Teil IV

Definition

4.1 Einführung: Definition

- Eingabe: vage, unvollständige, inkonsistente Anforderungen
- Ergebnis: vollständige, konsistente, eindeutige, erfüllbare Anforderungen
- erstellte Dokumente:
  - Pflichtenheft
  - Produkt-Modell (z.B. SA, OOA (UML))
  - GUI-Modell
- Was statt Wie

Requirements-Engineering umfasst:

- Systemanalyse (requirements analysis)
- Produktdefinition (requirements specification)

Requirements Engineering

umfasst Methoden, Beschreibungsmittel und Werkzeuge
zur Ermittlung, Analyse und Formulierung von Anforderungen
an Software-Systeme (und eingebettete Systeme)

Struktur eines Pflichtenhefts

1. **Zielbestimmung**
   Musskriterien, Wünsche, Abgrenzung (was nicht?)
2. **Produkteinsatz**
   Zielgruppen, Betriebsbedingungen, ...
3. **Produkt-Umgebung**
   Software (BS, DBMS, GUI), Hardware, Produkt-Schnittstellen
4. **Produktfunktionen**
   einzeln, nur Was, ggf. in Pseudocode
5. **Produkt-Daten**
   langfristig zu speichernde Daten, aus Benutzersicht
6. **Leistungsanforderungen**
   z.B. Antwortzeit, Platzbedarf, Genauigkeit
7. **Benutzeroberfläche**
   Bildschirmlayout, Dialogstruktur, ...
8. **Qualitätsanforderungen**
9. **Testfälle** (einzeln)
10. **Entwicklungsumgebung** (Software, Hardware, Schnittstellen)
11. **Ergänzungen** (Index, Glossar, ...)

detailliert!

Anforderungen

- funktionale Anforderungen
  - Eingaben und deren Einschränkungen
  - Funktionen des Systems
  - Ausgaben (Reaktionen)
- Qualitätsanforderungen
  - Zeit, Speicher
  - Wartbarkeit
  - Zuverlässigkeit (Ausfallsicherheit, Fehlerbehandlung)
  - Portabilität, Anpassbarkeit, Kompatibilität
  - Benutzerfreundlichkeit, Benutzerprofil

Anforderungen (Fortsetzung)

- Anforderung an Realisierung
  - durch Software und/oder Hardware
  - Geräte
  - Schnittstellen
  - zu verwendende Hilfsmittel (BS, Rechner, . . .)
  - Dokumentation
- Anforderungen an Prüfung, Einführung, Betreuung
- Anforderungen an die Durchführung der System-Erstellung
  - Vorgehensweise
  - Ressourcen (Personal, Kosten, Termine)
  - Vorschriften, Normen

4.2 Analysemethoden

- die gängigen Analysemethoden kombinieren Basistechniken
- heute dominierend:
  - OOA (Object Oriented Analysis): basierend auf UML
    Klassendiagramme, Zustandsautomaten, Interaktionsdiagramme, ...
- “herkömmliche” Methoden:
  - SA (Structured Analysis, DeMarco 1978):
    Hierarchie von DFDs (→ Funktionsbaum), DD, ET,
    Entscheidungsbäume, Pseudocode
  - SA/RT (Structured Analysis, Real Time):
    SA, Zustandsautomaten, (ERD)

4.3 Objektorientierte Analyse

- nach Aufkommen von OOP: Bedarf für OOA und OOD
- zunächst viele konkurrierende Ansätze und Notationen u.a. von: Booch, Rumbaugh (OMT), Coad/Yourdon, Jacobson (OOSE), Wirfs-Brock
- nun: Standardisierte Notation UML (Unified Modeling Language) durch Booch/Jacobson/Rumbaugh + OMG

Diagramme in UML 2 (u.a.)

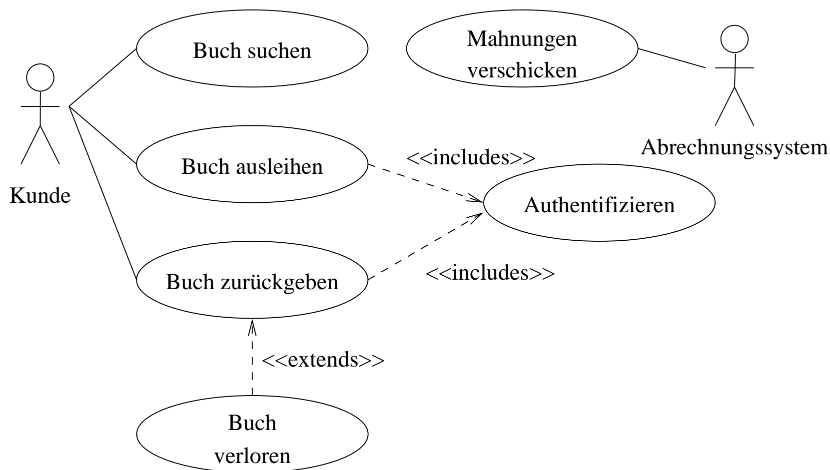
- Anwendungsfall-Diagramme (use case diagrams, → Jacobson)
- Klassendiagramme (und Varianten: Paket- und Objektdiagramme)
- Interaktionsdiagramme (u.a. Sequenz-, Kommunikations- u. Zeitdiagramme)
- Zustandsautomaten
- Aktivitätsdiagramme
- Kollaborationen, Montage- und Installationsdiagramme
- Spezifikationssprache OCL (Object Constraint Language)

4.3.1 Anwendungsfalldiagramme

Anwendungsfall (use case)

- typische Interaktion zwischen Benutzer und System
- z.B. (Bibliothek): Buch ausleihen, Buch zurückgeben, Buch bestellen
- extern sichtbare Funktion
- abgegrenzte funktionale Anforderung

Beispiel: Anwendungsfalldiagramm für Bibliothek



Anwendungsfalldiagramm

- zentrales Instrument beim Einstieg in Anforderungsanalyse
- Graph mit folgenden Knoten:
  - Akteur: (actor)
    - Rolle eines Benutzers im System
      (ggfs. mehrere Rollen pro Benutzer)
    - Ausgangspunkt für Bestimmung der Anwendungsfälle
    - auch nicht-menschliche Akteure möglich (z.B. anderes System)
    - dargestellt als Strichmännchen
  - Anwendungsfall: dargestellt durch Ellipse
- Kanten:
  - ungerichtete Kante zwischen Akteur und zugehörigem AF
  - gerichtete Kante und Beschriftung <<includes>> oder <<extends>> wischen AFs

Erweiterung eines Anwendungsfalls

- Anwendungsfall beschreibt nur “Normalfall” (→ einfacher)
- Sonderfallbehandlung als eigenständiger Fall
- <<extends>>-Kante zeigt von Sonderfall auf Normalfall

Einbeziehen von Anwendungsfällen

- haben mehrere Anwendungsfälle einen gleichen Verhaltensanteil, so kann dieser als eigener Fall "herausfaktorisiert" werden
- <<includes>>-Kante zeigt vom ursprünglichen Fall auf herausgezogenen Fall
- dies sollte sparsam genutzt werden
- denn: hier noch keine Entwurfsentscheidungen wie Zerlegung in Komponenten

Textuelle Beschreibung

zu jedem AF Text mit

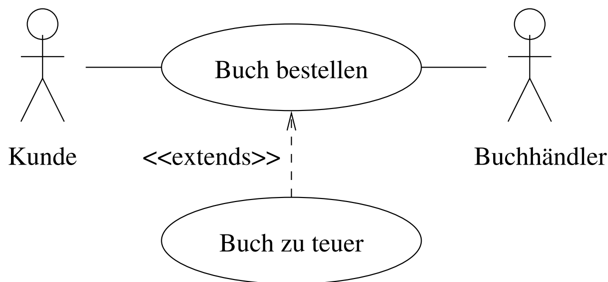
- Beschreibung der Interaktion/Funktion
- Vorbedingung, Nachbedingung, nicht-funktionale Anforderungen
- Varianten, Sonderfälle, ...

AF: Buch bestellen (in Buchhandlung)

Der Kunde nennt Titel (möglicherweise unvollständig) und/oder Autor(en) und ggfs. den Verlag und das Erscheinungsjahr des gewünschten Buches. Der Buchhändler sucht das Buch in der Datenbank und ermittelt die ISBN-Nummer, den Preis und die Adressen möglicher Großhändler. Von den Großhändlern werden Lieferfristen erfragt. Die Bestellung wird an den Großhändler mit der kürzesten Lieferfrist weitergeleitet. Dem Kunden werden Preis und Lieferfrist mitgeteilt. Die Bestellung wird in der Datenbank mit ISBN-Nummer, Name und Adresse des Kunden, geleisteter Anzahlung, Lieferfrist und Name des Großhändlers vermerkt.

AF: Buch zu teuer

Dem Kunden ist der genannte Preis zu hoch. Die Bestellung wird daher nicht an den Großhandel weitergeleitet, sondern der Buchhändler sucht in der Datenbank nach Werken mit ähnlichem Thema und nennt deren Preise.



Anwendungsfälle und Interaktionsdiagramme

- sinnvoll: ein Interaktionsdiagramm pro AF
- beschreibt Zusammenspiel von Objekten in AF
- in UML in verschiedenen Varianten:
  - Sequenzdiagramme
  - Kommunikationsdiagramme (äquivalent zu SDs)
  - Zeitdiagramme (→ Realzeitanwendungen)

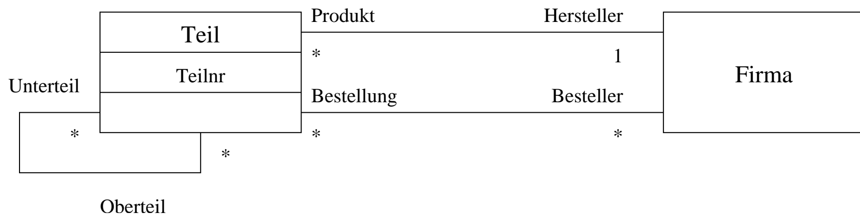
4.3.2 Klassendiagramm

- Bedeutung jedes Konzepts innerhalb eines Klassendiagramms abhängig von gewählter Perspektive
- konzeptionelle Perspektive:
  Größen des Problembereichs und Beziehungen dazwischen
- Spezifikationsperspektive:
  mehr Details; Einbeziehung der Schnittstellen
- Implementierungsperspektive

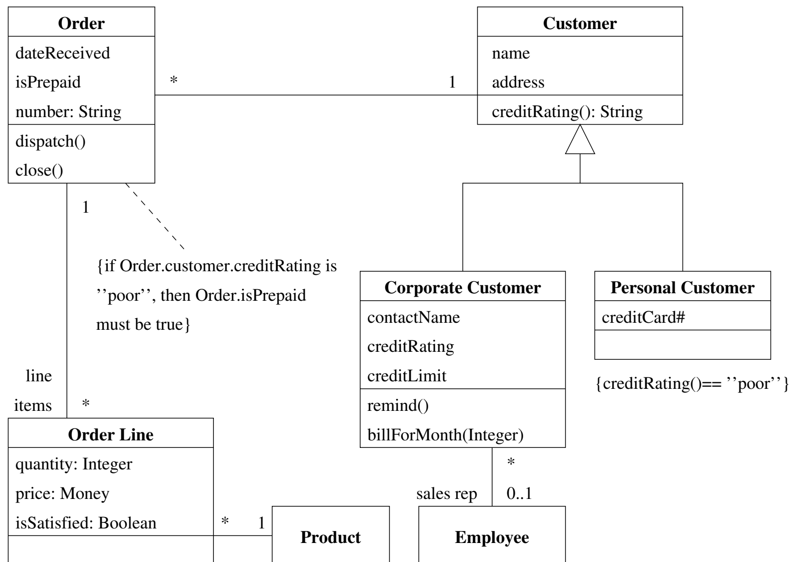
Assoziation aus konzeptioneller Perspektive

- modelliert Beziehung (zwischen Objekten) zweier Klassen
- aus Sicht beider Klassen: je ein Rollenname
  default: Name der Nachbarklasse;
  Rollennamen zwingend, wenn mehrere Assoziationen zwischen zwei Klassen;
  sonst nur, wenn Diagramm dadurch klarer
- alternativ: (gemeinsamer) Assoziationsname
- Rolle entspricht Objekt- oder Objektmengen-wertigem Attribut
- Kardinalitäten (Multiplizitäten): 1, n, 0..1, n..m, * ($\hat{=}$ 0..∞)
  oder Folge hiervon, Bsp.: 1,3..5,7

Beispiel: Klassendiagramm



Beispiel 2: Klassendiagramm



Restriktionen

- Restriktionen (constraints) bzgl. Objektzustand oder Assoziation können als Kommentare angefügt werden
- Notation: {Restriktion}
  Bsp.: {sorted}, {immutable}, {read-only}
- in OCL, natürlicher Sprache, Pseudocode, ...

Beispiele für Restriktionen

- Vor- und Nachbedingungen von Operationen

  Bsp.: Operation int sqrt()

  Vorbedingung: {this.value $>=$ 0}

  Nachbedingung: {result * result == this.value}

- Invarianten: bleiben nach jeder Operation erfüllt

  Bsp.: {balance == sum(entries);}

Verantwortung für Einhaltung von Vor- und Nachbedingungen

- Vorbedingung: Verantwortung für Einhaltung beim Aufrufer
- Operation verantwortlich für Einhaltung der Nachbedingung, sofern Vorbedingung erfüllt (analog bei Invariante)
- dadurch: kein Verdoppeln oder Vergessen von Überprüfung
- explizite Überprüfung von Restriktionen beim Debugging z.B. Operation checkInvariants

Assoziationen aus Spezifikationsperspektive

- entsprechen Verpflichtungen zur Bereitstellung von Operationen zum Zugriff auf Nachbarklassen und zur Pflege der Beziehung
  Beispiel: Firma bietet Operation zur Ermittlung von ihr hergestellter Teile
- mögliche Konvention:
  - bei 1-wertiger Beziehung: Operation liefert Nachbarobjekt
  - bei mehrwertiger Beziehung:
    Operation liefert Kollektion von Nachbarobjekten

Beispiel (Java)

```
class Teil{
   public List<Teil> oberteile();
   public List<Teil> unterteile();
   public Firma hergestelltVon();
   public Vector<Firma> bestelltVon();
}
```

Assoziationen aus Spezifikationsperspektive (Forts.)

- Diagramm aus Spezifikationsperspektive erlaubt unterschiedl. Implementierungen
- im Beispiel denkbare Implementierungen von oberteile():
  1) Teil enthält Verweis (OID) auf Liste von Oberteilen oder
  2) alle Teile durchlaufen und feststellen, ob Teil ein Unterteil hiervon ist

Operationen zur Beziehungspflege, z.B.:

- “Teil”-Konstruktor erstellt Verknüpfung zu Hersteller
- Operation addBesteller fügt Besteller hinzu

Kollektionen

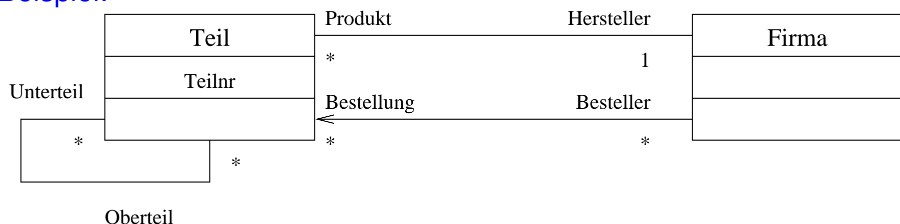
- default: Menge
- durch Restriktionen spezifizierbar:
  {bag}, {ordered bag}, {dag}, . . .

Assoziation aus Implementierungsperspektive

- entspricht Verweis (OID) auf Nachbarobjekt (bzw. Kollektion davon)
- Implementierung von Kollektion durch Vector, Array, Baum, Liste, …

Einschränkung der Navigationsrichtung

Beispiel:



- im Beispiel: von Firma auf bestellte Teile zugreifbar, aber nicht umgekehrt
- Navigationsrichtungen erst bei Verfeinerung einfügen (Spezifikations- oder Implementierungsperspektive)

Attribute

- aus konzeptioneller Sicht: ähnlich zu Assoziationen
- aus Spezifikationsperspektive:
  - Objekt ermöglicht Zugriff und Manipulation des Attributwerts
  - “Wert-Semantik” (statt “Referenz-Semantik” bei Assoziationen)
    $2 = 2$, aber i.allg. $O_1 \neq O_2$, auch bei gleichen Attributwerten (wegen OIDs)
- aus Implementierungsperspektive:
  - Attribut als Komponente des Objektzustandsraums (Instanz-Variable)
  - Wertsemantik
- UML: visibility name: type = defaultValue
  abh. von Detaillierung: nur name oder name: type

Abgeleitete Attribute und Assoziationen

- Attribute bzw. Assoziationen, die aus anderen berechnet werden
  Beispiel: /Dauer aus Anfangszeit und Endzeit
- aus Spezifikationsperspektive offen, was bei Implementierung gespeichert wird

Operationen

UML: visibility name(par-list): return-type {properties}

abh. von Detaillierung ggfs. nur name, …

Aggregation und Komposition

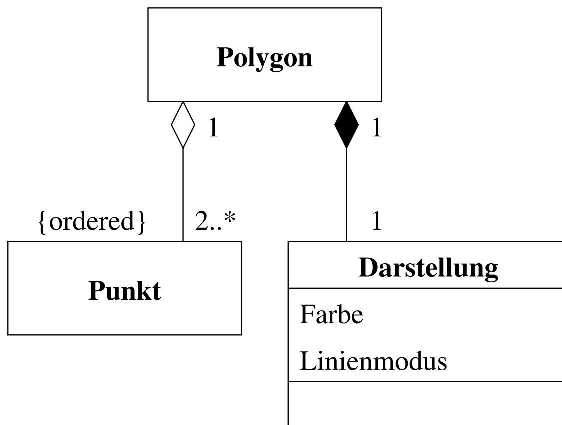
Aggregation

- Aggregation ist spezielle Assoziation (part-of)
- Bedeutung: Objekt ist Komponente von anderem Objekt
- in UML 2 nicht genau definiert ($\rightarrow$ nicht verwenden!)
- Notation: Kante mit weißer Raute als Pfeilspitze

Komposition

- Objekt ist nur Komponente genau eines anderen Objekts
- Komponente und Objekt "leben und sterben" gemeinsam
- Notation: Kante mit schwarzer Raute als Pfeilspitze

Beispiel: Aggregation und Komposition



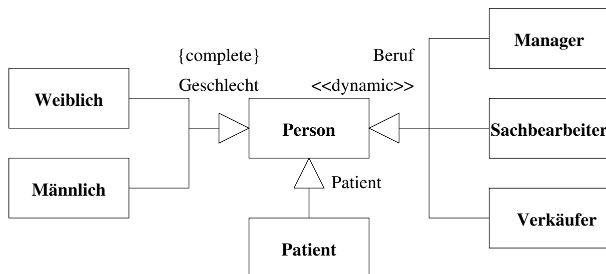
Mehrfach-Klassifikation

- OOP verlangt meist statische Einfachklassifikation, d.h.
  - Vererbung nur bzgl. eines Aspekts
  - Objekt kann Unterklasse nicht wechseln
- in Analyse manchmal angemessener:

  Klassifikation bzgl. mehrerer Aspekte
- beachte: Mehrfachklassifikation $\neq$ Mehrfachvererbung

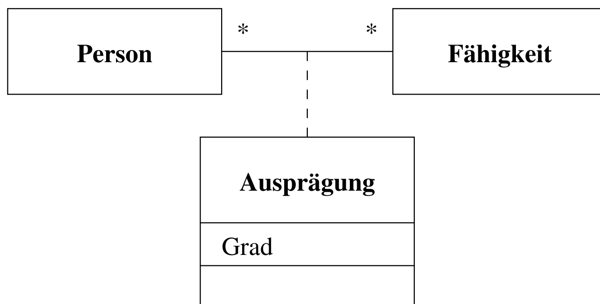
Dynamische Klassifikation

- in OOP: statische Klassifikation
- in Analyse ggf. sinnvoll: Objekte können Unterklasse wechseln



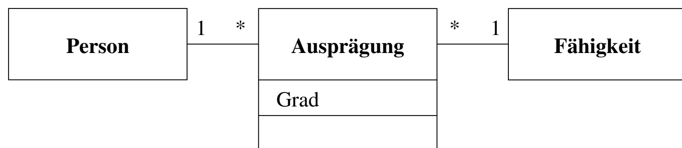
- für Implementierung müssen dynamische Klassifikation und Mehrfachklassifikation in statische Einfachklassifikation transformiert werden

Assoziationsklassen



- erlauben an Assoziation gebundene Attribute und Operationen
- Achtung: pro Paar assoziierter Objekte nur ein Assoziationsobjekt
- im Bsp.: jede Fähigkeit jeder Person hat nur eine Ausprägung

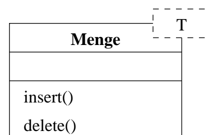
Alternative zu Assoziationsklasse



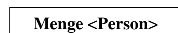
- hier: mehrere Ausprägungen einer Fähigkeit für jede Person möglich (hier ungewünscht)
- Programmiersprachen bieten keine direkte Realisierung von Assoziationsklassen
- AKs müssen daher durch obige Alternative implementiert werden
- ggf. ergänzt durch Zusatzcode zur Einhaltung der Zuordnungsrestriktion

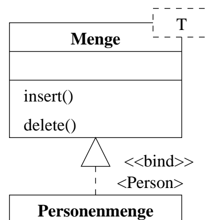
Parametrisierte Klassen

- erlauben parametrischen Polymorphismus (generische Typen)
- Einsatz vor allem bei Kollektionen




```
class Menge <T>{
  void insert(T element);
  void delete(T element);
}
⋮
```






```
Menge<Person> personen;
```

- im Unterschied zur Unterklassenbeziehung: unveränderte Attribute und Operationen
- Ersetzung zur Compilezeit ($\neq$ Spätes Binden)

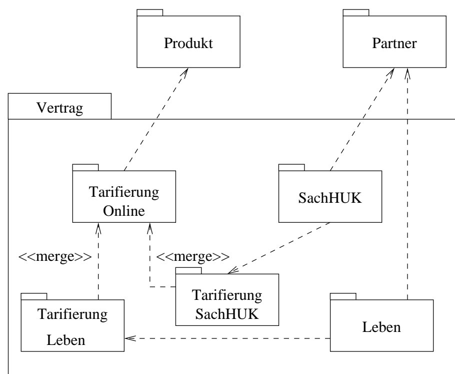
Sichtbarkeit

- in UML: Sichtbarkeit angelehnt an C++
- Attribute/Operationen:
  - public (UML: +): überall im Programm sichtbar
  - private (UML: -): nur in eigener Klasse sichtbar
  - protected (UML: #): sichtbar in Klasse und Unterklassen
- Problem: jede OO-Sprache hat eigenes Sichtbarkeitskonzept ($\rightarrow$ Anpassung)
- z.B. in Java: Sichtbarkeit package

Strukturierung großer Systeme

- Ziel: bessere Überschaubarkeit durch Zerlegung in Subsysteme
- Minimierung der Abhängigkeiten zwischen Subsystemen
- in UML hierzu:
  - Paket
    - Subsystem mit eigenem Klassendiagramm (ggfs. mit “Unterpaketen”)
    - dargestellt durch “Karteikarte”
  - Abhängigkeit zwischen Paketen
    - signalisiert, dass Änderung des Pakets eine Änderung des Nachbarpakets erfordern kann
    - dargestellt durch gestrichelten Pfeil
- Zerlegung in Pakete von Java unterstützt (Sichtbarkeit: package)

Beispiel: Zerlegung in Pakete



Abhängigkeit $P_2 - \rightarrow P_1$, z.B. wenn:

- Klasse C_2 in P_2 Nachricht an C_1 in P_1 sendet
- Operation in C_2 hat Argument vom Typ C_1
- Abhängigkeiten sind i.allg. nicht transitiv
- "Generalisierung" von Paketen möglich ($<<$ *merge* $>>$)

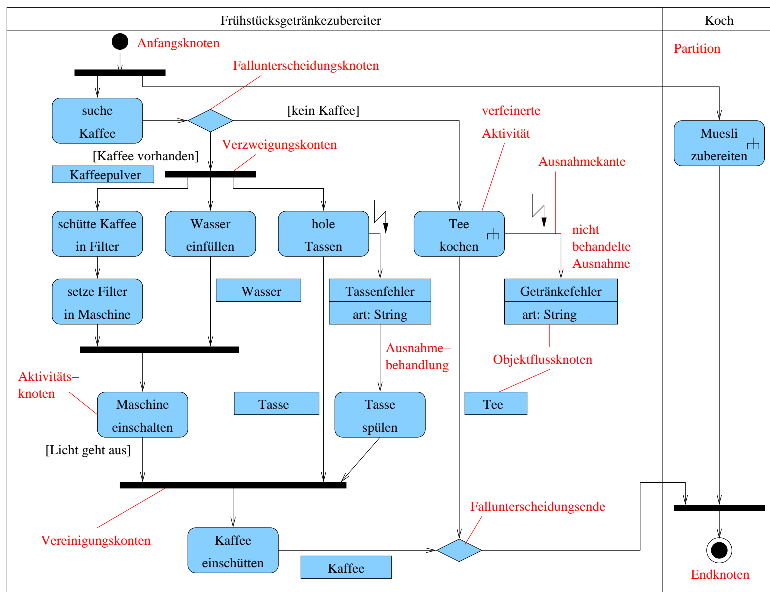
Zerlegung in Pakete

- gute Zerlegung schwierig
- top-down (oft besser!) oder bottom-up
- Faustregeln:
  - Zerlegung in logisch zusammengehörige Themenbereiche
  - Pakete sollen einzeln implementierbar und testbar sein
  - Vererbungshierarchie (wenn überhaupt) nur vertikal schneiden
  - keine Aggregation / Komposition durchtrennen
  - möglichst wenig Assoziationen auftrennen

4.3.3 Aktivitätsdiagramme

- "Erweiterung von Flussdiagrammen um Parallelität"
- Wurzeln in Petri-Netzen, Ereignisdiagrammen (Odell), Statecharts (Harel)
- geeignet zur Modellierung von Workflow, parallelen Aktivitäten
- geeignet zur Verfeinerung von Anwendungsfällen

Beispiel: Aktivitätsdiagramm



Bestandteile eines Aktivitätsdiagramms

- Anfangs- und Endknoten markieren Beginn und Ende eines Ablaufs
- Verzweigungs- und Vereinigungsknoten zeigen Aufspaltung bzw. zugehörige Zusammenführung des Kontrollflusses an
- Objektflussknoten repräsentieren übertragene Daten / Artefakte
- Fallunterscheidungsknoten modellieren Verzweigungen und Zusammenflüsse
- Partitionen zeigen den Bearbeiter an (optional)
- Ausnahmekanten (mit Blitz) zeigen Ausnahmen an
- jede verfeinerte Aktivität (→Symbol) wird durch eigenes Diagramm beschrieben
- Ausnahmen werden abgefangen oder zum übergeordneten Ablauf weitergegeben

4.3.4 OOA-Muster

- bei der Analyse wiederholen sich bestimmte Muster (patterns) immer wieder
- bei Erkennen solcher Muster: vorhandene Ansätze wiederverwenden

Muster 1: Abstrakte Oberklasse

- Situation: mehrere Klassen haben gleiche Attribute und Operationen
- Vorgehen: übernehme diese in abstrakte Oberklasse
- Bsp.: Kunde, Dozent $\rightarrow$ Person

Muster 2: Konkrete Oberklasse

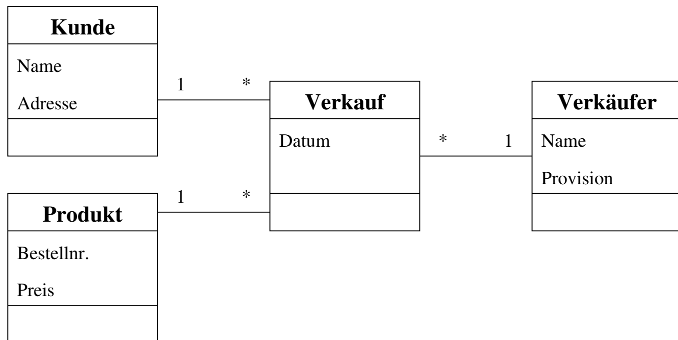
- Situation: Klasse *C* enthält nur Attribute und Operationen, die auch in anderen Klassen vorkommen
- Vorgehen: *C* wird Oberklasse

Muster 3: Assoziationen mit Eigenschaften

- Situation: über eine Assoziation müssen Informationen gespeichert werden
- Vorgehen: kopple Klassen über Klasse mit benötigten Informationen
  (alternativ: Assoziationsklasse)

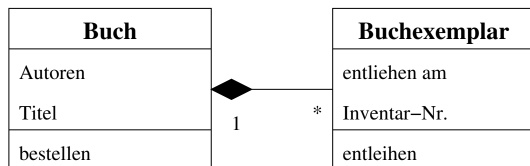
Muster 4: Koordination von Objekten

- Klassen auf hohem Abstraktionsniveau haben kaum eigene Attribute, sondern koordinieren das Zusammenwirken anderer Klassen



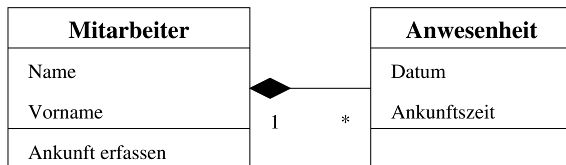
Muster 5: Exemplare

- Situation: Attributwerte bei vielen Objekten gleich
- Vorgehen: schaffe abstrakteren Begriff, der Gemeinsamkeiten “herausfaktorisiert”, und erstelle Aggregation oder Komposition



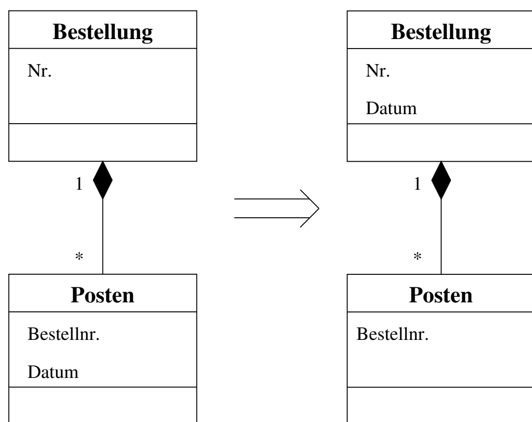
Muster 6: Ereignisse registrieren

- Situation: für ein Objekt müssen Ereignisse registriert werden
- Vorgehen: Klasse des Objekts durch Komposition mit Ereignis-Klasse verknüpfen



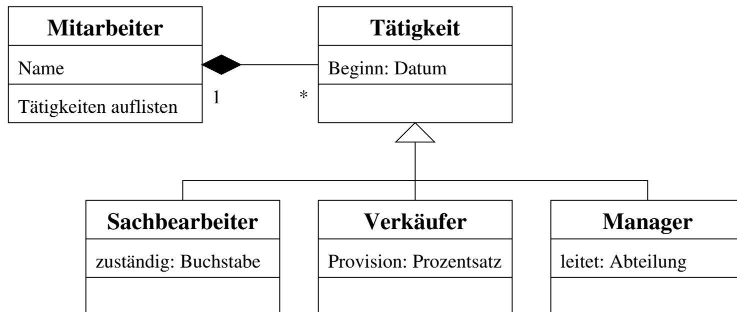
Muster 7: Hochziehen von Attributwerten

- Situation: bei Komposition: alle Komponenten haben gemeinsamen Attributwert
- Vorgehen: diesen Attributwert ins Gesamtobjekt hochziehen



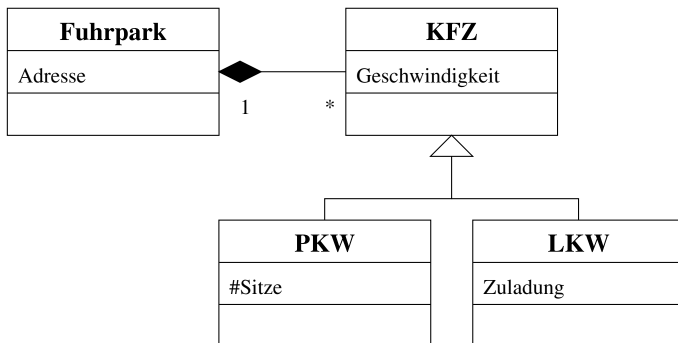
Muster 8: Historie dynamischer Klassifikation

- Situation: dynamische Klassifikation eines Objektes soll nachvollzogen werden
- Vorgehen: Klasse “aggregiert” Historie



Muster 9: “Potenz-Typen”

- Situation: Objekte des Potenz-Typs “aggregieren” Objekte von Unterklassen einer anderen Klasse



4.3.5 OOA-Methodik

1) datenorientierte Ansätze
   - Rumbaugh’91
   - Shlaer/Mellor’92
2) funktionsorientierte Ansätze
   - Wirfs-Brock’90
   - Jacobson’92 (use cases)
   - Rubin/Goldberg’92
3) Synthese von 1) und 2)
   - Coad/Yourdon’91
   - Booch’94

4.3.5.1 10 Schritte zum OOA-Modell nach Heide Balzert

1. Klassen finden
2. Assoziationen und Kompositionen finden
3. Attribute und Operationen für jede Klasse
4. Objektlebenszyklus erstellen
5. Vererbung einführen
6. interne Operationen finden
7. Operationen spezifizieren (z.B. in Pseudocode)
8. Vererbung überprüfen
9. Assoziationen und Kompositionen überprüfen
10. Zerlegung in Subsysteme

Schritt: Klassen finden

1. konkrete Objekte identifizieren

   z.B. in techn. Systemen: reale Objekte

   in kommerziellen Systemen: → Formulare
2. top-down:
   - verbale Anforderungen durchsuchen
   - hat Begriff Attribute und/oder Operationen?

   bottom-up:
   - Attribute (Daten) und Operationen sammeln
   - zu Klassen zusammenfassen
3. Klassenname: konkretes Substantiv, singular, beschreibt Gesamtheit der Objekte (keine Rolle)
4. bei unveränderlichen 1:1-Beziehungen Klassen ggfs. zusammenfassen, z.B.: Kunde, Teilnehmer

Schritt: Assoziationen und Kompositionen

- existieren permanente Beziehungen zwischen Objekten?
- in verbalen Anforderungen auf Verben achten
- bei fachlicher Über-/Unterordnung: Komposition
- Kommunikation zwischen Objekten → Assoziation
- Rollen ermitteln
- Schnappschuss oder Historie benötigt?
- Restriktionen?
- hat Assoziation eigene Attribute/Operationen?
  → z.B. Assoziationsklasse
- Kardinalitäten herausfinden (1, 0..1, *, . . .)

Seminar-Beispiel: Formular-Analyse

Anmeldeformular

Teilnehmer:

Titel | Vorname | Name

Veranstaltungnr. | Thema | Datum

Anmeldebestätigung und Rechnung an:

Titel | Vorname | Name

Firma | Straße

PLZ | Ort | Telefon

Seminarveranstaltung
Veranstaltungsnr.
Thema
Datum

Teilnehmer
Titel
Vorname
Name

Rechnungsempfänger
Titel
Vorname
Name
Firma
Straße
PLZ
Ort
Telefon

Klassen aus verbalen Anforderungen

Kunde
Kunden–Nr. Name Adresse Funktion Umsatz
Ersterfassung Änderung Löschen Adressaufkleber erstellen Serienbrief erstellen

Seminarbuchung
Angemeldet am Bestätigung am ...
Ersterfassung Änderung Löschen Anmeldebestätigung versenden Abmelden Benachrichtigung versenden Rechnung erstellen

Seminartyp
Thema ...
Ersterfassung Änderung Löschen

Mitarbeiter
Name Berechtigung Passwort

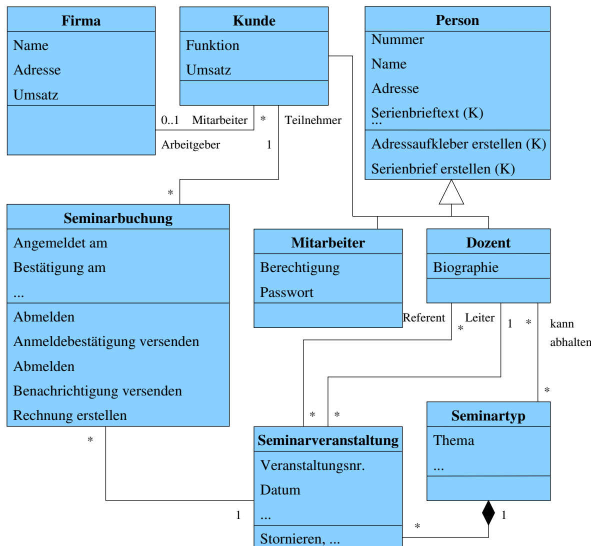
Firma
Name Adresse Umsatz

Seminarveranstaltung
Veranstaltungsnr. Datum ...
Ersterfassung... Stornieren Durchführung eintragen Teilnehmerliste erstellen

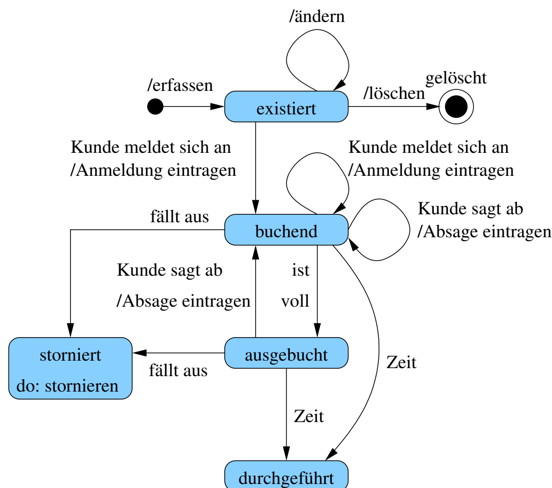
Dozent
Name Adresse Biographie ...
Ersterfassung ... Seminarveranstaltung zuordnen Seminartyp zuordnen Serienbrief erstellen

Zahlungsverzug
Rechnungsdatum Rechnungsbetrag
Zahlungsverzüge eintragen

Klassendiagramm im Beispiel (vereinfacht)



Automat zu Seminarveranstaltung



→ interne Operationen Anmeldung eintragen und Absage eintragen

Operationen als Pseudocode

```
Operation Anmeldung_eintragen(out Ergebnis):
  if Zustand = existiert or Zustand = buchend
    then Teilnehmer_anzahl_inkrementieren
         Ergebnis := ”ok”
    else Ergebnis := ”keine Anmeldung möglich”
  if Zustand = existiert then Zustand := buchend
  if Teilnehmer_anzahl = Teilnehmer_max then
     Zustand := ausgebucht
```

4.3.5.2 Methodik von Jacobson

3 Arten von Objekten:

1) Interface-Objekte

   zur Kommunikation mit Außenwelt

2) Kontroll-Objekte

   koordinieren andere Objekte

   angelehnt an Anwendungsfälle (use cases)

3) Entitätsobjekte

   kapseln Daten

   geringe Änderungswahrscheinlicheit

4.3.5.3 CRC-Karten

- CRC = Class-Responsibility-Collaboration (Wirfs-Brock)
- einer Klasse (statt Attribute und Operationen) zunächst Aufgaben und Kollaborationsklassen zuordnen
- Kollaborationsklassen helfen bei Erfüllung der Aufgaben
- $\leq$ 3 Aufgaben pro Karte (sonst teilen)