Objekten der beteiligten Klassen eine Beziehung besteht, die sich als *ist Teil von* oder *besteht aus* beschreiben lässt.

Akteur (*actor*): Ein Akteur ist eine Rolle, die ein Benutzer des Systems spielt. Akteure befinden sich außerhalb des Systems. Akteure können Personen oder externe Systeme sein.

Aktion (*action*): Eine Aktion ist eine atomare Operation, die durch ein Ereignis ausgelöst wird und sich selbst beendet. Sie kann mit einer → *Transition* verbunden sein. Eine *entry*-Aktion wird bei Eintritt und eine *exit*-Aktionen bei Verlassen des → *Zustands* ausgeführt.



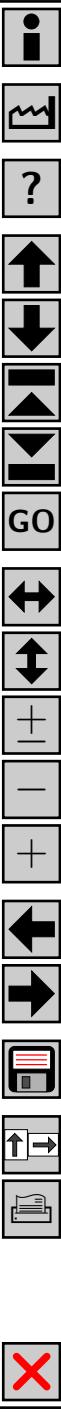
Aktivität (*activity*): Eine Aktivität ist eine Operation, die mit einem → Zustand eines → Zustandsautomaten verbunden ist. Sie beginnt bei Eintritt und endet bei Verlassen des Zustands. Sie kann alternativ durch ein Paar von Aktionen, eine zum Starten und eine zum Beenden der Aktivität, beschrieben oder durch ein weiteres → Zustandsdiagramm verfeinert werden.

Aktivitätsdiagramm (*activity diagram*): Ein Aktivitätsdiagramm ist ein Sonderfall eines → Zustandsdiagramms, bei dem (fast) alle Zustände mit einer Verarbeitung verbunden sind. Ein → Zustand wird verlassen, wenn die Verarbeitung beendet ist. Außerdem ist es möglich, eine Verzweigung des Kontrollflusses zu spezifizieren und zu beschreiben, ob die Verarbeitungsschritte in festgelegter oder beliebiger Reihenfolge ausgeführt werden können.

Analyse (*analysis*): Aufgabe der Analyse ist die Ermittlung und Beschreibung der Anforderungen eines Auftraggebers an ein Software-System. Das Ergebnis soll die Anforderungen vollständig, widerspruchsfrei, eindeutig, präzise und verständlich beschreiben.

Analyseprozess: Der Analyseprozess beschreibt die methodische Vorgehensweise zur Erstellung eines objektorientierten AnalysemodeLLs. Er besteht aus einem → Makroprozess, der die grundlegenden Vorgehensschritte vorgibt, und der situations- und anwendungsspezifischen Anwendung von methodischen Regeln.

Assoziation (*association*): Eine Assoziation modelliert Verbindungen zwischen Objekten einer oder mehrerer Klassen. Binäre Assoziationen verbinden zwei Objekte. Eine Assoziation zwischen Objekten einer Klasse heißt reflexiv. Jede Assoziation wird beschrieben durch → Kardinalitäten und einen optionalen Assoziationsnamen oder Rollennamen. Sie kann um Restriktionen ergänzt werden. Besitzt eine Assoziation selbst wieder Attribute und gegebenenfalls Operationen und Assoziationen zu anderen Klassen, dann wird sie zu einer → assoziativen Klasse. Die Qualifikationsangabe (*qualifier*) zerlegt die Menge der Objekte am anderen Ende der Assoziation in Teilmengen. Eine abgeleitete Assoziation liegt vor, wenn die gleichen Abhängigkeiten bereits durch andere Assoziationen beschrieben werden. Sonderfälle der Assoziation sind die → Aggregation und die → Komposition. In der Analyse ist jede Assoziation inhärent bidirektional.



Assoziative Klasse (*association class*): Eine assoziative Klasse besitzt sowohl die Eigenschaften der → *Assoziation* als auch die der → *Klasse*.

Attribut (*attribute*): Attribute beschreiben Daten, die von den → *Objekten* der → *Klasse* angenommen werden können. Alle Objekte einer Klasse besitzen dieselben Attribute, jedoch im Allgemeinen unterschiedliche Attributwerte. Jedes Attribut ist von einem bestimmten → *Typ* und kann einen Anfangswert (*default*) besitzen. Bei der Implementierung muss jedes Objekt Speicherplatz für alle seine Attribute reservieren. Der Attributname ist innerhalb der Klasse eindeutig.

Attributspezifikation (*attribute specification*): Ein → *Attribut* wird durch folgende Angaben spezifiziert:

Name: Typ = Anfangswert

{ mandatory, unique, readOnly, Einheit: ..., Beschreibung: ... }

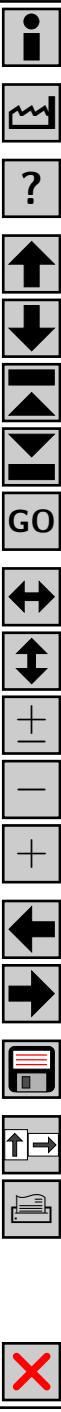
Dabei gilt: mandatory=Muss-Attribut, unique=Schlüsselattribut, readOnly=Attributwert nicht änderbar.

Balancierter Makroprozess: Der balancierte → *Makroprozess* unterstützt die Gleichgewichtigkeit von statischem und dynamischem Modell. Er beginnt mit dem Erstellen von → *Geschäftsprozessen* und der Identifikation von Klassen. Dann werden statisches und dynamisches Modell parallel erstellt und deren Wechselwirkungen berücksichtigt.

Botschaft (*message*): Eine Botschaft ist eine Aufforderung eines Senders (*client*) an einen Empfänger (*server, supplier*), eine Dienstleistung zu erbringen. Der Empfänger interpretiert diese Botschaft und führt eine Operation aus.

Datenbasierter Makroprozess: Beim datenbasierten → *Makroprozess* wird zunächst das Klassendiagramm erstellt und aufbauend darauf werden die → *Geschäftsprozesse* und die anderen Diagramme des dynamischen Modells entwickelt.

Dynamisches Modell: Das dynamische Modell ist der Teil des OOA-Modells, welches das Verhalten des zu entwickelnden Systems beschreibt. Es realisiert außer den Basiskonzepten (Objekt, Klasse, Operation) die dynamischen Konzepte (Geschäftsprozess, Botschaft, Zustandsautomat).



Einfachvererbung: Bei der Einfachvererbung besitzt jede Unterklasse genau eine direkte Oberklasse. Es entsteht eine Baumstruktur.

Elementare Klasse (*support class*): → *Strukturtyp*

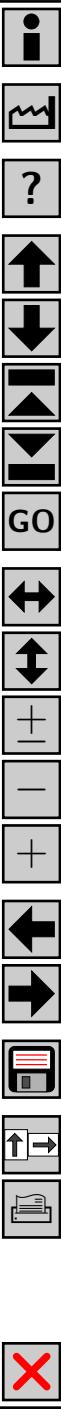
Entwurf (*design*): Aufgabe des Entwurfs ist – aufbauend auf dem Ergebnis der Analyse – die Erstellung der Software-Architektur und die Spezifikation der Komponenten, d. h. die Festlegung von deren Schnittstellen, Funktions- und Leistungsumfang. Das Ergebnis soll die zu realisierenden Programme auf einem höheren Abstraktionsniveau widerspiegeln.

Ereignis (*event*): Ein Ereignis tritt immer zu einem Zeitpunkt auf und besitzt keine Dauer. Es kann sein: eine wahr werdende Bedingung, ein Signal, eine Botschaft (Aufruf einer Operation), eine verstrichene Zeitspanne oder das Eintreten eines bestimmten Zeitpunkts. In den beiden letzten Fällen spricht man von zeitlichen Ereignissen.

Geheimnisprinzip (*information hiding*): Die Einhaltung des Geheimnisprinzips bedeutet, dass die Attribute und die Realisierung der Operationen außerhalb der Klasse nicht sichtbar sind.

Geschäftsprozess (*use case*): Ein Geschäftsprozess besteht aus mehreren zusammenhängenden Aufgaben, die von einem → Akteur durchgeführt werden, um ein Ziel zu erreichen bzw. ein gewünschtes Ergebnis zu erstellen.

Geschäftsprozessdiagramm (*use case diagram*): Ein Geschäftsprozessdiagramm beschreibt die Beziehungen zwischen → Akteuren und → Geschäftsprozessen in einem System. Auch Beziehungen zwischen Geschäftsprozessen (*extends* und *uses*) können eingetragen werden. Es gibt auf einem hohen Abstraktionsniveau einen guten Überblick über das System und seine Schnittstellen zur Umgebung.



Geschäftsprozessschablone (*use case template*): Die Geschäftsprozessschablone ermöglicht eine semiformale Spezifikation von → *Geschäftsprozessen*. Sie enthält folgende Informationen: Name, Ziel, Kategorie, Vorbedingung, Nachbedingung Erfolg, Nachbedingung Fehlschlag, Akteure, auslösendes Ereignis, Bechreibung des Standardfalls sowie Erweiterungen und Alternativen zum Standardfall.

Interaktionsdiagramm (*interaction diagram*): In der UML ist Interaktionsdiagramm der Oberbegriff von → *Sequenz-* und → *Kommunikationsdiagramm*. Bei anderen Methoden wird (oft) der Begriff Interaktionsdiagramm für das Sequenzdiagramm verwendet.

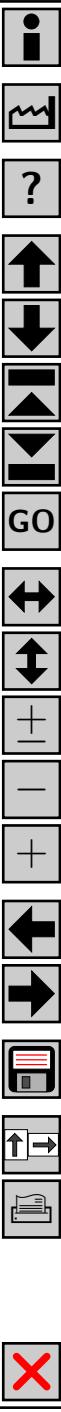
Kardinalität (*multiplicity*): Die Kardinalität bezeichnet die Wertigkeit einer → *Assoziation*, d. h. sie spezifiziert die Anzahl der an der Assoziation beteiligten Objekte.

Klasse (*class*): Eine Klasse definiert für eine Kollektion von → *Objekten* deren Struktur (Attribute), → *Verhalten* (Operationen) und Beziehungen (Assoziationen, Vererbungsstrukturen). Klassen besitzen – mit Ausnahme von abstrakten Klassen – einen Mechanismus, um neue Objekte zu erzeugen. Der Klassename muss mindestens im Paket, besser im gesamten System eindeutig sein.

Klassenattribut (*class scope attribute*): Ein Klassenattribut liegt vor, wenn nur ein Attributwert für alle → *Objekte* der → *Klasse* existiert. Klassenattribute sind von der Existenz der Objekte unabhängig.

Klassendiagramm (*class diagram*): Das Klassendiagramm stellt die Klassen, die → *Vererbung* und die → *Assoziationen* zwischen Klassen dar. Zusätzlich können → *Pakete* modelliert werden.

Klassenoperation (*class scope operation*): Eine Klassenoperation ist eine Operation, die für eine → *Klasse* statt für ein → *Objekt* der Klasse ausgeführt wird.



Kommunikationsdiagramm (*communication diagram*): Ein Kommunikationsdiagramm beschreibt die Objekte und die Verbindungen zwischen diesen Objekten. An jede Verbindung (*link*) kann eine → *Botschaft* in Form eines Pfeiles angetragen werden. Die Reihenfolge und Verschachtelung der Operationen wird durch eine hierarchische Numerierung angegeben.

Komposition (*composition*): Die Komposition ist eine besondere Form der → *Aggregation*. Beim Löschen des Ganzen müssen auch alle Teile gelöscht werden. Jedes Teil kann – zu einem Zeitpunkt – nur zu einem Ganzen gehören. Es kann jedoch anderen Ganzem zugeordnet werden. Die dynamische Semantik des Ganzen gilt auch für seine Teile.

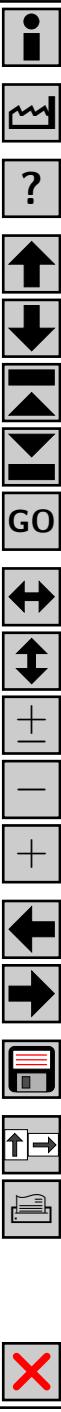
Konzept (*concept*): Der Begriff des Konzepts wird in der Informatik im Sinne von Leitidee verwendet, z. B. Konzepte der Programmierung, Konzepte der Objektorientierung. Ein Konzept beschreibt einen definierten Sachverhalt (z. B. eine Klasse) unter einem oder mehreren Gesichtspunkten.

Makroprozess: Der Makroprozess beschreibt auf einem hohen Abstraktionsniveau die einzelnen Schritte, die zur systematischen Erstellung eines OOA-Modells durchzuführen sind. Der Makroprozess kann die Gleichgewichtigkeit von statischem und dynamischem Modell (→ *balancierter Makroprozess*) unterstützen oder → *datenbasiert* bzw. → *szenariobasiert* sein.

Methode (*method*): Der Begriff Methode beschreibt die planmäßig angewandte, begründete Vorgehensweise zur Erreichung von festgelegten Zielen. In der Software-Technik wird der Begriff Methode als Oberbegriff von → *Konzepten*, → *Notation* und → *methodischer Vorgehensweise* verwendet.

Methodische Vorgehensweise (*method*): Eine methodische Vorgehensweise ist eine planmäßig angewandte, begründete Vorgehensweise zur Erreichung von festgelegten Zielen. Sie wird häufig als → *Methode* bezeichnet.

Nachbedingung (*postcondition*): Die Nachbedingung beschreibt die Änderung, die durch eine Verarbeitung bewirkt wird, unter der Voraussetzung, dass vor ihrer Ausführung die → *Vorbedingung* erfüllt war.



Nachricht (*message*): → *Botschaft*

Notation (*notation*): Darstellung von → *Konzepten* durch eine festgelegte Menge von grafischen und/oder textuellen Symbolen, zu denen eine Syntax und Semantik definiert ist.

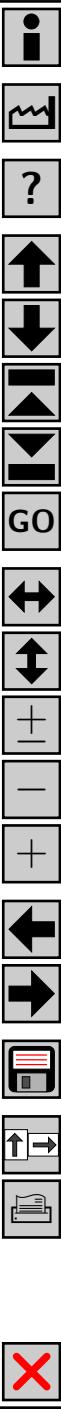
Oberklasse (*super class*): In einer Vererbungsstruktur heißt jede Klasse, von der eine Klasse Eigenschaften und Verhalten erbt, Oberklasse dieser Klasse. Mit anderen Worten: Eine Oberklasse ist eine Klasse, die mindestens eine Unterklasse besitzt.

Objekt (*object*): Ein Objekt besitzt einen → *Zustand* (Attributwerte und Verbindungen zu anderen Objekten), reagiert mit einem definierten → *Verhalten* (Operationen) auf seine Umgebung und besitzt eine → *Objektidentität*, die es von allen anderen Objekten unterscheidet. Jedes Objekt ist Exemplar einer → *Klasse*.

Objektdiagramm (*object diagram*): Das Objektdiagramm stellt → *Objekte* und ihre Verbindungen untereinander dar. Objektdiagramme werden im Allgemeinen verwendet, um einen Ausschnitt des Systems zu einem bestimmten Zeitpunkt zu modellieren. Objekte können einen – im jeweiligen Objektdiagramm – eindeutigen Namen besitzen oder es können anonyme Objekte sein. In verschiedenen Objektdiagrammen kann der gleiche Name unterschiedliche Objekte kennzeichnen.

Objektidentität (*object identity*): Jedes → *Objekt* besitzt eine Identität, die es von allen anderen Objekten unterscheidet. Selbst wenn zwei Objekte zufällig dieselben Attributwerte besitzen, haben sie eine unterschiedliche Identität. Im Speicher wird die Identität durch unterschiedliche Adressen realisiert.

Objektorientierte Analyse (*object oriented analysis*): Ermittlung und Beschreibung der Anforderungen an ein Software-System mittels objektorientierter Konzepte und Notationen. Das Ergebnis ist das OOA-Modell.



Objektorientierter Entwurf (*object oriented design*): Aufbauend auf dem OOA-Modell erfolgt die Erstellung der Software-Architektur und die Spezifikation der Klassen aus der Sicht der Realisierung. Das Ergebnis ist das OOD-Modell, das ein Spiegelbild der objektorientierten Programme auf einem höheren Abstraktionsniveau bildet.

Objektorientierte Software-Entwicklung (*object oriented software development*): Bei einer objektorientierten Software-Entwicklung werden die Ergebnisse der Phasen Analyse, Entwurf und Implementierung objektorientiert erstellt. Für letztere werden objektorientierte Programmiersprachen verwendet. Auch die Verteilung auf einem Netz kann objektorientiert erfolgen.

Objektverwaltung (*class extension, object warehouse*): In der Systemanalyse besitzen Klassen implizit die Eigenschaft der Objektverwaltung. Das bedeutet, dass die Klasse weiß, welche → *Objekte* von ihr erzeugt wurden. Damit erhält die Klasse die Möglichkeit, Anfragen und Manipulationen auf der Menge der Objekte einer → *Klasse* durchzuführen.

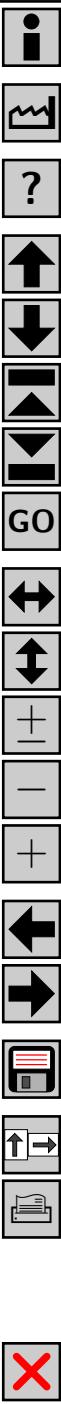
OOA: → *Objektorientierte Analyse*

OOA-Modell: Fachliche Lösung des zu realisierenden Systems, die in einer objektorientierten → *Notation* modelliert wird. Das OOA-Modell besteht aus dem → *statischen* und dem → *dynamischen Modell* und ist das wichtigste Ergebnis der → *Analyse*.

OOD: → *Objektorientierter Entwurf*

OOD-Modell: Technische Lösung des zu realisierenden Systems, die in einer objektorientierten → *Notation* modelliert wird. Das OOD-Modell ist ein Abbild des späteren (objektorientierten) Programms.

Operation (*operation*): Eine Operation ist eine Funktion, die auf interne Daten (Attributwerte) eines → *Objekts* Zugriff hat. Auf alle Objekte einer → *Klasse* sind dieselben Operationen anwendbar. Für Operationen gibt es in der Analyse im Allgemeinen eine fachliche Beschreibung. Abstrakte Operationen besitzen nur einen Operationsrumpf. Externe Operationen werden vom späteren Bediener des Systems aktiviert. Interne Operationen werden dagegen immer von anderen Operationen aufgerufen.



Paket (*package*): Ein Paket fasst Modellelemente (z. B. Klassen) zusammen. Ein Paket kann selbst wieder Pakete enthalten. Es wird benötigt, um die Systemstruktur auf einer hohen Abstraktionsebene auszudrücken. Pakete können im Paketdiagramm dargestellt werden.

Protokoll (*protocol*): Die Menge der → *Botschaften*, auf die Objekte einer Klasse reagieren, wird als Protokoll der Klasse bezeichnet.

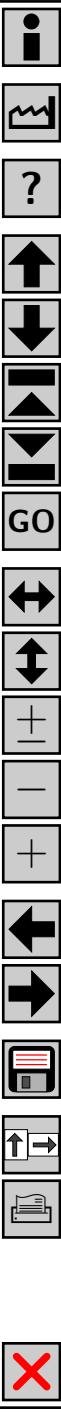
Prototyp (*prototype*): Der Prototyp dient dazu, bestimmte Aspekte vor der Realisierung des Software-Systems zu überprüfen. Der Prototyp der Benutzeroberfläche zeigt die vollständige Oberfläche des zukünftigen Systems, ohne dass bereits Funktionalität realisiert wird.

Qualifikationsangabe (*qualifier*): Die Qualifikationsangabe ist ein spezielles Attribut der → *Assoziation*, dessen Wert ein oder mehrere Objekte auf der anderen Seite der Assoziation selektiert. Mit anderen Worten: Die Qualifikationsangabe zerlegt die Menge der Objekte am anderer Ende der Assoziation in Teilmengen. Der *qualifier* kann auch aus mehreren Attributen bestehen.

Rolle (*role name*): Die Rolle beschreibt, welche Bedeutung ein Objekt in einer → *Assoziation* wahrnimmt. Eine binäre Assoziation besitzt maximal zwei Rollen.

Sequenzdiagramm (*sequence diagram*): Ein Sequenzdiagramm besitzt zwei Dimensionen. Die vertikale repräsentiert die Zeit, auf der horizontalen werden die Objekte angetragen. In das Diagramm werden die → *Botschaften* eingetragen, die zum Aktivieren der Operationen dienen.

Statisches Modell: Das statische Modell realisiert außer den Basiskonzepten (Objekt, Klasse, Attribut) die statischen Konzepte (Assoziation, Vererbung, Paket). Es beschreibt die Klassen des Systems, die Assoziationen zwischen den Klassen und die Vererbungsstrukturen. Außerdem enthält es die Daten des Systems (Attribute). Die Pakete dienen dazu, Teilsysteme zu bilden, um bei großen Systemen einen besseren Überblick zu ermöglichen.



Systemanalyse: → Analyse

Szenario (scenario): Ein Szenario ist eine Sequenz von Verarbeitungsschritten, die unter bestimmten Bedingungen auszuführen sind. Diese Schritte sollen das Hauptziel des → Akteurs realisieren und ein entsprechendes Ergebnis liefern. Ein → Geschäftsprozess wird durch eine Kollektion von Szenarios dokumentiert.

Szenariobasierter Makroprozess: Der szenariobasierte → Makroprozess beginnt mit dem Erstellen von → Geschäftsprozessen und → Interaktionsdiagrammen und leitet daraus das Klassendiagramm ab.

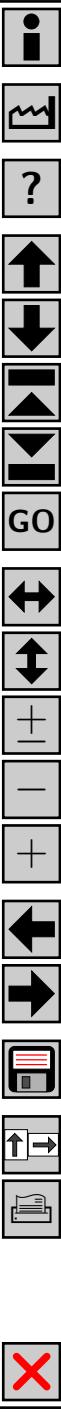
Transition (transition): Eine Transition (*Zustandsübergang*) verbindet einen Ausgangs- und einen Folgezustand. Sie kann nicht unterbrochen werden und wird stets durch ein → Ereignis ausgelöst. Ausgangs- und Folgezustand können identisch sein.

Typ (type): Jedes → Attribut ist von einem bestimmten Typ. Er kann ein Standardtyp (z. B. Int), ein Aufzählungstyp, eine → Strukturtyp oder eine Liste (list of Typ) sein.

Der Typbegriff wird auch im Sinne von Klassenspezifikationen verwendet. Er legt fest, auf welche Operationsaufrufe die → Objekte einer → Klasse reagieren können, d. h. der Typ definiert die Schnittstelle der Objekte. Ein Typ wird implementiert durch eine oder mehrere Klassen.

UML: *Unified Modeling Language*, die von BOOCHE, RUMBAUGH und JACOBSON bei der *Rational Software Corporation* entwickelt und 1997 von der OMG (Object Management Group) als Standard akzeptiert wurde.

Unterklasse (sub class): Jede Klasse, die in einer Vererbungshierarchie Eigenschaften und Verhalten von anderen Klassen erbt, ist eine Unterklasse dieser Klasse. Mit anderen Worten: Eine Unterklasse besitzt immer (mindestens) eine direkte Oberklasse.



Vererbung (inheritance): Die Vererbung beschreibt die Beziehung zwischen einer allgemeineren Klasse (Basisklasse) und einer spezialisierten Klasse. Die spezialisierte Klasse erweitert die Liste der Attribute, Operationen und → Assoziationen der Basisklasse. Operationen der Basisklasse dürfen redefiniert werden. Es entsteht eine Klassenhierarchie.

Verhalten (behavior): Unter dem Verhalten eines → Objekts sind die beobachtbaren Effekte aller → Operationen zu verstehen, die auf das Objekt angewendet werden können. Das Verhalten einer → Klasse wird bestimmt durch die Operationsaufrufe, auf die diese Klasse bzw. deren Objekte reagieren.

Vorbedingung (precondition): Die Vorbedingung beschreibt, welche Bedingungen vor dem Ausführen einer Verarbeitung erfüllt sein müssen, damit die Verarbeitung definiert ausgeführt werden kann.

Zustand (state): Der Zustand eines → Objekts wird bestimmt durch seine Attributwerte und seine Verbindungen (*link*) zu anderen Objekten, die zu einem bestimmten Zeitpunkt existieren.

Zustand (state): Ein Zustand eines → Zustandsautomaten ist eine Zeitspanne, in der ein Objekt auf ein Ereignis wartet. Ein Zustand besteht so lange, bis ein → Ereignis eintritt, das eine → Transition auslöst.

Zustandsautomat (finite state machine): Ein Zustandsautomat besteht aus → Zuständen und → Transitionen. Er hat einen Anfangszustand und kann einen Endzustand besitzen.

Zustandsdiagramm (statechart diagram): Das Zustandsdiagramm ist eine grafische Repräsentation des → Zustandsautomaten.