Inhaltsverzeichnis

Seite

1. DEFINITION VON SYSTEMANALYSE 2

1.1 EXKURS IN DIE SYSTEMTHEORIE 3

1.1.1 Informationssysteme 4

1.1.2 Probleme im Software – Entwicklungsprozess 4

1.1.3 Entwicklungsphasen 5

1.2 KONZEPT OBJEKTORIENTIERTES PARADIGMA 6

1.2.1 Objekt 6

1.2.2. Kapselung: 6

1.2.3 Attribute 7

1.2.4. Klassen 8

1.2.5. Methoden 9

1.2.6. Polymorphismus: „viele Erscheinungsarten“ 11

1.2.7. Vererbung 12

1.2.8 Klassenbeziehungen 15

1.2.9 Botschaften 18

1.3 UNIFIED MODELING LANGUAGE 20

1.3.0 Gemeinsamkeiten der oo - Analysemodelle 20

1.3.1 Vorgänger der UML 22

1.3.2. Unified Modeling Language (UML) 27

1.3.3. UML-Notastion der Klassen 30

1. Definition von Systemanalyse

Zweck:
- Verstehen des Systems
- Übersicht
- Fehleranalyse
- Modell zur Umsetzung

Brockhaus: „Methode zur Untersuchung und Gestaltung von Systemen; gliedert ein System in Subsystem, Elemente, Verknüpfungen, Ein- und Ausgabegrößen.“

Duden: „Disziplin zur Untersuchung von Eigenschaften technischer, wirtschaftlicher und informationsverarbeitender Systeme.“

eng gefasst: Disziplin, die Methoden bereitstellt, um zu einer realen Ausgangssituation eine Software zu entwerfen. Dabei liegt der Fokus auf Informationssysteme.

Einordnung der Systemanalyse im Softwareentwicklungsprozes:

1) Analysephase
2) Entwurfsphase
3) Implementationsphase
4) Test- und Einführungsphase
5) Wartungsphase

Prototyping: Projekt wird in kleinere Zeitabschnitte unterteilt. In jedem Zeitabschnitt werden die Punkte 1-4 durchlaufen.

1.1 Exkurs in die Systemtheorie

Eigenschaften von Systemen:

a) Jedes System existiert in einer Umgebung.
b) Jedes System ist durch einen Rand von seiner Umgebung getrennt.
c) Systeme haben Ein- und Ausgaben.
d) Systeme haben Schnittstellen.
e) Ein System kann Subsysteme enthalten.
f) Ein System kann Kontrollmechanismen haben.
g) Die Kontrollmechanismen basieren auf Feedback.
h) Ein System hat Eigenschaften, die über die Summe der Eigenschaften der Subsysteme hinausgehen.

Anmerkungen:

zu a), b)	Wichtig in der Analysephase: Was gehört zum System und was nicht?
zu c), d)	Zur Kommunikation mit dem System müssen Schnittstellen definiert werden. Nur über diese sind Eingaben aus bzw. Ausgaben in die Umgebung möglich.
zu e)	Über die Erfassung oder Definition von Teilaufgaben des Systems können Subsysteme beschrieben werden.
zu f), g)	Unterscheide: *Negatives Feedback* (ein Systemgleichgewichtszustand muss hergestellt werden) und *positives Feedback* (Systemoutput wird gesteigert).
zu h)	Unterscheide: *Holistischer Ansatz* (Bsp.: Auto ist ein Transportmittel, aber die Subsysteme wie Räder, Motor, Karosserie,... besitzen diese Eigenschaft nicht.) und *Reduktionistischer Ansatz* (Komplexe Systeme können durch die Eigenschaften der Subsysteme komplett beschrieben werden).

© S. Meißner und A. Vogl

1.1.1 Informationssysteme

Allgemein betrachtet man in der Systemanalyse Systemeigenschaften. Im Folgenden betrachten wir Informationssysteme. Diese kann man in drei Kategorien einteilen.

1. Operationale Informationssysteme:

Systeme, die die Alltagsroutine eines Unternehmens automatisieren.

2. Management – Systeme:

Liefern Informationen, die für das Management eines Unternehmens (Abteilung, Projekte) wichtig sind.

3. Echtzeit Kontrollsysteme:

Direkte Kontrolle von Systemoperationen (Produktionsstraßen, Flugnavigationssystem,...)

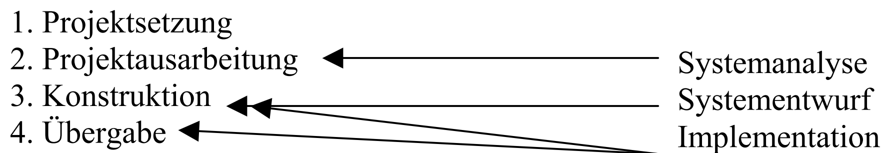
1.1.2 Probleme im Software – Entwicklungsprozess

Folgende Probleme lassen sich ermitteln:

- schlechtes Projektmanagement (alte Systeme in neuem Gewand; das neue System wurde nie realisiert)
- schlechte Analyse (Organisation stimmt nicht mit der Arbeitsorganisation beim Auftraggeber überein)
- mangelhafte Schulungen
- Analyse entspricht nicht den Anforderungen
- Projektsetzung mangelhaft
- Zusätze von Auftraggebern verlängern den Entwicklungsprozess
- beim Auftraggeber: Vorbereitung der Implementation schlecht
- mangelnde Qualifikation des Teams
- Vorbereitung der Implementation im Entwicklungsteam
- Äußere Umstände führen zum Wechsel der Voraussetzungen
- Kommunikationsprobleme (schlechte oder falsche Analyse wird betrieben)
- Inkompetenz der Entwickler
- schlecht dokumentierte, bestehende Systeme, die integriert bzw. ersetzt werden sollen

1.1.3 Entwicklungsphasen

Wir können die Softwareentwicklung in verschiedene Phasen einteilen:



Wir werden uns mit der Systemanalyse beschäftigen. Das ist der Prozess, der die fachlichen Anforderungen erfasst, analysiert und schließlich modelliert.

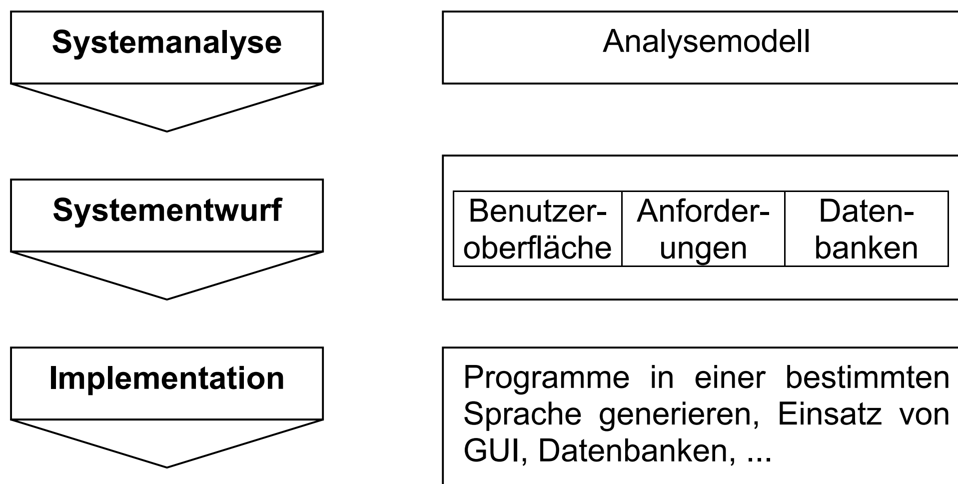
Ergebnis der Systemanalyse ist ein **Modell**. Dieses Modell muss drei Bedingungen genügen: es muss konsistent, vollständig und durchführbar sein.

Konsistenz: Das Modell genügt der Semantik (Geschäftsregeln, Integritätsregeln,...) der realen Ausgangssituation.

Vollständigkeit: Alle wesentlichen Aspekte und Anforderungen und Funktionalitäten sind erfasst.

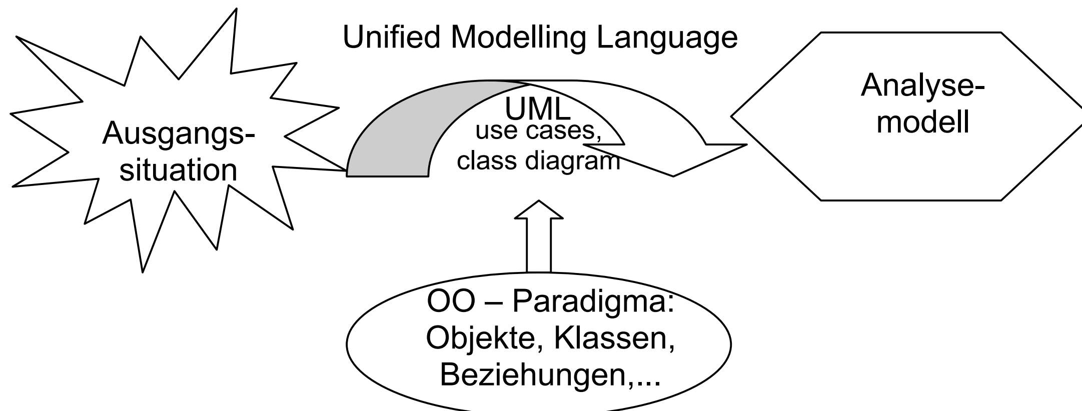
Durchführbarkeit: Das Modell ist in sich widerspruchsfrei und prinzipiell realisierbar.

Abgrenzung zwischen Systemanalyse und Systementwurf:



Die Frage nach konkreter Programmiersprache o. ä. stellt sich in der Analysephase nicht!

1.2 Konzept Objektorientiertes Paradigma



1.2.1 Objekt

Definition (Objekte):

> Objekte sind wohlunterscheidbare Modellierungseinheiten mit eindeutigem Bezug zu realexistierenden Dingen (Auto, Produkt,...) oder Personen oder Begriffen (Musik, Krankheit,...)

Objekte werden beschrieben durch Eigenschaften (Attribute) und Verhalten (Methoden). Im System bekommt jedes Objekt durch das Programmiersystem eine eindeutige Identifizierungsnummer, der OID (object identifier). Objekte mit gleichen Eigenschaften (Attributen) und gleichen Verhalten (Methoden) werden zu Klassen zusammengefasst.

1.2.2. Kapselung

Ziel ist, Veränderungen der Werte der Attribute eines Objekts nur über wohldefinierte Operatoren zuzulassen. Damit definiert die Kapselung für jede Klasse (jedes Objekt) eine Innen- und eine Außenwelt. Praktisch realisiert wird das über die Keywords „private“ bzw. „public“.

Beispiel:

```
class Kunde {
      private int Kdnr;             // nur innerhalb dieser Klassen
      private String name;          // können die Daten noch
      private String vorname;       // geändert werden!
      private Date gebdatum;
      private char geschlecht;
      ...
      public get.kdnr ( ) {...};    // Mit diesen Methoden können die
      public set.kdnr (int Wert)    // Daten verändert werden.
      {...};
      public kunde (int ikdnr; String iname, String ivorname;…)
      {
      this.name = iname;            // Wahl des letztgenannten Kunden
      this.Kdnr = iKdnr
      this.vorname = ivorname,….
      } }

class Applik {
      ...
      public main (args []) {
      kunde k=new kunde(1,’Mentär’,’Rudi’,….};
      k.get.kdnr();
      k.set.name(‘Mentär1’);        // falsch, da nur über set / get
      k.name = ’Mentär1’;           // aufrufbar
...} }
```

1.2.3 Attribute

Attribute beschreiben die Eigenschaften, Attributwerte die Zustände von Objekten. Alle Objekte einer Klasse besitzen die gleichen Attribute d.h. insbesondere im Kontext einer Klasse müssen die Attributnamen unterschiedlich sein!
Es gibt :

a) einfache Attribute (name, vorname, farbe)
b) zusammengesetzte Attribute (Adresse besteht aus plz, ort, str, usw.)
  Notation: `adresse (plz, ort, str)`
c) mehrwertige Attribute z.B. Hobbies
  Notation: `{hobbies}`

Viele Attribute genügen gewissen Regeln, die durch die Ausgangssituation bestimmt sind.

Beispiel:

```
class Produkt {
      private int prodnr;
      private String bezeichnung;
      private float preis;
      private String material;
      private String farbe;
      ... }
```

Produkte haben semantische Beschränkungen, z. B.:

> Preis > 0
> material ∈ {Aluminium, Eisen, Kupfer}
> farbe ∈ {rot, gelb, silber}
> wenn material = Aluminium dann farbe = Silber;

Die Geschäfts- oder Integritätsregeln müssen im Modell erfasst sein. Eine weiter wichtige Kategorie von Attributen sind die **Schlüsselattribute**. Ein Schlüssel ist ein Attribut oder eine Attributmenge, mit deren Hilfe jedes Objekt eindeutig identifiziert werden kann! Beispiel: kdnr in der Klasse Kunde; prodnr in der Klasse Produkt.

1.2.4. Klassen

Eine Klasse ist eine Menge von Objekten gleichen Typs, d.h. mit gleichen Attributen und gleichen Methoden und gleichen Beziehungen zwischen verschieden Objekttypen! Es gibt zwei Möglichkeiten, Klassen zu beschreiben.

1. Die Klassenintension:

> Beschreibung der Klassen durch Attribute und Methoden.

2. Die Klassenextension:

> Beschreibung der Klasse als Menge bis dahin erzeugten Objekten.

Anmerkung: Objektorientierte Programmiersprachen kennen eigentlich nur die Klassenintension. Objektorientierte Datenbanken müssen mit beiden Beschreibungen arbeiten.

Möglichkeiten der Zuordnung von Objekten zu Klassen (die Existenz eines Objekts und die Zuordnung zu einer Klasse sind zwei verschiedene Dinge!):

a) Instanzierung

> Objekt wird über den Konstruktor einer Klasse erzeugt und dieser Klasse automatisch zugeordnet!

b) Objektmigration

> Eine Objekt ändert im Laufe seines Lebenszyklus die Zugehörigkeit zu einer Klasse. Beispiel: Es existiert in der Applikation eine Klasse Mitarbeiter und eine Klasse Rentner. Ab einem gewissen Alter müssen die Mitarbeiterobjekte in die Klasse Rentner wechseln (unter Beibehaltung des object identifier).

c) Rollenkonzept

Ein Objekt kann zur gleichen Zeit Instanz mehrerer Klassen (gleicher Spezialisierungsstufe) sein. Beispiel: In einer Unternehmensapplikation existieren die Klassen Kunde, Mitarbeiter, Aushilfen. Ein Mitarbeiterobjekt kann auch der Klasse Kunde zugeordnet sein.

Eine Klasse kann eigene Attribute haben, sogenannte Klassenattribute. Eigenschaft: Der Wert eines Klassenattributs ist für alle Objekte gleich!

Beispiel:

```
class Kunde {
      private int Kdnr;
      ...
      private static int Anzahl;
}
```

Anzahl: die Anzahl der bis dahin erzeugten Kundenobjekte

1.2.5. Methoden

Beschreibungen das Verhalten / Funktionalitäten der Ausgangssituationen. Es gibt drei Arten:

a) Objektmethode:

Beschreiben die Kommunikationsmöglichkeit der Innenwelt einer Klasse mit ihrer Außenwelt. Objektmethoden können verschiedene Aufgaben haben:

- lesend auf Attribute zugreifen
- schreibend auf Attribute zugreifen (Attributwerte verändern)
- Berechnungen durchführen
- Aufruf anderer Methoden
- Implementierung von Geschäftsregeln

b) Klassenmethode:

Klassenmethoden beschreiben das Verhalten bezüglich der Gesamtheit der Objekte der Klasse. Beispiel: (Im Applikationsprogramm: Aufruf durch `Kunde.anzahlberechnen ()`)

```
class Kunde {
      private int Kdnr;
      private String name;
      ...
      public static int anzahl      // Klassenattribut
      ...
      // Klassenmethode typisch durch static gekennzeichnet:
      public static int anzahlberechne ()
      {berechne die Anzahl der bis dahin erzeugten Objekte}
      ... }
```

c) Implizierte Methoden:

Methoden, die notwendig sind zum Arbeiten mit Klassen:

- Konstruktoren
- Destruktoren

Bezüglich aller Arten gilt: jede Methode wird beschrieben durch Signatur und Rumpf.
Methodensignatur besteht aus:

- Name (muss innerhalb der Klasse eindeutig sein)
- Übergabeparameter
- Rückgabewert

Im **Rumpf** steht die Programmfunktionalität.

Beschreibung von Klassen und Klassenverhalten durch **CRC-Karten (Class Responsibilities Collaborators):**

<name>	konkret / abstrakt
Liste der Oberklassen	
Liste der Unterklassen	
Responsibilities	Collaborators
<attribut>	
<attribut>	
...	
<methode1>	<klasse x>
<methode2>	
...	

Hinweis abstrakte Klasse: Klasse, zu der keine Objekte erzeugt werden dürfen.

Beispiel: Klasse Person vererbt an Klasse Kunde; Klasse Auftrag kann von Kunden erzeugt werden.

Kunde	konkret
Person	
kdnr;	
name;	
vorname;	
...	
set()	
get()	
...	
neuer Auftrag ( )	Klasse Auftrag

Achtung! Beschreibung ist nicht vollständig: Es fehlen die Referenzattribute, die über die Klassenbeziehungen hinzukommen!

1.2.6. Polymorphismus: „viele Erscheinungsarten“

Es gibt mehrere Arten:

1.) Typkonvertierung

Bezieht sich auf elementare Datentypen. Beim Einsatz der Methoden werden die Datentypen „passend gemacht“; der Rückgabewert bleibt derselbe! Beispiel:

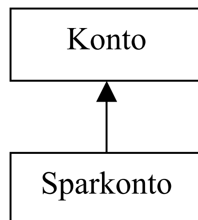
```
String a,b;
int i;
b = a + i;    //’t’ konkatenation von Strings
```

`+` ist die Methode, die zwei vorgegebene Werte konkateniert und als String zurückgibt! Hier findet ein Typkonvertierung integer→ String statt

2.) Inklusionspolymorphismus

Konvertierung von Klassen als Datentypen. Klassen, die in einer Vererbungshierarchie stehen, zum Beispiel Applikation im Bankumfeld:

```
Klasse Konto
Klasse Sparkonto
```



Im Konto existiert eine Methode `einzahlen (float wert) {...}`(Objektmethode). In der Applikation:

```
Konto k=new Konto (4711, 12345000, ...)
Sparkonto sk=new Sparkonto (4712, 12345000, ...)
k.einzahlen (2000,00)
sk.einzahlen (1500,00)
```

`einzahlen ()` kann im Sparkonto benutzt werden, ohne dort definiert zu sein!

3.) Overriding

Methoden in einer allgemeinen Klassen können in den speziellen Klassen überschrieben werden d.h. in der allgemeinen und der speziellen Klasse existieren Methoden gleichen Namens, aber mit unterschiedlichen Programmfunktionen (mehr dazu in 1.2.7.).

4.) Overloading

Methoden gleichen Namens können verschieden Übergabeparameter und / oder Rückgabeparameter haben. Beispiel:

```
Botschaftenclass Kunde {
private int kdnr;
private String name;
...
public void update (int kdnr) {tu was};
public void update (int kdnr, String name) {...};
public void update (String name, String vorname) {...};
}
```

1.2.7. Vererbung

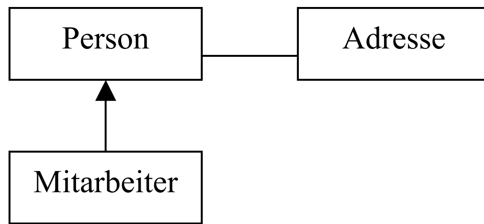
Vererbung ist eine gerichtete Beziehung zwischen verschiedenen Klassen in der Form „ist ein“. Beispielsweise ein Kunde ist eine Person, genauer: ein Kundenobjekt ist ein Personenobjekt.

Konsequenzen:

1. Dadurch werden gewisse Klassen als speziell und andere als allgemein ausgezeichnet.
2. Die speziellen Klassen „erben“ von den allgemeinen Klassen:
   - alle Objektattribute
   - alle Klassenattribute
   - alle Objektmethoden
   - alle Klassenmethoden
   - alle „normalen“ Beziehungen der allgemeinen Klassen zu irgendwelchen anderen Klassen!

Beispiel:

```
class Person {
      private int pnr;
      private String name;
      private String vorname;
      ...
      private static int anzahl;
      public getpnr() {...};
      ...
      public setpnr() {};
      ...
      public static int anzahlberechnungen(){...};
      class mitarbeiter extends Person {}
      private float gehalt;
      private String einstelldatum;
      public float gehaltberechnen(){...};
      }
```



Spezialisierungskonzept muss auf 3 verschiedenen Ebenen realisiert werden. Im Folgenden werden wir uns diese genauer ansehen.

a) intensionale Spezialisierung (Klassenintension)

Forderung: Die Intension der speziellen Klasse muss ein Subtyp sein oder Intension der allgemeinen Klassen. Def. Subtyp:

i) bzgl. atomarer Attribute

Sind t_1, t_2 zwei Datentypen atomarer Attribute, dann ist t_1 Subtyp von t_2, also dom (t_1) $\subseteq$ dom (t_2). dom steht für Domain = Wertebereich.

Bsp.: t_1 ist int, t_2 ist float => t_1 ist Subtyp von t_2: dom (t_1) '=' Z, dom (t_2) '=' R

ii) bzgl. Arrays

array [1...n] of t_1 ist Subtyp von array [1...m] of t_2 => n $\geq$ m und t_1 ist Subtyp von t_2

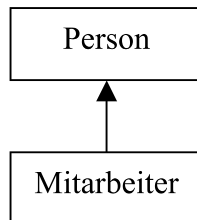
iii) bzgl. set, bag, list

set (t_1) ist Subtyp von set (t_2) => t_1 ist Subtyp von t_2, analog bag und list

iv) bzgl. tuple – Datentypen

T_1: tuple (int a, int b) und T_2: tuple (int a) dann sind allgemein T_1, T_2 Tupletypen. Ist T_1 Subtyp von T_2 => T_1 enthält mehr Datentypn als T_2 oder die Datentypen in T_1 sind Subtypen der Datentypen von T_2

Beispiele:



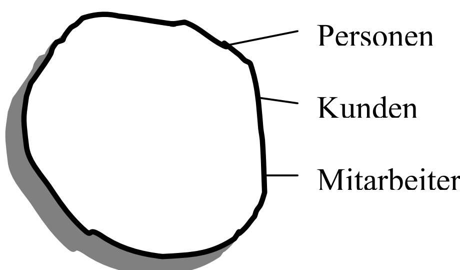

```
class Person {
      int pnr; String name; String gebdat;
      ...}  // Methoden

class Mitarbeiter extends Person {
      float gehalt; String einstelldat;
      ...}  // Methoden
```

Über die intensionale Spezialisierung wird eine Typhierarchie definiert!

b) extensionale Spezialisierung (Klassenextension)

Forderung: Die Extension der speziellen Klasse ist eine Teilmenge der Extension der allgemeinen Klasse. Beispiel:



Jedes Mitarbeiterobjekt ist Mitglied der Klasse Person. Mehr dazu später unter „overriding“!

c) Methodenspezialisierung

i) Jede Methode der allgemeinen Klasse ist auch Methode der speziellen Klasse.
ii) Methoden können in der speziellen Klasse redefiniert werden. Beispiel: In der Klasse Person existiert eine Objektmethode:

```
public void show ( ) {
      System.out.println (pnr+ ´´ ´´ +name+ ´´ ´´ +gebdat); }
```

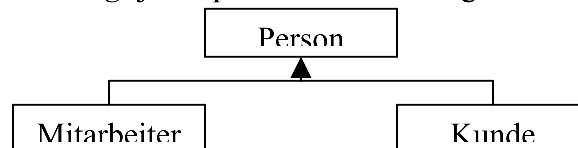
In der Klasse Mitarbeiter wird diese Methode redefiniert. Dabei unterscheidet man:

- ersetzende Spezialisierung: komplett neuer Programmcode
- verfeinerte Spezialisierung: Aufruf der allgemeinen Methode, zusätzlicher Code:

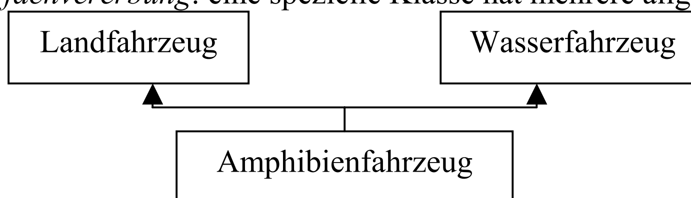
```
public void show ( ) {
show ( );
System.out.println (gehalt+ ´´ ´´ +einstelldat); }
```

Anmerkung: Es existiert die Unterscheidung in

- *Einfachvererbung*: jede spezielle Klasse hat genau eine allgemeine Klasse



- *Mehrfachvererbung*: eine spezielle Klasse hat mehrere allgemeine Klassen

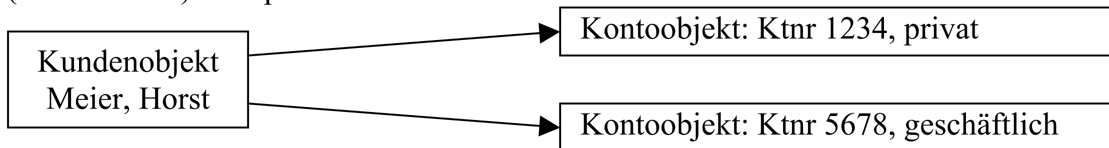


1.2.8 Klassenbeziehungen

Beziehungen zwischen Objekten (links) werden zusammengefasst zu Beziehungen auf Klassenebene. Man unterscheidet drei Arten. Alle Beziehungen sind Beziehungen zwischen Klassen gleicher Spezialisierungsstufe.

a) Assoziationen

Beziehungen allgemeiner Art, die beteiligten Klassen sind gleichberechtigt (bidirektional). Beispiel:

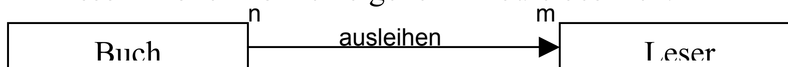


Zusammengefasst wird durch alle Objektbeziehungen zwischen Kunden- und Kontoobjekten eine Klassenbeziehung „besitzt“ definiert:

Eigenschaften:

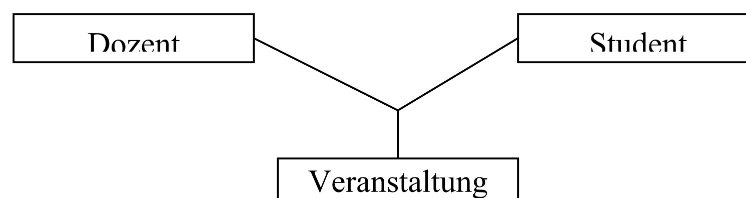
- Assoziationen können einen Namen haben.
- Assoziationen haben eine Kardinalität, d. h. die Angabe, wie viele Objekte der einen Klasse mit wie vielen Objekten der anderen Klasse in Beziehung stehen dürfen. Kardinalitäten sind abhängig von der Ausgangssituation, die modelliert werden soll.

- Assoziationen können eigene Attribute besitzen.



„ausleihen“ hat das Attribut Rückgabedatum.

- Assoziationen kann zwischen beliebig vielen Klassen Beziehungen schaffen (Stelligkeit der Beziehung). Beispiel: „prüfen“ ist eine Assoziation zwischen den drei Klassen Dozent, Student, Veranstaltung, d. h. zu prüfen ist eine 3stellige Assoziation:



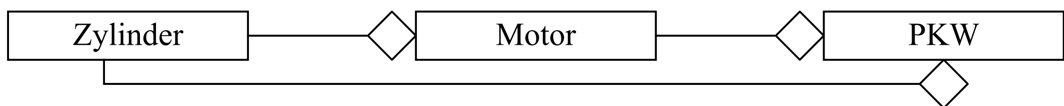
- zwischen 2 Klassen können mehrere Assoziationen bestehen

b) Aggregationen

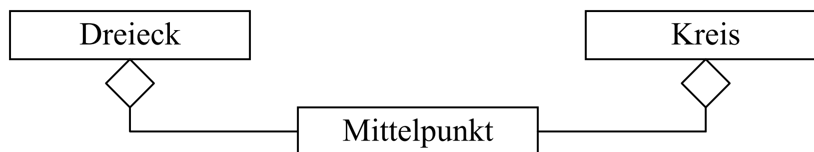
Aggregationen sind spezielle Assoziationen. Es sind gerichtete Beziehungen im Sinne von „ist Teil von“ oder „ist Komponente von“. Beispiel:

Die Aggregation teilt die beteiligten Klassen ein in Aggregatklasse und Komponentenklasse. Zusätzliche Eigenschaften:

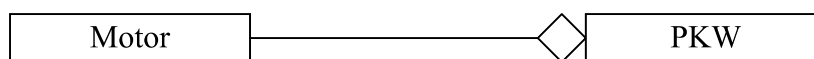
- *irreflexiv*: Eine Klasse kann nicht Komponente von sich selbst sein
- *asymmetrisch*: Eine Klasse ist Komponente und die andere das Aggregat und nicht umgekehrt.
- *transitiv*: Beispiel:



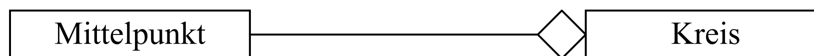
Es gibt vier Typen von Komponenten:
i) gemeinsame Komponenten:
Eine Klasse kann Komponente mehrere Aggregatklassen sein. Beispiel in einer Geometrieanwendung:



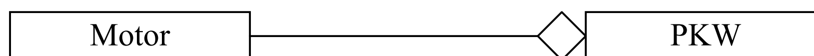
ii) private Komponente:
Eine Klasse kann immer nur Komponente gegen eine Aggregatklasse sein. Beispiel:



iii) abhängige Komponente:
Das Komponentenobjekt darf ohne Aggregatobjekt nicht existieren. Beispiel:



iv) unabhängige Komponente:
Komponentenobjekte können auch nach dem Löschen des Aggregatobjekts weiter existieren. Beispiel:



Bei Aggregation gibt es :
1) Propergierung von Werten

Aggregat	Komponenten
hat Auto ein Attribut Farbe hat Karosserie ein Attribut Farbe	ist Auto rot, muss auch das zugehörige Karosserieobjekt rot sein!

2) Propagierung von Methoden:
existiert z.B. in einer Graphikanwendung eine Klasse „Fenster“ als Aggregat und existiert im Aggregatobjekt gewisse Komponentenobjekte (Dreiecke, Kreise,...) und ist bezüglich des Aggregats eine Methode „verschieben“ definiert, so müssen alle Komponentenobjekte im Aggregatobjekt mitverschoben werden!

c) Kompositionen

Komposition ist eine spezielle Aggregation im Sinne, dass die Komponentenobjekte privat und abhängig sind.

Modellierung der Klassenbeziehung

1) bezüglich Assoziationen:
Realisation über „Referenzattribute“, d.h. z.B.

Modellierung mittels CRC – Karten:

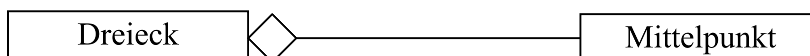
Person	konkret
Responsibilities	Collaborators
name vorname gebdat … {adresse}	 Adresse

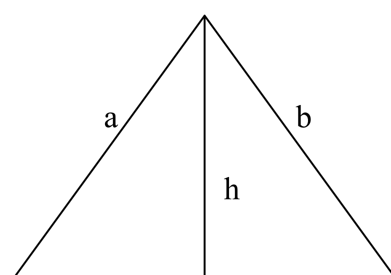
Adresse	konkret
Responsibilities	Collaborators
plz ort straße … bewohner	 Person

Hinweis: { ... } für mehrwertige Attribute.

Referenzattribute erhalten als Werte die OID’s der Objekte die zum gegebenen Objekt in Beziehung stehen (analog: Aggregationen und Kompositionen).

2) Besonderheit bei Aggregationen (Kompositionen):
Modellierung durch Objekteinbettung, d.h. die Attribute der Komponentenklasse werden den Attributen der Aggregatklasse zugefügt.





Dreieck	konkret
Responsibilities	Collaborators
höhe1 a b c höhe2 ... flächeninhalt mx-Koordinaten my-Koordinaten	

Problem der Objekteinbettung: die eingebetteten Komponenten können nicht referenziert werden!

1.2.9 Botschaften

Jede objektorientierte Software besteht aus Klassen, deren Objekte miteinander kommunizieren und auf diesem Wege Anwendungsfunktion umsetzt. Diese Kommunikation besteht im Austausch von Botschaften
Botschaften bestehen aus Namen und eventuell Parametern von existierenden Methoden des kommunizierten Objekts.

Voraussetzung:

1) Die Methode muss existieren
2) Botschaften können nur durch längst vorher definierte Beziehungen ausgetauscht werden!

Beispiel:

Bezug: Klasse Person mit Methoden zur Erzeugung von Personenobjekten und mit Methoden zur Änderung von Attributwerten

```
class Applik {
public void main (...) {
      Person p = new Person (‘Ana’,’Bolika’);
      /**  Klasse Applik schickt eine Botschaft an die Klasse Person,
      den Konstruktor zur Erzeugung eines neuen
      Personenobjektes aufzurufen und die übergebenen Attributwerte
      zuzuordnen */
p.setpnr (12345);
      // analog
```

Zusammenfassung Oo - Programmierung:

Zur objektorientierten Programmierung werden benötigt:

- Informationen über die notwendigen Klassen (in Form von CRC – Karten, Klassendiagrammen)
- Informationen über die Umsetzung von Anwendungsfunktionalität via Objektverhalten
- Informationen über den Lebenszyklus komplexer Objekte als endliche Automaten

1.3 Unified Modeling Language

Zunächst wollen wir Allgemeines zum Instrumentarium der oo – Systemanalyse betrachten. Allen oo – Analysemethoden gemeinsam: Die Bestimmung der Klassen! Aus der ersten „prosaischen“ Beschreibung der Anwendungssituation werden alle Substantive extrahiert. Durch entfernen von unsinnigen Substantiven und Substantiven, die Attribute anderer Substantive sind oder überflüssige Attribute beschreiben, erhält man iterativ das Ergebnis: Eine Liste möglicher Klassenkandidaten. Nun kann man CRC – Karten zusammenstellen.

1.3.0 Gemeinsamkeiten der oo - Analysemodelle

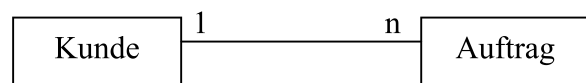
a) statisches Klassenstrukturdiagramm (class diagram)

Diagramm, in dem die Klassen mit ihren Beziehungen eingezeichnet sind. Jede oo – Analysemethode hat ihre eigene Notation.
=> *Beispiel:* siehe 1. Übungsaufgabe „Geometrie“

b) Zustandübergangsdiagramm (state diagram)

Beschreibt den Lebenszyklus eines Objekts. Zustand eines Objekts ist definiert über eine konkrete Bewertung der Attribute seiner Intension. Zustandsänderung ist definiert durch eine Wertänderung ein oder mehrer Attribute. Also: Objekte werden als endliche Automaten interpretiert.
Im state diagram: alle definierten Zustände eines komplexen Objekts und die Ereignisse Aktionen, die zu Zustandsänderungen führen.
=> *Beispiel*: Klasse Kunde ungeeignet, da kaum Veränderungen. Betrachten wir Klasse Auftrag:

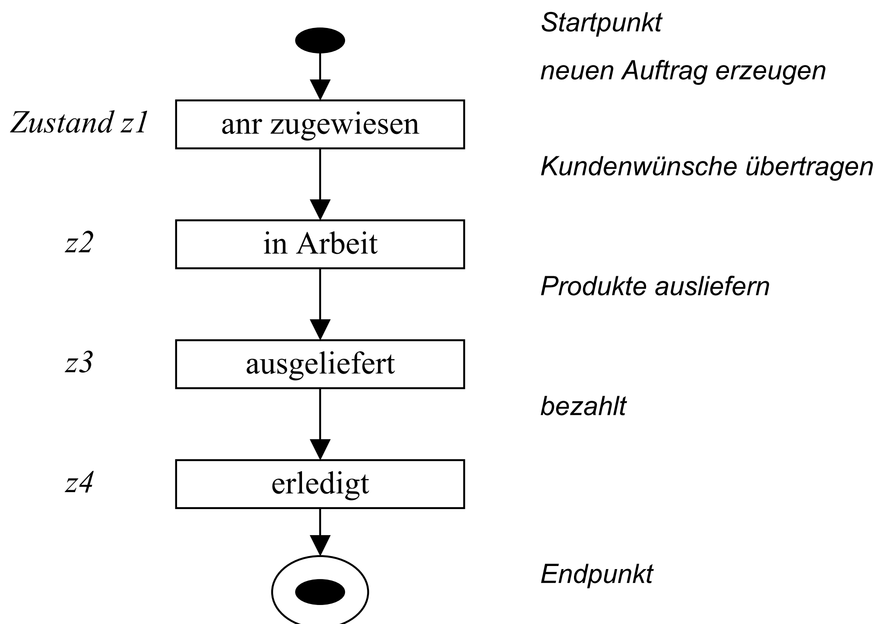
Auftrag	konkret
anr adatum status kunde rechnung	kunde



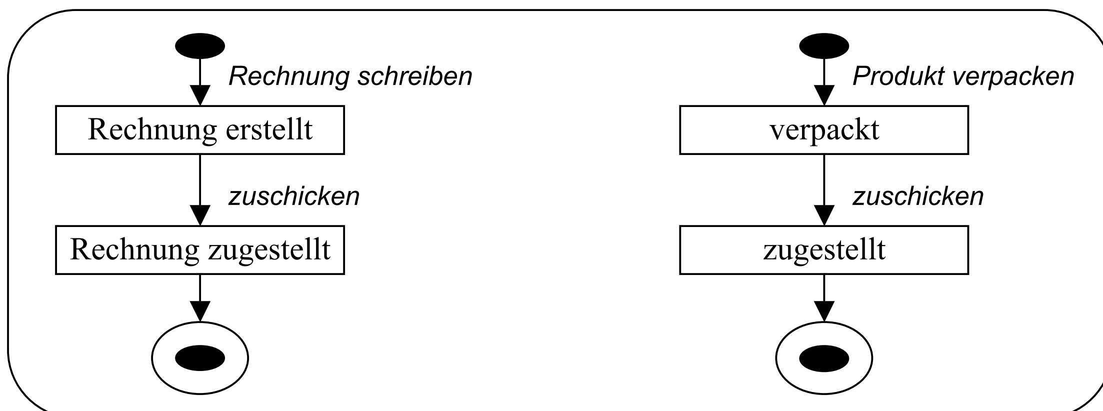
Interessant hier:
Geschäftsregeln für status:

- in Arbeit
- ausgeliefert
- erledigt

Daraus lässt sich das state diagram für ein Auftragsobjekt erstellen (*Pseudo – Notation)*:

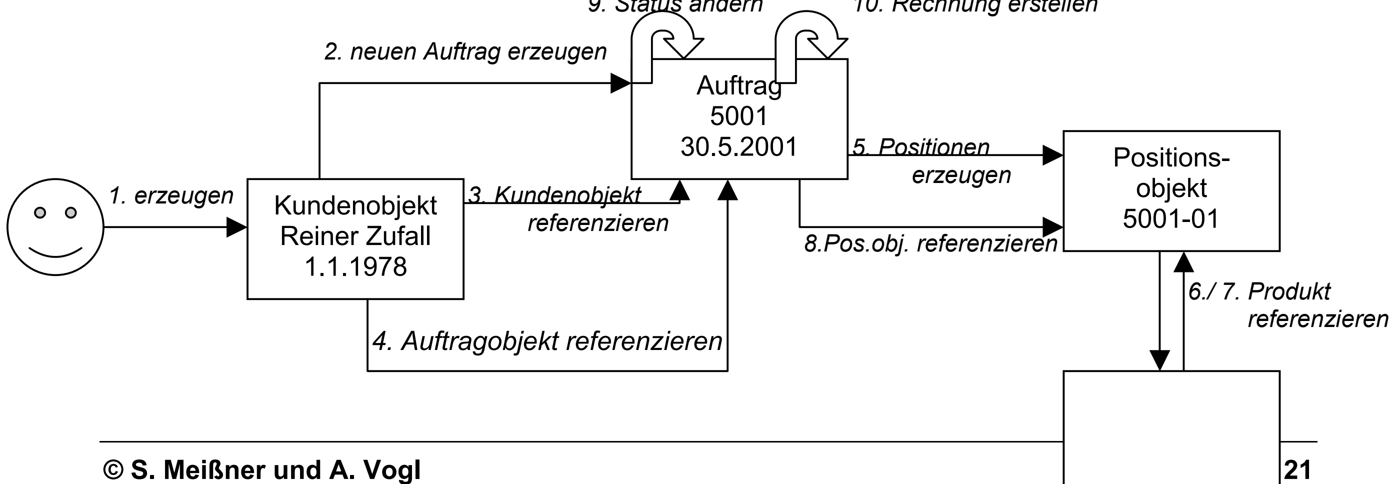


state diagrams können verschiedene Detaillierungsebenen beschreiben. Im Beispiel: ein detailliertes state diagram zum Zustand „ausgeliefert“:



c) dynamische Modelle der Ereignisse, Szenarien, Objektinteraktion

Beschreibt Interaktionsverhalten des Systems auf Objektebene. Basis dafür sind charakteristische (typische) Szenarien. In jedem Szenarium: Bestimmung der beteiligten Objekte und ihrer Interaktionen.
=> *Beispiel*: Szenario sei Auftrag einen neuen Kunden zu bearbeiten.



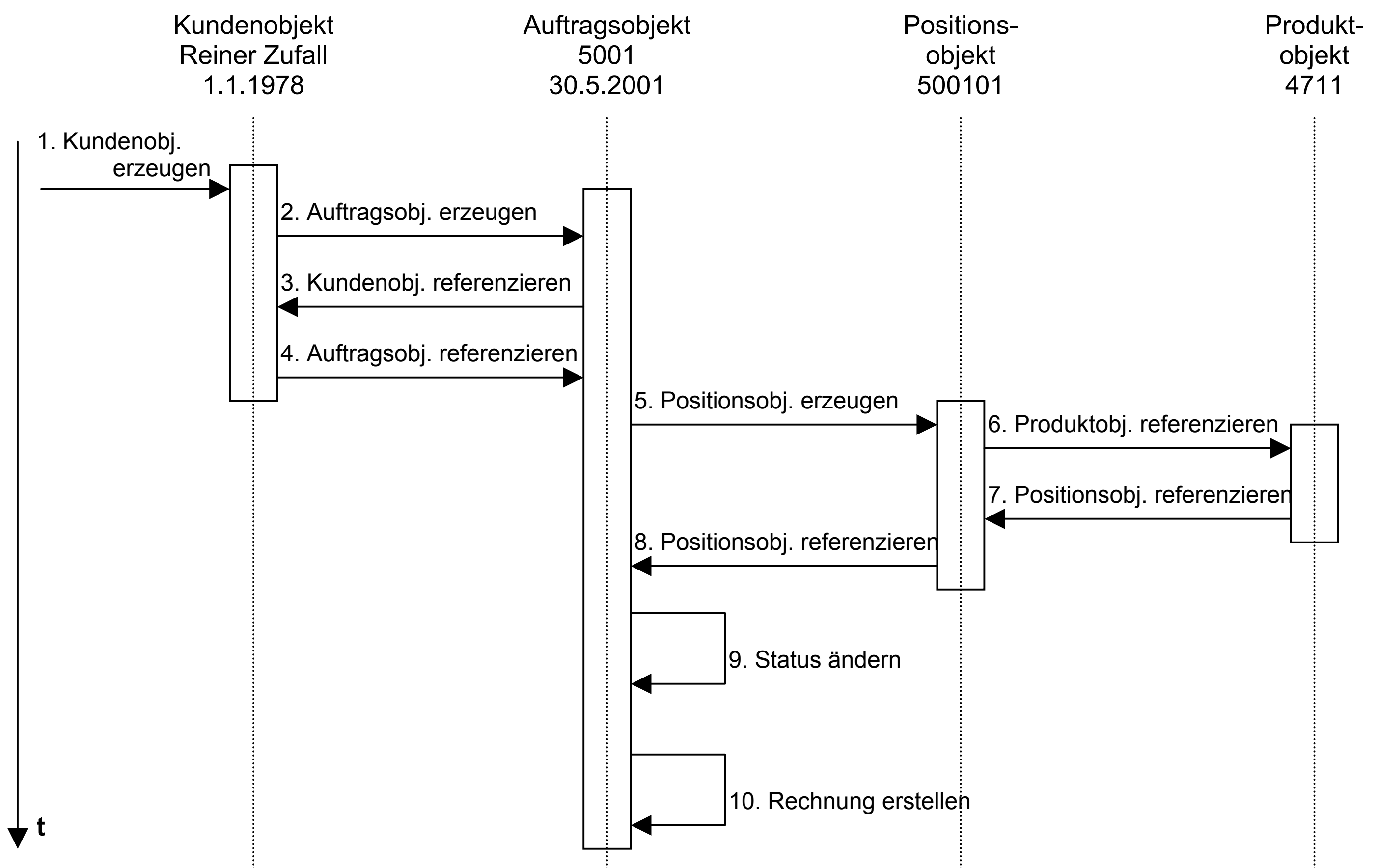
1.3.1 Vorgänger der UML

1. Object Modeling Technique (OMT) von James Rumbaugh

1991 wurde das Prinzip veröffentlicht. Benutzt werden drei verschiedene Modelle:

- statisches Objektmodell (Klassenstrukturdiagramm)
- dynamisches Modell (=> state diagram, Ereignisdiagramm)
- Funktionenmodell (=> aus der strukturierten Analyse!): Beschreibung der Funktionalitäten System/Anwendungssituation in hierarchischer Anordnung

Zum Ereignisdiagramm: beschreibt Interaktion von Objekten in einem bestimmten Szenarium. => *Beispiel*: Auftrag eines neuen Kunden bearbeiten:



Eigenschaften:

- Ereignisdiagramm hat eine Zeitachse (vertikal: von oben nach unten).
- Jedes Objekt hat einen header, eine gestrichelte vertikale Lebenslinie und ein Rechteck, das angibt, wie lange das Objekt am Szenarium beteiligt ist.
- Die Ereignisse sind durch horizontale Pfeile dargestellt, mit einem Namen und eventuell einer laufenden Nummer versehen.

Einsatz der OMT:

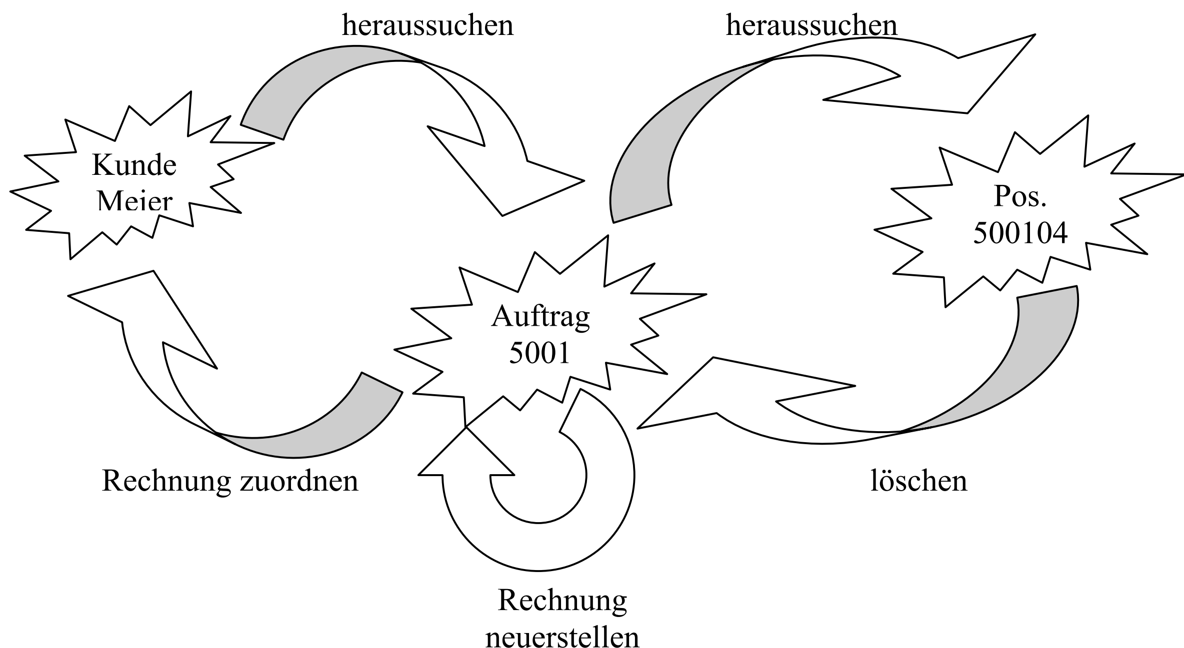
- Beschreibung der „normalen“ Abläufe in einem charakteristischen Szenarium.
- Überprüfung der Modellierung (sind alle Klassen vorhanden? Sind alle Klassenbeziehungen vorhanden? Ist das Modell ausgewogen?

2 Booch-Methode

Grady Booch 1991-1994 die Booch-Methode entwickelt. Benutzt folgende Diagramme:

- Klassendiagramm
- Zustandsdiagramm
- Objektdiagramm: beschreibt die Objektinteraktion für ein Szenario

  Szenario: Stornierung einer Position eines Auftrages



- Ereignisdiagramm: siehe OMT
  Gemeinsamkeiten von Objekt- und Ereignisdiagramm: beide beschreiben Objektinteraktionen von Anwendungsszenarien. *Unterschied*: im Ereignisdiagramm dominiert die zeitliche Komponente.
  Einsatz: viele Objekte, weniger Botschaften: Objektdiagramm
  wenige Objekte viel Botschaften: Ereignisdiagramm
  (Daumenregel !)

- Moduldiagramm: beschreibt Zuordnungen von Modulen zu Dateien (für die Analyse irrelevant)

- Prozessdiagramm: beschreibt Zuordnungen von Prozessen zur Hardware (für die Analyse irrelevant)

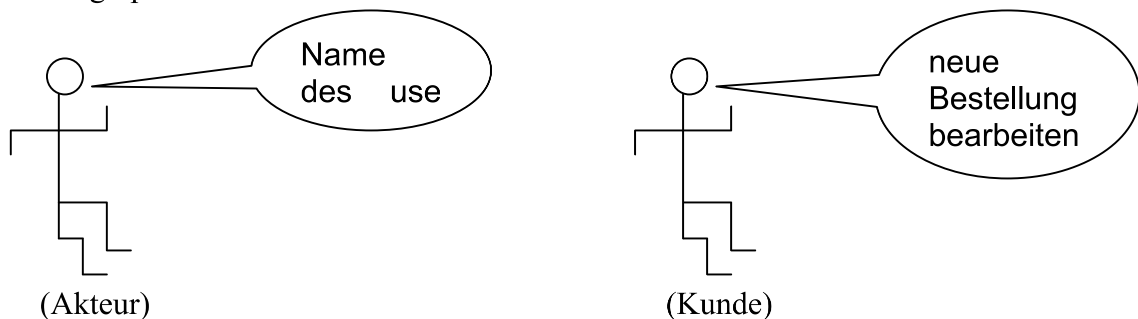
© S. Meißner und A. Vogl

3. Object Oriented Software Engineering (OOSE) Ivar Jacobson

Benutzte Diagramme:

- use cases (use case diagram)
  Idee: Beschreibung des Systems über eine Sammlung charakteristischer Anwendungsfälle. Herangehensweiße:
  i) welche Person (oder andere Systeme) stellen typische Aufgaben an das System (Akteure, Personen oder externe Systeme)
  ii) auf welche Ereignisse muss mein System reagieren können?!
  Beschreibung der use cases :
  i) Textuell und graphisch
  ii) Textuell: Auflistung der Arbeitsschritte der Interaktion des Systems mit den Akteuren dieses Anwendungsfalls.

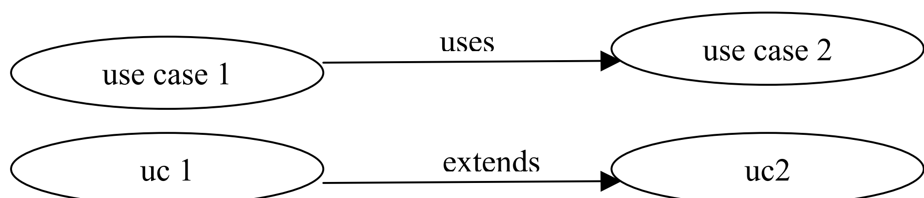
  graphisch:



  Vorgehensweise: ein use case für den Normalfall (ohne Besonderheiten). Danach: weiter use cases, die die Variationen des Normalfalls beschreiben

  use cases können auf verschiedenen Detaillierungsstufen beschrieben werden! Bis hin zu Objekt- bzw. Ereignisdiagrammen.
  Zwischen use cases sind zwei Arten von Beziehungen definiert:
  1. use case1 benutzt (uses) use case2 Einsatz um Wiederholungen zu vermeiden
  2. use case1 erweitert (extends) use case2 Einsatz z.B. Variationen des Normalfalls

  grafisch:



Beispiel: Szenario: Ein Versicherter möchte einen Schadensfall von seiner Versicherung bezahlt bekommen.

System: Versicherungsgesellschaft
Akteur: Versicherte
Grafisch: siehe Block 4!
Textuell:

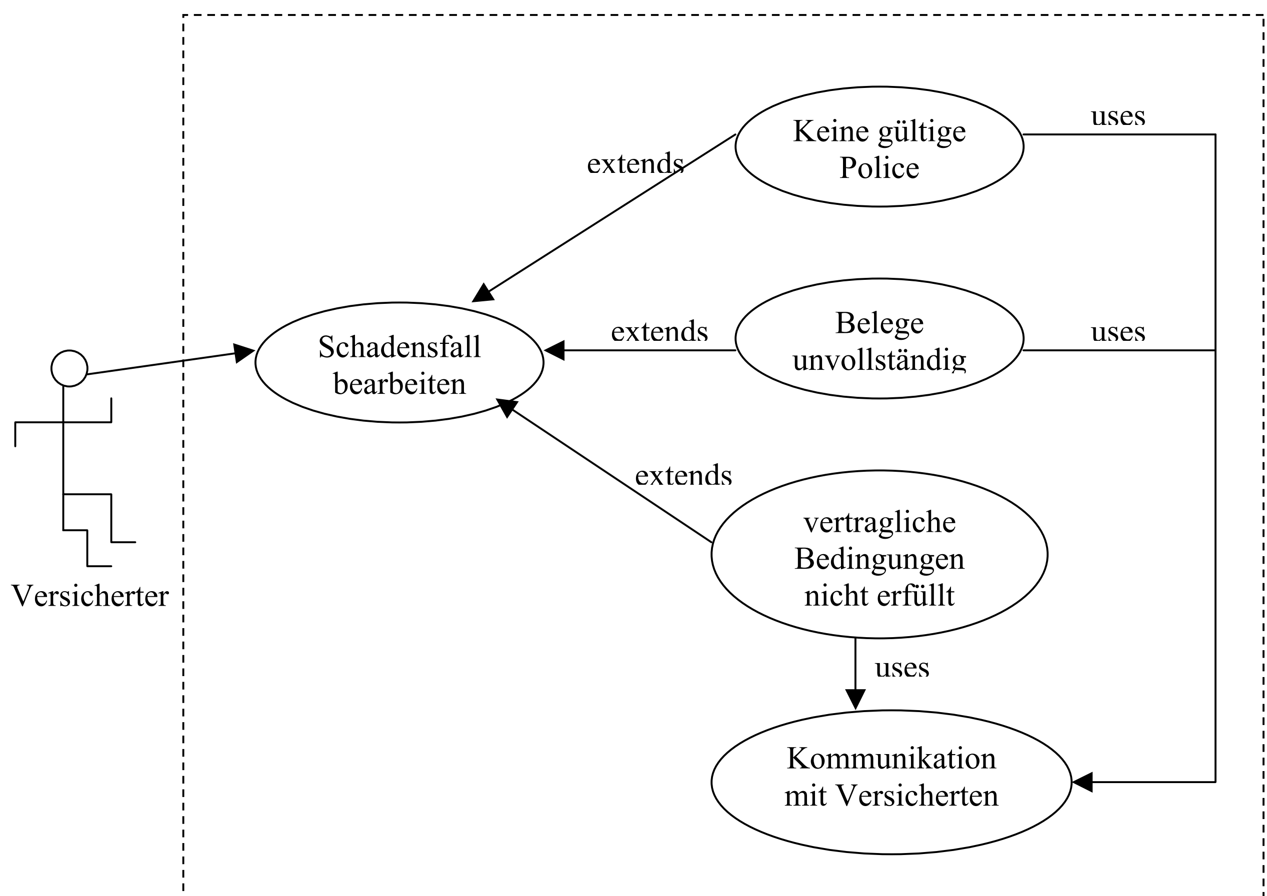
1. Versicherter schickt seine Ansprüche mit Belegen an Versicherung
2. System prüft ob der Versicherte eine gültige Police hat
3. System beauftragt den Schadensfall zu überprüfen
4. Mitarbeiter stellen fest, dass alle vertraglichen Bedingungen erfüllt
5. System bezahlt den Schaden

Variationen:

V1: 2a. Der Versicherte hat kein gültige Police
2a1. System meldet dem Versicherten, dass er kein Ansprüche stellen kann
2a2. Vorgang wird geschlossen

V2: 3a. Belege sind nicht vollständig
3a1. Meldung an den Versicherten, fehlende Belege nachzureichen
3a2. Versicherte schickt fehlende Belege
weiter mit 3

V3: 4a. Mitarbeiter stellt fest, dass die vertraglichen Bedingungen nicht komplett erfüllt sind
4a1. Verhandlung mit dem Versicherten
4a2. Einigung auf eine Teilsumme



- Problem Domain Model: ist ein Klassendiagramm

- Object Model: ist eine weitere Spezifikation des problem domain model unter dem Gesichtspunkt der Wahrscheinlichkeit von Änderungen. Resultat: Einteilung der Klassen in 3 Kategorien:
  - Datenklassen (Datencontainer)
  - Funktions-, prozeß, vorgangsorientierte Klassen
  - Schnittstellenklassen

- Design-Model: beschreibt den Übergang in die Entwurfsphase

- Sequence-Diagram: ist das Ereignisdiagramm

1.3.2. Unified Modeling Language (UML)

Ein Instrumentarium zur Softwareentwicklung von der Analyse bis zur Implementation! Zur Verfügung stehen:

a) use cases (diagram)

beschreiben Anwendungsszenarien über die Akteure und Systemaktionen bzw. –interaktionen.

b) class diagram

beschriebt statisch die logische Struktur des Systems über die beteiligten Objekttypen und ihre Beziehungen

c) interaction diagram : sequence diagram und collaboration diagram

beschreiben auf Objektebene die zu einer Aufgabenbewältigung notwendigen Objektaktionen.

sequence diagram: Ereignisdiagramm von J. Rambaugh
collaboration diagram: Object diagram von G. Booch
Gemeinsamkeiten. Beschreibung von Objektaktionen
Unterschiede: sequence diagram hat den Schwerpunkt auf der zeitlichen Komponente, collaboration diagram ist allgemeiner

d) package diagram

ist ein aggregiertes Klassendiagramm! Besteht aus packages, die Klassen enthalten, die ein Teil des System beschreiben.

e) state diagram

beschreibt komplexe Objekte als endliche Automaten durch Zustände und Aktionen, die Zustände verändern.

f) activity diagram

beschreibt die Aufgaben des Systems mittels work flows. (Prozesscharakter)

Alle Diagramme beschreiben das gleiche System, allerdings aus unterschiedlichen Blickwinkeln und unterschiedlichen Detaillierungsgrad!

Sichtweisen auf das System

i) statische Beschreibung der logischen Struktur: class diagram, package diagram
ii) Lebenszyklus komplexer Objekte: state diagram
iii) Dynamik des Systems auf verschiednen Detaillierungssystemen: interaction diagram, activity diagram, use cases

Praktisches Arbeiten mit der UML:

Ausgangsphase sind die use cases, d.h. der erste Schritt der Analyse besteht im Auffinden der charakteristischen Anwendungsfälle!

Voraussetzung dafür: erste Informationen über das zu entwickelnde System (Interviews, Arbeitsplatzbeschreibungen, Organigramme, sonstige Unterlagen,...)

Ergebnis: prosaische Grobbeschreibung des Systems

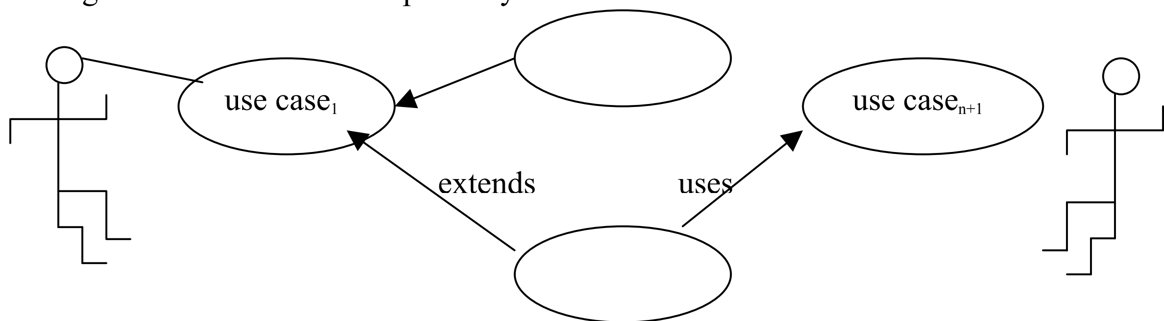
für das Finden von use cases: 2 Möglichkeiten

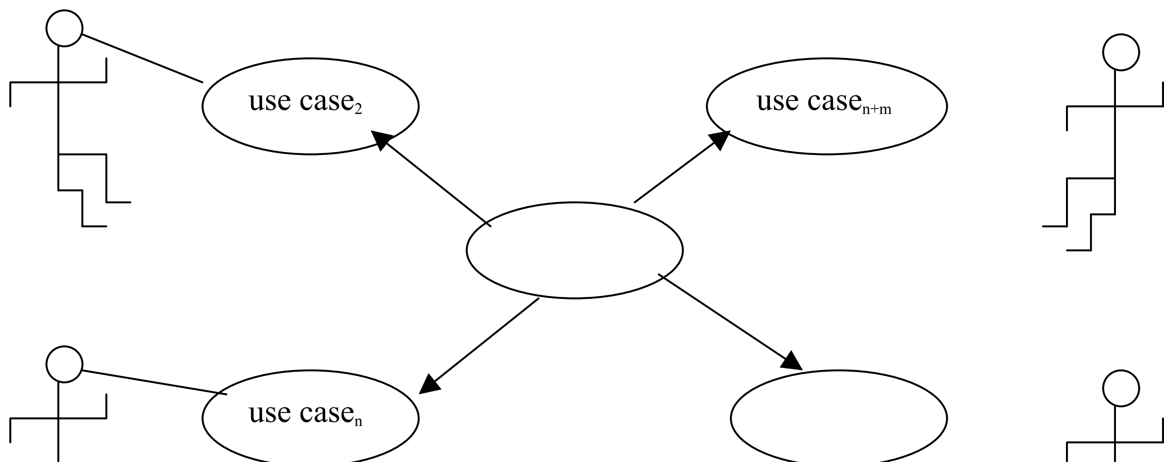
- 1. Möglichkeit: Bestimmung der Akteure (nicht Personen, sondern die Rollen, die die Personen haben!). ACHTUNG: Nicht zu sehr ins Detail gehen! Akteure können auch externe Systeme sein!
- 2. Möglichkeit: Identifikation der Ereignisse, auf die das System reagieren muss! ACHTUNG: use case sind nicht eindeutig bestimmt; sie können (müssen) unterschiedliche Detaillierungsstufen haben.

Vorgehensweise:

1.) Beschreibung des Normalfalls (keine Besonderheiten)
2.) Beschreiben von Variationen
3.) alles über „extends“ bzw. „uses“ in Beziehung setzen
für jeden einzelnen Anwendungsfall

Alle erhaltenen use cases können dann mit ihren Akteuren in einem use case diagram beschrieben und ggf. in Beziehung gesetzt werden! Dieses resultierende use case diagram beschreibt das komplette System!





ACHTUNG: textuelle Beschreibung aller use cases ist sehr wichtig!

Ansatz zur Formalisierung:

<Zeit und/oder Sequenznummer> <Akteur> <Aktion> <Constraints>

Beispiel 1:

1. Versicherter schickt Belege und den Anspruch an Versicherungsgesellschaft

Sequenznr. Akteur Aktion

Beispiel 2:

n. Am Ende jeden Monats schickt die Sachbearbeiterin einen Kreditübersicht an

Sequenznr. Zeitfaktor Aktion Akteur Aktion (Forts.)

alle Kunden, deren Kredit über 10.000 DM liegt

constraints

Constraint: Einschränkungen
ACHTUNG: durch Verfeinerung des Detaillierungsgrads kommt man zu interactio diagrams oder zum activity diagram!

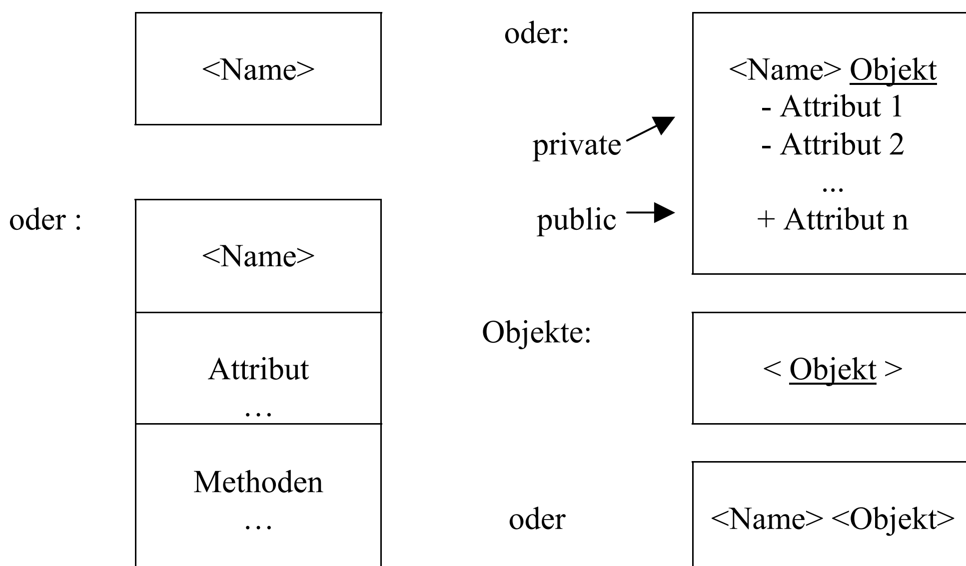
Verfeinerung: Herausnehmen einzelner Arbeitsschritte eines uns cases und Formulierung dieses Arbeitsschrittes als eigener use case oder Detailliter Beschreibung aller Arbeitsschritte in gegebene use case.

nächster Schritt: Entwicklung des Klassendiagramms

Frage: wie bekommt man die Klassen?
Ausgangspunkt: die textuelle Beschreibung der use cases!
Für jeden use case:

1. aus der textuellen Beschreibung eine Liste aller Substantive erstellen
2. Entfernen aller unsinnigen und überflüssigen Kandidaten
3. Entfernen aller Kandidaten (Substantive) die offensichtlich Attribute anderer Substantive sind

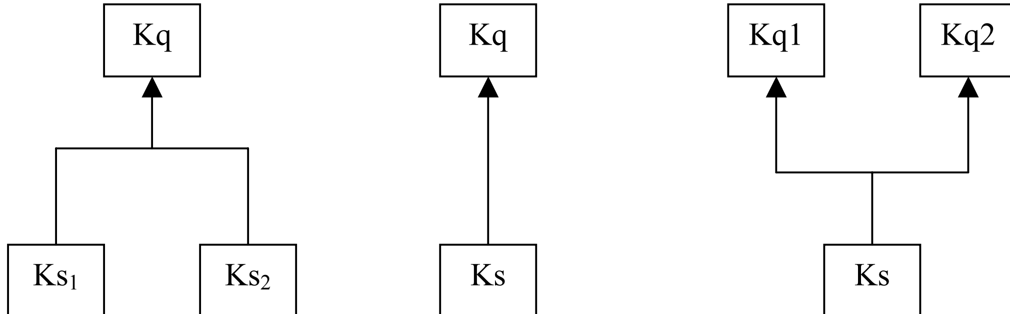
Dann folgt eine erste Beschreibung der erhalten Klassen mittels CRC- Karten (kein UML – Instrument!). Grafische Darstellung mittels benannter Rechtecke:



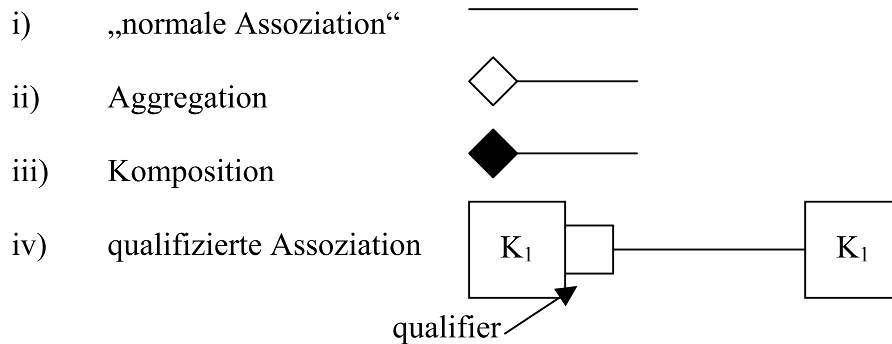
1.3.3. UML-Notation der Klassen

Beziehungen: Vererbung - Assoziation

- Vererbung: graphische Notation

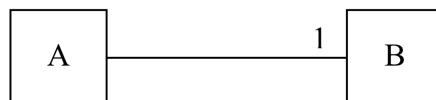


- Assoziationen:

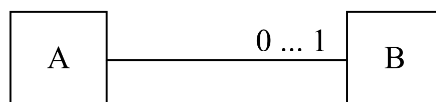


Kardialität: für i) - iii) am Beispiel einer Assoziation:

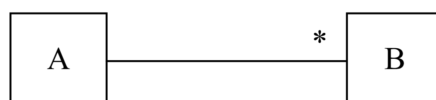
- zu einem Objekt aus A steht genau 1 Objekt aus B in Beziehung



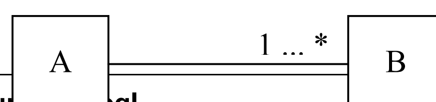
- ein Objekt von A steht mit keinem oder einem Objekt von B in Beziehung



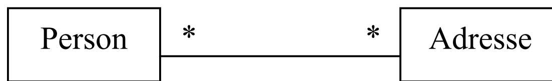
- ein Objekt von A steht mir keinem, einem oder beliebig vielen Objekten von B in Beziehung

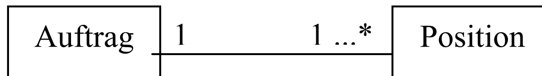


- ein Objekt von A steht mit mindestens 1 Objekt von B in Beziehung

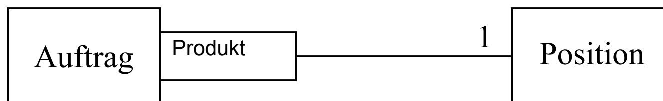


Beispiel: Eine Person kann mehrere Adressen haben, an einer Adresse können mehrere Personen wohnen.



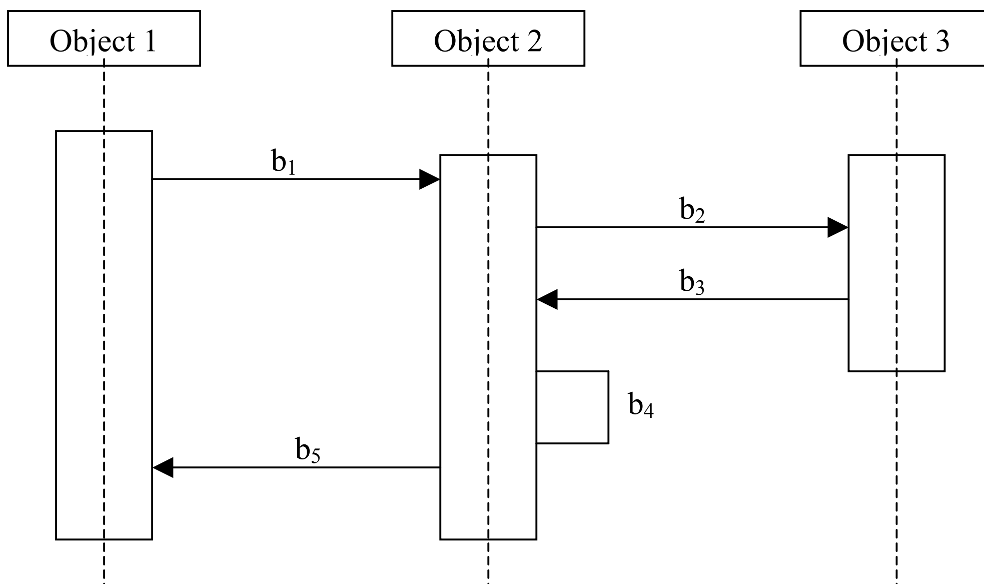


Zusätzliche Informationen bezüglich Beziehungen zwischen Auftrag und Position; in einem Auftrag darf es zu einem Produkt höchstens eine Position geben.

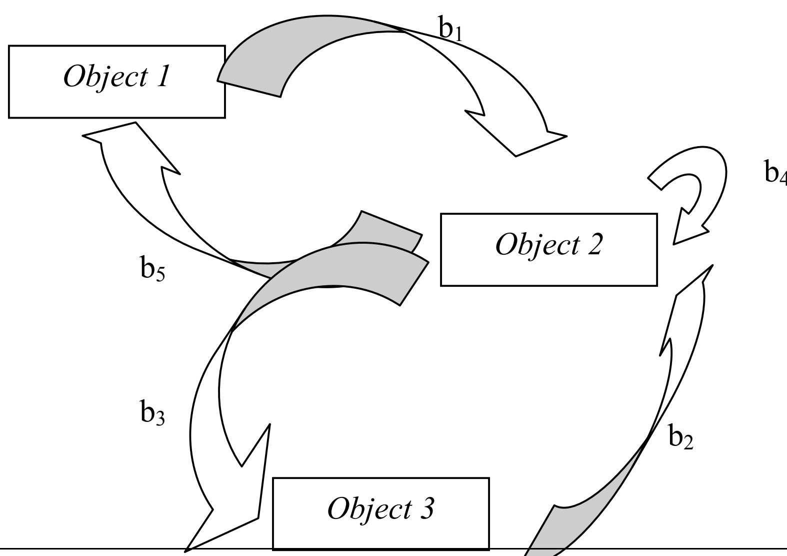


interaction diagram

- sequence diagram



- colaboration diagram



interaction diagrams haben als Basis use case oder verfeinerte use cases!

package diagram:

packages bestehen aus Klassen. Voraussetzung: man hat Kriterien, nach denen man die Klassen einzelner packages zugeordnet werden. ACHTUNG! Einsatz nur bei komplexer Systeme (viele Klassen)

state diagram (Zustandsdiagramm):

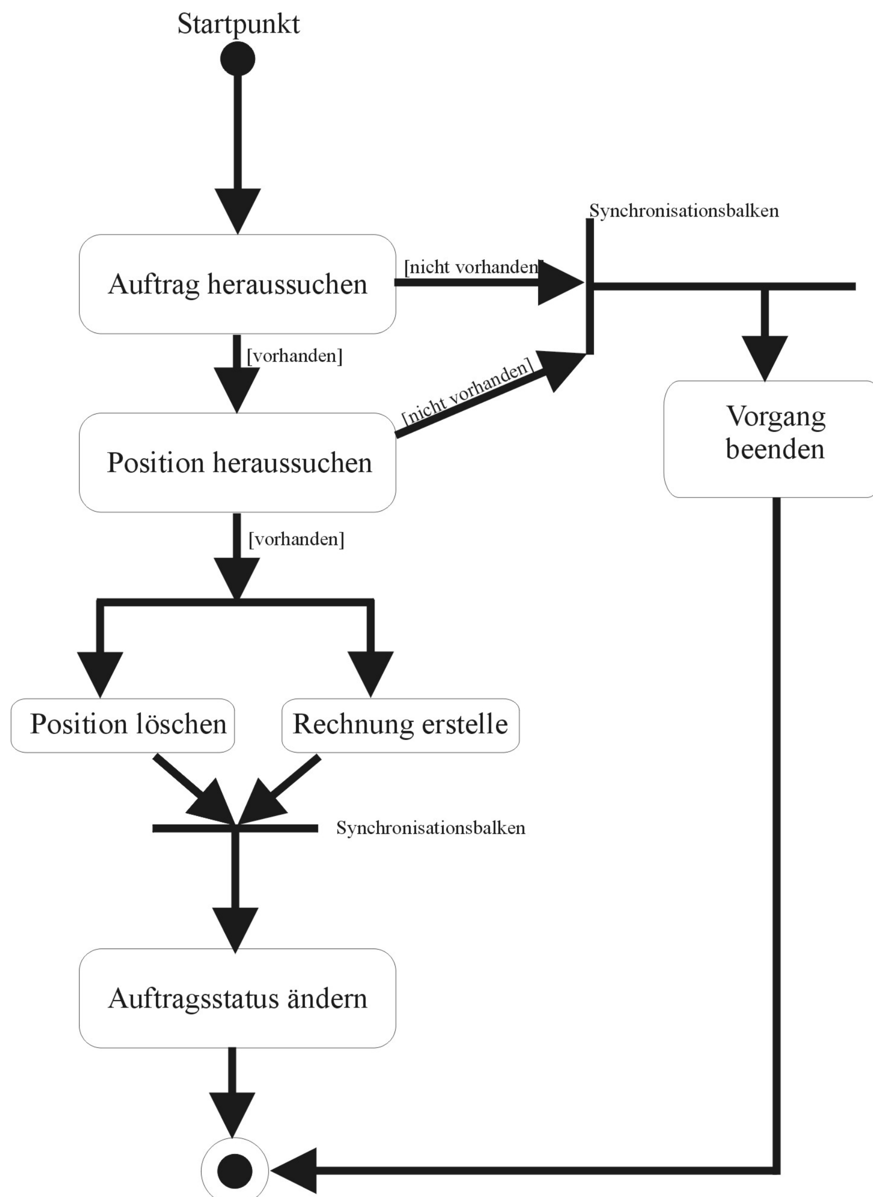
beschreibt ein Objekt als endlichen Automaten mit Zuständen und Zustandsübergängen. Ist nur Sinnvoll bezüglich komplexer Objekte

activity-diagram:

beschreibt die Aufgabenerledigung eines Systems mittels work-flows
Einsatz: 2 Extreme

- Beschreibung der funktionalen Abläufe in einem use case
- Beschreibung der Prozessabläufe einer Methode einer bestimmten Klasse

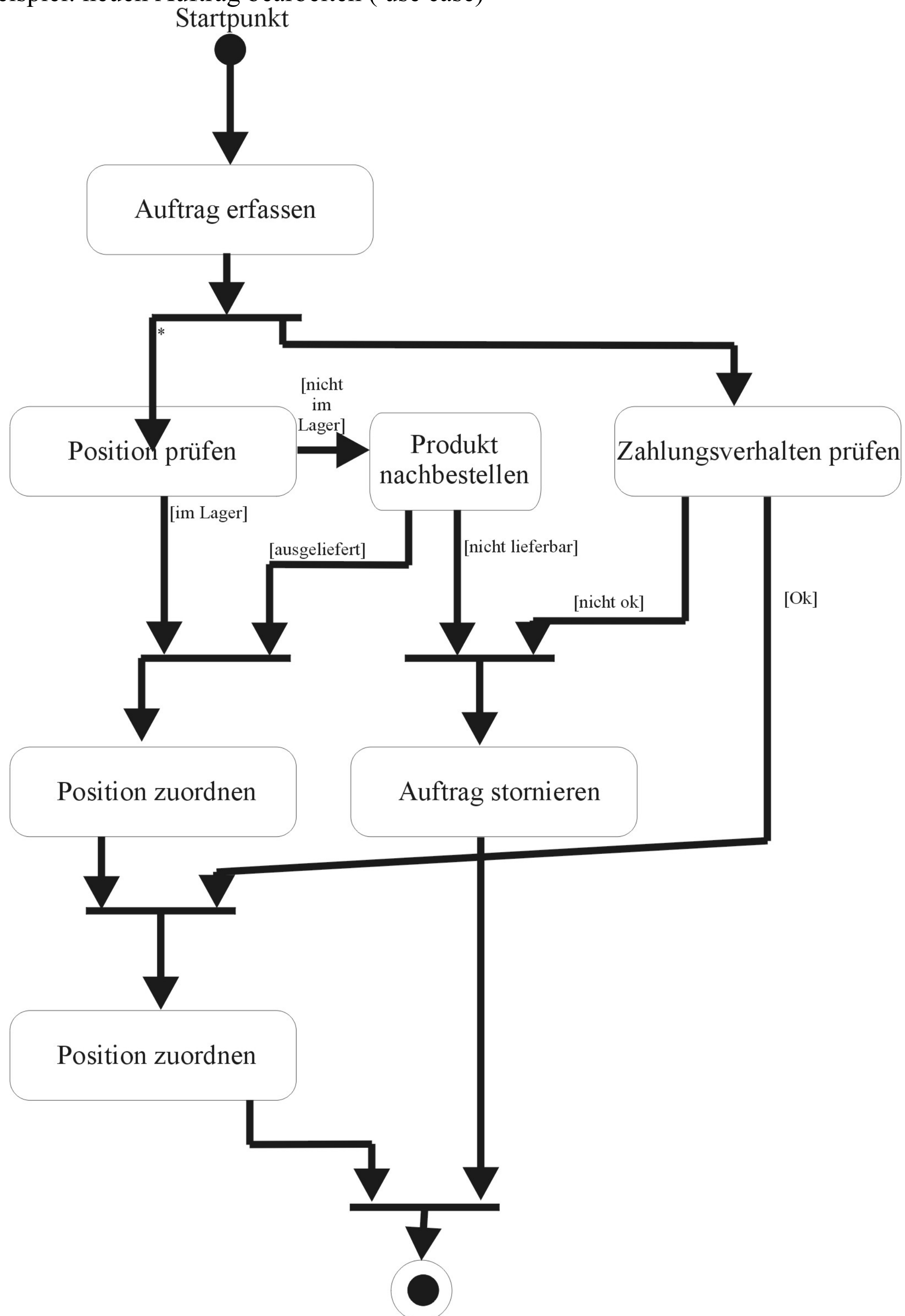
Beispiel: Löschen einer bestimmten Auftragsposition



graphische Notation:

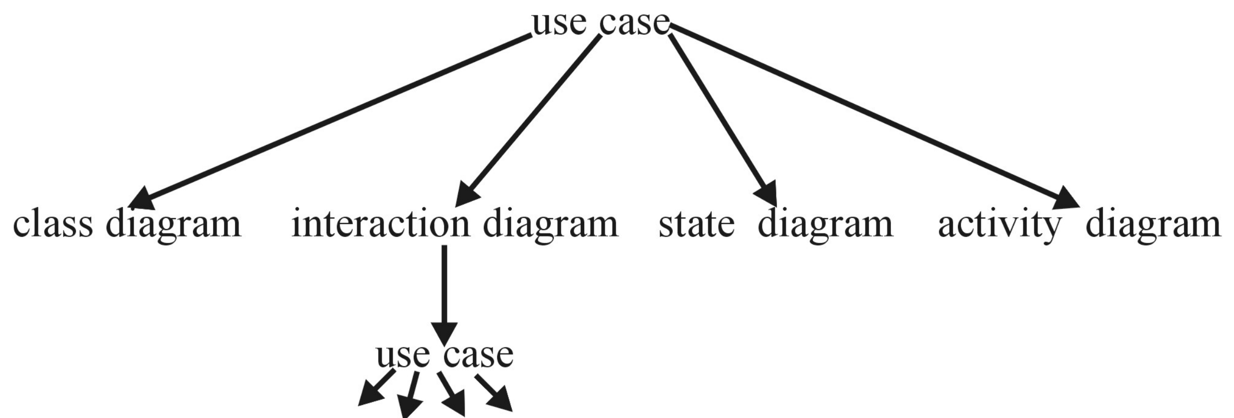
- Aktivitäten: abgerundete Rechtecke
- Entscheidungen: eckige Klammern
- Synchronisation
  paralleler Prozess: Synchronisationsbalken ----------
- * : mehrfaches Durchlaufen

Beispiel: neuen Auftrag bearbeiten (use case)



Fazit:

1.) Ausgangspunkt der Analyse sind die use cases



2.) Alles, was nicht direkt in einem Diagramm modelliert werden kann, aber wichtig für die weiterer Verarbeitung ist, muss als Kommentar irgendwo schriftlich fixiert werden. Ergibt ein Dictionary zusätzlich zur Beschreibung über Diagramme (z.B. CRC-Karten).