

# **Softwarepraktikum nach dem Wintersemester 13/14**

---

Prof. Dr. Markus Müller-Olm  
Institut für Informatik, WWU Münster

Nur für den persönlichen Gebrauch!  
Nicht zur Weitergabe bestimmt!

# Betreuer

Vorlesungen/Leitung:

Prof. Dr. Markus Müller-Olm  
Sebastian Kenter  
Benedikt Nordhoff  
Alexander Wenner

Tutoren:

Kai Kientopf  
Olaf Köhler  
Ivaylo Leonov  
Sebastian Lichtenfels  
Jan Staggenborg

# Lernziele

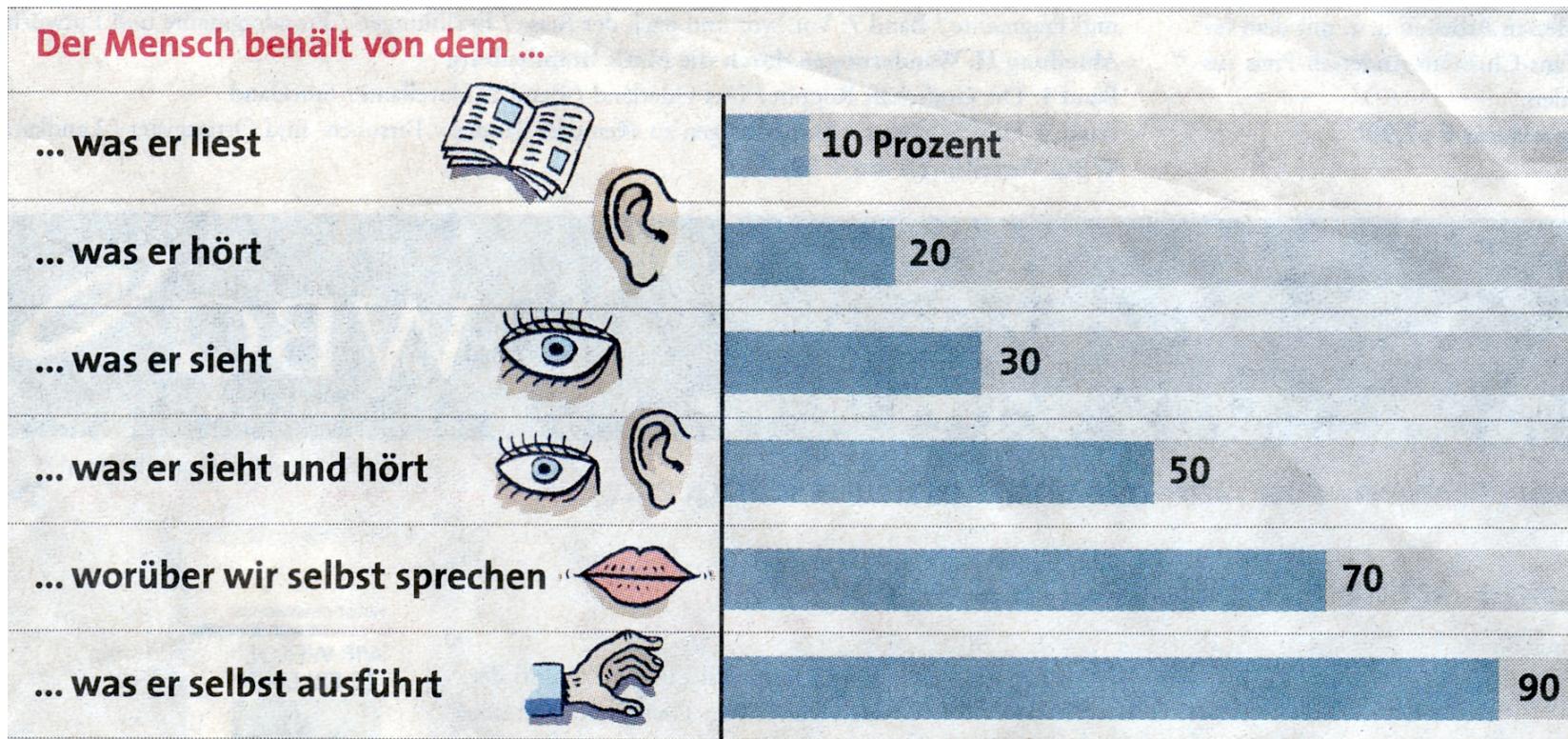
„Hard Skills“:

- OO Softwaretechnik und bewährte Programmierrichtlinien kennen und anwenden lernen
- Java Programmierkenntnisse vertiefen
- Softwareentwicklungswerkzeuge anwenden

„Soft Skills“:

- Gruppenarbeit
- Projektarbeit
  - Selbständige Informationsrecherche + Arbeitsaufteilung
- (enge) Deadlines einhalten
- Präsentation

# Handlungsorientiertes Lernen



# Aufgabenstellung

Eine Social Media-Plattform

OO-Softwarekonstruktionsprozess anwenden

## Zusätzlich als Einzelabgabe

- „Warm-Up“-Programmieraufgabe: Adressbuch

# Zum Ablauf

## Vorlesungen + GRUPPENARBEIT

Heute:

- Gruppeneinteilung
  - Liste ausfüllen
  - Gruppeneinteilung hängt gegen 14:00 an der Tür des M2 aus
- 11:15-12:45: Vorlesung
- 14:15: Erstes Gruppentreffen, Teambildungsaufgabe lösen
- ab 17:00: Präsentation der Lösung hier im M2

Weitere Vorlesungen:

- Di, 18.2., und Do, 20.2., jeweils 16:00 (s.t.!) - 17:30 im M2.
- Mi, 26.2., 9:00 (s.t.!) - 10:30 Uhr, voraussichtlich im M3.

# Schedule 1. Woche: 17.-21. Februar

| Mo, 17.2. | Di, 18.2.  | Mi, 19.2.                     | Do, 20.2.  | Fr, 21.2.                  |
|-----------|--|-------------------------------|--|----------------------------|
|           |  |                               |  |                            |
| 8:00      |  |                               |  |                            |
| 9:00      |  | 9:00:Tutoren<br>in A-Gruppen  |  |                            |
| 10:00     |  | 10:00:Tutoren<br>in B-Gruppen |  |                            |
| 11:00     | 11:15-12:45<br>Vorlesung 1                             | 11:00 Tutoren<br>in C-Gruppen |  |                            |
| 12:00     | Mittagspause   |                               |  |                            |
| 13:00     |  |                               | Gruppenarbeitsphase 1<br>(inkl. Bearbeitung der Warm-Up-Aufgabe) |                            |
| 14:00     | 14:15-17:00<br>Bearbeitung<br>Teambildungs-<br>aufgabe |                               |  |                            |
| 15:00     |  |                               |  |                            |
| 16:00     |  | 16:00-17:30<br>Vorlesung 2    |  | 16:00-17:30<br>Vorlesung 3 |
| 17:00     | 17:00-17:30<br>Präsentation                            |                               |  |                            |
| 18:00     |  |                               |  |                            |

# Schedule 2. Woche: 24.-28. Februar

| Mo, 24.2. | Di, 25.2.  | Mi, 26.2.                 | Do, 27.2.                  | Fr, 28.2.                  |
|-----------|--|---------------------------|----------------------------|----------------------------|
|           |  |                           |                            |                            |
| 8:00      |  |                           |                            |                            |
| 9:00      |  |                           |                            |                            |
| 10:00     |  | 9:00-10:30<br>Vorlesung 4 | 9:00-9:45:<br>Feedback A   |                            |
| 11:00     | Präsentation und<br>Besprechung<br>der Abgabe 1<br>&<br>Finalisierung der<br>Warm-up-<br>Programmier-<br>aufgabe | 10:30<br>Tutoren in A     | 10:00-10:45:<br>Feedback B |                            |
| 12:00     |  | 11:00<br>Tutoren in B     | 11:00-10:45:<br>Feedback C |                            |
| 13:00     |  | 11:30<br>Tutoren in C     |                            | Gruppenarbeits-<br>phase 2 |
| 14:00     |  |                           |                            |                            |
| 15:00     |  |                           |                            |                            |
| 16:00     |  |                           |                            |                            |
| 17:00     | 17:00: Abgabe 1  | 17:00: Abg. Warm-Up       |                            |                            |
| 18:00     |  |                           |                            |                            |

# Schedule 3. Woche: 3.-7. März

|       | Mo, 3.3.     | Di, 4.3.               | Mi, 5.3.        | Do, 6.3. | Fr, 7.3.               |
|-------|--------------|------------------------|-----------------|----------|------------------------|
| 8:00  |              |                        |                 |          |                        |
| 9:00  |              |                        |                 |          |                        |
| 10:00 |              |                        |                 |          |                        |
| 11:00 |              |                        |                 |          |                        |
| 12:00 | Rosen-montag |                        |                 |          |                        |
| 13:00 |              | Gruppenarbeits-phase 2 |                 |          | Gruppenarbeits-phase 3 |
| 14:00 | Kein SoPra!  |                        |                 |          |                        |
| 15:00 |              |                        |                 |          |                        |
| 16:00 |              |                        |                 |          |                        |
| 17:00 |              |                        | 17:00: Abgabe 2 |          |                        |
| 18:00 |              |                        |                 |          |                        |

# Schedule 4. Woche: 10.-14. März

|       | Mo, 10.3.                                   | Di, 11.3.        | Mi, 12.3.   | Do, 13.3.                   | Fr, 14.3. |
|-------|---|------------------|---|-----------------------------|-----------|
| 8:00  |   |                  |   |                             |           |
| 9:00  |   |                  |   | 9:00-10:30<br>Präsentation  |           |
| 10:00 |   |                  |   | Kaffeepause                 |           |
| 11:00 |   |                  | Prüfungsphase:<br>Einzelvorführungen<br>und<br>Prüfungsgespräch | 11:00-12:30<br>Präsentation |           |
| 12:00 |   |                  | &   | Mittagspause                |           |
| 13:00 | Gruppenarbeits-<br>phase 3<br>(Fortsetzung) |                  | Vorbereitung der<br>Abschlusspräsentation                       | 13:30-15:00<br>Präsentation |           |
| 14:00 |   |                  |   | Kaffeepause                 |           |
| 15:00 |   |                  |   | 15:30-17:00<br>Präsentation |           |
| 16:00 |   |                  |   |                             |           |
| 17:00 |   | 17:00: Endabgabe |   |                             |           |
| 18:00 |   |                  |   |                             |           |

# Wichtige Termine

- Mo, 24.2., 17:00 Uhr:  
Analysedokument (Pflichtenheft)
- Di, 25.2., 17:00 Uhr:  
Warm-Up-Aufgabe
- Mi, 5.3., 17:00 Uhr:  
Entwurfsdokument + Implementierung: Erste Iteration
- Di, 11.3., 17:00 Uhr:  
Endgültiges Entwurfsdokument, Implementierung und Benutzerhandbuch
- Mi, 12.3.:  
Prüfungsphase & Vorbereitung der Abschlusspräsentation  
Einzelpräsentation und –vorführung der einzelnen Gruppen  
Prüfungsgespräch
- Do, 13.3.:  
Abschlusspräsentation + Besuch der Abschlusspräsentation der anderen Gruppen
- ab Mo, 17.3., nach Absprache:  
Feedback durch die Betreuer

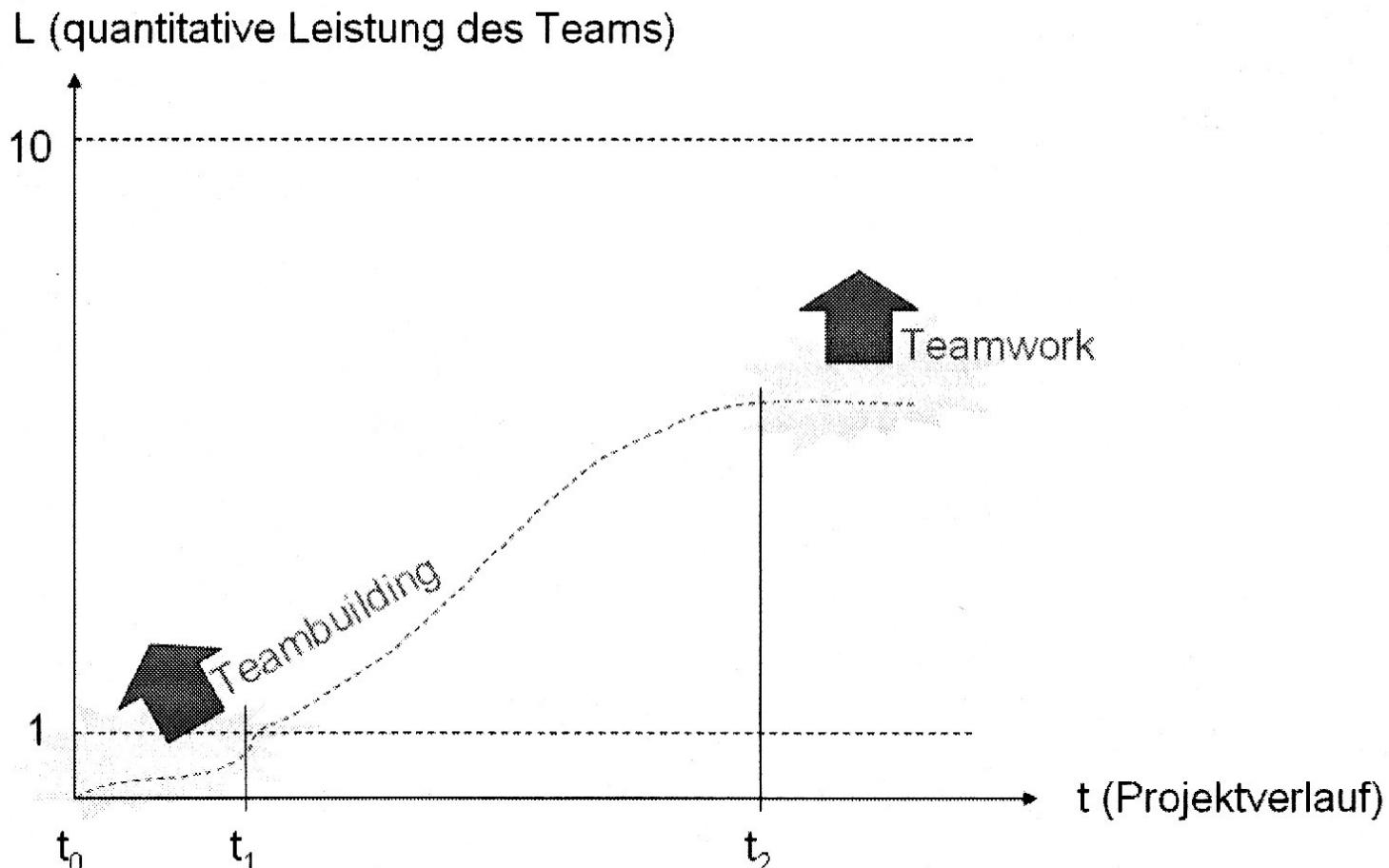
# Spielregeln I

- Fest zugeordneter Raum  
Raum steht Mo-Fr von 9:00-18:00 Uhr zur Verfügung
- Jeder Tutor betreut 2-3 Gruppen;  
Tutor ist partiell bei Gruppendiskussionen anwesend
- Rechnerarbeiten an eigenen Laptops (vorzugsweise) oder in den Pools
- Festgelegte Abgabedokumente und -termine

# Spielregeln II

- „Anwesenheitspflicht“: Anwesenheit von ca. 8 Stunden pro Tag:  
9:00 – 17:00 Uhr
- Gruppenabgaben + Einzelnoten
- Selbständige Aufgabenverteilung in der Gruppe
- Heuristik, Kreativität + Diskussion statt stark formalisiertes Vorgehen
- „Pair Programming“ bei der Implementierung
- Integriertes Testen mit JUnit
- Kontrakte spezifizieren und prüfen
- **Jeder Teilnehmer programmiert einen Teil des Systems !!**

# Teamkurve



Quelle: Fleischmann/Spies/Neumeyer, SEUH 2005

# Information + Kommunikation

- Webseite:
  - <http://cs.uni-muenster.de/sev/teaching/ws1314/sopra/>
  - aus Urheberrechtsgründen ggf. z.T. Passwort-geschützt (siehe Tafel)
- BSCW (Link auf Webseite)
- Vorlesungs-Folien:
  - Im BSCW
- Git zur Versionsverwaltung
- Literatur: Liste auf Webseite + Semesterapparat in der Bibliothek

# Zur Bewertung

In die Bewertung fließen u.a. ein:

- Qualität des Analyse- und des Entwurfsdokuments
- Konsistenz von Entwurfsdokument und Implementierung
- Qualität, Robustheit und Umfang des realisierten Prototyps
- Gruppenzusammenarbeit
- Einhalten der Implementierungs- und Test-Richtlinien

# Überblick

- Organisatorisches zum Sopra
- Einführung in das Sopra-Vorgehensmodell

Grundlage (hauptsächlich):

Bernd Brügge und Allen H. Dutoit  
*Objekt-Oriented Software Engineering*  
Pearson, 3rd Edition, 2010.

Begleitfolien zu diesem Buch von Bernd Brügge (TU München)

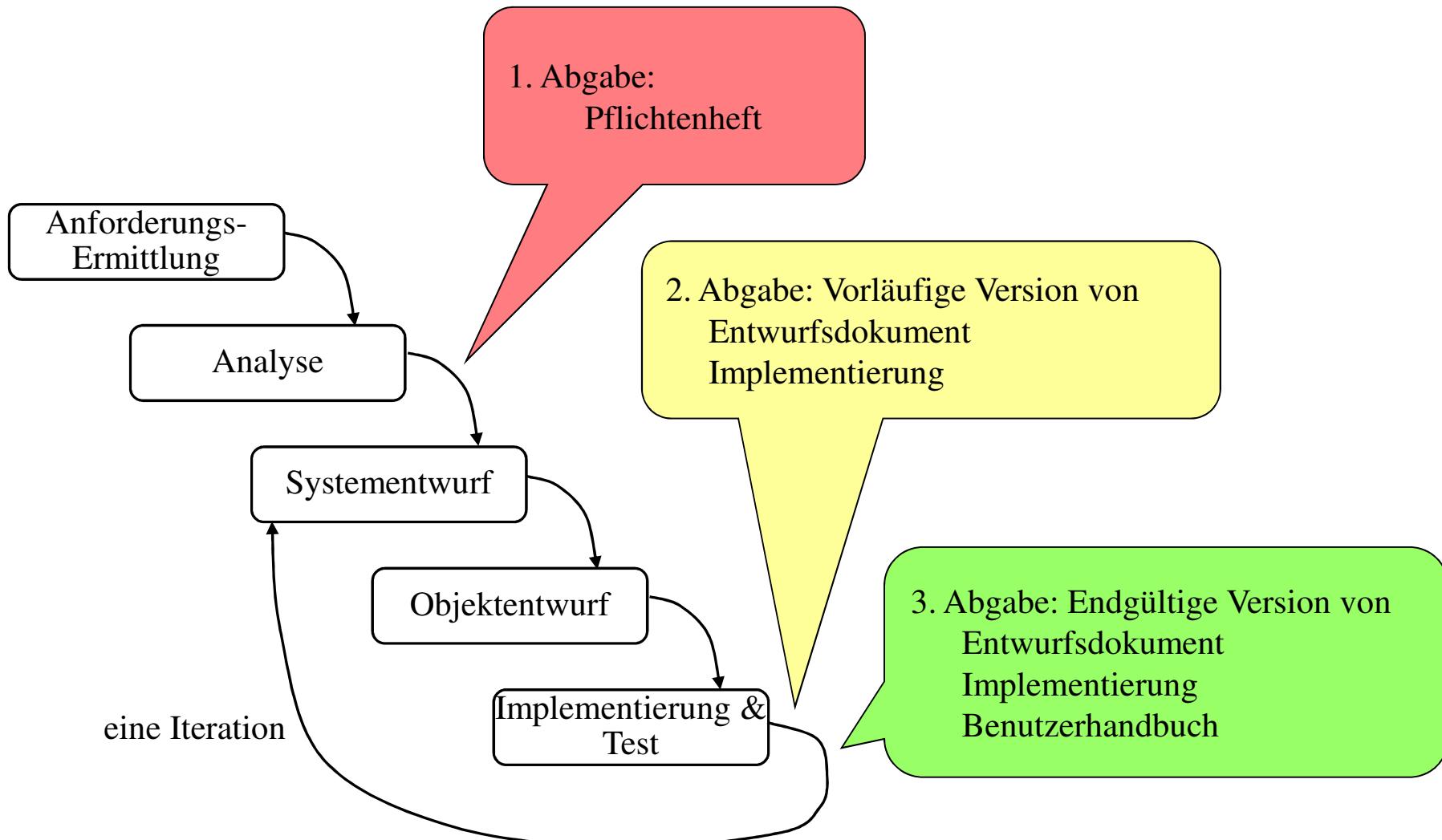
# Überblick

- Einführung in das Sopra-Vorgehensmodell

# Vorgehensmodelle

- Die universelle Methode der Softwareentwicklung gibt es nicht.
- Wahl der Vorgehensweise hängt von vielen Faktoren ab:
  - Anwendungsgebiet: Businessanwendung, Echtzeitsystem, Computerspiel,...
  - Programmierparadigma: Objektorientiert, funktional, ...
  - Qualität, Qualifikation, und Vorwissen der Softwareentwickler
  - Neu- oder Weiterentwicklungsprojekt
  - ...
- Hier im Sopra: Anwendungsfallgetriebene OOA und OOD  
(OOA = *object-oriented analysis*; OOD = *object-oriented design*)
- Ziele u.a.:
  - Techniken der OOA und OOD genauer kennen & anwenden lernen
  - Bewährte Programmierrichtlinien anwenden:  
Dokumentation, Integriertes Testen, Design-by-Contract, Versionsverwaltung

# SoPra-Vorgehensmodell



# Inhaltsverzeichnis des SoPra-Pflichtenhefts

## 1. Einführung

Textuelle Beschreibung von Zweck und Umfang des geplanten Systems

## 2. Vorgeschlagenes System

### 2.1. Übersicht:

Kurze textuelle Beschreibung

### 2.2. Funktionale Anforderungen

Kurze textuelle Beschreibung der funktionalen Anforderungen auf hohem Niveau

### 2.3. Nichtfunktionale Anforderungen

Textuelle Beschreibung der relevanten nichtfunktionalen Anforderungen

### 2.4. Systemmodelle

#### 2.4.1. Szenarien

2.4.2. Anwendungsfallmodell: Use case-Diagramm + Use case-Beschreibungen

2.4.3. Statisches Modell: Klassendiagramm für Entitätsklassen +  
Klassenbeschreibungen für Entitäts-, Grenz- und Kontrollklassen

2.4.4. Dynamisches Modell: Sequenzdiagramme + Zustandsdiagramm(e)

## 3. Glossar

Lexikonartige Auflistung und Kurzerklärung wichtiger Begriffe

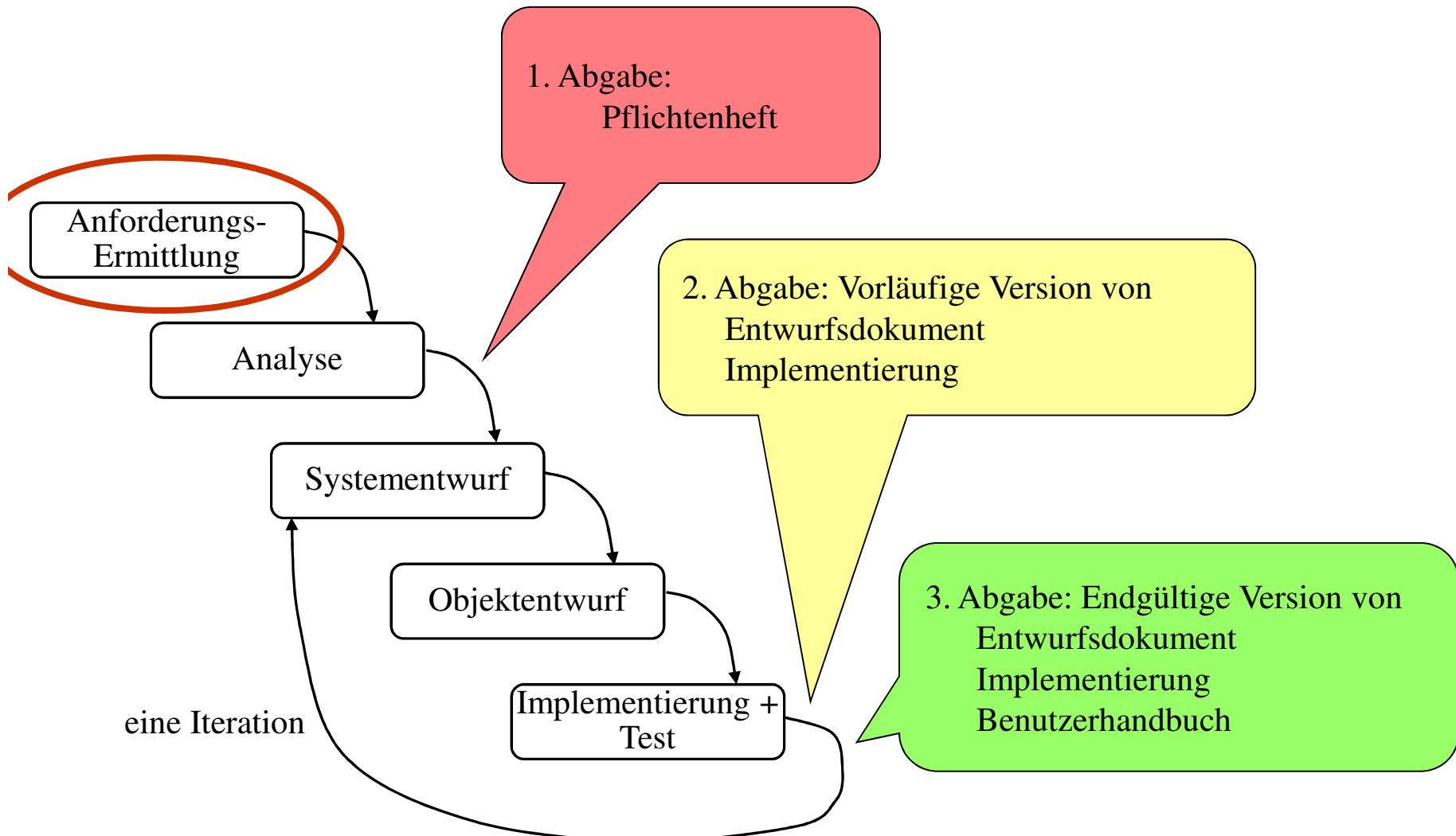
## Anhang A: GUI-Skizzen

GUI-Skizzen der zentralen Grenzklassen

[ 2.4.1 + 2.4.2 entstehen in der Anforderungsermittlungsphase ]

[ 2.4.3 + 2.4.4. entstehen in der Analysephase ]

# SoPra-Vorgehensmodell



# Ziel der Anforderungsermittlung (*requirements elicitation*)

- Aufgabenbeschreibung des Systems auf hohem Niveau aus Sicht der Benutzer: **Was** sollen wir bauen? Und **nicht**: **Wie** sollen wir es bauen
- Konsequenz: Beschäftigen uns mit dem Anwendungsgebiet (der „**Anwendungsdomäne**“) und **nicht** mit den Konzepten, die zur Realisierung benötigt werden (der „**Lösungsdomäne**“)  
Gilt auch noch für Analyse !
- Aufgabenbeschreibung dient in der Praxis insbesondere der (frühen) Kommunikation von Entwicklern, Kunden und Benutzern
- Dies soll verhindern, dass das „falsche System“ gebaut wird:  
Frühe „Validierung“
- Entstehende Dokumente sind Teil des Pflichtenheft
- Das Pflichtenheft ist Teil des Vertrags zwischen Kunde und Entwickler

# Zielmodelle der Anforderungsermittlung

## Funktionales Modell

Identifikation und Beschreibung von Szenarien, Akteuren und Anwendungsfällen:

- Szenarien (Text)
- Anwendungsfalldiagramm (*use case diagram, UML*)
- Anwendungsfallbeschreibungen (Text)

## Nichtfunktionales Modell

Erfasst nichtfunktionale Anforderungen, z.B. hinsichtlich:

- Bedienbarkeit, Zuverlässigkeit, Implementierung, Betrieb, Verpackung, Lizenierung etc. (Text)

Beide Modelle werden zusammen mit den Modellen der  
(Anforderungs-) Analyse Teil des „Pflichtenhefts“ !

# Anforderungsermittlung: Funktionales Modell

Identifikation von  
Akteuren und Anwendungsfällen  
mit Hilfe von Szenarien

# Requirement Engineering in der Praxis

- Problem:
  - Softwarespezialist kennt die „Lösungsdomäne“ aber nicht die „Anwendungsdomäne“
  - Anwender kennt die „Anwendungsdomäne“ aber nicht die „Lösungsdomäne“
- Es ist Aufgabe des Softwarespezialisten, Verständnis der Anwendungsdomäne zu gewinnen und die dazu notwendigen Informationen aus Kunden und Endanwendern „herauszufragen“.
- Dazu Interviews und Gespräche mit Kunde und Endanwendern:
  - Softwareentwickler hilft Kunde, die Anforderungen zu formulieren
  - Kunde hilft Softwareentwickler, die Anforderungen zu verstehen
  - Gutes Mittel zur Aufdeckung der Anforderungen:  
**Szenarien durchspielen und beschreiben**

# Szenarien

- “*A narrative description of what people do and experience as they try to make use of computer systems and applications*”  
[John M. Carroll, *Scenario-based Design*, Wiley, 1995]
- Konkrete, informelle Beschreibung eines einzelnen (oder weniger) Systemfeatures, das (die) durch einen einzelnen (oder wenige) Akteur(e) des Systems genutzt wird
- Szenarien können in verschiedenen Phasen des Software-Lebenszykluses zu verschiedenen Zwecken verwendet werden:
  - **Anforderungsermittlung:** As-is Szenario, **visionäres Szenario**
  - **Akzeptanztest durch den Kunden:** Evaluations-Szenario
  - **Systemeinführung:** Training-Szenario

# Typen von Szenarien

## As-is Szenario:

Zur Beschreibung einer aktuellen Situation. Wird in Re-Engineering Projekten benutzt. Der Benutzer beschreibt das System.  
Beispiel: Beschreibung einer Briefschach-Partie.

## Visionäres Szenario:

**Zur Beschreibung eines zukünftigen Systems. Wird bei Neusystemen und Re-Engineering-Projekten benutzt.**

**Benötigt häufig Kooperation von Benutzer und Entwickler:**

**Beispiel: Beschreibung eines interaktiven Internet-basierten Schach-Turniers.**

## Evaluations-Szenario

Dient der Evaluation des fertigen Systems.

Beispiel: Vier Benutzer spielen Internet-basiert ein Schach-Turnier.

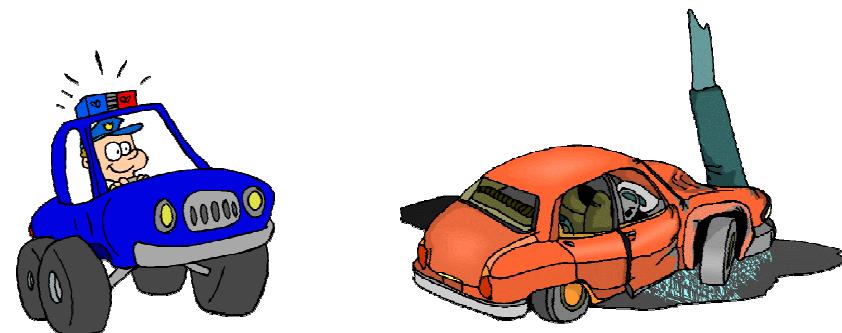
## Training-Szenario:

Schrittweise beschriebenes Beispiel zur Benutzung eines Systems für einen Anfänger

Beispiel: Wie man ein Internet-Schach-Turnier durchführt.

# Beispiel: Notruf-Management-System

- Betrachten FRIEND, ein Beispiel aus [Bruegge/Dutoit, 2010]:  
FRIEND ist ein Notruf-Management-System
- System soll helfen die Tätigkeiten von Polizisten, Feuerwehrmännern etc.  
im Außendienst mit einer Notrufzentrale zu koordinieren
- Akteure sind hier:
  - Außenbeamte (Polizist, Feuerwehrmann, etc. vor Ort)
  - Dienstleiter (sitzt in der Notrufzentrale und  
koordiniert den Einsatz)



# Ansatzpunkte für Szenarien für das Notruf-Management-System

- Was muss man tun, um den Vorfall “Katze im Baum” zu melden?
- Was muss man tun, wenn eine Person “Feuer im Kaufhaus” meldet?
- Wer ist alles an der Meldung eines Vorfalls beteiligt?
- Was tut das System, wenn kein Streifenwagen verfügbar ist? Oder dann, wenn der Streifenwagen auf dem Weg zum “Katze im Baum”-Vorfall einen Unfall hat?
- Was muss man tun, wenn sich der Vorfall “Katze im Baum” zum Vorfall “Großmutter ist von der Leiter gefallen” entwickelt hat?
- Kann das System mit der gleichzeitigen Meldung “Feuer im Kaufhaus” umgehen?”



# Beispiel-Szenario: Feuer im Kaufhaus

- Hans fährt in seinem Streifenwagen die Hauptstraße entlang und bemerkt Rauch, der aus einem Kaufhaus kommt. Seine Partnerin Monika aktiviert die Funktion “MeldeNotfall” auf ihrem Meldegerät.
- Monika gibt die Adresse des Gebäudes, eine kurze Beschreibung der Lage des Brandes und das Ausmaß des Notfalls ein. Zusätzlich zu einer Feuerwehreinheit fordert sie einige Krankenwagen an, da das Gebiet ziemlich belebt zu sein scheint. Sie schickt ihre Eingabe ab und wartet auf Bestätigung.
- Franz, der Dienstleiter, wird auf den Notfall durch einen Signalton seiner Workstation aufmerksam gemacht. Er überprüft die Informationen, die von Monika gesendet wurden, und bestätigt die Meldung. Er beordert eine Feuerwehreinheit und zwei Krankenwagen zum Einsatzort und sendet die voraussichtliche Ankunftszeit an Monika.
- Monika erhält die Bestätigung und die voraussichtliche Ankunftszeit.

# Beobachtungen zum Beispiel-Szenario

- Konkretes Szenario:
  - Beschreibt eine einzelne Instanz eines Vorfalls
  - Beschreibt **nicht** alle möglichen Umstände, unter denen über eine Feuer berichtet werden muss
- Beteiligte Akteure:
  - Hans, Monika, Franz

# Heuristiken zum Finden von Akteuren und Szenarien

- Problemstellung durchdenken und mit Kommilitonen/Kollegen diskutieren
- Fragenkataloge auf den Folgefolien durchgehen
- In der Praxis:
  - Gespräche mit Kunden und Endanwendern führen
  - Ggf.: Vorläufersysteme untersuchen
- Heuristiken und Fragenkataloge bieten nur Ansatzpunkte und garantieren keine Vollständigkeit  
⇒ Heuristiken nie schematisch und unkritisch verwenden!

# Fragen zur Identifikation von Akteuren (nach [Bruegge/Dutoit])

- Welche Benutzergruppen werden vom System unterstützt?
- Welche Benutzergruppen nutzen die Hauptfunktionen?
- Welche Benutzergruppen führen Sekundärfunktionen wie Wartung und Verwaltung durch?
- Mit welcher externen Hard- oder Software wird das System interagieren?

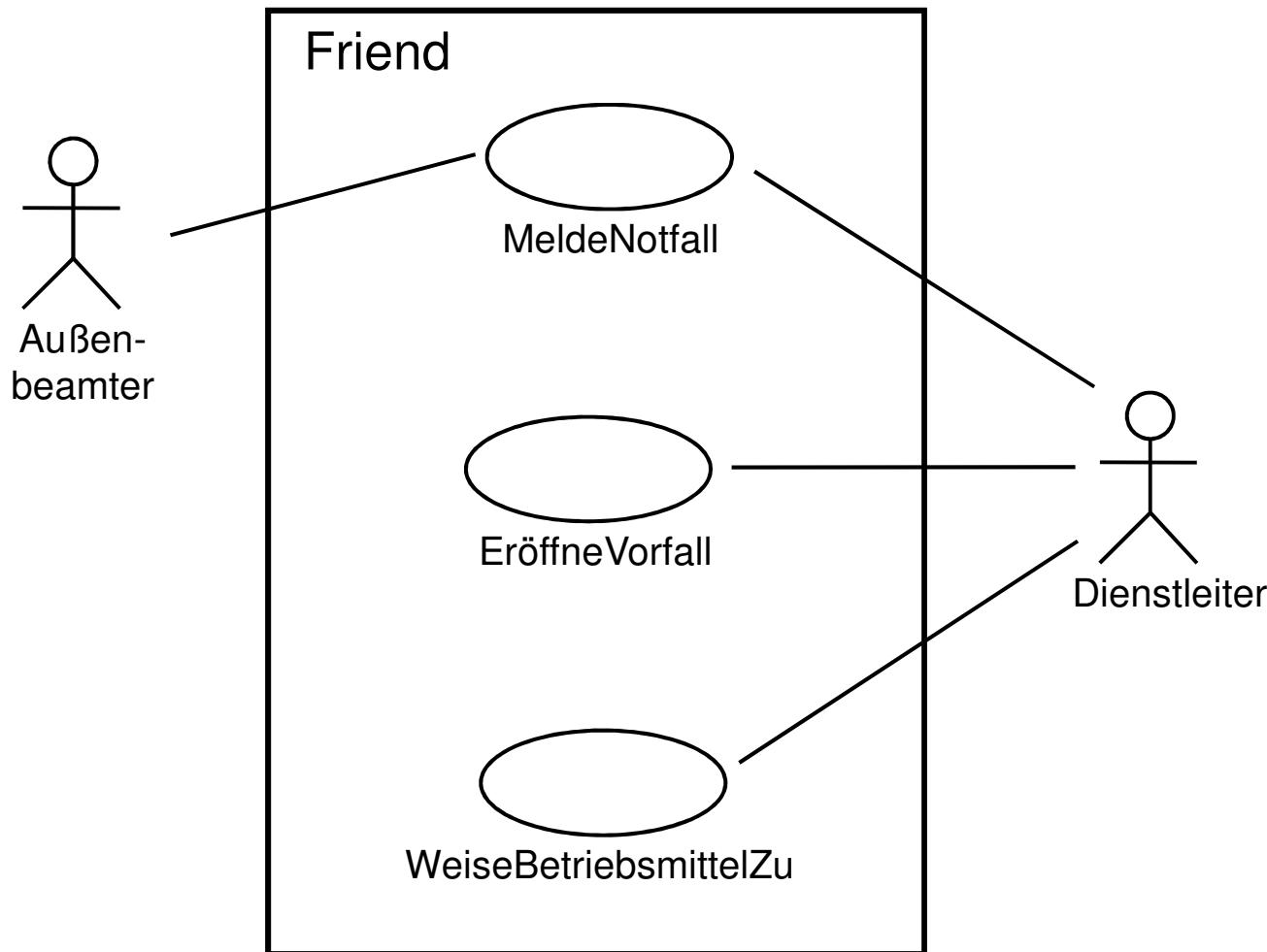
# Fragen zur Identifikation von Szenarien (nach [Bruegge/Dutoit])

- Was sind die Aufgaben, die die Akteure vom System erwartet?
- Auf welche Informationen greifen die Akteure zu? Wer generiert diese Informationen? Können Sie modifiziert oder entfernt werden? Von wem?
- Über welche externen Änderungen müssen die Akteure das System informieren? Wie oft und wann?
- Über welche Ereignisse muss das System die Akteure informieren?

# Ziel nach Formulierung der Szenarien:

- Aus den konkreten Verwendungen des Systems in den Szenarien **Anwendungsfälle identifizieren**:
  - Beispiel: “MeldeNotfall” im ersten Absatz ist Kandidat für eine Anwendungsfälle.
- Bemerkung: Der Anwendungsfall “MeldeNotfall” repräsentiert jetzt alle möglichen Instanzen, einen Notfall zu melden.
- Konstruiere ein Anwendungsfalldiagramm und beschreibe jeden Anwendungsfall im Detail

# Beispiel: Teil des Anwendungsfallmodells des Vorfallmanagement-Systems



# Erinnerung: Form einer AF-Beschreibung

Name des Anwendungsfalls

Beteiligte Akteure:

Beschreibung der am Anwendungsfall beteiligten Akteure

Anfangsbedingungen:

Was muss am Anfang gelten? Wann wird der Anwendungsfall gestartet ?

Ereignisfluss:

Beschreibung des Ablaufs im Normalfall: Freie Form, informelle natürliche Sprache

Abschlussbedingungen

Was gilt am Ende? Unter welchen Bedingungen ist der AF vollständig abgearbeitet?

Ausnahmen

Beschreibung, was passiert, wenn irgend etwas schief geht

Spezielle Anforderungen

Nichtfunktionale Anforderungen, Einschränkungen

# Beispiel: MeldeNotfall-Anwendungsfall

Name des Anwendungsfalls: MeldeNotfall

Beteiligte Akteure:

Außenbeamter (im Beispiel Monika), Dienstleiter (im Beispiel Franz)

Anfangsbedingungen:

Der Außenbeamte ist beim System angemeldet.

Ereignisfluss:

SIEHE NÄCHSTE FOLIE

Abschlussbedingungen

Der Außenbeamte hat eine Bestätigung und eine Nachricht über die vom Dienstleiter ausgewählte Reaktion erhalten

ODER

Der Außenbeamte hat eine Erklärung erhalten, warum seine Anforderung nicht bearbeitet werden konnte.

Ausnahmen

Der Außenbeamte wird sofort informiert, wenn seine Meldegerät keine Verbindung zur Zentrale hat.

Der Dienstleiter wird sofort informiert, wenn das Meldegerät eines angemeldeten Außenbeamten den Kontakt zur Zentrale verliert.

Spezielle Anforderungen

Der Bestätigung auf den Bericht des Außenbeamten erfolgt innerhalb von 30 Sekunden.

Die Nachricht über die ausgewählte Lösung kommt innerhalb von 30 Sekunden an, nachdem der Dienstleiter sie gesendet hat.

# **Beispiel: MeldeNotfall-Anwendungsfall**

## **Ereignisfluss**

1. Der Außenbeamte aktiviert die “MeldeNotfall”-Funktion auf seinem Meldegerät. Das System antwortet durch Bereitstellung eines Formulars für den Außenbeamten.
2. Der Außenbeamte füllt die Formularfelder “Grad des Notfalls”, “Notfalltyp”, “Ort” und “Kurzbeschreibung der Situation” aus. Er beschreibt auch mögliche Lösungen für die Notfallsituation. Sobald er das Formular ausgefüllt hat, schickt er es ab. Das System benachrichtigt den Dienstleiter.
3. Der Dienstleiter beurteilt die eingegebene Information und erzeugt einen Vorfall in der Datenbank, indem er den EröffneVorfall-Anwendungsfall aufruft. Der Dienstleiter wählt eine Lösung aus und bestätigt den Bericht. Das System zeigt dem Akteur Außenbeamter die Bestätigung und die ausgewählte Lösung an.

# Typische Reihenfolge beim Formulieren von Anwendungsfällen

- Erster Schritt: Anwendungsfall benennen
  - Name des Anwendungsfalls: MeldeNotfall
- Zweiter Schritt: Akteure Identifizieren
  - Konkrete Namen im Szenario (“Monika”) zu den beteiligten Akteuren (“Außenbeamter”) abstrahieren
  - Beteiligte Akteure:
    - Außenbeamter (Monika im Szenario)
    - Dienstleiter (Franz im Szenario)
- Dritter Schritt: Ereignisfluss beschreiben
  - informelle natürlichsprachliche Beschreibung
- Danach die übrigen Teile ausfüllen

# Beziehungen zwischen Anwendungsfällen

- Neben Akteuren und Anwendungsfällen werden in einem AF-Diagramm auch Assoziationen (Beziehungen) zwischen Anwendungsfällen dargestellt
- Wichtige Typen von AF-Assoziationen:

**Includes:** AF benutzt einen anderen

ermöglicht Wiederverwendung und Ausgliederung unabhängig nutzbarer Teilfunktionalität

**Extends:** Ein AF erweitert einen anderen

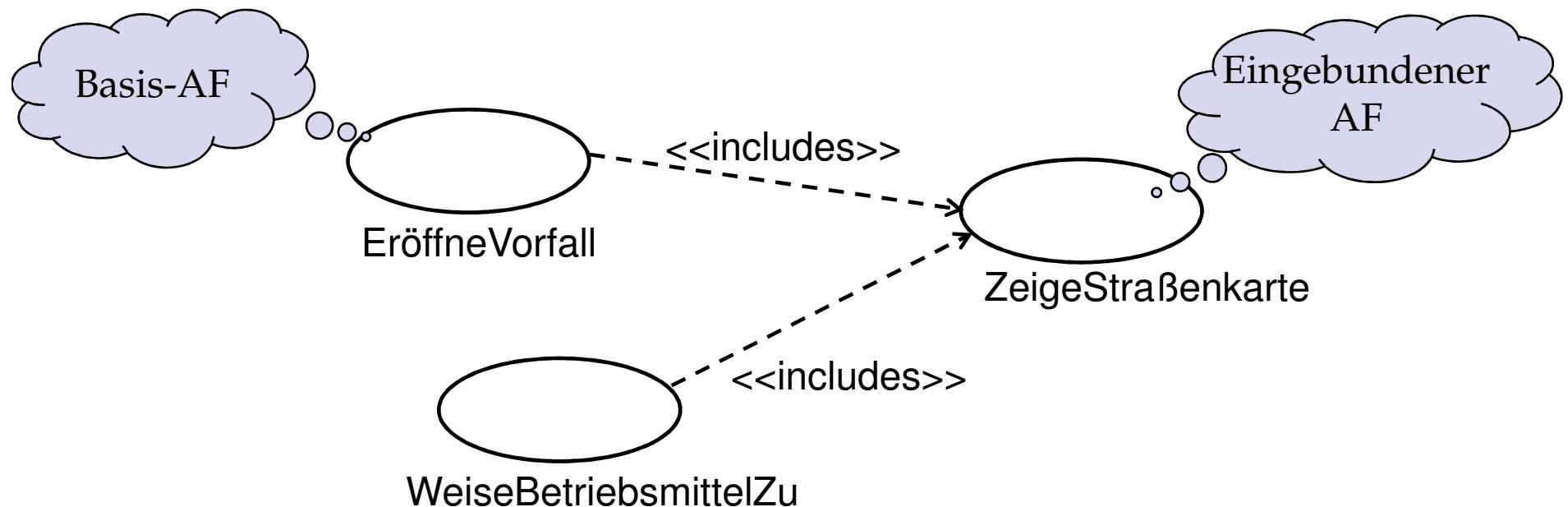
zur übersichtlichen Beschreibung von Sonder- oder Ausnahmefällen

**Generalisierung:**

Ein abstrakter Anwendungsfall hat verschiedene Spezialisierungen

# Beispiel: <<includes>> ermöglicht Wiederverwendung existierender Funktionalität

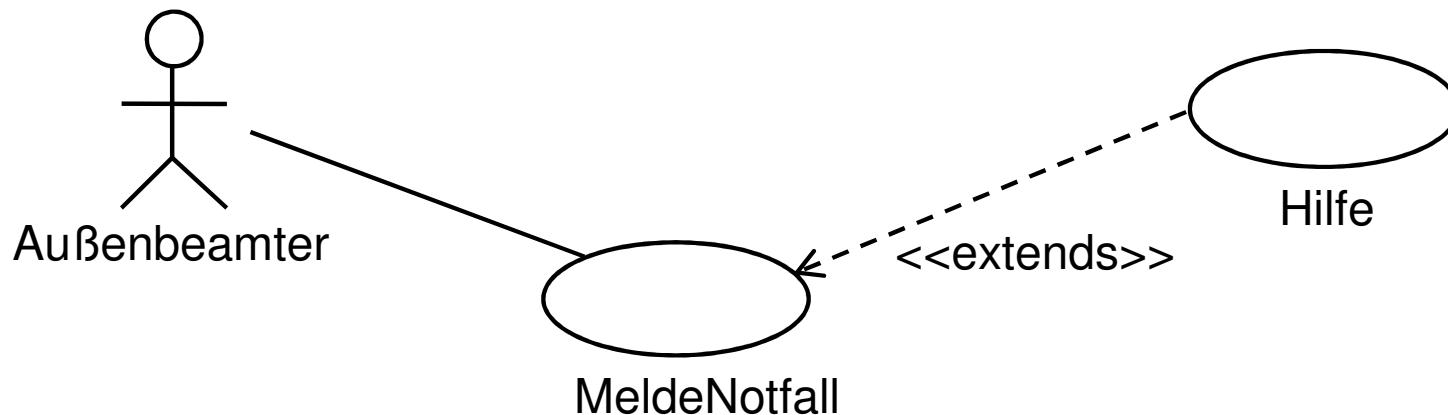
Der AF “ZeigeStraßenkarte” beschreibt Verhalten, das sowohl vom AF “EröffneVorfall” als auch vom AF “WeiseBetriebsmittelZu” benutzt wird. Das Verhalten des AF “ZeigeStraßenkarte”, dass vielleicht ursprünglich nur bei EröffneVorfall beschrieben war, kann also durch Auslagern in den AF “ZeigeStraßenkarte” für den AF “WeiseBetriebsmittelZu” wiederverwendet werden.



# <<extends>> beschreibt Sonder- und Ausnahmefälle

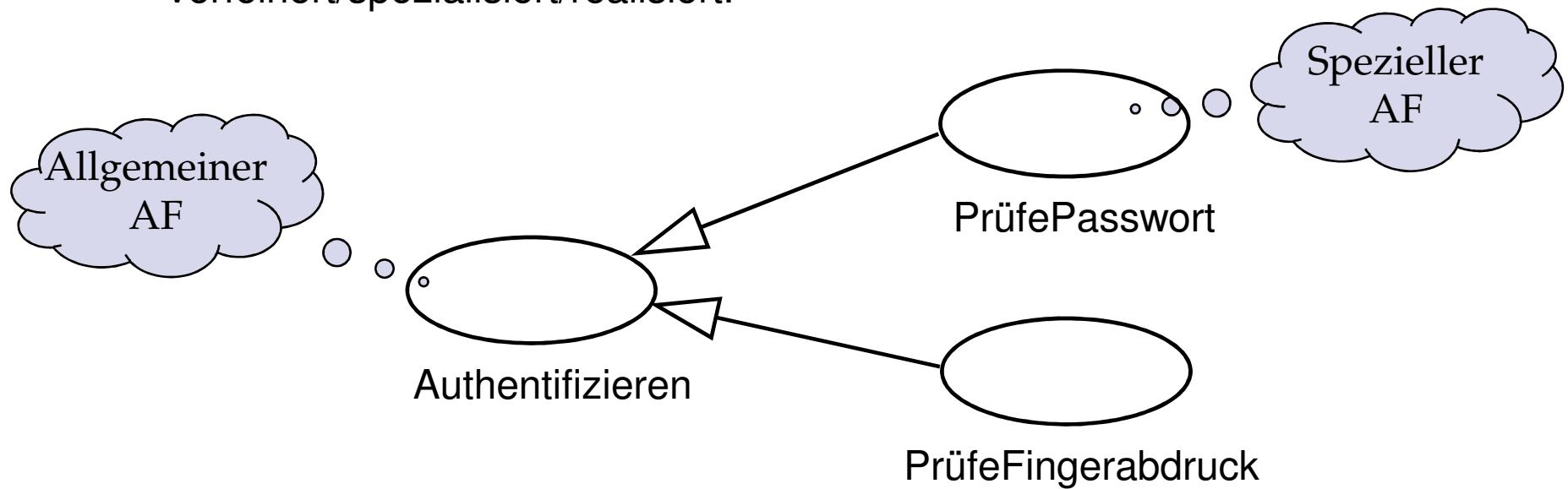
Der AF "MeldeNotfall" ist in sich vollständig.

Er wird aber für spezifische Szenarien, in denen der Außenbeamte selbst Hilfe benötigt, durch den AF "Hilfe" erweitert



# Generalisierungsbeziehung zwischen Anwendungsfällen

- Ein Anwendungsfall kann allgemeineren Anwendungsfall durch Hinzufügen/Festlegen von Einzelheiten spezialisieren. Dies wird durch eine Generalisierungsbeziehung zwischen dem allgemeinen und dem speziellen Anwendungsfall modelliert
- Beispiel:
  - Allgemeiner Anwendungsfall "Authentifizieren" wird durch konkretere Anwendungsfälle "PasswortPrüfen" und "FingerabdruckPrüfen" verfeinert/spezialisiert/realisiert.



# Anforderungsermittlung:

## Nichtfunktionales Modell

# Nichtfunktionales Modell

- Neben den funktionalen Anforderungen ist die Erfüllung nicht-funktionaler Anforderungen sehr wichtig für Akzeptanz des Systems
- Nicht-funktionale Anforderungen beziehen sich z.B. auf
  - Benutzerfreundlichkeit (z.B. textuelles Interface oder GUI, spezielles look-and-feel,...)
  - Zuverlässigkeit und Verfügbarkeit
  - Leistungsanforderungen
  - Unterstützung
  - Implementierung
  - Schnittstellen
  - Betrieb
  - Installation
  - Rechtliches (z.B. Lizensierung)
- Das meiste davon ist zwar in der Praxis äußerst wichtig aber für das Sopra weniger relevant...
- Die folgenden Folien zeigen Beispielfragen zur Identifikation nicht-funktionaler Anforderungen  
⇒ **Diskutieren, was für SOPRA relevant ist und entsprechende Anforderungen ins Pflichtenheft aufnehmen!**

# Nichtfunktionales Modell

- Neben den funktionalen Anforderungen ist die Erfüllung nicht-funktionaler Anforderungen sehr wichtig für Akzeptanz des Systems
- Nicht-funktionale Anforderungen beziehen sich z.B. auf
  - Benutzerfreundlichkeit (z.B. textuelles Interface oder GUI, spezielles look-and-feel,...)
  - Zuverlässigkeit und Verfügbarkeit
  - Leistungsanforderungen
  - Unterstützung
  - Implementierung
  - Schnittstellen
  - Betrieb
  - Installation
  - Rechtliches (z.B. Lizensierung)
- Nicht-funktionale Anforderungen sind in der Praxis äußerst wichtig, einige sind aber für das Sopra weniger relevant...
- Die folgenden Folien zeigen Beispielfragen zur Identifikation nicht-funktionaler Anforderungen
  - ⇒ **Nichtfunktionale Anforderungen diskutieren**  
**Im Pflichtenheft: Für Sopra relevante Anforderungen beschreiben, Irrelevanz der übrigen Anforderungen begründen**

# Fragen zur Ermittlung nicht-funktionaler Anforderungen 1 (nach Bruegge/Dutoit)

## Benutzerfreundlichkeit

- Was ist der Kenntnisstand der Benutzer?
- Welche Normen sollen für die Benutzerschnittstelle verwendet werden?
- Welche Dokumentation soll Benutzern übergeben werden?

## Zuverlässigkeit, Verfügbarkeit, Robustheit

- Wie zuverlässig, verfügbar und robust soll das System sein?
- Ist ein Systemneustart im Falle eines Fehlers akzeptabel?
- Wie viele Daten darf das System verlieren?
- Wie soll das System mit Ausnahmen umgehen?
- Gibt es Betriebssicherheitsanforderungen?
- Gibt es Datenschutzanforderungen?

# Fragen zur Ermittlung nicht-funktionaler Anforderungen 2 (nach Bruegge/Dutoit)

## Leistungsanforderungen

- Wie schnell muss das System reagieren?
- Gibt es zeitkritische Benutzeroberaufgaben?
- Wie viele Anwender soll das System gleichzeitig unterstützen?
- Wie groß sind Datenspeicher vergleichbarer Systeme?
- Welche Latenzzeit wird noch akzeptiert?

## Unterstützung

- Welche Erweiterungen des Systems sind geplant?
- Wer wartet das System?
- Gibt es Pläne, das System auf andere Hard- oder Softwareumgebungen zu portieren?

# Fragen zur Ermittlung nicht-funktionaler Anforderungen 3 (nach Bruegge/Dutoit)

## Implementierung

- Gibt es Beschränkungen auf gewisse Hardware- und oder Softwareplattform?
- Gibt es Beschränkungen durch die Wartungsarbeitsgruppe?
- Gibt es Beschränkungen durch die Testarbeitsgruppe?

## Schnittstelle

- Soll das System mit anderen bereits vorhandenen Systemen interagieren?
- Wie werden Daten in das System exportiert/aus dem System exportiert?
- Welche Normen sollen vom System unterstützt werden?

# Fragen zur Ermittlung nicht-funktionaler Anforderungen 4 (nach Bruegge/Dutoit)

## Betrieb

- Wer kümmert sich um das laufende System?

## Installation

- Wer installiert das System?
- Wie viele Installationen sind geplant?
- Gibt es zeitliche Beschränkungen für die Installation?

## Rechtliches

- Wie soll das System lizenziert werden?
- Wer trägt die Verantwortung bei Systemfehlern?
- Werden durch die Nutzung bestimmter Algorithmen oder Komponenten Lizenzen fällig?

# Aufgabe ab morgen

- Szenarien finden/diskutieren/aufschreiben
- Akteure und Anwendungsfälle identifizieren und beschreiben
- Nicht-funktionale Anforderungen diskutieren und beschreiben, ggf. begründen, warum einzelne Anforderungen für das Sopra nicht relevant sind
- Parallel dazu: Warm-up-Programmieraufgabe beginnen

# Inhaltsverzeichnis des SoPra-Pflichtenhefts

## 1. Einführung

Textuelle Beschreibung von Zweck und Umfang des geplanten Systems

## 2. Vorgeschlagenes System

### 2.1. Übersicht:

Kurze textuelle Beschreibung

### 2.2. Funktionale Anforderungen

Kurze textuelle Beschreibung der funktionalen Anforderungen auf hohem Niveau

### 2.3. Nichtfunktionale Anforderungen

Textuelle Beschreibung der relevanten nichtfunktionalen Anforderungen

### 2.4. Systemmodelle

#### 2.4.1. Szenarien

2.4.2. Anwendungsfallmodell: Use case-Diagramm + Use case-Beschreibungen

2.4.3. Statisches Modell: Klassendiagramm für Entitätsklassen +  
Klassenbeschreibungen für Entitäts-, Grenz- und Kontrollklassen

2.4.4. Dynamisches Modell: Sequenzdiagramme + Zustandsdiagramm(e)

## 3. Glossar

Lexikonartige Auflistung und Kurzerklärung wichtiger Begriffe

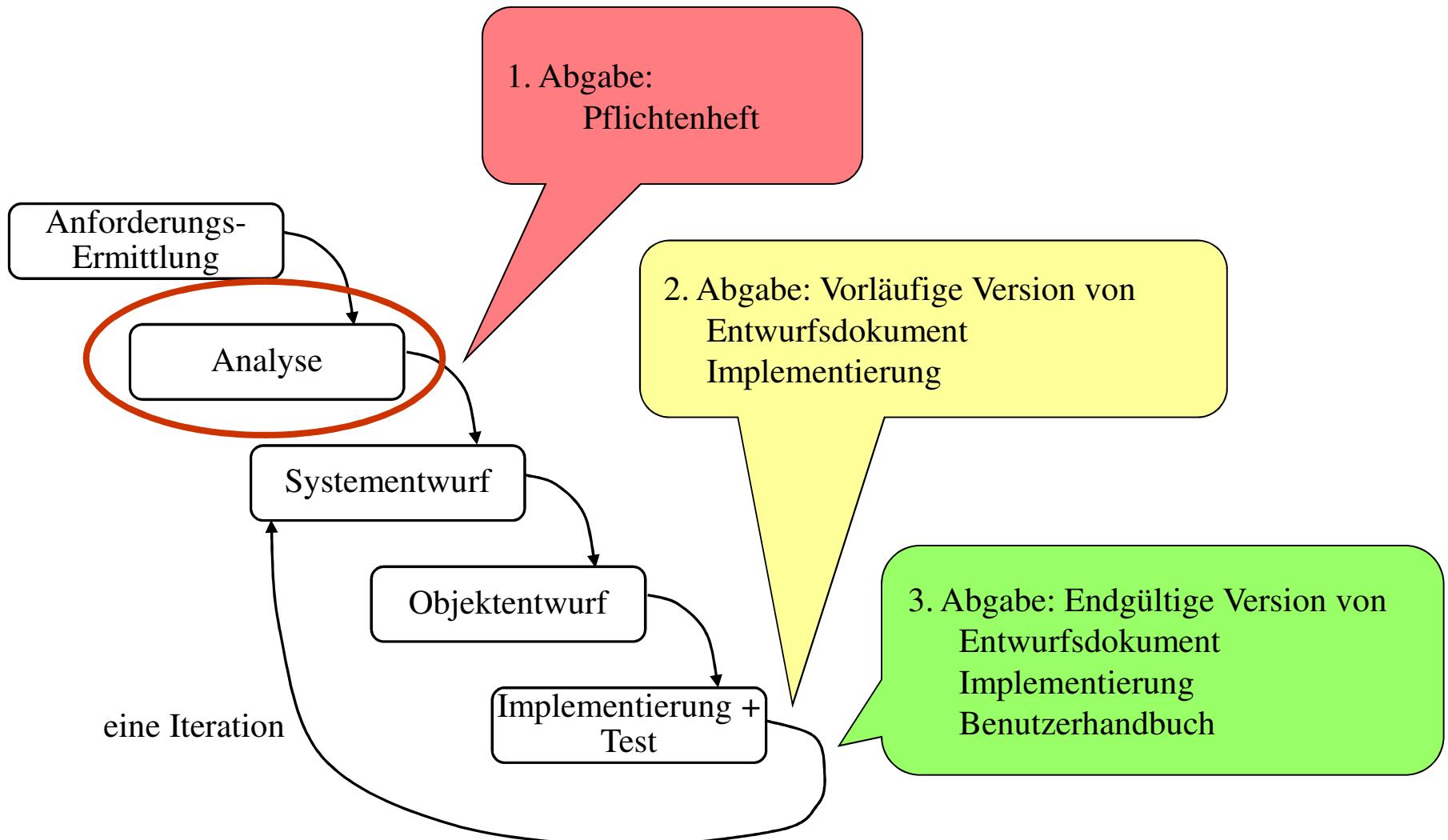
## Anhang A: GUI-Skizzen

GUI-Skizzen der zentralen Grenzklassen

[ 2.4.1 + 2.4.2 entstehen in der Anforderungsermittlungsphase ]

[ 2.4.3 + 2.4.4. entstehen in der Analysephase ]

# SoPra-Vorgehensmodell



# Ziel der (Anforderungs-) Analyse

- Ausarbeitung und Präzisierung der Anforderungsdefinition
- Finden und Beseitigen von Inkonsistenzen
- Modellierung der Anwendungsdomäne und der Anforderungen
- Es geht dabei immer noch primär um das WAS und nicht um das WIE !

# Zielmodelle der Analyse

- Statisches Modell

Besteht aus:

- Klassendiagrammen (UML)
  - Entitätsklassen der Anwendungsdomäne (nicht der Lösungsdomäne !) und ihre Beziehungen
- Klassenbeschreibungen (Text)
  - Entitäts-, Kontroll- und Grenzklassen der Anwendungsdomäne

- Dynamisches Modell

- Sequenzdiagramme (UML)
- Zustandsdiagramme (UML)

- Konstruktion beider Modelle schreitet simultan voran.

Dynamisches Modell dient insbesondere dazu, Auslassungen und Unklarheiten im statischen Modell aufzudecken

- Modelle werden zusammen mit den Modellen der Anforderungsanalyse im „Pflichtenheft“ dokumentiert

# Anwendungsdomäne vs. Lösungsdomäne

- Anwendungsdomäne:
  - Der Problembereich
  - Beispiele: Finanzbereich, Buchhandelsbereich, Flugzeugsteuerung, Restaurantmanagement, ...
- Anwendungsdomänen-Klasse:
  - Eine Abstraktion des Problembereichs.
  - Beispiele: Konten, Aktien, Bücher, Buchungen, Bestellungen ...
- Lösungsdomäne:
  - Konzepte, die bei der Lösung helfen
  - Beispiele: Datenbanken, Webservices, Datenstrukturen, Algorithmen, ...
- Lösungsdomänen-Klasse
  - Eine Abstraktion der Lösungsdomäne
  - Oft aus technischen Gründen eingeführt
  - Beispiele: Listen, Bäume, interne Datenbank, ...

# Was uns während der Analyse interessiert

- Während der Analyse interessieren wir uns für:
  - Anwendungsklassen
  - Assoziationen beschreiben Beziehungen in der Anwendungsdomäne
  - Vererbung repräsentiert Taxonomie der Anwendungsdomäne
- Wir interessieren uns (in der Regel) *noch nicht* für:
  - Genaue Signatur von Operationen
  - Lösungsdomänenklassen

Während der Analyse nur Anwendungsdomänen-Klassen verwenden,  
keine Klassen der Lösungsdomäne!

# Gute und schlechte Beispiele von Klassen bei der Analyse einer E-commerce-Anwendung

Konzepte des Anwendungsbereichs, die im Analysemodell repräsentiert sein sollten

Kunde

Buchung

Rechnung

Konzepte des Lösungsbereichs, die im Analysemodell nicht explizit auftauchen sollten

Kundendatenbank

Bestellliste

Rechnungsdatei

# Die drei Arten von Klassen im Analysemodell

Die wichtigsten Fragen, die während der Analyse geklärt werden sollen:

1. Welche **Informationen** werden vom System **verwaltet**?
2. Welche **Systemschnittstellen** zu den Akteuren müssen realisiert werden?
3. Welche **Abläufe** müssen realisiert werden?

Dementsprechend interessieren wir uns für drei Arten von Klassen:

1. **Entitätsklassen**
2. **Grenzklassen**
3. **Steuerungsklassen**

# Die drei Arten von Klassen im Analysemodell (Forts.)

## Entitätsklassen (entity classes)

- repräsentieren Informationen, die vom System verwaltet werden
- in der Regel persistent
- E-commerce-Beispiel: Kunde, Rechnung, Bestellung

## Grenzklassen (Schnittstellenklassen, boundary classes, interface classes)

- repräsentieren die Systemschnittstelle zu den Akteuren
- in Softwaresystemen u.a.: die wesentlichen Bildschirmansichten und –masken
- E-commerce-Beispiel: BestellFormular, RechnungsAusgabe

## Steuerungsklassen (control classes)

- realisieren die Anwendungsfälle
- koordinieren dabei die Grenz- und Entitätsobjekte
- E-commerce-Beispiel: KundenÄndern

# Erstellung des statischen und dynamischen Modells

# Natürliche Sprachanalyse von Abbott [Abbott, 1983]

- Klassische Heuristik, um Objekte, Attribute, Assoziationen in Problembeschreibungen zu identifizieren

| Sprachkonstrukt | Modellkomponente | Beispiele         |
|-----------------|------------------|-------------------|
| Eigenname       | Instanz          | Monika            |
| Substantiv      | Klasse           | Außenbeamter      |
| Tat-Verb        | Operation        | erzeugt, weist zu |
| Sein-Verb       | Vererbung        | ist ein...,       |
| Haben-Verb      | Zustand          | hat, besteht aus  |
| Modalverb       | Zwang            | muss sein         |
| Adjektiv        | Attribut         | grün, dunkel      |

- **Kritik:** zu schematisch; zu abhängig vom Sprachstil der Problembeschreibung; Synonyme problematisch
- Nicht schematisch anwendbar, aber interessante Idee für den Hinterkopf ... 
- Weitere Heuristiken: Siehe z.B. Bruegge/Dutoit-Buch (siehe auch unten)

# Identifikation von Entitätsklassen

Entitätsklassen:

repräsentieren persistente, vom System verwaltete Informationen

Entitätsklassen haben meist:

- viele Attribute
- wenig (echte) Operationen
- einfachen Lebenszyklus

# Beispiel: Identifikation von Entitätsklassen aus Anwendungsfallbeschreibungen

1. Der Außenbeamte aktiviert die "MeldeNotfall"-Funktion auf seinem Meldegerät. Das System antwortet durch Bereitstellung eines Formulars für den Außenbeamten.
2. Der Außenbeamte füllt die Formularfelder "Grad des Notfalls", "Notfalltyp", "Ort" und "Kurzbeschreibung der Situation" aus. Er beschreibt auch mögliche Lösungen für die Notfallsituation. Sobald er das Formular ausgefüllt hat, schickt er es ab. Das System benachrichtigt den Dienstleiter.
3. Der Dienstleiter beurteilt die eingegebene Information und erzeugt einen Vorfall in der Datenbank, indem er den EröffneVorfall-Anwendungsfall aufruft. Der Dienstleiter wählt eine Lösung aus und bestätigt den Bericht. Das System zeigt dem Akteur Außenbeamter die Bestätigung und die ausgewählte Lösung an.

# Beispiel: Beschreibung der identifizierten Entitätsklassen

|                |  |
|----------------|--|
| Dienstleiter   | Polizeibeamter, der den Vorfall leitet. Ein Dienstleiter eröffnet, dokumentiert und schließt Vorfälle ab als Antwort auf Notfall-Meldungen und andere Kommunikation mit dem Außenbeamten. Dienstleiter werden durch Dienstnummern identifiziert.   |
| NotfallMeldung | Anfangsbericht über einen Vorfall von einem Außenbeamten an einen Dienstleiter. Eine NotfallMeldung erfordert das Erstellen eines Vorfalls durch den Dienstleiter. Eine NotfallMeldung besteht aus einer Notfallebene, einem Typ (Brand, Verkehrsunfall, Sonstiges), einem Ort und einer Beschreibung.         |
| Außenbeamter   | Polizist oder Feuerwehrmann im Einsatz. Ein Außenbeamter kann zu jeder Zeit nur einem Vorfall zugewiesen sein. Außenbeamte werden durch die Dienstnummer identifiziert.  |
| Vorfall        | Ereignis, das die Aufmerksamkeit eines Außenbeamten erfordert. Ein Vorfall kann von einem Außenbeamten dem System mitgeteilt werden. Ein Vorfall besteht aus einer Beschreibung, einer Antwort, einem Status (offen, beendet, dokumentiert), einer Örtlichkeit und einer Anzahl von zugewiesenen Außenbeamten. |

**Tabelle 5.2:** Entitätsobjekte für den MeldeNotfall-Anwendungsfall

aus: Bruegge/Dutoit, Objektorientierte Softwaretechnik, 2004

# Heuristiken zum Identifizieren von Entitätsklassen (nach Bruegge/Dutoit)

- Immer wieder auftretende Substantive in den Anwendungsfällen
  - z.B. „Vorfall“
- Dinge aus dem Anwendungsbereich, deren Zustand vom System in Auge behalten werden muss
  - z.B. „Außenbeamter“, „Dienstleiter“, „Ressourcen“
- Aktivitäten in der Wirklichkeit, die das System verfolgen muss
  - z.B. „NotfallOperationsPlan“
- Ausdrücke, die Entwickler und Benutzer noch klären müssen, damit der Anwendungsfall verständlich wird
  - z.B. „Informationen, die vom Außenbeamten...“

# Bemerkungen zur Beschreibung von Klassen

- Alle identifizierten Klassen werden textuell beschrieben.
- Textuelle Beschreibung der Klassen ist wichtig, um ...
  - ... Terminologie zu vereinheitlichen
  - ... Missverständnisse zu vermeiden
  - ... Inkonsistenzen und Synonyme zu entlarven.
- Tipp:
  - Zunächst nicht allzu viel Zeit auf Klassenbeschreibungen verwenden, da Änderungen wahrscheinlich sind.
  - Später präzisieren !

# Identifikation von Grenzklassen

Grenzklassen:

repräsentieren die Systemschnittstelle zu den Akteuren

- Benutzerschnittstelle auf grober Abstraktionsebene
- Beispiele für Grenzobjekte
  - Formular, um Informationen bestimmter Art entgegenzunehmen
  - Bildschirmsicht oder -fenster für bestimmten Zweck
  - externe Eingabegeräte
  - ...
- Dinge wie „Menüeintrag“, „Scrollleiste“ etc. sind hier viel zu detailliert !

# Beispiel: Identifikation von Grenzklassen aus Anwendungsfallbeschreibungen

1. Der Außenbeamte aktiviert die "MeldeNotfall"-Funktion auf seinem Meldegerät. Das System antwortet durch Bereitstellung eines Formulars für den Außenbeamten.
2. Der Außenbeamte füllt die Formularfelder "Grad des Notfalls", "Notfalltyp", "Ort" und "Kurzbeschreibung der Situation" aus. Er beschreibt auch mögliche Lösungen für die Notfallsituation. Sobald er das Formular ausgefüllt hat, schickt er es ab. Das System benachrichtigt den Dienstleiter.
1. Der Dienstleiter beurteilt die eingegebene Information und erzeugt einen Vorfall in der Datenbank, indem er den EröffneVorfall-Anwendungsfall aufruft. Der Dienstleiter wählt eine Lösung aus und bestätigt den Bericht. Das System zeigt dem Akteur Außenbeamter die Bestätigung und die ausgewählte Lösung an.

# Beispiel: Beschreibung der identifizierten Grenzklassen

|                      |  |
|----------------------|--|
| Empfangsbestätigung  | Bestätigung, um dem Außenbeamten den Empfang der Meldung beim Dienstleiter anzuzeigen.   |
| DienstleiterStation  | Vom Dienstleiter benutzter Rechner.  |
| MeldeNotfallTaste    | Vom Außenbeamten benutzte Taste, um den MeldeNotfall-Anwendungsfall zu initiieren.   |
| NotfallMeldeFormular | Formular, das für die Eingabe von MeldeNotfall benutzt wird. Dieses Formular wird dem Außenbeamten auf der Mobilen Station angeboten, wenn die „Melde Notfall“-Funktion gewählt wurde. Das NotfallMeldeFormular enthält Felder, um alle Attribute einer NotfallMeldung zu spezifizieren und einen Knopf (oder etwas Ähnliches), um das ausgefüllt Formular abzuschicken. |
| MobileStation        | Vom Außenbeamten benutzter mobiler Rechner.  |
| VorfallFormular      | Formular, das für die Erstellung eines Vorfalls benutzt wird. Dieses Formular wird dem Dienstleiter auf der DienstleiterStation angeboten, wenn die NotfallMeldung empfangen wurde. Der Dienstleiter benutzt dieses Formular auch, um Betriebsmittel zuzuweisen und den Bericht des Außenbeamten zu bestätigen.  |

Tabelle 5.3: Grenzobjekte für den MeldeNotfall-Anwendungsfall

aus: Bruegge/Dutoit, Objektorientierte Softwaretechnik, 2004

# Heuristiken zum Identifizieren von Grenzklassen (nach Bruegge/Dutoit)

- Als Grenzklassen identifizieren:
  - Benutzerschnittstellen, die Benutzer benötigt, um AFs zu initiieren (z.B. MeldeNotfallTaste)
  - Formulare, die Benutzer benötigt, um Daten in das System einzutragen (z.B. NotfallMeldeFormular)
  - Aufzeichnungen und Nachrichten, die das System benötigt, um dem Benutzer zu antworten (z.B. Empfangsbestätigung)
  - Wenn mehrere Akteure am AF beteiligt: Endgeräte (z.B. DienstleiterStation)
- Visuelle Aspekte der Schnittstelle **nicht** durch Grenzklassen modellieren. dafür sind **Skizzen und GUI-Prototypen** besser geeignet!
- Grenzobjekte immer aus der Sicht und mit Ausdrücken des Endbenutzers beschreiben, keine Terminologie aus Lösungsbereich

# Identifikation von Steuerungsklassen

- Steuerungsobjekte koordinieren die Grenz- und Entitätsobjekte
- In der Regel enge Beziehung  
Steuerungsobjekte \$ Anwendungsfall
- Steuerungsobjekte realisieren Anwendungsfall
- Im Beispiel: Für den AF Notfall melden je ein Steuerungsobjekt
  - Außenbeamten → MeldeNotfallSteuerung
  - Dienstleiter → VerwalteNotfallSteuerung

# Beispiel: Beschreibung der identifizierten Steuerungsklassen

|                           |   |
|---------------------------|---|
| MeldeNotfall-Steuerung    | Verwaltet die Benachrichtigungsfunktion MeldeNotfall auf der MobilenStation. Dieses Objekt wird kreiert, wenn der Außenbeamte den „Melde Notfall“-Knopf auswählt. Es erzeugt dann ein Notfall-MeldeFormular und bietet es dem Außenbeamten an. Nach dem Abschicken des Formulars sammelt dieses Objekt die Informationen des Formulars, erzeugt eine NotfallMeldung und leitet sie zum Dienstleiter. Das Steuerungsobjekt wartet dann auf eine Empfangsbestätigung von der DienstleiterStation. Sobald die Empfangsbestätigung eingetroffen ist, erzeugt das MeldeNotfallSteuerung-Objekt eine Empfangsbestätigung und zeigt sie dem Außenbeamten an. |
| VerwalteNotfall-Steuerung | Verwaltet die Benachrichtigungsfunktion MeldeNotfall auf der DienstleiterStation. <b>Die VerwalteNotfallSteuerung nimmt Notfallmeldungen entgegen.</b> Es erzeugt dann ein VorfallFormular und bietet es dem Dienstleiter an. Sobald der Dienstleiter einen Vorfall erzeugt, Betriebsmittel bereitgestellt und eine Empfangsbestätigung eingegeben hat, leitet VerwalteNotfallSteuerung die Empfangsbestätigung an die MobileStation weiter.  |

Tabelle 5.4: Steuerungsobjekte für den Anwendungsfall MeldeNotfall

aus: Bruegge/Dutoit, Objektorientierte Softwaretechnik, 2004

# Heuristiken zum Identifizieren von Steuerungsklassen (nach Bruegge/Dutoit)

- Jeder Anwendungsfall hat mindestens ein Steuerungsobjekt
- Jeder Akteur im Anwendungsfall hat ein zugehöriges Steuerungsobjekt
- Die Lebensdauer eines Steuerungsobjekts sollte die Gesamtdauer des Anwendungsfalls oder die Dauer einer Benutzersitzung abdecken. Wenn es schwierig ist, die Lebensdauer zu bestimmen, dann hat der zugehörige AF möglicherweise keine definierte Anfangs- oder Endbedingung

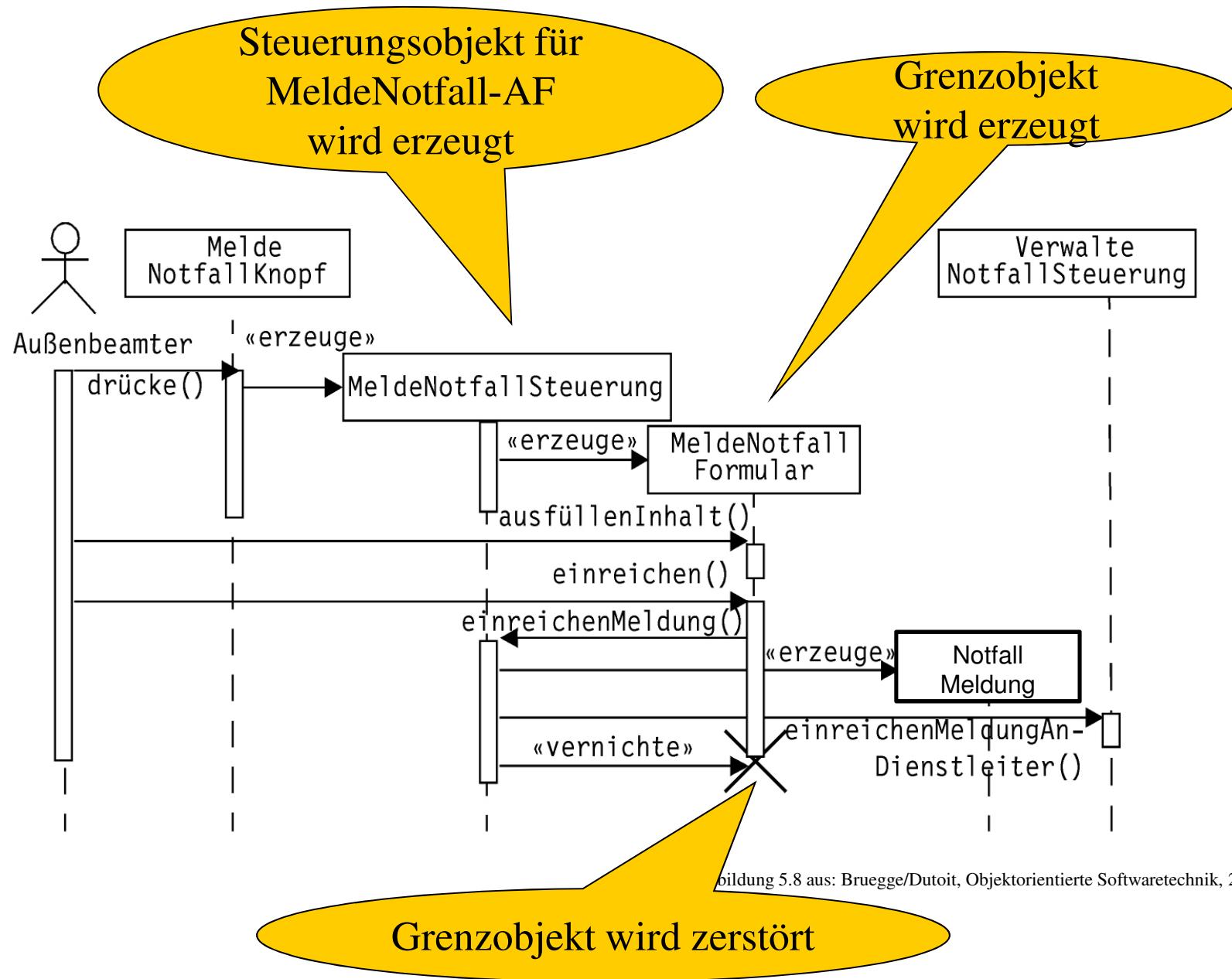
# Sequenzdiagramme: Abbildung von Anwendungsfällen auf Objekte

Sequenzdiagramme: Teil des dynamischen Modells

Ziele:

- Konkretisierung der Anwendungsfälle
- Wie ist das Verhalten auf beteiligte Objekte verteilt?
- **Fehlende Klassen und Unklarheiten aufdecken!**
- Konventionen:
  - Spalten repräsentieren partizipierende Objekte und Akteure
  - Linkste Spalte: Akteur, der den Anwendungsfall initiiert
  - Zweite Spalte: in der Regel ein Grenzobjekt
  - Dritte Spalte: in der Regel Steuerungsobjekt; kann u.a. weitere Grenzobjekte erzeugen

# Sequenzdiagramm für MeldeNotfall-Anwendungsfall (Teil 1)



Bildung 5.8 aus: Bruegge/Dutoit, Objektorientierte Softwaretechnik, 2004

# Sequenzdiagramm für MeldeNotfall-Anwendungsfall (Teil 2)

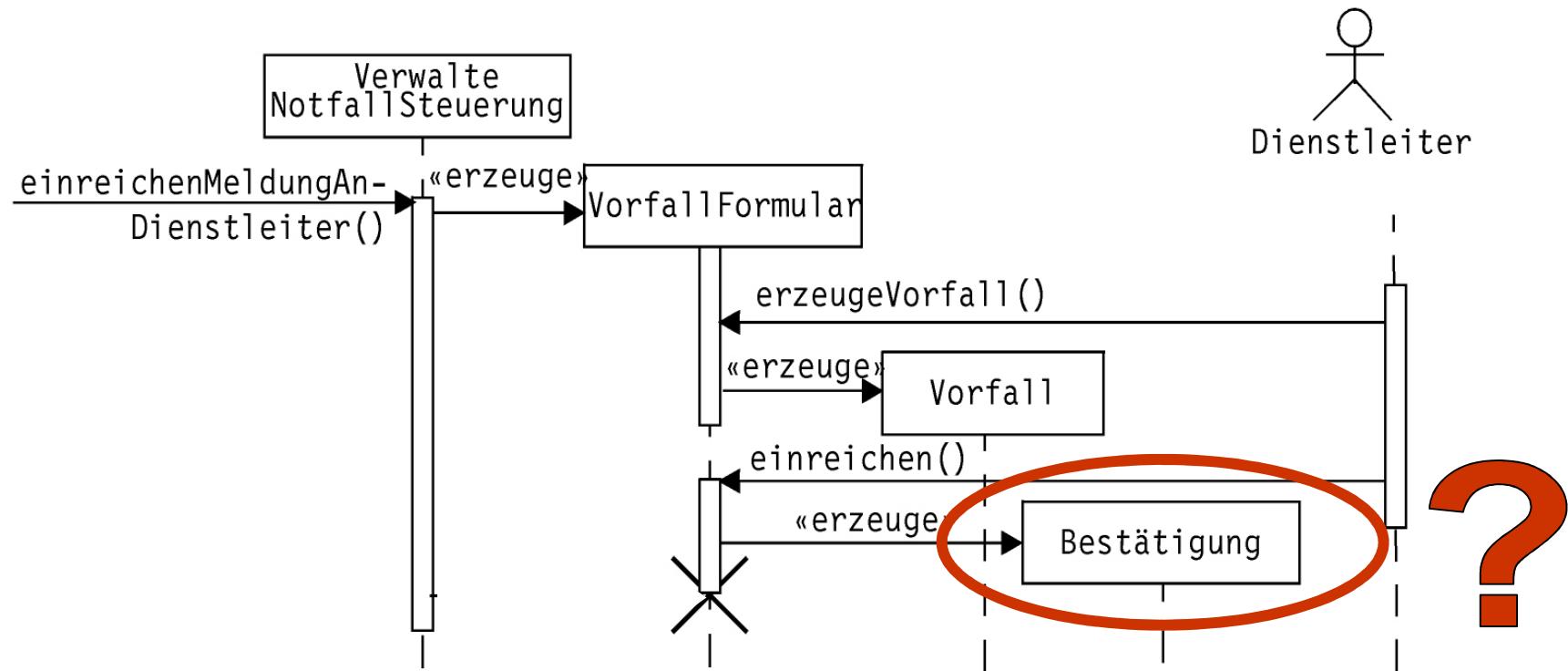
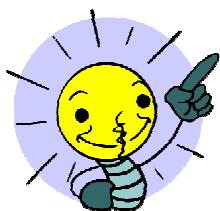


Abbildung 5.9 aus: Bruegge/Dutoit, Objektorientierte Softwaretechnik, 2004



1. Entitätsklasse „Bestätigung“ ergänzen !
2. Inhalt der Bestätigung im AF „MeldeNotfall“ genauer spezifizieren !

# Beschreibung der neu identifizierten Entitätsklasse

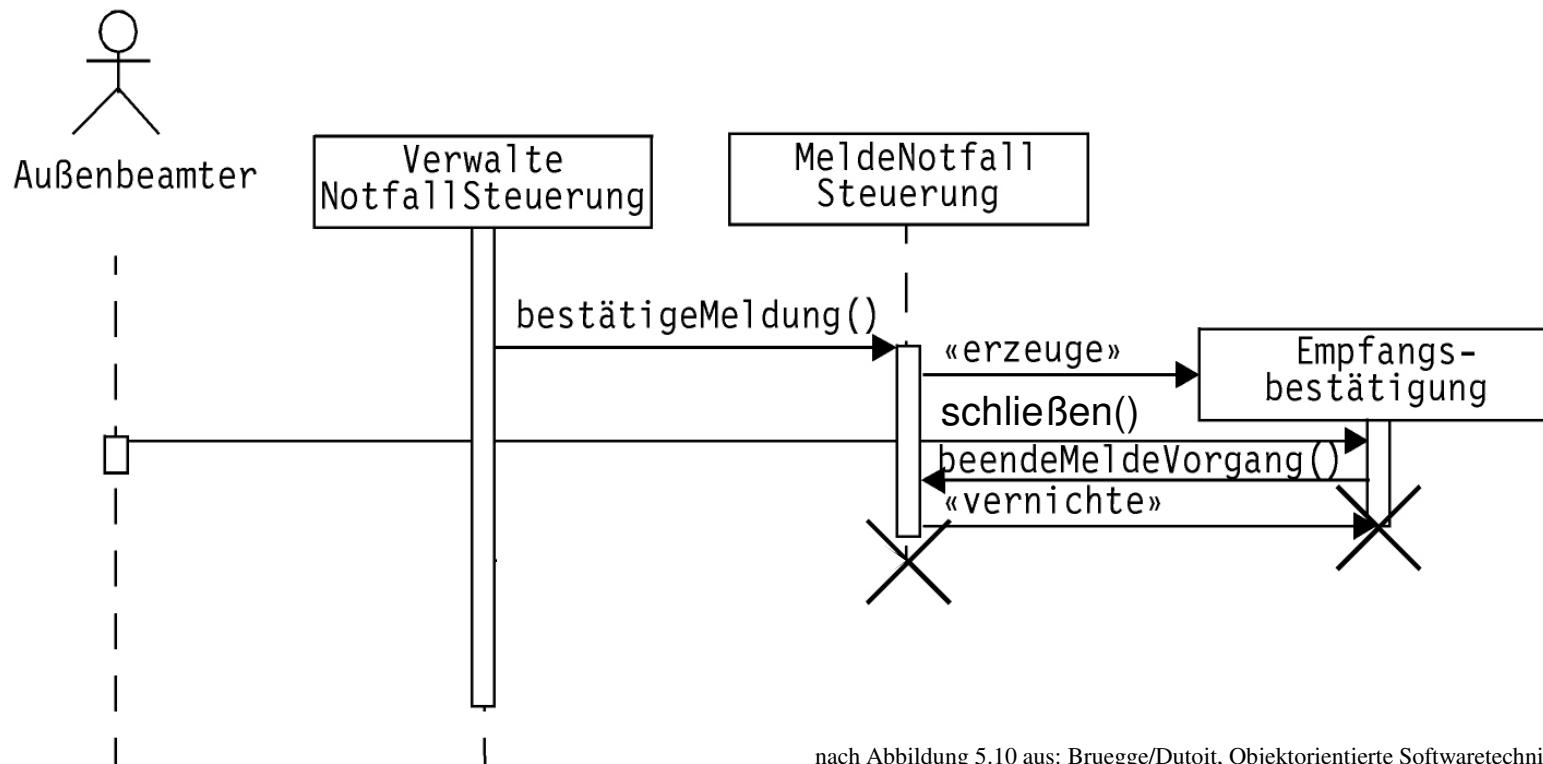
Bestätigung:

Antwort eines Dienstleiters auf eine NotfallMeldung eines Außenbeamten.  
Durch das Senden der Bestätigung teilt der Dienstleiter dem Außenbeamten mit,  
dass die NotfallMeldung empfangen wurde, dass ein Vorfall erzeugt wurde und  
dass diesem Betriebsmittel zugewiesen wurden. Die Bestätigung gibt Auskunft  
über die zugewiesenen Betriebsmittel und die voraussichtliche Ankunftszeit.

# Beispiel: Verfeinerter Ereignisfluss des MeldeNotfall-Anwendungsfalls

1. Der Außenbeamte aktiviert die "MeldeNotfall"-Funktion auf seinem Meldegerät. Das System antwortet durch Bereitstellung eines Formulars für den Außenbeamten.
2. Der Außenbeamte füllt die Formularfelder "Grad des Notfalls", "Notfalltyp", "Ort" und "Kurzbeschreibung der Situation" aus. Er beschreibt auch mögliche Lösungen für die Notfallsituation. Sobald er das Formular ausgefüllt hat, schickt er es ab. Das System benachrichtigt den Dienstleiter.
3. Der Dienstleiter beurteilt die eingegebene Information und erzeugt einen Vorfall in der Datenbank, indem er den EröffneVorfall-Anwendungsfall aufruft. Der Dienstleiter wählt eine Lösung aus und bestätigt den Bericht. Das System zeigt dem Akteur Außenbeamter die Bestätigung und die ausgewählte Lösung an. **Die Empfangsbestätigung zeigt dem Außenbeamten an, dass die Notfallmeldung erhalten und ein Vorfall angelegt wurde. Sie gibt auch Auskunft über die zugewiesenen Betriebsmittel (z.B. Löschzug, Krankenwagen, etc.) und deren voraussichtliches Eintreffen.**

# Sequenzdiagramm für MeldeNotfall-Anwendungsfall (Teil 3)



# Richtlinien für Sequenzdiagramme (nach Bruegge/Dutoit)

- Die erste Spalte sollte dem Akteur entsprechen, der den AF initiiert
- Die zweite Spalte sollte ein Grenzobjekt sein, das der Akteur benutzt, um AF zu initiieren.
- Die dritte Spalte sollte das Steuerungsobjekt sein, das den Rest des Anwendungsfalls behandelt
- Steuerungsobjekte werden von den Grenzobjekten erzeugt, die Anwendungsfälle initiieren
- Steuerung- und Grenzobjekte greifen auf Entitätsobjekte zu
- Entitätsobjekte greifen niemals auf Grenz- oder Steuerungsobjekte zu (dadurch können verschiedene AFs leichter dieselben Entitätsobjekte verwenden; vgl. auch: Schichtenarchitektur)

# Assoziationen identifizieren

Erinnerung: Assoziation:

- repräsentiert Beziehung zwischen Objekten
- durch Kante im Klassendiagramm dargestellt
- hat (genauer: kann haben)
  1. Namen
  2. Rolle
  3. Multiplizität

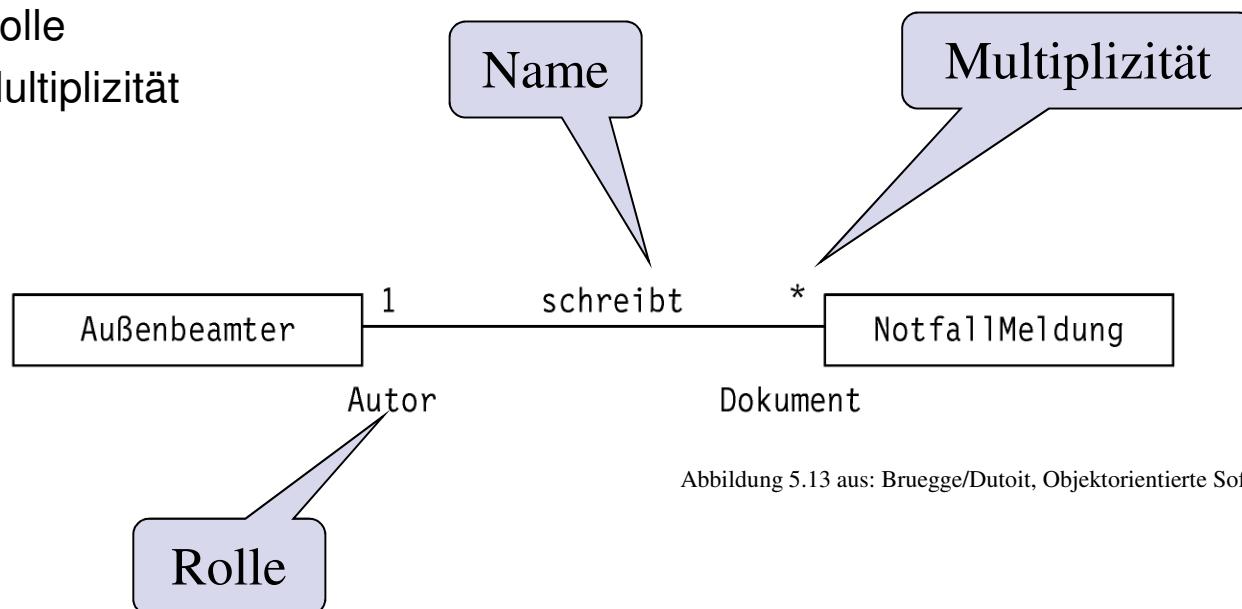


Abbildung 5.13 aus: Bruegge/Dutoit, Objektorientierte Softwaretechnik, 2004

# Assoziationen identifizieren (Forts.)

Tipps:

- nicht zu viele Assoziationen verwenden
- ableitbare Assoziationen in der Regel vermeiden:

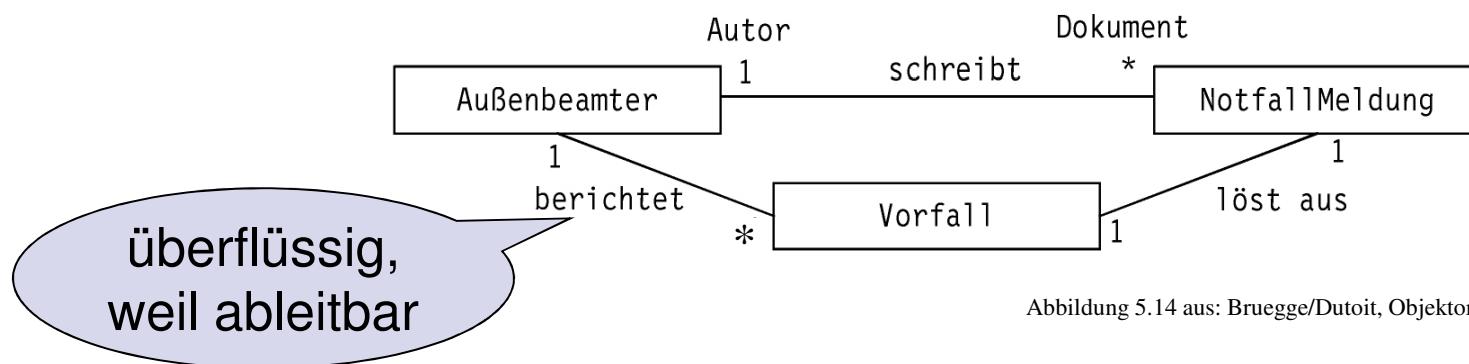


Abbildung 5.14 aus: Bruegge/Dutoit, Objektorientierte Softwaretechnik, 2004

- Multiplizitäten nicht zu früh klären, da Modelländerungen wahrscheinlich sind

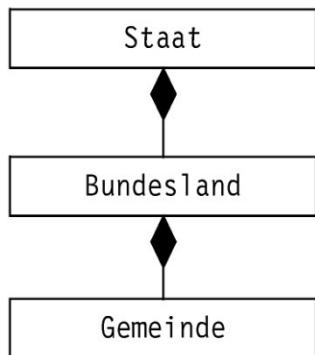
# Ratschläge zum Identifizieren von Assoziationen (nach Bruegge/Dutoit)

- Verbalphrasen überprüfen (z.B. „schreibt ...“, „sendet ...“, etc.)
- Assoziationen und Rollen treffend und genau benennen
- Untersuchen, welche Objekte mit welchen Objekten in den Sequenzdiagrammen Nachrichten austauschen

# Zusammensetzungbeziehungen identifizieren

Erinnerung: Zwei Typen von Zusammensetzungsbeziehungen

Komposition



Aggregation

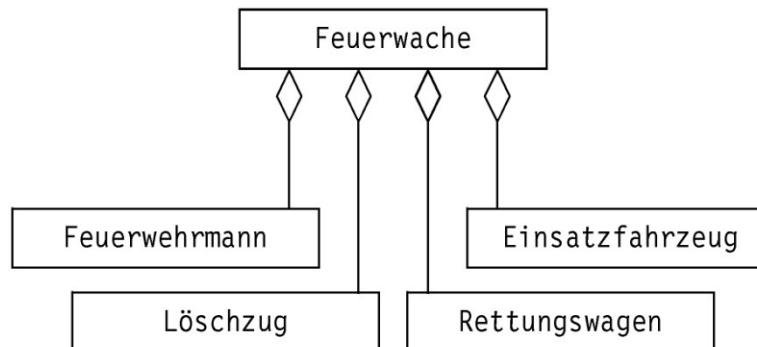


Abbildung 5.15 aus: Bruegge/Dutoit, Objektorientierte Softwaretechnik, 2004

- Teil-Ganzes-Beziehung
- Teile können ohne das Ganze nicht existieren

Tipp: Überstrukturierung vermeiden !

- Teile können ohne das Ganze existieren
- Zuordnung kann sich verändern
- Mehrere Beziehungen gleichzeitig möglich

# Attribute identifizieren

- Attribute repräsentieren Eigenschaften von Objekten
  - Nur für das System aus Sicht der Anwendungsdomäne relevante Eigenschaften sollen modelliert werden
    - Gegenbeispiel: Haarfarbe des Dienstleiters ist irrelevant...
  - Beschreibung von Attributen:  
In der Klassenbeschreibung:
    - Namen
    - kurze Beschreibung
    - Typ
  - Im Klassendiagramm:      Bsp.
- NotfallMeldung

notfallTyp:{Brand,Verkehr,sonst}

ort:String

beschreibung:String
- Tipp:  
Während der Analyse nicht zu viel Zeit auf Identifikation von Attributen verwenden. Diese später ergänzen !

# Heuristiken zum Identifizieren von Attributen (nach Bruegge/Dutoit und Rumbaugh)

- Possessivphrasen überprüfen (z.B. „hat“, „besitzt“, ...)
- Zu speichernde Werte als Attribute eines Entitätsobjekts darstellen
- Objekte bereits identifizierter Klassen sind keine Attribute in einer anderen Klasse; stattdessen Assoziation benutzen
- Keine Zeit mit Beschreibung von Feinheiten verschwenden, bevor die Objektstruktur feststeht; Details aber später ergänzen !

# Modellierung von Vererbungsbeziehungen

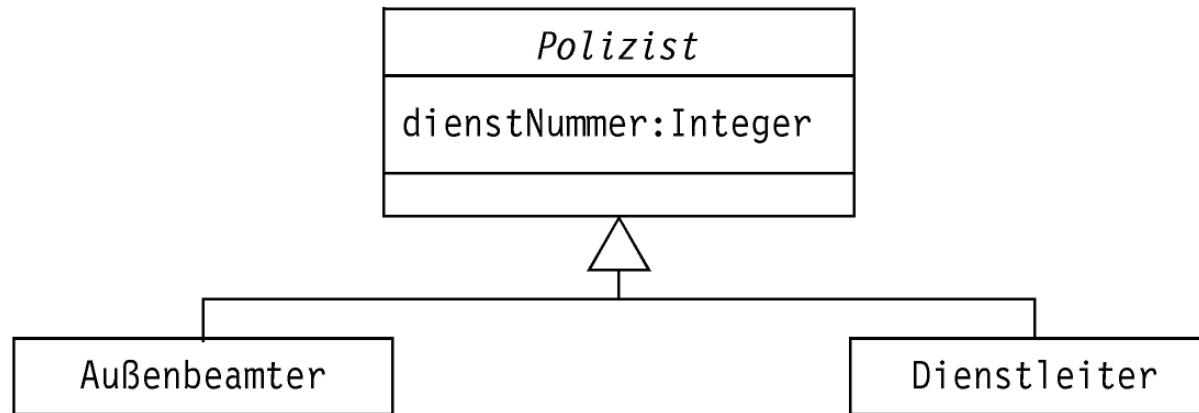


Abbildung 5.18 aus: Bruegge/Dutoit, Objektorientierte Softwaretechnik, 2004

**Erinnerung:** Vererbungsbeziehungen verwenden um ...

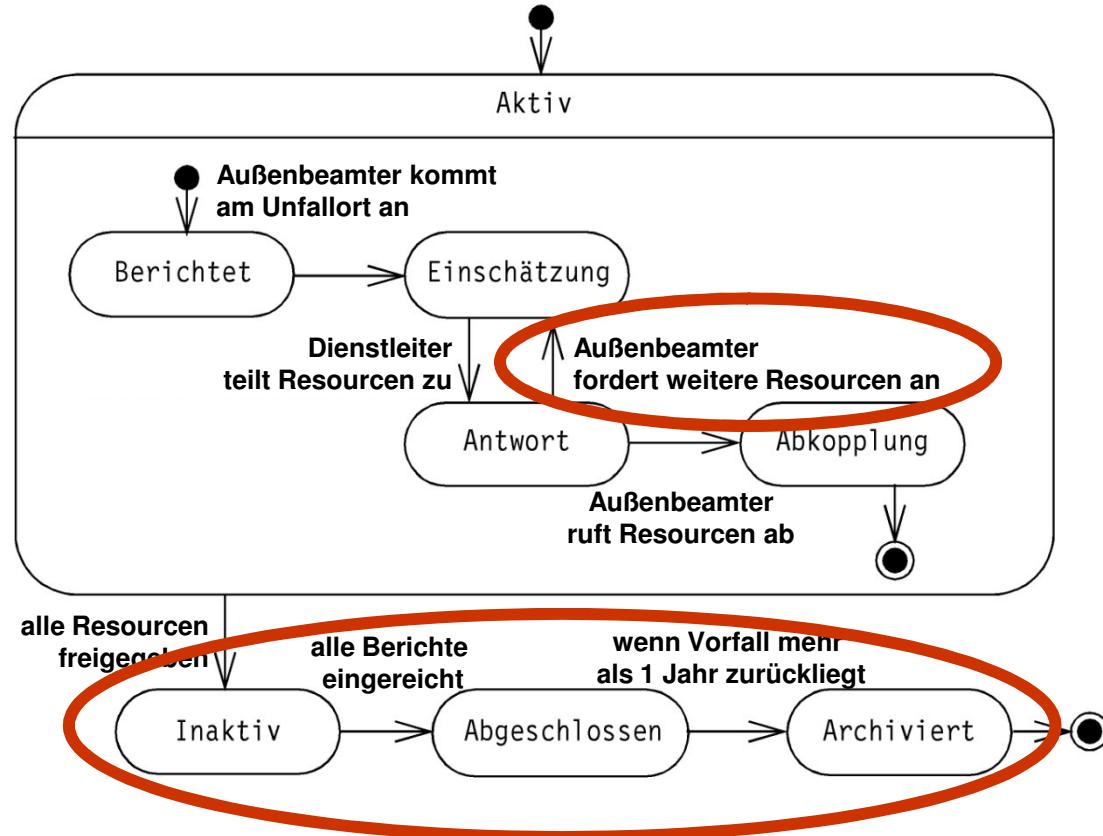
- ... Taxonomien der Anwendungsdomäne zu erfassen
- ... Redundanz zu vermeiden

# Modellierung des zustandsabhängigen Verhaltens von Objekten

- Schon betrachtet: Sequenzdiagramme:
  - modellieren Nachrichtentausch zwischen Objekten
  - beschreiben Verhalten des Systems aus Sicht eines Anwendungsfalls
  - Ziel u.a.: Finden fehlender Klassen, Assoziationen, Operationen
- Jetzt: Zustandsdiagramme:
  - Verhaltensbeschreibung aus Sicht eines einzelnen Objekts
  - Übergänge zwischen Zuständen des Objekts
  - Ziele: Objektverhalten verstehen, fehlende Anwendungsfälle und fehlerhafte Ereignisflüsse identifizieren
- Zustandsdiagramme nur für Objekte mit langer Lebensdauer und interessantem zustandsabhängigem Verhalten betrachten:
  - Steuerungsobjekte: oft
  - Entitätsobjekte: weniger oft
  - Grenzobjekte: fast nie

# Beispiel: Notruf-Management System FRIEND

Dynamisches Verhalten von Instanzen der Klasse Vorfall:



nach Abbildung 5.17: Bruegge/Dutoit, Objektorientierte Softwaretechnik, 2004

Lebenszyklus kann neue Anwendungsfälle aufdecken:

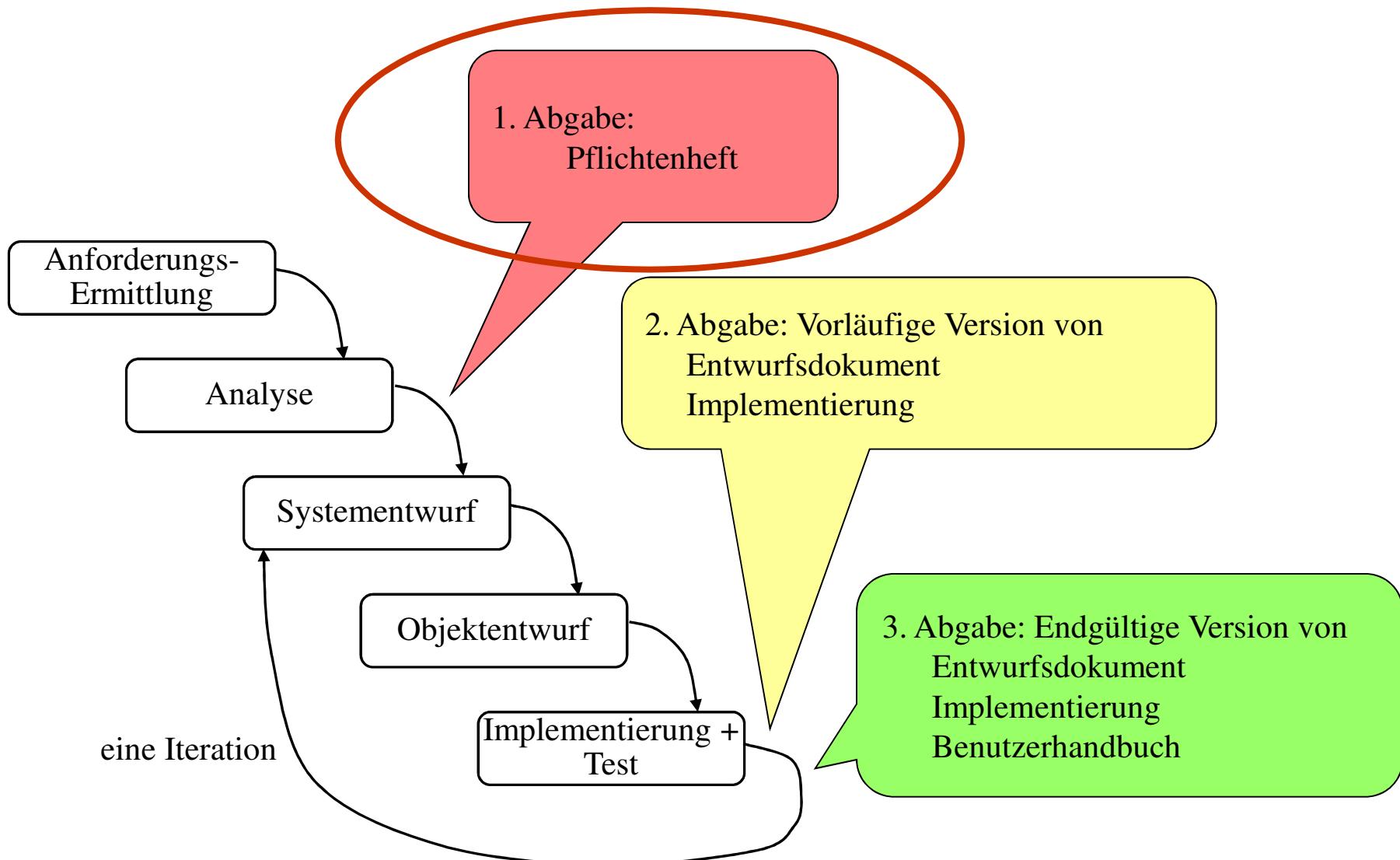
- Es sollte dem Außenbeamten möglich sein weitere Betriebsmittel anzufordern ! 😊
- Anwendungsfälle für Dokumentieren, Schließen, Archivierung von Vorfällen vorhanden ?

# Zusammenfassung: Tätigkeiten zum Erstellen des Analysemodells

- Identifikation von
  - Entitäts-, Grenz- und Steuerungsklassen
  - Entitätsklassen und ihre Beziehungen werden in UML-Klassendiagramm erfasst
  - Entitäts-, Grenz- und Steuerungsklassen werden textuell beschrieben
- Sequenzdiagramme: Anwendungsfälle auf Nachrichtenaustausch zwischen Objekte abbilden
  - Konkretisierung / fehlende Klassen + Unklarheiten aufdecken;

**Zum Umfang des dynamischen Modells im SOPRA-Pflichtenheft: Siehe unten!**
- Identifikation von
  - Assoziationen mit Multiplizitäten, Aggregationen, Attributen
- Modellierung von:
  - Vererbungsbeziehungen
  - Zustandsabhängigem Verhalten von Objekten mit interessantem Verhalten
    - Konkretisierung / fehlende AFs und fehlerhafte Ereignisflüsse aufdecken
- Diskutieren, Prüfen, Iterieren !

# SoPra-Vorgehensmodell



# Inhaltsverzeichnis des SoPra-Pflichtenhefts

## 1. Einführung

Textuelle Beschreibung von Zweck und Umfang des geplanten Systems

## 2. Vorgeschlagenes System

### 2.1. Übersicht:

Kurze textuelle Beschreibung

### 2.2. Funktionale Anforderungen

Kurze textuelle Beschreibung der funktionalen Anforderungen auf hohem Niveau

### 2.3. Nichtfunktionale Anforderungen

Textuelle Beschreibung der relevanten nichtfunktionalen Anforderungen

### 2.4. Systemmodelle

#### 2.4.1. Szenarien

2.4.2. Anwendungsfallmodell: Use case-Diagramm + Use case-Beschreibungen

2.4.3. Statisches Modell: Klassendiagramm für Entitätsklassen +  
Klassenbeschreibungen für Entitäts-, Grenz- und Kontrollklassen

2.4.4. Dynamisches Modell: Sequenzdiagramme + Zustandsdiagramm(e)

## 3. Glossar

Lexikonartige Auflistung und Kurzerklärung wichtiger Begriffe

## Anhang A: GUI-Skizzen

GUI-Skizzen der zentralen Grenzklassen

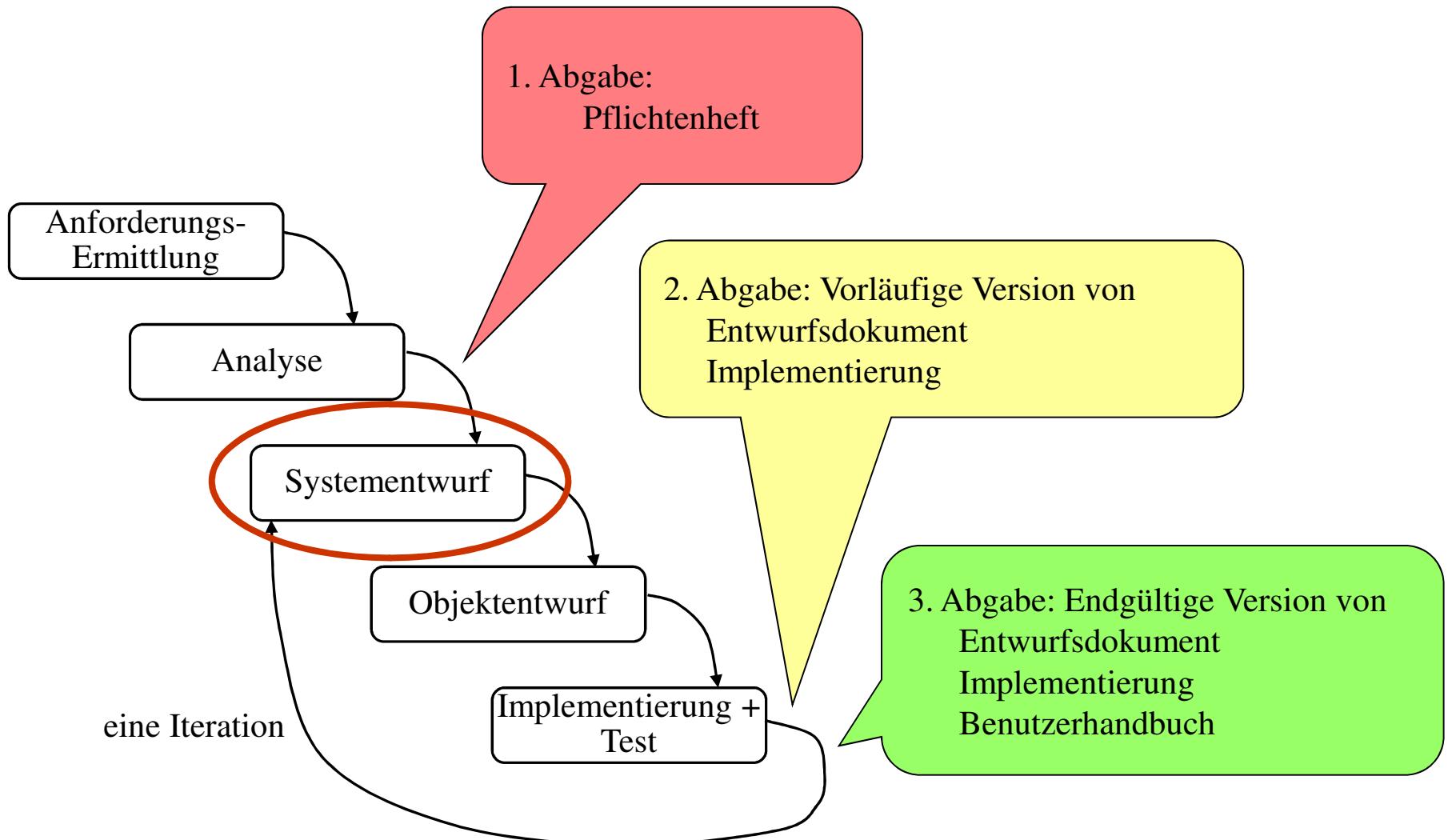
[ 2.4.1 + 2.4.2 entstehen in der Anforderungsermittlungsphase ]

[ 2.4.3 + 2.4.4. entstehen in der Analysephase ]

# Wichtig: Vereinbarungen für den Umfang des dynamischen Modells im WS 13/14

1. Es genügt Sequenzdiagramme abzugeben, die den typischen Ablauf der beiden Anwendungsfälle mit dem kompliziertesten Verhalten darstellen.
2. Bei den Zustandsdiagrammen genügt es, Zustandsdiagramme für den Lebenszyklus von Objekten derjenigen Klasse anzugeben, die die Medien repräsentiert.
3. Weitere Sequenzdiagramme und Zustandsdiagramme sind erlaubt, sofern Sie sie als sinnvoll erachten, aber nicht unbedingt gefordert.

# SoPra-Vorgehensmodell



# Analyse vs. Entwurf

- Analyse:
  - Es geht primär um das WAS
  - System wird aus Sicht der Anwendungsdomäne betrachtet
- Entwurf:
  - Nachdem das WAS geklärt ist, geht es jetzt primär um das WIE
  - Fragestellungen der Lösungsdomäne geraten jetzt in's Blickfeld
- Unterscheiden:
  - Systementwurf
  - Objektentwurf
  - [ Implementierung + Test ]
- Phasen sind zwar logisch nacheinander angeordnet, können sich aber zeitlich überlappen

# Inhaltsverzeichnis des SoPra-Entwurfsdokuments

## 1. Einleitung

Textuelle Beschreibung des Zwecks des Systems

## 2. Systementwurf

### 2.1. Entwurfsziele

Textuelle Beschreibung der Entwurfsziele und ihrer Priorisierung;  
kurze Begründung für die getroffenen Entscheidungen

### 2.2. Systemzerlegung

Paketdiagramm der Subsystemzerlegung; kurze textuelle Beschreibung der Aufgaben der Subsysteme und Begründung für diese Zerlegung

### 2.3. Verwendung existierender Software-Komponenten

kurze Beschreibung und Begründung

### 2.4. Management persistenter Daten

Auflistung, welche Objekte persistent zu halten sind;  
Beschreibung und Begründung des gewählten Persistenzmechanismus

### 2.5. Sonstiges

ggf. Beschreibung und Begründung weiterer Systementwurfsentscheidungen;  
dieser Abschnitt kann entfallen

## 3. Objektentwurf → Struktur folgt

## 4. Glossar

## 5. Anhang → Struktur folgt

# Aufgaben & Ziele des Systementwurfs

1. Entwurfsziele festlegen und priorisieren
2. Zerlegung des Systems in Subsysteme („Softwarearchitektur“)
3. Auswahl von Softwarekomponenten
4. Persistente Daten identifizieren und Persistenzmechanismus festlegen
5. Weitere Entwurfsfragen klären

# Aufgaben & Ziele des Systementwurfs

1. Entwurfsziele festlegen und priorisieren
- 2.
- 3.
- 4.
- 5.

# **Entwurfziele können sich auf unterschiedlichste Aspekte beziehen**

## **Verlässlichkeitskriterien**

- Robustheit
- Zuverlässigkeit
- Verfügbarkeit
- Fehlertoleranz
- Schutz vor feindlichen Angriffen (security)
- Sicherheit (safety)

## **Wartungskriterien**

- Erweiterbarkeit
- Modifizierbarkeit
- Anpassungsfähigkeit
- Portierbarkeit
- Lesbarkeit
- Rückverfolgbarkeit der Anforderungen

## **Leistungskriterien**

- Antwortzeit
- Datendurchsatz
- Speicherbedarf

## **Kostenkriterien**

- Entwicklungskosten
- Aufstellungskosten
- Umrüstungskosten
- Wartungskosten
- Administrationskosten

## **Endbenutzerkriterien**

- Nützlichkeit
- Nutzbarkeit

Definitionen: siehe Folgefolien

# Verlässlichkeitskriterien

|   |  |
|---|--|
| Robustheit  | Fähigkeit, ungültige Benutzereingaben zu überstehen                                      |
| Zuverlässigkeit   | Übereinstimmung zwischen erwünschtem und beobachtetem Verhalten                          |
| Verfügbarkeit   | Anteil der Zeit, in der das System im Normalbetrieb verwendet werden kann                |
| Fehlertoleranz  | Fähigkeit, unter fehlerhaften Umständen zu arbeiten                                      |
| Schutz vor feindlichen Angriffen<br>( <i>security</i> ) | Fähigkeit, feindlichen Angriffen standzuhalten   |
| Sicherheit<br>( <i>safety</i> )                         | Fähigkeit, Gefährdung menschlichen Lebens zu vermeiden, selbst bei Fehlern und Abstürzen |

nach Bruegge/Dutoit, Objektorientierte Softwaretechnik, 2004, Tabelle 6.3

# Wartungskriterien

|                     |   |
|---------------------|---|
| Erweiterbarkeit     | Wie einfach ist es, neue Funktionalität hinzuzufügen?                           |
| Modifizierbarkeit   | Wie leicht kann Funktionalität geändert/korrigiert werden?                      |
| Anpassungsfähigkeit | Wie leicht kann das System an andere Anwendungsdomänen angepasst werden?        |
| Portierbarkeit      | Wie leicht ist das System auf andere Plattformen zu übertragen?                 |
| Lesbarkeit          | Wie verständlich wird das System durch Lesen des Codes?                         |
| Rückverfolgbarkeit  | Wie leicht können Codeteile auf spezifische Anforderungen zurückgeführt werden? |

nach Bruegge/Dutoit, Objektorientierte Softwaretechnik, 2004, Tabelle 6.5

# Leistungskriterien

|                |  |
|----------------|--|
| Antwortzeit    | Zeit zwischen Stellen einer Benutzeranfrage und deren Bestätigung/Bearbeitung  |
| Durchsatz      | Anzahl der Aufgaben, die ein System innerhalb einer festen Zeit erledigen kann |
| Speicherbedarf | Vom System benötigter Speicherplatz  |

nach Bruegge/Dutoit, Objektorientierte Softwaretechnik, 2004, Tabelle 6.2

# Kostenkriterien

|                       |   |
|-----------------------|---|
| Entwicklungskosten    | Kosten für die Konstruktion des Systems                       |
| Aufstellungskosten    | Kosten für Installation des Systems und Schulung der Benutzer |
| Umrüstungskosten      | Kosten für Migration von Daten aus dem Vorgängersystem        |
| Wartungskosten        | Kosten zur Behebung von Fehlern und für Systemverbesserungen  |
| Administrationskosten | Kosten für die Administration des Systems                     |

nach Bruegge/Dutoit, Objektorientierte Softwaretechnik, 2004, Tabelle 6.4

# Endbenutzerkriterien

|              |  |
|--------------|--|
| Nützlichkeit | Wie gut unterstützt das System die Arbeit des Benutzer?      |
| Nutzbarkeit  | Wie einfach ist es für den Benutzer, das System zu bedienen? |

nach Bruegge/Dutoit, Objektorientierte Softwaretechnik, 2004, Tabelle 6.6

# Entwurfsziele auswählen und priorisieren

- Es ist unrealistisch, alle wünschenswerten Kriterien zu erfüllen
  - Typische trade-offs:
    - Speicherplatz vs. Geschwindigkeit
    - Lieferzeit vs. Funktionalitätsumfang
    - Lieferzeit vs. Qualität
    - Wartungskriterien vs. Leistungskriterien
    - ...
- ) Wichtigste Kriterien, an denen sich der Entwurf orientieren soll, wählen und priorisieren

Auswahl dient als Richtschnur beim weiteren Entwurf

Entscheidung aus nicht-funktionalen Anforderungen und aus der Natur der Anwendungsdomäne ableiten

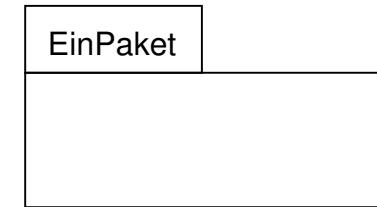
# Aufgaben & Ziele des Systementwurfs

- 1.
2. Zerlegung des Systems in Subsysteme („Softwarearchitektur“)
- 3.
- 4.
- 5.

# Strukturierung großer Systeme

- Ziel: bessere Überschaubarkeit durch Zerlegung in **Subsysteme**
- Leitlinie: **Minimierung der Abhängigkeiten** zwischen Subsystemen
- Subsystemzerlegung kann auch **Basis für Arbeitsteilung** im Team sein
- Darstellung der Subsystemzerlegung in UML durch **Paketdiagramme** (siehe Folgefolien)
- Zerlegung in Pakete auch von Java unterstützt (**package-Konzept**)

# UML: Paketdiagramme



## Paket

- Gruppiert Modellierungselemente (z.B. Klassen und/oder Teilpakete)
- Dargestellt durch stilisierte „Hängeregistermappe“
- Paketname wird in den Karteireiter geschrieben
- Zeigt das Paket die Substruktur des Pakets nicht (z.B. Teilpakete oder enthaltene Klassen), wird der Paketname manchmal auch „auf die Karteikarte“ geschrieben
- Jedes Paket bildet eigenen Namensraum, d.h. verschiedene Pakete können Klassen oder Pakete gleichen Namens enthalten.

## Abhängigkeit zwischen Paketen

- Signalisiert, dass Änderung des Pakets eine Änderung des Nachbarpakets erfordern kann
- Dargestellt durch gestrichelten Pfeil
- Abhängigkeit von P2 nach P1, z.B. wenn gilt:
  - Klasse C2 in P2 sendet Nachricht an Klasse C1 in P1
  - Operation von C2 in P2 hat Argument vom Typ C1 in P1
- Verschieden Abhängigkeitstypen in UML (<<import>> und <<access>>):
  - Für uns im SoPra nicht relevant; benutzen implizit <<access>>, was zu Java-import-Klausel passt

# Beispiel: Paketdiagramm der Systemzerlegung eines Notrufmanagementsystems

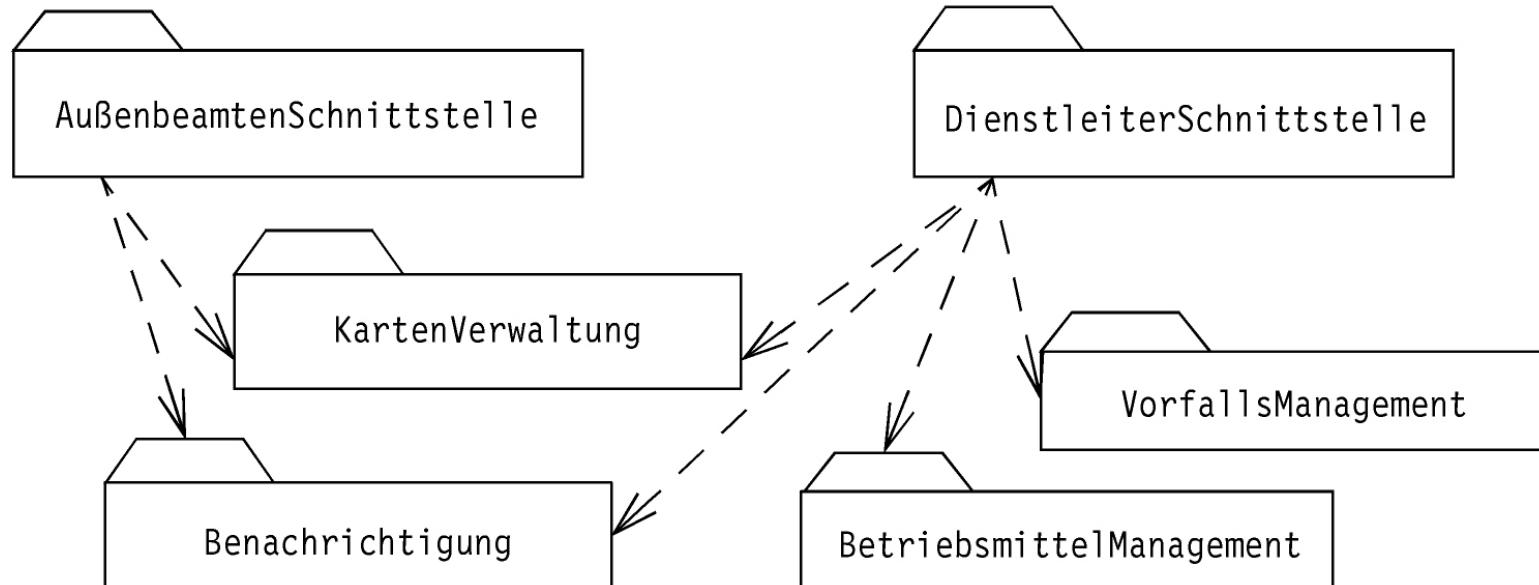
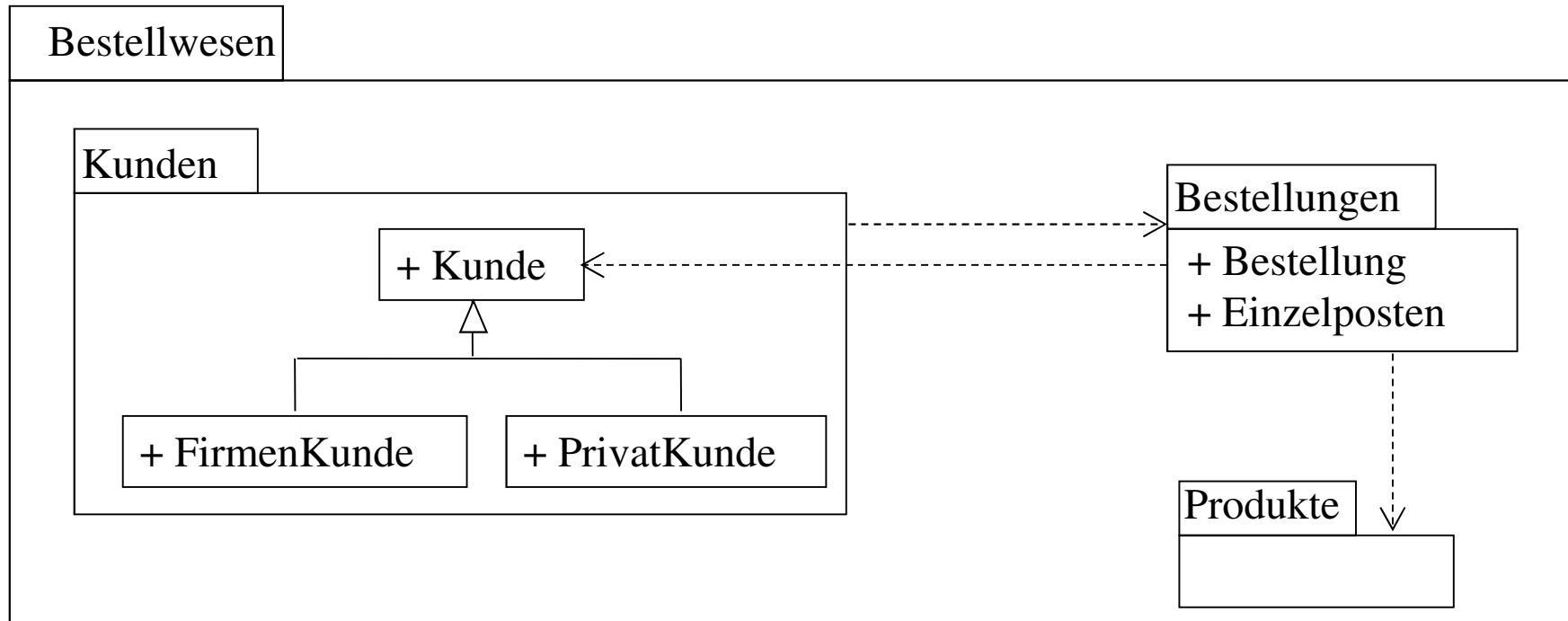


Abbildung 6.4 aus: Bruegge/Dutoit, Objektorientierte Softwaretechnik, 2004

# Beispiel: Ein Paket, mit Substruktur



Beispiel nach M. Winter, Methodische objektorientierte Softwareentwicklung, dpunkt-Verlag, 2005

## Legende:

- +: öffentliche Sichtbarkeit: Modellelement außerhalb des Paketes sichtbar
- Bestellung, Einzelposten: Kurzrepräsentation im Paket enthaltener Klassen

# Bemerkungen zur Zerlegung in Pakete

- Gute Zerlegung ist nicht immer einfach und muss aktiv entworfen werden!
- Top-down (oft besser!) oder bottom-up
- Faustregel:
  - **Schwache Kopplung ...** (zwischen Paketen)
    - Grund: Änderungen in einem Paket sollen möglichst geringe Auswirkungen auf andere Pakete haben
    - Pakete sollen einzeln implementierbar und testbar sein
    - Vererbungshierarchie (wenn möglich) nicht schneiden
    - keine Aggregation / Komposition durchtrennen
    - möglichst wenig Assoziationen auftrennen
  - **... und starke Kohäsion (hoher Zusammenhalt)** (innerhalb von Paketen)
    - Zerlegung in logisch zusammengehörige Themenbereiche
    - Paket sollte nicht nur Ansammlung unzusammengehöriger Klassen sein
- Beachte aber: Starke Kopplung ist unkritisch, wenn Änderungen wenig wahrscheinlich

# Architekturstile / -muster

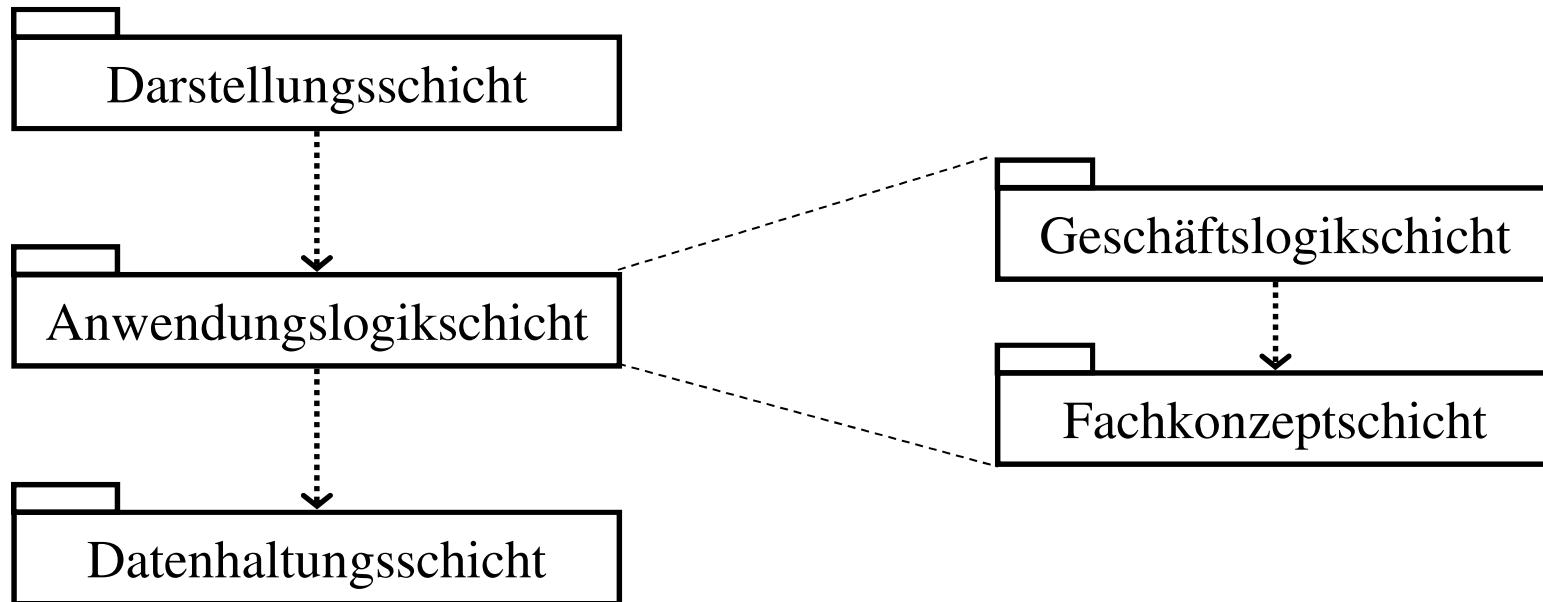
Architekturstil (Architekturmuster):

- Schema für den Aufbau von Architekturen, dass sich in verschiedenen Systemen als nützlich erwiesen hat
- Beispiele:
  - Schichtenarchitekturen
  - Model/View/Controller-Architektur von Systemen mit graphischer Benutzeroberfläche
  - Client-Server
  - Depot
  - Pipe and Filter-Architekturen
  - ...

Besonders wichtiges Prinzip für Grob-Architektur:

- Schichtenarchitektur:
  - Hierarchische Struktur der Teilsysteme
  - Teilsystem darf nur auf Teilsysteme darunterliegender Schichten zugreifen
  - Vorteil: Abhängigkeit nur in einer Richtung

# Erinnerung: Häufige Grobarchitektur von Softwaresystemen: Drei-Säulen-Architektur



- Geschlossene Schichtenarchitektur
- Vorteile z.B.:
  - Darstellungsschicht oder Datenhaltungsschicht (relativ) leicht austauschbar
- Nur Grobarchitektur: Weitere Subsystemzerlegung im Inneren der Schichten muss zusätzlich festgelegt werden !
- Dabei ggf. weitere Schichtenzerlegung oder komplexere Architektur

# Aufgaben & Ziele des Systementwurfs

- 1.
- 2.
3. Auswahl von Softwarekomponenten
- 4.
- 5.

# Auswahl der Softwarekomponenten

- Welche existierenden Softwarekomponenten sollen/können (wieder-) verwendet werden ?
- Beispiele:
  - GUI-Frameworks, z.B. Java AWT oder Swing
  - Persistenz-Frameworks, z.B. Hibernate, Serialisierung, ...
  - Middleware als Kommunikationsinfrastruktur bei verteilten Anwendungen...
  - Teile von Altsystemen...

# Aufgaben & Ziele des Systementwurfs

- 1.
- 2.
- 3.
4. Persistente Daten identifizieren und Persistenzmechanismus festlegen
- 5.

# Persistenz

- *Persistente Daten (dauerhafte Daten)* sind Daten, die über eine einmalige Ausführung des Systems hinaus Bestand haben.
  - Bsp. 1: Dokumente in Textverarbeitungssystemen
  - Bsp. 2: Benutzerdaten, Buchbestand, Entleihdaten in Bibliothekssystem

# Persistenz

## Aufgaben im Systementwurf:

- Persistente Daten identifizieren
  - Typischerweise sind dies die Entitätsobjekte !
  - Ansonsten: Analysemodell überprüfen/überarbeiten
- Mechanismus zur Sicherstellung der Persistenz auswählen

# Typische Mechanismen zur persistenten Datenhaltung

(Text-) Dateien in Standard-Format (z.B. CSV oder XML)

Dateien + (Java-) Serialisierung

- class Beispiel implements Serializable
- Variante: XML-Serialisierung (z.B. mit XStream-Bibliothek)

(Relationale) Datenbank

- Datenbankanbindung in Java z.B. mit Hilfe von
  - JDBC: grundlegender Unterstützung des Zugriffs auf Datenbanken; erfordert Programmierung der Abbildung der Objektstrukturen von Hand
  - Hibernate: Persistenzframework; unterstützt die Abbildung der Objektstrukturen
  - Beides erfordert gewisses Maß an Einarbeitung
  - Empfehlung, wenn Entscheidung für Datenbank:  
Java Persistence API mit Hibernate: Einführendes Beispiel im BSCW!

Mischung davon ...

# Zur Abwägung zwischen Dateien und Datenbanken als Persistenzmechanismus

## Dateien + Serialisierung

Vorteil:

- Einfacherer Zugang

Nachteile:

- Skaliert schlechter
- Austausch von Daten zwischen verschiedenen Anwendungen problematisch
- Änderungen des Datenmodells problematisch

## Datenbanken

Vorteile:

- skaliert besser
- stabiler bei Änderungen des Datenmodells
- einfacherer Austausch von Daten zwischen verschiedenen Anwendungen
- komplexe Anfragen an Datenbestand unterstützt

Nachteil:

- Schwererer Zugang
- Aufwändiger Realisierung

# Aufgaben & Ziele des Systementwurfs

- 1.
- 2.
- 3.
- 4.
5. Weitere Entwurfsfragen klären

# Weitere Entwurfsfragen

- Zugriffskontrolle und Sicherheit
  - Wer hat Zugriff auf was? Wie soll die Zugriffskontrolle realisiert werden ?
- Randanwendungsfälle
  - Wie wird das System gestartet, beendet, konfiguriert, ... ?
- ...

# Inhaltsverzeichnis des SoPra-Entwurfsdokuments

## 1. Einleitung

Textuelle Beschreibung des Zwecks des Systems

## 2. Systementwurf

### 2.1. Entwurfsziele

Textuelle Beschreibung der Entwurfsziele und ihrer Priorisierung;  
kurze Begründung für die getroffenen Entscheidungen

### 2.2. Systemzerlegung

Paketdiagramm der Subsystemzerlegung; kurze textuelle Beschreibung der Aufgaben der Subsysteme und Begründung für diese Zerlegung

### 2.3. Verwendung existierender Software-Komponenten

kurze Beschreibung und Begründung

### 2.4. Management persistenter Daten

Auflistung, welche Objekte persistent zu halten sind;  
Beschreibung und Begründung des gewählten Persistenzmechanismus

### 2.5. Sonstiges

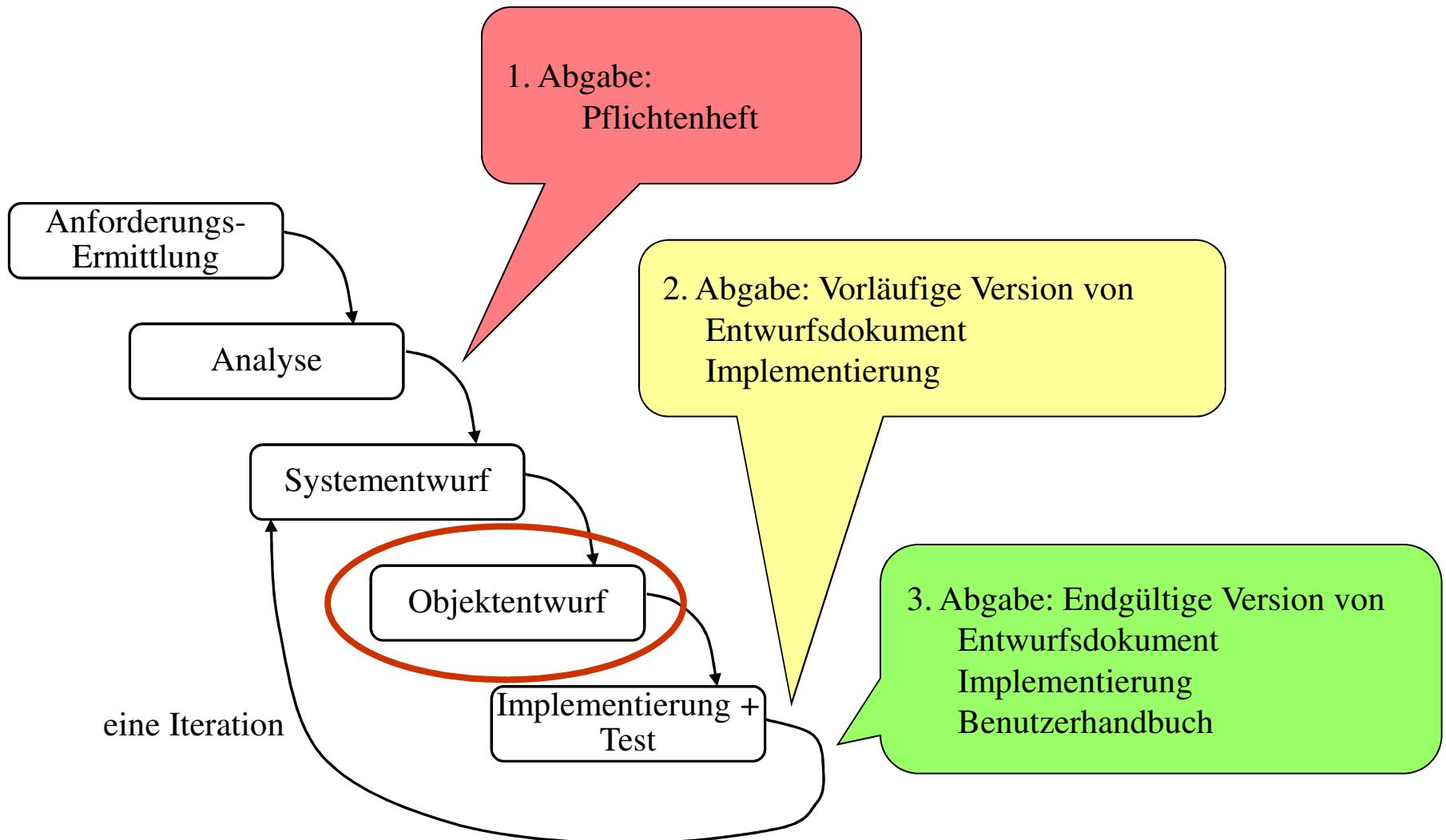
ggf. Beschreibung und Begründung weiterer Systementwurfsentscheidungen;  
dieser Abschnitt kann entfallen

## 3. Objektentwurf → Struktur folgt

## 4. Glossar

## 5. Anhang → Struktur folgt

# SoPra-Vorgehensmodell



# Inhaltsverzeichnis des SoPra-Entwurfsdokuments

## 1. Einleitung

Textuelle Beschreibung des Zwecks des Systems

## 2. Systementwurf

→ Struktur siehe oben

## 3. Objektentwurf

### 3.1. Abwägungen des Objektentwurfs

Textuelle Beschreibung der Überlegungen, die zur Entscheidung für den vorliegenden Objektentwurf geführt haben

### 3.2. Klassenmodell der Entitätsklassen

UML-Klassendiagramm der überarbeiteten Entitätsklassen und ihrer Beziehungen inkl. Klassenschnittstellen (Attribute, Methoden inkl. Typen, Signatur, Sichtbarkeit).

Die Entitätsklassen sind (in der Regel) die Klassen, deren Objekte persistent gehalten werden

### 3.3. Dokumentation weiterer interessanter Ausschnitte des Entwurfsklassenmodells

Jeweils UML-Klassendiagramme für Systeme von Klassen mit interessanten/nicht-trivialen Beziehungen, z.B. bei Verwendung von Entwurfsmustern.

Zusätzlich jeweils kurze textuelle Erläuterung des gezeigten Ausschnitts des Klassenmodells.

**Anmerkung:** Die textuelle Beschreibung der obigen und aller weiteren Klassen des Entwurfsklassenmodells erfolgt mit Javadoc im Java-Quelltext.

## 4. Glossar

## 5. Anhang

Javadoc des Codes inkl. Spezifikation nicht-trivialer Verträge und Invarianten (textuell; Formulierung in OCL nicht notwendig); Anhang nur elektronisch abgeben

# Ziele des Objektentwurfs

Ausgangspunkt:

Statisches Modell (Objektmodell) aus der Analyse

Ziele:

1. Identifikation + Ergänzung fehlender
  - Klassen (Anwendungsdomänen- und Lösungsdomänenklassen)
  - Attribute
  - Methoden (Operationen)
  - Assoziationen
2. Festlegung von Sichtbarkeit, Typen/Signatur von Attributen und Methoden
3. Spezifikation von Verträgen für Methoden und von Klasseninvarianten
4. Wiederverwendung: Benutzung von Mustern, Komponenten, Bibliotheken ...
5. Modelltransformationen

Erstelltes Modell bildet Basis für die Implementierung !

# Ziele des Objektentwurfs

Ziele:

1. Identifikation + Ergänzung fehlender
  - Klassen (Anwendungsdomänen- und Lösungsdomänenklassen)
  - Attribute
  - Methoden (Operationen)
  - Assoziationen
- 2.
- 3.
- 4.
- 5.

# Erinnerung: Technik zum Finden fehlender Klassen, Methoden und Assoziationen

- Szenarien für Anwendungsfälle im Detail durchspielen
- Erinnerung: Hilfsmittel dazu: Interaktionsdiagramme
- Zwei Typen von Interaktionsdiagrammen
  - Sequenzdiagramme: Fokus auf zeitlicher Abfolge von Nachrichten
  - Kommunikationsdiagramme (früher: Kollaborationsdiagramme): Fokus auf Topologie des Objektgeflechts (d.h. auf den Assoziationen)
- Beachte:
  - Nachrichtenaustausch von Objekt vom Typ A zu Objekt vom Typ B:
    - entspricht Methodenaufruf auf Objekt vom Typ B
      - ) Methoden identifizieren
    - erfordert, dass aufrufendes Objekt eine Referenz auf das Objekt vom Typ B besitzt, z.B. durch Assoziation von der Klasse A zur Klasse B oder vorherigen Nachrichtenaustausch
      - ) Assoziationen identifizieren
- Durchführung während des Objektentwurfs nur für interessante Teilespekte.  
Dies muss **nicht** im Entwurfsdokument dokumentiert werden.
- In der Praxis auch: Iterieren zwischen Objektentwurf und Implementierung

# Ziele des Objektentwurfs



1.

- 
- 
- 
- 

2. Festlegung von Sichtbarkeit, Typen bzw. Signaturen von Attributen bzw. Methoden

3.

4.

5.

# Sichtbarkeits-Informationen

UML und Java definieren vier Sichtbarkeitslevel:

- Privat (Java: private) (nur Klassen-Implementierer): UML-Kennzeichnung: -
  - Auf ein privates Attribut kann nur von der Klasse aus, in der es definiert es, zugegriffen werden.
  - Eine private Methode kann nur von der Klasse aus, in der sie definiert ist, aufgerufen werden.
  - Auf private Attribute und Methoden kann von Subklassen oder anderen Klassen aus nicht zugegriffen werden.
- Geschützt (Java: protected) (auch Klassen-Erweiterer): UML-Kennzeichnung: #
  - Auf ein geschütztes Attribut (eine geschützte Methode) kann von der Klasse, in der es definiert ist, und von jeder ihrer Subklassen aus zugegriffen werden.
- Öffentlich (Java: public) (alle Klassen-Entwickler): UML-Kennzeichnung: +
  - Auf ein öffentliches Attribut (eine öffentliche Methode) kann von jeder Klasse aus zugegriffen werden.
- Paket (package) (alle im selben Paket): UML-Kennzeichnung: ~
  - Auf ein Attribut (eine Operation) mit Sichtbarkeit package, kann von allen Klassen innerhalb des Pakets, in dem sich die Klasse, in der das Attribut (die Operation) definiert ist, zugegriffen werden
  - In Java gilt Sichtbarkeit im Packet immer, wenn nichts Anderes spezifiziert wird.

# Information-Hiding-Prinzipien

- Öffentliche Schnittstelle von Klassen und Subsystemen sorgfältig definieren
- Das “Need to know”-Prinzip anwenden:
  - Zugriffsrechte so restriktiv wie möglich festlegen:
    - Nur Information, auf die tatsächlich von anderen Klassen aus zugegriffen werden muss, zugänglich machen.
    - Vorteile: Schwächere Kopplung, bessere Kapselung, leichtere Modifizierbarkeit, leitere Lokalisierung von Fehlern
  - Öffentliche Attribute im Code vermeiden; ggf. öffentliche set- und get-Methoden einführen, um Zugriff ggf. leichter überwachen zu können

# Ziele des Objektentwurfs



1.

- 
- 
- 
- 

2.

3. Spezifikation von Verträgen für Methoden und von Klasseninvarianten

4.

5.

# Typ- und Signaturinformation

## Beobachtung:

Spezifikation des Typs von Attributen und der Signatur von Operationen ist allein oft zu schwach, um legitime Wertebereiche von Attributbelegungen, Parametern und Resultaten von Methoden zu spezifizieren

⇒ **Design-by-contract:**  
zusätzlich „Verträge“ spezifizieren

# Design by Contract

## Ziel:

- Möglichst fehlerfreie („korrekte“) Software
- Populär gemacht von Bertrand Meyer im Kontext der OO-Sprache Eiffel

## Mittel:

- Spezifikation von Methoden mit Vor- und Nachbedingungen
- Spezifikation von Klasseninvarianten

## Vor-/Nachbedingungen und Klasseninvarianten ...

- ... bieten eine (i.A. partielle) Spezifikation für das Verhalten der Instanzen der Klasse
- ... abstrakte Verhaltensbeschreibung (WAS statt WIE)
- ... dokumentieren Annahmen und Verantwortlichkeiten von Klassenbenutzern und Klassenimplementierern
- ... bieten Ansatzpunkte für formale Korrektheitsbeweise von Programmen oder, pragmatischer, **Überprüfen von Annahmen zur Laufzeit**

# Design by Contract

## Idee:

Aufrufer (Benutzer) einer Methode m() ...

- muss sicherstellen, dass die Vorbedingung von m() gilt
- kann sich darauf verlassen, dass die Nachbedingung bei Terminierung von m() gilt

Implementierer einer Methode m() ...

- kann sich darauf verlassen, dass beim Aufruf die Vorbedingung von m() gilt
- muss sicherstellen, dass nach Ausführung des Methodenrumpfes die Nachbedingung von m() gilt

Klasseninvariante einer Klasse K

- Bedingung, die in jedem stabilen Zustand eines Objektes der Klasse K gilt, d.h. nach der Initialisierung und vor und nach jedem Aufruf einer Methode durch ein anderes Objekt

# Beispiel: Klasse, die einen Stack von int-Werten beschränkter Kapazität (max. 42 Einträge) realisieren soll

| BoundedStack       |
|--------------------|
| anzElem:int        |
| isEmpty(): boolean |
| isFull():boolean   |
| push(x:int)        |
| pop()              |
| top():int          |

Klasseninvariante:

anzElem  $\leq$  0

isEmpty():

pre true

post result=true , anzElem = 0

push(x:int):

pre not isFull()

post not isEmpty() and anzElem = anzElem@pre + 1

isFull():

pre true

post result=true , anzElem = 42

pop():

pre not isEmpty()

post not isFull() and anzElem = anzElem@pre - 1

top():

pre not isEmpty()

post anzElem = anzElem@pre

# Design by Contract

## Vertrag:

- Vereinbarung, von der beide Vertragsparteien profitieren
- Jede Partei zieht aus Vertrag einen Nutzen und geht eine Verpflichtung ein

## Beispiel: Kaufvertrag

|               | Käufer                       | Verkäufer                         |
|---------------|------------------------------|-----------------------------------|
| Verpflichtung | Muss Geld bezahlen           | Muss Eigentum an der Ware abgeben |
| Nutzen        | Erwirbt Eigentum an der Ware | Bekommt Geld                      |

# Vor- und Nachbedingung als Vertrag

## Vertragspartner:

- Implementierer des Methodenrumpfes
- Aufrufer der Methode

|               | Implementierer   | Aufrufer  |
|---------------|--|---|
| Verpflichtung | Muss dafür sorgen, dass bei Terminierung die Nachbedingung gilt    | Muss sicherstellen, dass die Vorbedingung beim Aufruf der Methode erfüllt ist         |
| Nutzen        | Kann sich darauf verlassen, dass beim Aufruf die Vorbedingung gilt | Kann sich darauf verlassen, dass die Nachbedingung nach Terminierung der Methode gilt |

# Zur Notation von Vor-/Nachbedingungen und Klasseninvarianten

## Informell:

- textuell
- oft hilfreich und klar genug

## Formal

- in UML: z.B. OCL (Object Constraint Language)  
→ Zusatzfolien
- in Java: z.B. JML (Java Modelling Language)

## Manchmal die beste Lösung:

- beides, informell und formal:
  - Informelle Beschreibung erleichtert Zugang zu formaler Beschreibung
  - Formale Beschreibung präzisiert informelle Beschreibung

Richtlinien zum Dokumentieren und Prüfen von Kontrakten: Siehe unten!

# Ziele des Objektentwurfs

•

1.

- 
- 
- 
- 

2.

3.

4. Wiederverwendung von Mustern, Komponenten, Programmgerüsten,  
Klassenbibliotheken

5.

# Wiederverwendung

## Vorteile:

- Geringerer Entwicklungsaufwand
- Niedrigeres Risiko
- Verbreiteter Gebrauch von Standardbegriffen;  
dadurch einfachere und klarere Kommunikation (z.B. Musternamen)
- Erhöhte Verlässlichkeit

# Häufige Wiederverwendungsmechanismen

## Entwurfsmuster

- Wiederverwendbarer, bewährter Entwurf für ein gegebenes Problem

## Komponenten:

- in sich geschlossene Instanzen von Klassen, die zu Anwendungen zusammengesetzt werden

## Programmgerüst (*Framework*):

- kooperierendes System von Klassen als Architektskelett für gewissen Typ verwandter Anwendungen
- oft aktiv, d.h. Framework ruft Methoden der Anwendung auf:
  - Hollywood-Prinzip: „don't call us, we'll call you“ ...
- GUI-Frameworks: z.B. Java AWT, Swing, SWT
- Kommunikations-Frameworks für verteilte Anwendungen: z.B. CORBA

## Klassenbibliothek:

- stellt grundlegende Funktionalität als Sammlung unabhängiger Klassen zur Verfügung
- in der Regel passiv: d.h. Methoden der Klassen der Bibliothek werden aufgerufen, aber rufen selber keine Methoden der Anwendung auf

# Ziele des Objektentwurfs

•

1.

- 
- 
- 
- 

2.

3.

4.

5. Modelltransformationen: siehe folgenden Teil über „Transformationen“

# Inhaltsverzeichnis des SoPra-Entwurfsdokuments

## 1. Einleitung

Textuelle Beschreibung des Zwecks des Systems

## 2. Systementwurf

→ Struktur siehe oben

## 3. Objektentwurf

### 3.1. Abwägungen des Objektentwurfs

Textuelle Beschreibung der Überlegungen, die zur Entscheidung für den vorliegenden Objektentwurf geführt haben

### 3.2. Klassenmodell der Entitätsklassen

UML-Klassendiagramm der überarbeiteten Entitätsklassen und ihrer Beziehungen inkl. Klassenschnittstellen (Attribute, Methoden inkl. Typen, Signatur, Sichtbarkeit).

Die Entitätsklassen sind (in der Regel) die Klassen, deren Objekte persistent gehalten werden

### 3.3. Dokumentation weiterer interessanter Teile des Entwurfsklassenmodells

jeweils UML-Klassendiagramme für Systeme von Klassen mit interessanten/nicht-trivialen Beziehungen, z.B. bei Verwendung von Entwurfsmustern.

Zusätzlich jeweils kurze textuelle Erläuterung des gezeigten Ausschnitts des Klassenmodells.

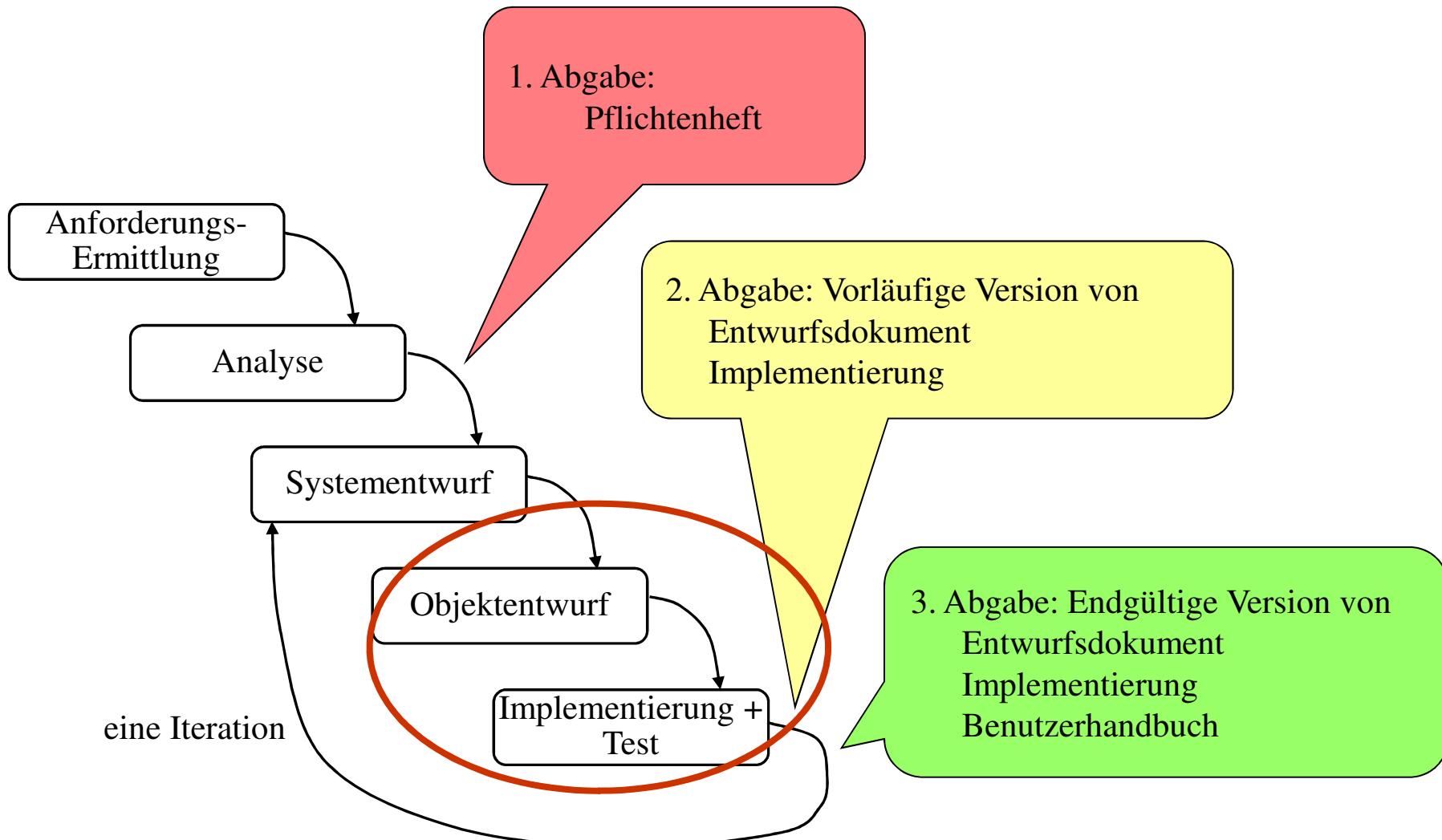
**Anmerkung:** Die textuelle Beschreibung der obigen und aller weiteren Klassen des Entwurfsklassenmodells erfolgt mit Javadoc im Java-Quelltext.

## 4. Glossar

## 5. Anhang

Javadoc des Codes inkl. Spezifikation nicht-trivialer Verträge und Invarianten (textuell; Formulierung in OCL nicht notwendig); Anhang nur elektronisch abgeben

# SoPra-Vorgehensmodell



# Transformationen

Software-Entwickler transformieren...

- ... Objektmodelle zur Verbesserung von Modularität und Effizienz
- ... Objektmodelle in Code
- ... dabei u.a. Assoziationen in (Kollektionen von) Referenzen
- ... Kontrakte in Prüfcode
- ... Klassenmodelle in Datenbankschemata
- ... Code in Code zur Verbesserung von Modularität und Effizienz
- ...

# Arten von Transformationen

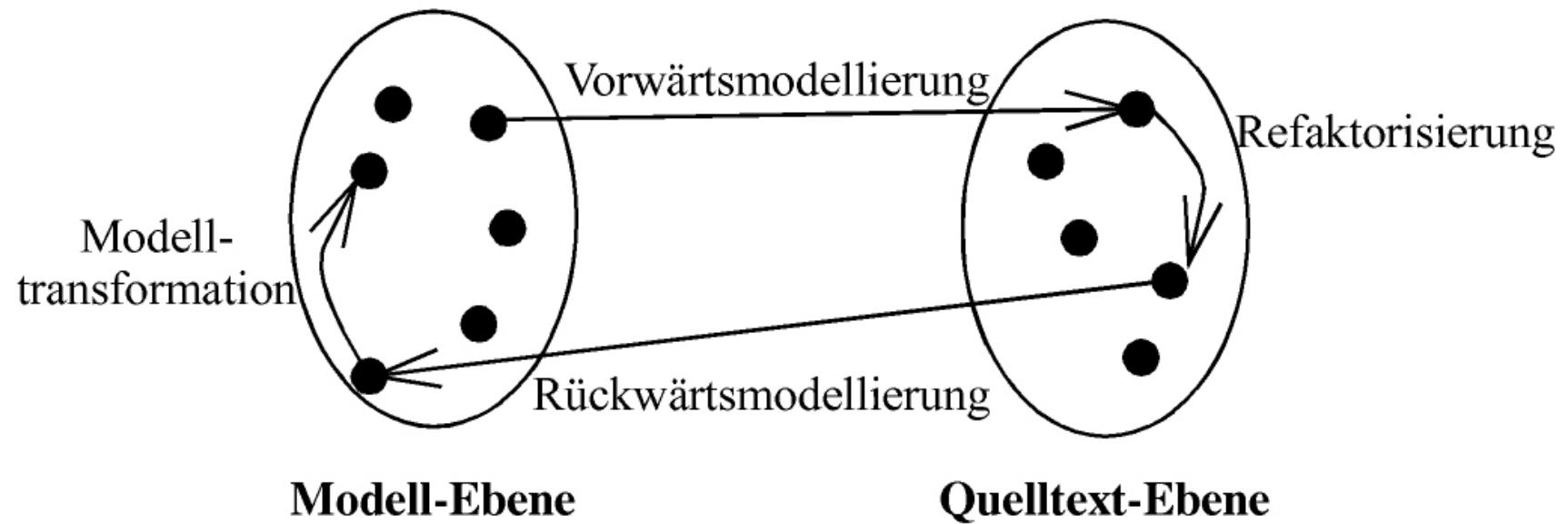


Abbildung 10.1 aus: Bruegge/Dutoit, Objektorientierte Softwaretechnik, 2004

# Beispiel einer Modelltransformation

Elimination redundanter Attribute durch Einführen einer Oberklasse:

Objektentwurfsmodell vor der Transformation



Objektentwurfsmodell nach der Transformation

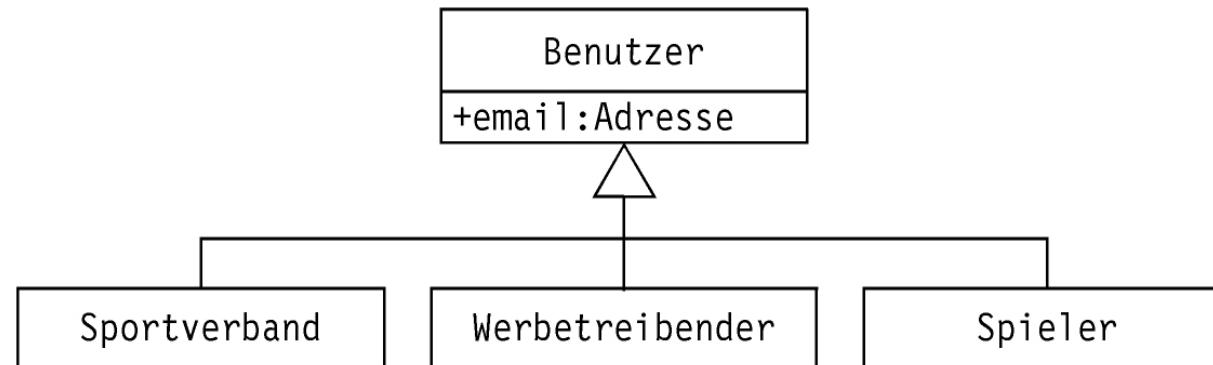


Abbildung 10.2 aus: Bruegge/Dutoit, Objektorientierte Softwaretechnik, 2004

# Anwendung einer Refaktorisierung: Feld-Ausklammerung

## Vor der Refaktorisierung

```
public class Spieler {  
    private String email;  
    //...  
}  
  
public class Sportverband {  
    private String email;  
    //...  
}  
  
public class Werbetreibender {  
    private String email;  
    //...  
}
```

## Nach der Refaktorisierung

```
public class Benutzer {  
    private String email;  
}  
  
public class Spieler extends Benutzer {  
    //...  
}  
  
public class Sportverband  
    extends Benutzer {  
    //...  
}  
  
public class Werbetreibender  
    extends Benutzer {  
    //...  
}
```

Abbildung 10.3 aus: Bruegge/Dutoit, Objektorientierte Softwaretechnik, 2004

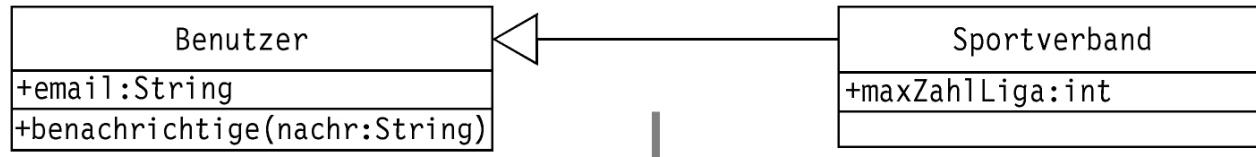
# Anwendung der Refaktorisierung Konstruktor-Ausklammerung

| Vor der Refaktorisierung   | Nach der Refaktorisierung   |
|--|---|
| <pre>public class Benutzer {<br/>    private String email;<br/>}<br/><br/>public class Spieler<br/>    extends Benutzer {<br/>    public Spieler(String email) {<br/>        this.email = email;<br/>        //...<br/>    }<br/>}<br/>public class Sportverband<br/>    extends Benutzer{<br/>    public Sportverband<br/>        (String email) {<br/>        this.email = email;<br/>        //...<br/>    }<br/>}<br/>public class Werbetreibender<br/>    extends Benutzer{<br/>    public Werbetreibender<br/>        (String email) {<br/>        this.email = email;<br/>        //...<br/>    }<br/>}</pre> | <pre>public class Benutzer {<br/>    private String email;<br/>    public Benutzer(String email)<br/>    {<br/>        this.email = email;<br/>    }<br/>}<br/>public class Spieler<br/>    extends Benutzer {<br/>    public Spieler(String email) {<br/>        super(email);<br/>        //...<br/>    }<br/>}<br/>public class Sportverband<br/>    extends Benutzer {<br/>    public Sportverband<br/>        (String email) {<br/>        super(email);<br/>        //...<br/>    }<br/>}<br/>public class Werbetreibender<br/>    extends Benutzer {<br/>    public Werbetreibender<br/>        (String email) {<br/>        super(email);<br/>        //...<br/>    }<br/>}</pre> |

Abbildung 10.4 aus: Bruegge/Dutoit, Objektorientierte Softwaretechnik, 2004

# Beispiel für Vorwärtsmodellierung

Objektentwurfsmodell vor der Transformation



Quelltext nach der Transformation

```
public class Benutzer {
    private String email;
    public String gibEmail() {
        return email;
    }
    public void setzeEmail
        (String wert){
        email = wert;
    }
    public void benachrichtige
        (String nachr) {
        // ....
    }
    /* Andere Methoden
       ausgelassen */
}

public class Sportverband extends
Benutzer {
    private int maxZahlLiga;
    public int gibMaxZahlLiga() {
        return maxZahlLiga;
    }
    public void setzeMaxZahlLiga
        (int value) {
        maxZahlLiga = value;
    }
    /* Andere Methoden
       ausgelassen */
}
```

Abbildung 10.5 aus: Bruegge/Dutoit, Objektorientierte Softwaretechnik, 2004

Beachte (Best Practice):

- Öffentliche Sichtbarkeit der Attribute `email` und `maxZahlLiga` wird in **öffentliche sichtbare set- und get- Methoden** übersetzt.
- Die Java-Attribute `email` und `maxZahlLiga` selbst haben im Code Sichtbarkeit **private**.

# Toolunterstützung für Transformationen

- Viele **Transformationen** sind mehr oder weniger **systematisch**
- Mechanische Tätigkeit;  
daher bei **manueller Durchführung fehlerträchtig**
- Transformationen werden durch **Tools unterstützt**:
  - Entwicklungsumgebungen wie Eclipse und Netbeans unterstützen Refaktorisierungen
  - UML-Tools unterstützen Vorwärtsmodellierung
  - Persistenzframeworks wie Hibernate unterstützen Abbildung von Objektmodellen (in Form von Code) in Datenbankschemata
  - ...

# Häufige Transformationen

- Optimierung des detaillierten Objektmodells
- Abbildung von Assoziationen im Programmcode
- Dokumentation von Verträgen und Abbildung auf Ausnahmen
- Abbildung von Objektmodellen auf Datenbankschemata

# Häufige Transformationen

- Optimierung des detaillierten Objektmodells
- 
- 
-

# Optimierung des detaillierten Objektentwurfs

**Ziel:** Vermeiden von Ineffizienz/Verbesserung der Laufzeit

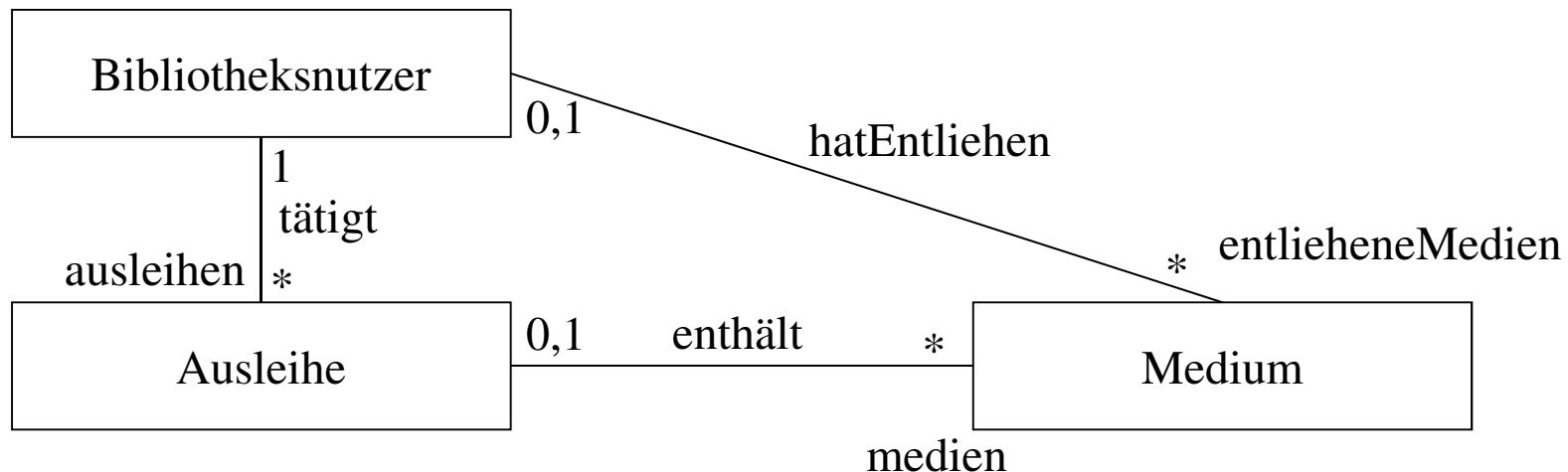
Betrachten hier:

- Optimierung von Zugriffspfaden
- Umwandlung von Klassen zu Attributen
- Verzögerung aufwendiger Berechnungen
- Zwischenspeichern von Resultaten

# Optimierung von Zugriffspfaden

- Zum Abkürzen häufig durchlaufener Navigationsketten:
  - abkürzende Assoziationen hinzufügen (nächste Folie)
- Bei Assoziationen mit „viele“-Multiplizität:
  - Suchzeit für einzelne Objekte in der Kollektion durch Verwendung von Indizierungen oder Ordnen der Objekte in der Kollektion verringern
- Schlecht platzierte Attribute verschieben
  - Dabei ggf. ganze Klassen (die durch übertriebene Modellierung entstanden sind) in Attribute verwandeln (siehe übernächste Folie)

# Abkürzen von Navigationsketten



- Assoziation „hatEntliehen“ wird dem Objektentwurf hinzugefügt, um Navigieren zwischen Medien und Nutzern besser zu unterstützen
- Klasseninvariante der Klasse Bibliotheksnutzer:  
**informelle Formulierung:**  
Das Attribut „entlieheneMedien“ eines Bibliotheksnutzers enthält genau die Medien, die in seinen Ausleihen enthalten sind  
**formal in OCL:**  
context Bibliotheksnutzer  
inv: self.entlieheneMedien = self.ausleihen.medien

# Umwandlung einer Klasse zu einem Attribut

Beispiel:

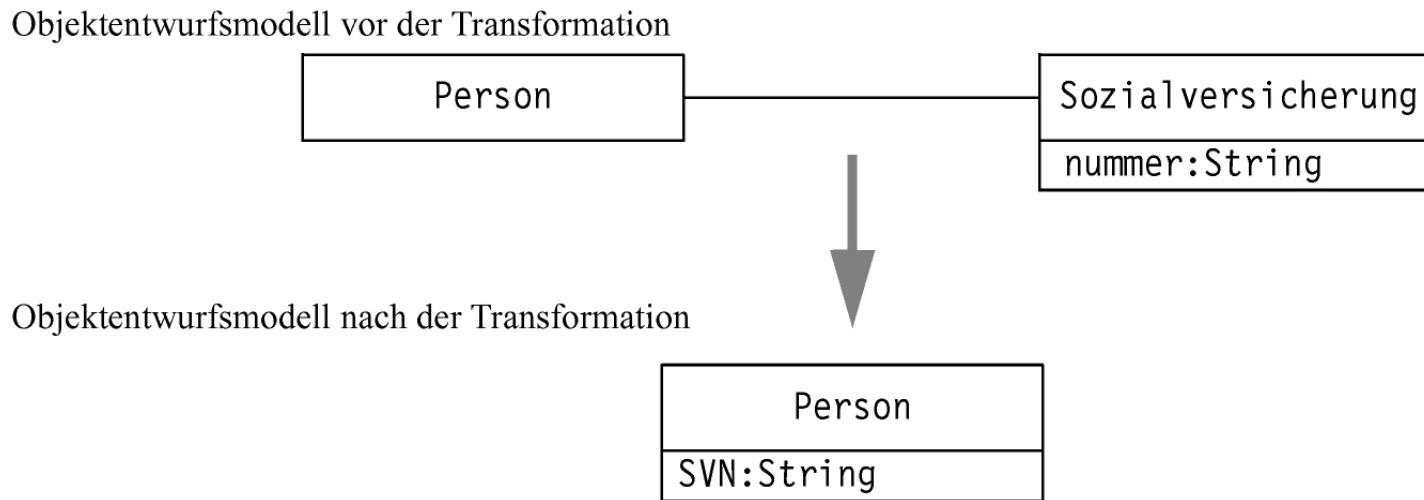


Abbildung 10.6 aus: Bruegge/Dutoit, Objektorientierte Softwaretechnik, 2004

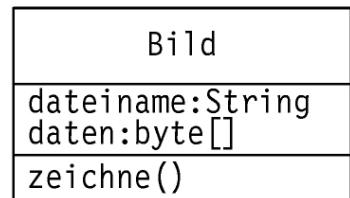
Diese Transformation ggf. anwenden, wenn:

- Klasse hat kein interessantes Verhalten
- wenig Attribute und
- nur eine Assoziation

# Verzögerung aufwendiger Berechnungen mit Hilfe des Stellvertretermusters

Beispiel:

Objektentwurfsmodell vor der Transformation



Objektentwurfsmodell nach der Transformation

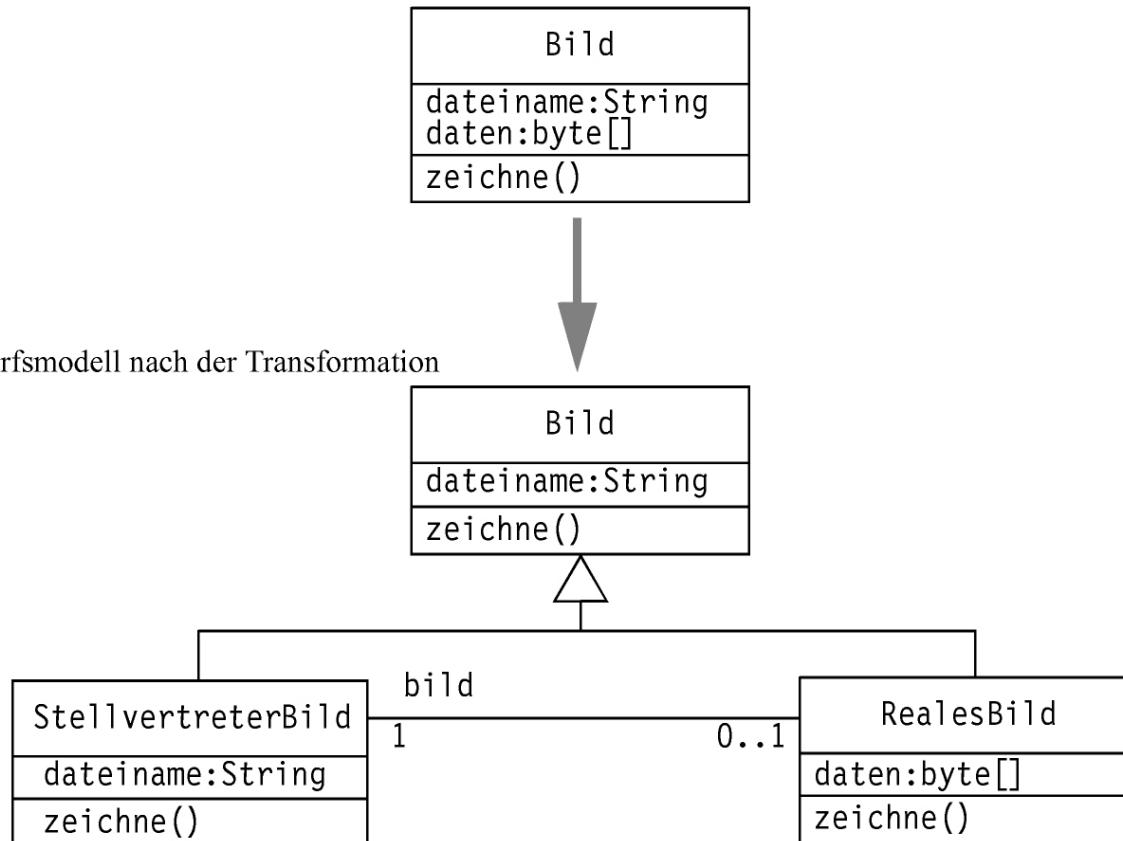


Abbildung 10.7 aus: Bruegge/Dutoit, Objektorientierte Softwaretechnik, 2004

# Verzögerung aufwendiger Berechnungen mit Hilfe des Stellvertretermusters (Forts.)

- Anwendung verwendet Oberklasse Bild
- Dadurch aus Sicht der Anwendung transparent, ob StellvertreterBild oder RealesBild verwendet wird
- Zur Laufzeit wird Instanz von StellvertreterBild anstelle des realen Bildes verwendet.
- Diese Instanz lädt das reale Bild nur (und erst dann), wenn es tatsächlich angezeigt werden soll. Anschließend delegierte es Methodenaufrufe an das reale Bild.
- Vorteil: Aufwand für Laden des Bildes entsteht nur (und erst dann), wenn das Bild tatsächlich benötigt wird.

# Zwischenspeichern von Resultaten

Idee:

- Ergebnis aufwendiger Berechnung in privatem Attribut („cache“) zwischenspeichern („cachen“)
- Bei erneuter Anfrage, Ergebnis direkt aus diesem Attribut auslesen, anstatt es neu zu berechnen
- Bei Veränderung der Daten, die der Berechnung zugrunde liegen, Attribut als ungültig kennzeichnen

Bemerkung:

Nur nützlich, wenn sich Datengrundlage selten ändert

Beispiel: Statistiken u.ä.

Bemerkung:

Sicherstellen der Konsistenz zwischengespeicherter Daten kann aufwendig und fehlerträchtig sein

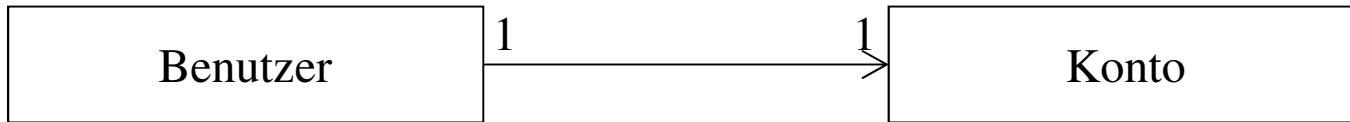
# Schlussbemerkungen zur Optimierung des detaillierten Objektentwurfs

- Optimierungstransformationen können die **Effizienz steigern**.
- Aber: Optimierungen können **Programmieraufwand erhöhen** und **Wartbarkeit verschlechtern**.
- Deshalb:
  - Optimierungstransformationen nicht „blind“ anwenden;  
Kosten und Nutzen zuvor stets abwägen!
  - Quelle der Ineffizienz zuvor genau analysieren.
  - „*First make it run, then make it run fast!*“ (Brian Kernighan zugeschrieben)

# Häufige Transformationen

- 
- Abbildung von Assoziationen im Programmcode
  - wir zeigen hier diese Transformation an Beispielen
- 
-

# Realisierung einer unidirektionalen Eins-zu-Eins Assoziation



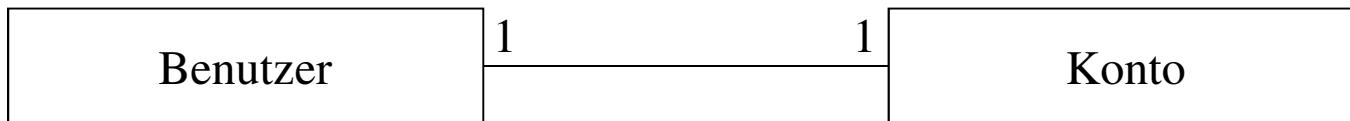
```
public class Benutzer {  
    /* Das Feld konto wird im Konstruktor  
     * initialisiert und nie verändert */  
    private Konto konto;  
  
    public Benutzer() {  
        konto = new Konto();  
    }  
  
    public Konto getKonto() {  
        return konto;  
    }  
}
```

nach Abbildung 10.8 aus: Bruegge/Dutoit, Objektorientierte Softwaretechnik, 2004

Annahme:

- Kontoobjekte werden nur bei Erzeugen von Instanzen von Benutzer erzeugt

# Realisierung einer bidirektionalen Eins-zu-Eins Assoziation



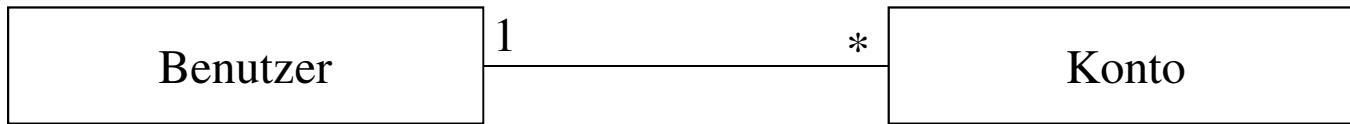
```
public class Benutzer {  
    /* Das Feld konto wird im Konstruktor  
     * initialisiert und nie verändert */  
    private Konto konto;  
  
    public Benutzer() {  
        konto = new Konto(this);  
    }  
  
    public Konto getKonto() {  
        return konto;  
    }  
}
```

```
public class Konto {  
    /* Das Feld besitzer wird im Konstruktor  
     * initialisiert und nie verändert */  
    private Benutzer besitzer;  
  
    public Konto(Benutzer besitzer) {  
        this.besitzer = besitzer;  
    }  
  
    public Benutzer getBesitzer() {  
        return besitzer;  
    }  
}
```

nach Abbildung 10.9 aus: Bruegge/Dutoit, Objektorientierte Softwaretechnik, 2004

Gleiche Annahme wie auf vorheriger Folie...

# Bidirektionale Eins-zu-Viele-Assoziation



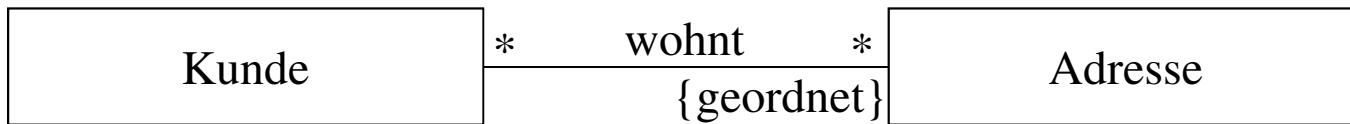
```
public class Benutzer {  
    private Set<Konto> konten;  
  
    public Benutzer() {  
        konten = new HashSet<Konto>();  
    }  
  
    public void fügeHinzuKonto(Konto k) {  
        konten.add(k);  
        k.setzeBesitzer(this);  
    }  
  
    public void entferneKonto(Konto k) {  
        konten.remove(k);  
        k.setzeBesitzer( null );  
    }  
}
```

```
public class Konto {  
    private Benutzer besitzer;  
  
    public void setzeBesitzer(Benutzer neuerBesitzer) {  
        if (besitzer!=neuerBesitzer) {  
            // Assoziation mit altem Besitzer ggf. löschen  
            Benutzer alt = besitzer;  
            if (alt != null) {  
                besitzer = null;  
                alt.entferneKonto(this);  
            }  
            // Assoziation mit neuem Besitzer herstellen  
            besitzer = neuerBesitzer;  
            if (neuerBesitzer != null)  
                neuerBesitzer.fügeHinzuKonto(this);  
        }  
    }  
}
```

nach Abbildung 10.10 aus: Bruegge/Dutoit, Objektorientierte Softwaretechnik, 2004

- Kontoobjekte werden unabhängig von Instanzen von Benutzer erzeugt:  
    ⇒ Operationen zur konsistenten Pflege der Assoziation ergänzen.
- Beachte: Klasse Benutzer hat jetzt ein kollektionswertiges Feld konten.

# Bidirektionale Viele-zu-Viele-Assoziation



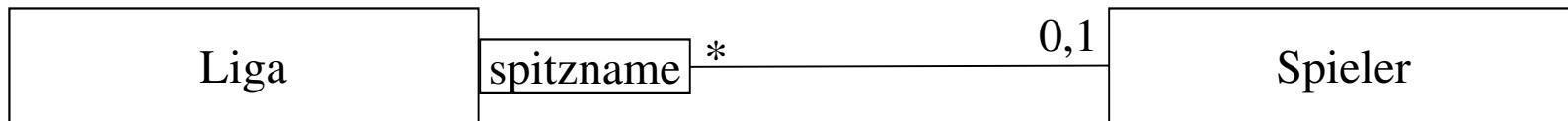
```
public class Kunde {  
    private List<Adresse> adressen;  
  
    public Kunde() {  
        adressen = new ArrayList<Adresse>();  
    }  
  
    public void fügeHinzuAdresse(Adresse a) {  
        if (!adressen.contains(a)) {  
            adressen.add(a);  
            a.fügeHinzuKunde(this);  
        }  
    }  
}
```

```
public class Adresse {  
    private Set<Kunde> kunden;  
  
    public Adresse() {  
        kunden = new HashSet<Kunde>();  
    }  
  
    public void fügeHinzuKunde(Kunde k) {  
        if (!kunden.contains(k)) {  
            kunden.add(k);  
            k.fügeHinzuAdresse(this);  
        }  
    }  
}
```

nach Abbildung 10.11 aus: Bruegge/Dutoit, Objektorientierte Softwaretechnik, 2004

- Wieder werden Operationen zur konsistenten Pflege der Assoziation ergänzt.
- Beachte auch: Geordnete Assoziation durch Liste realisiert.

# Bidirektionale qualifizierte Assoziation



```
public class Liga {  
    private Map<String, Spieler> spieler;  
  
    ...  
  
    public void fügeHinzuSpieler  
        (String spitzname, Spieler p) {  
        if (!spieler.containsKey(spitzname)) {  
            spieler.put(spitzname, p);  
            p.fügeHinzuLiga(spitzname, this);  
        }  
    }  
}
```

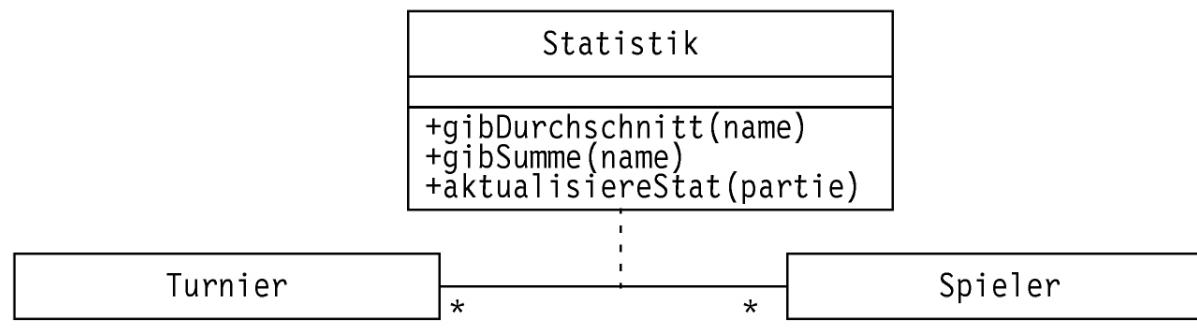
```
public class Spieler {  
    private Map<Liga, String> liga;  
  
    ...  
  
    public void fügeHinzuLiga(String spitzname, Liga l) {  
        if (!liga.containsKey(l)) {  
            liga.put(l, spitzname);  
            l.fügeHinzuSpieler(spitzname, this);  
        }  
    }  
}
```

nach Abbildung 10.12 aus: Bruegge/Dutoit, Objektorientierte Softwaretechnik, 2004

- Realisierung der qualifizierten Assoziation mit Hilfe von Maps

# Transformation einer Assoziationsklasse in eine Koordinatorklasse

Objektentwurfsmodell vor der Transformation



Objektentwurfsmodell nach der Transformation

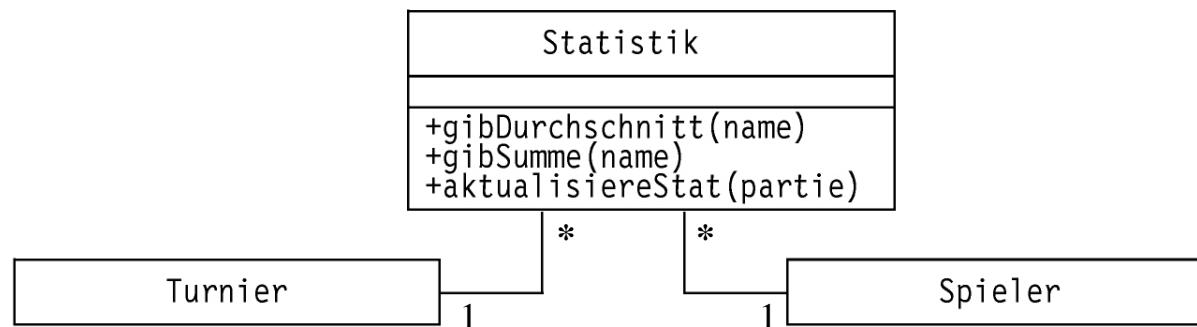


Abbildung 10.13 aus: Bruegge/Dutoit, Objektorientierte Softwaretechnik, 2004

# Häufige Transformationen

- 
- 
- Dokumentation von Verträgen und Abbildung auf Ausnahmen
-

# Beispiel: Klasse, die einen Stack von int-Werten beschränkter Kapazität realisieren soll

| BoundedStack         |
|----------------------|
| - anzElem:int        |
| + isEmpty(): boolean |
| + isFull(): boolean  |
| + push(x:int)        |
| + pop()              |
| + top():int          |

Klasseninvariante:

$$\text{anzElem} \geq 0 \wedge \text{anzElem} \leq \text{BOUND}$$

BOUND: int wird dem Konstruktor übergeben

isEmpty():

pre true

post result=true , anzElem = 0

**push(x:int):**

**pre not isFull()**

**post not isEmpty() and anzElem = anzElem@pre + 1**

isFull():

pre true

post result=true , anzElem = BOUND

**pop():**

**pre not isEmpty()**

**post not isFull() and anzElem = anzElem@pre - 1**

**top():**

**pre Not isEmpty()**

**post true**

# Javadoc-Kommentar für Klasse inklusive Dokumentation der Klasseninvariante

```
/**  
 * Klasse BoundedStack realisiert einen Stack beschränkter Kapazität  
 * Hier könnte eine ausführlichere Beschreibung stehen...  
 * @author Markus Müller-Olm  
 * @version 1.0  
 * @inv anzElem >= 0 && anzElem <= bound  
 */  
  
public class BoundedStack {  
    private Vector<Integer> vec = new Vector<Integer>();  
    private int bound;  
    private int anzElem = 0;  
  
    public BoundedStack(int bound) { this.bound = bound; }  
  
    public boolean IsEmpty() { return anzElem == 0; }  
  
    public boolean isFull() { return anzElem == bound; }  
    ...  
}
```

# Javadoc-Kommentar für Methode inklusive Dokumentation der Vor- und Nachbedingung

```
/**  
 * Legt ein Element auf den Stack  
 * Hier könnte eine ausführlichere Beschreibung  
 * stehen, wenn dies angemessen ist  
 * @param x das Element, das auf den Stack gelegt werden soll  
 * @pre der Stack ist nicht voll  
 * @post anzElem ist inkrementiert worden  
 */  
  
public void push(int x) {  
    vec.addElement(x);  
    anzElem++;  
}  
  
/**  
 * Gibt das oberste Element auf dem Stack zurück  
 * Dabei wird der Stack selbst nicht verändert  
 * @return der oberste Wert auf dem Stack  
 * @pre der Stack ist nicht leer  
 */  
  
public int top() { return vec.lastElement(); }
```

- Javadoc generiert dokumentierende html-Seiten aus derart annotiertem Code.
- Hinweise zu javadoc auf Homepage beachten, damit javadoc die Tags für Kontrakte interpretiert !

# Überprüfen der Kontrakte zur Laufzeit

- Neben Dokumentation der Kontrakte:  
Überprüfung zur Laufzeit, ob Kontrakte tatsächlich eingehalten werden  
⇒ stabilerer Code; erleichterte Fehlersuche:  
Fehler werden dichter an der Fehlerursache aufgedeckt
- Dazu assert-Anweisung (ab Java 1.4) verwenden:  
`assert <Bedingung> [ :<Ausdruck> ] ;`
- Assert erzeugt `java.lang.AssertionError`, falls die angegebene Bedingung bei Ausführung der assert-Anweisung verletzt ist; Wert des Ausdrucks wird als Fehlerbeschreibung mit ausgegeben
- Überprüfen von assertions muss explizit eingeschaltet werden:  
`java -ea ...` (ea = enableassertion)  
(z.B. im Konfigurations-Dialog der IDE)

# Überprüfen der Kontrakte zur Laufzeit

Überprüft werden kann:

- Gültigkeit der Vorbedingung zu Beginn des Methodenrumpfes
- Gültigkeit der Nachbedingung am Ende des Methodenrumpfes
- Gültigkeit der Klasseninvariante zu [Beginn und] Ende von Methodenrümpfen

# Beispiel: Überprüfen der Verträge mit Hilfe von assert-Anweisungen

```
public void push(int x) {  
    // Vorbedingung prüfen  
    assert !isFull() :  
        "Vorbedingung von push verletzt: push auf vollem Stack  
        aufgerufen";  
  
    // Vorzustand zur Überprüfung der Nachbedingung retten  
    int anzElemAtPre = anzElem;  
  
    vec.addElement(x);  
    anzElem++;  
  
    // Nachbedingung prüfen  
    assert anzElem == anzElemAtPre + 1 :  
        "Nachbedingung von push verletzt";  
  
    // Klasseninvariante prüfen  
    assert anzElem >= 0 && anzElem <= bound :  
        "Klasseninvariante verletzt bei Terminierung von push";  
}
```

Erklärung der Fehlerursache ist optional

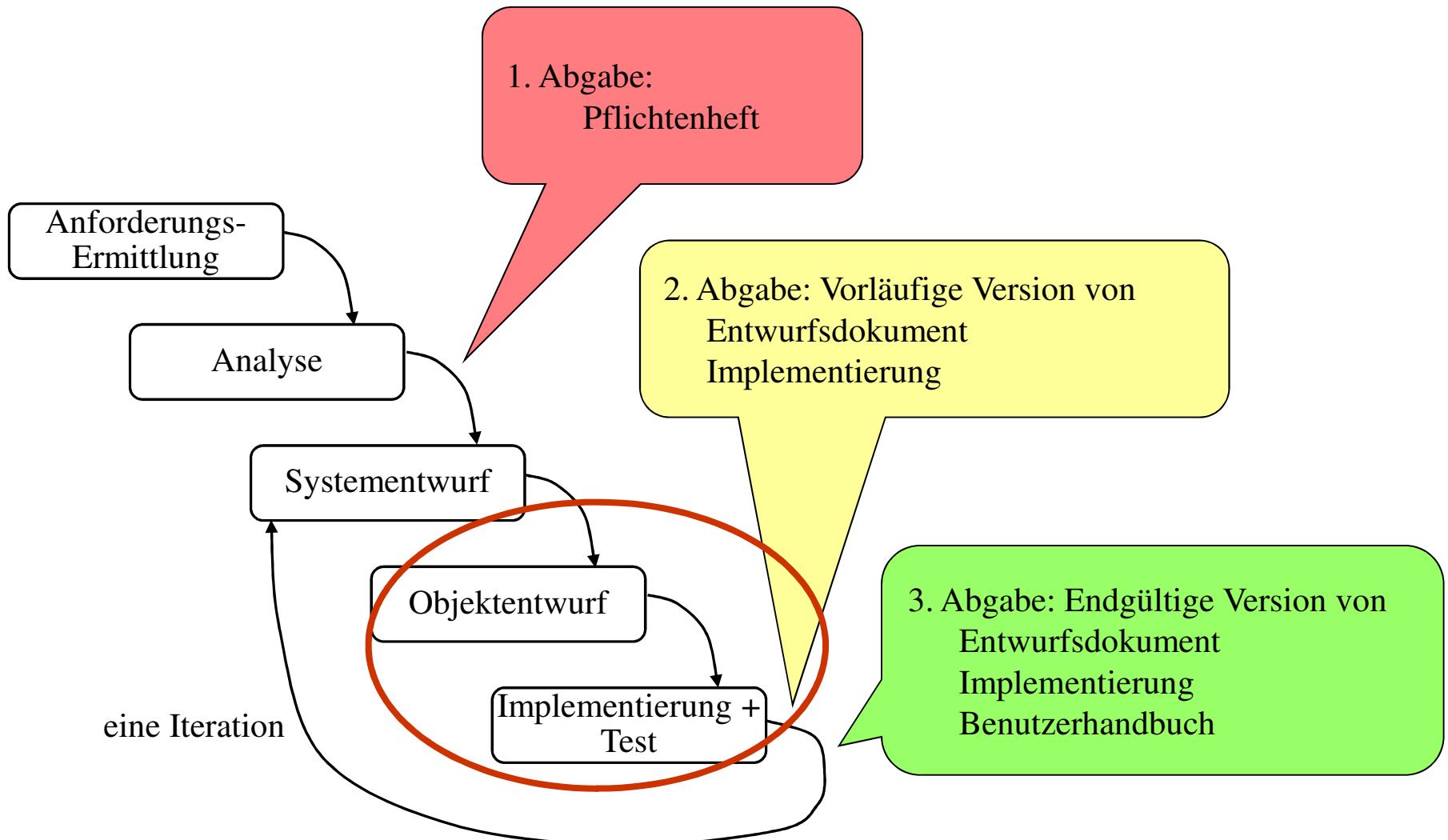
# Bemerkungen zur Prüfung der Verträge

- Prüfcode darf keine Seiteneffekte haben (d.h. keine Attribute verändern)
- Vertrag einer Oberklassenmethode muss auch in den Unterklassenmethoden überprüft werden, die diese Oberklassenmethode überschreiben (Spezifikationsvererbung !)
- Komplexen Prüfcode in Prüfmethoden kapseln:
  - übersichtlicherer Code
  - einfachere Wiederverwendung des Prüfcodes für verschiedene Methoden mit gleichen Bedingungen und für überschreibende Unterklassenmethoden
- Prüfcode kann...
  - ... komplexer sein als produktiver Code
  - ... zu ineffizient sein, um ihn ständig zu durchlaufen
  - ... selber Fehler enthalten⇒ pragmatischer Ansatz zum Prüfen von Kontrakten (siehe nächste Folie)

# Spielregeln zur Dokumentation und zum Prüfen von Kontrakten im Sopra

- Nicht-triviale Vor-/Nachbedingungen und kritische Klasseninvarianten zu allen Methoden und Klassen in javadoc-Kommentaren spezifizieren (textuell oder semi-formal, optional zusätzlich OCL)
- Nicht-triviale Vorbedingungen mit assert-Anweisungen prüfen
- Ausnahme:
  - Komplexität des Prüfcodes nicht gerechtfertigt durch erwarteten Nutzen der Prüfung
- Nachbedingungen und Klasseninvarianten nur in kritischen Fällen prüfen

# SoPra-Vorgehensmodell



# Häufige Transformationen

- 
- 
- 
- Abbildung von Objektmodellen auf Datenbankschemata

# Relationale Datenbanken

- **Relationale Datenbanken** speichern Datensätze konzeptionell in Tabellen ab.
- Bsp.: Datenbank mit nur einer Tabelle

Tabelle Benutzer

| name         | kennung | email             |
|--------------|---------|-------------------|
| Grete Schulz | gschulz | gr.schulz@wwu.de  |
| Hans Meier   | h.meier | hans.meier@wwu.de |

- Die obige Tabelle hat drei **Attribute** (name, kennung und email) und enthält zwei **Datensätze**.
- Die Spezifikation der in einer Datenbank abgelegten Tabellen, ihrer Attribute und deren Typ sowie weiterer für die Datenbank relevanter Festlegungen bezeichnet man als **Datenbankschema**.

# Kandidaten- und Primärschlüssel

- **Kandidatenschlüssel** einer Tabelle: Eine minimale Menge von Attributen, deren Werte die Datensätze in der Tabelle eindeutig identifizieren.
- **Primärschlüssel** einer Tabelle: Ein ausgewählter, im Datenbankschema festgelegter Kandidatenschlüssel.
- Primärschlüssel werden beim Einfügen, Aktualisieren oder Auswählen zur Identifikation der Datensätze benutzt.
- Erhaltung der **Konsistenz der DB**: Beim Einfügen/Aktualisieren von Datensätzen prüft die Datenbank, ob die Schlüsseleigenschaft erhalten bleibt. Ansonsten wird die Ausführung der Operation verweigert.
- Bsp.:

| Tabelle Benutzer    |         |                     |
|---------------------|---------|---------------------|
| Kandidatenschlüssel |         | Kandidatenschlüssel |
| name                | kennung | telefonnummer       |
| Grete Schulz        | gschulz | +49-251-649038      |
| Hans Meier          | h.meier | +49-251-946380      |

Primärschlüssel

# Fremdschlüssel

- Eine Menge von Attributen, die auf den Primärschlüssel einer anderen Tabelle verweist, bezeichnet man als **Fremdschlüssel**.

- Fremdschlüssel werden im Datenbankschema mit festgelegt.
  - **Erhaltung der Konsistenz:**

Beim Einfügen/Aktualisieren/Löschen von Datensätzen überprüft die Datenbank, dass zu Fremdschlüsseleinträgen stets ein entsprechender Primärschlüsseleintrag in der referenzierten Tabelle existiert; ansonsten verweigert sie die Operation.

- Bsp.:

Tabelle E-Mail-Adressen

| email                 | kennung |
|-----------------------|---------|
| grete@uni-muenster.de | gschulz |
| gs@wwu.de             | gschulz |
| hans@uni-muenster.de  | h.meier |

Fremdschlüssel

- Das Attribut „kennung“ der Tab. Mailaliases verweist auf den Primärschlüssel der Tabelle Benutzer.

# Aufgaben beim Abbilden eines Objektmodells in ein Datenbankschema

- Abbildung der elementaren UML- bzw. Java-Datentypen auf die Datentypen der Datenbank
- Objektidentität garantieren
  - Objekte mit gleichen Attributwerten sind nicht unbedingt gleich...
- Abbilden von Assoziationen
- Abbilden von Vererbung

# Aufgaben beim Abbilden eines Objektmodells in ein Datenbankschema

- Abbildung der elementaren UML- bzw. Java-Datentypen auf die Datentypen der Datenbank
  - Im Wesentlichen: Korrespondierende Datentypen benutzen
  - Aber: Manche Datenbank-Managementsysteme unterstützen nur Strings fester Länge. Dann:
    - Invariante ergänzen (und einhalten!), dass Strings nicht die im Datenbankschema vereinbarte Länge überschreiten
    - Bei Eingabe von Namen etc. prüfen, dass diese Bedingung eingehalten wird
- 
- 
- 
-

# Aufgaben beim Abbilden eines Objektmodells in ein Datenbankschema

- 
- Objektidentität garantieren
  - Objekte mit gleichen Attributwerten sind nicht unbedingt gleich...
- 
-

# Abbildung einer Klasse in eine Datenbanktabelle

**Vorgehensweise** (siehe Beispiel auf Folgefolie):

- Zusätzlich zur Repräsentation der Attribute wird eine zusätzliche Spalte hinzugefügt.
- Diese Spalte repräsentiert die **Objektidentität** und ist **Primärschlüssel** der Tabelle; geeignete Werte werden durch Programm oder Datenbank generiert.

**Übliches Vorgehen:**

Zur Repräsentation der Objektidentität werden **keine** fachlichen Attribute der Klasse verwendet, selbst dann nicht, wenn diese die Schlüsseleigenschaft haben.

Beispiel:

Attribute „kennung“ oder „Personalnummer“ nicht zur Repräsentation der Objekidentität verwenden.

**Bemerkung:**

Das Objektidentitätsattribut fügt man in Praxi dann auch der Klasse als zusätzliches „künstliches“ (nicht-fachliches) Attribut hinzu, um eine Beziehung zwischen den Objekten im Speicher und den Tabelleneinträgen in der Datenbank herzustellen.

# Beispiel: Abbildung einer Klasse in eine Datenbanktabelle

Klassendiagramm:

|               |
|---------------|
| Benutzer      |
| name          |
| kennung       |
| telefonnummer |

Zugehöriges Datenbankschema:

Tabelle Benutzer

| <b>id</b> | <b>name</b> | <b>kennung</b> | <b>telefonnummer</b> |
|-----------|-------------|----------------|----------------------|
| ...       | ...         | ...            | ...                  |



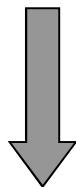
**Primärschlüssel**

# Abbildung einer Klasse in eine Datenbanktabelle (Forts.)

## Objektdiagramm:

|                                |
|--------------------------------|
| <u>:Benutzer</u>               |
| name = "Grete Schulz"          |
| kennung = "gschulz"            |
| telefonnummer = +49-251-649038 |

|                                |
|--------------------------------|
| <u>:Benutzer</u>               |
| name = "Hans Meier"            |
| kennung = "h.meier"            |
| telefonnummer = +49-251-946380 |



## Zugehörige Tabelle:

Tabelle Benutzer

| <b>id</b> | <b>name</b>  | <b>kennung</b> | <b>telefonnummer</b> |
|-----------|--------------|----------------|----------------------|
| 99        | Grete Schulz | gschulz        | +49-251-649038       |
| 42        | Hans Meier   | h.meier        | +49-251-946380       |

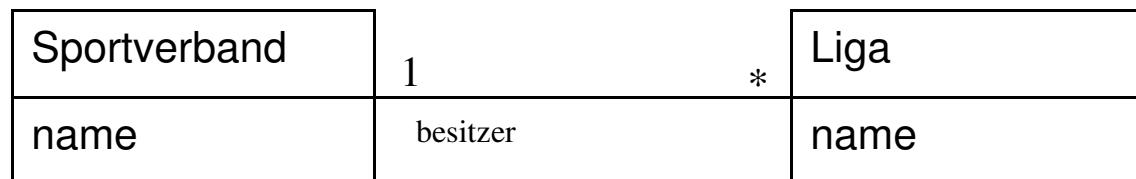
# Aufgaben beim Abbilden eines Objektmodells in ein Datenbankschema

- 
- 
- 
- Abbilden von Assoziationen
-

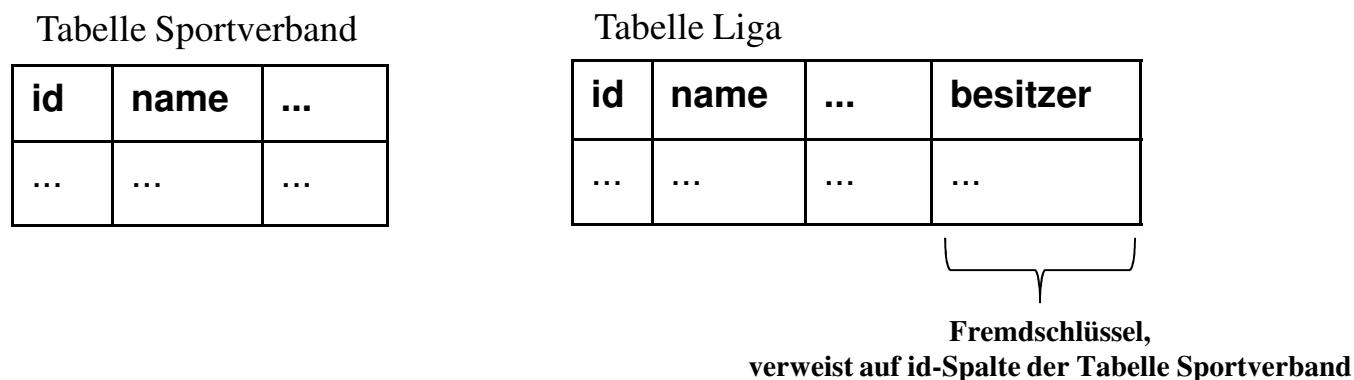
# Abilden einer Eins-zu-n-Assoziation durch einen Fremdschlüssel

- Eine Eins-zu-n-Assoziation (mit beliebiger Kardinalität n auf der anderen Seite) kann durch einen Fremdschlüssel repräsentiert werden.
- Beispiel (unten und Folgefolie):  
Spalte **besitzer** der Tabelle **Liga** enthält Verweise auf die **id**-Spalte der zugeordneten Tabelle **Sportverband** als Fremdschlüssel.

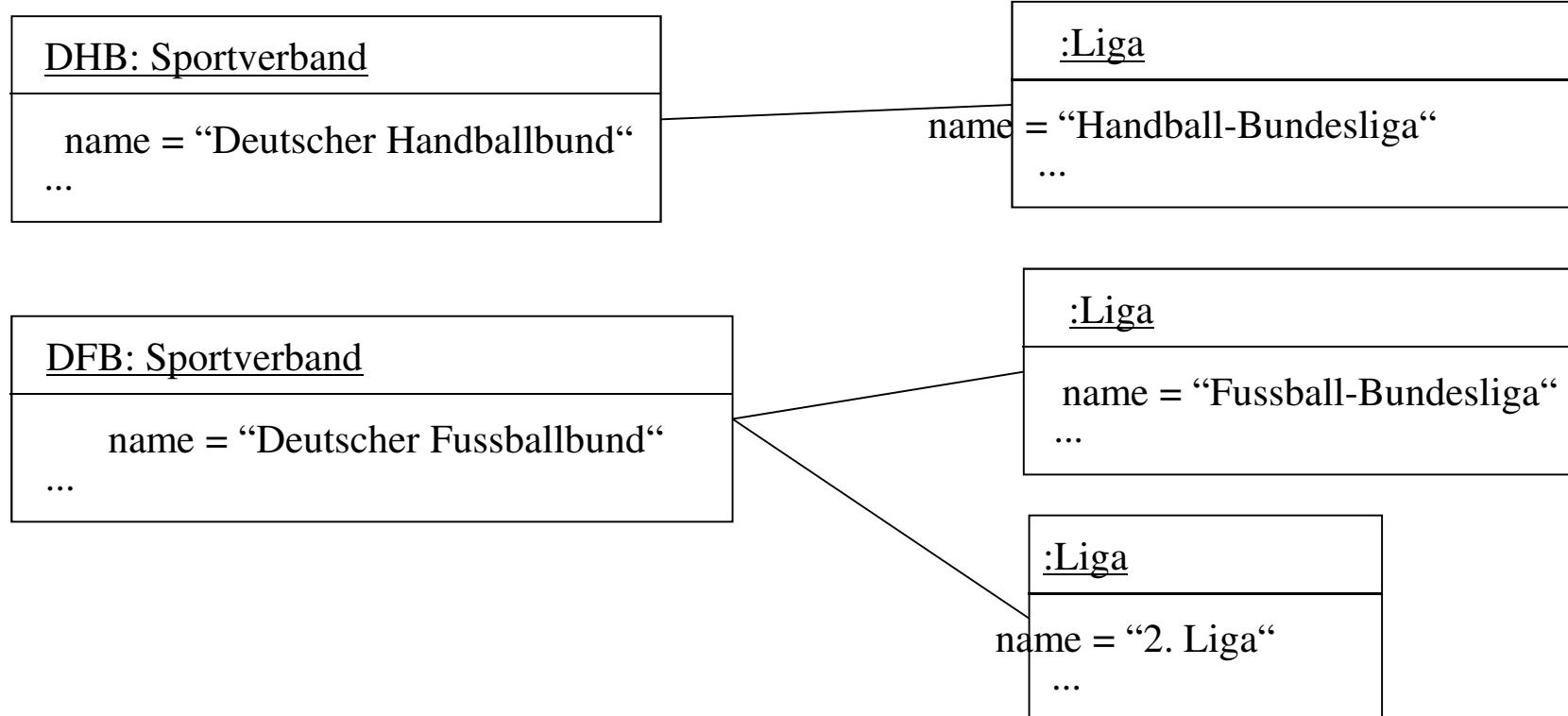
Klassen-diagramm:



Zugehöriges Datenbankschema:



## Objektdiagramm:



## Zugehörige Tabellen:

Tabelle Sportverband

| <b>id</b> | <b>name</b>            | ... |
|-----------|------------------------|-----|
| 99        | Deutscher Handballbund | ... |
| 42        | Deutscher Fussballbund | ... |

Tabelle Liga

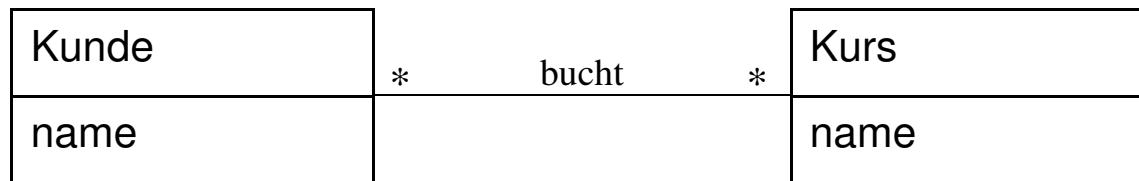
| <b>id</b> | <b>name</b>         | ... | <b>besitzer</b> |
|-----------|---------------------|-----|-----------------|
| 1         | 2. Liga             | ... | 42              |
| 2         | Handball-Bundesliga | ... | 99              |
| 3         | Fussball-Bundesliga | ... | 42              |

# Abbildung einer Assoziation auf eine separate Assoziationstabelle

- Assoziationen mit beliebigen Kardinalitäten können durch eine sog. **Assoziationstabelle** repräsentiert werden.  
Beispiel: Siehe Folgefolien
- Vorteil: Allgemeine Technik für beliebige Assoziationstypen
- Nachteil: Bei Repräsentation von 1-zu-n Assoziationen i.d.R. ineffizienter als Fremdschlüssel, da beim Zugriff der „join“ mehrerer Tabellen berechnet werden muss.
- Bemerkung:  
Eine Assoziationstabelle entspricht (fast) einer mit Hilfe von Fremdschlüsseln repräsentierten Koordinatorklasse.  
Bsp: Siehe Tafel !

# Beispiel für Verwendung einer Assoziationsstabelle

Klassendiagramm:



Zugehöriges Datenbankschema:

Tabelle Kunde

| <b>id</b> | <b>name</b> | ... |
|-----------|-------------|-----|
| ...       | ...         | ... |

Tabelle Kurs

| <b>id</b> | <b>name</b> | ... |
|-----------|-------------|-----|
| ...       | ...         | ... |

Tabelle Buchung

| <b>kunde</b> | <b>kurs</b> |
|--------------|-------------|
| ...          | ...         |

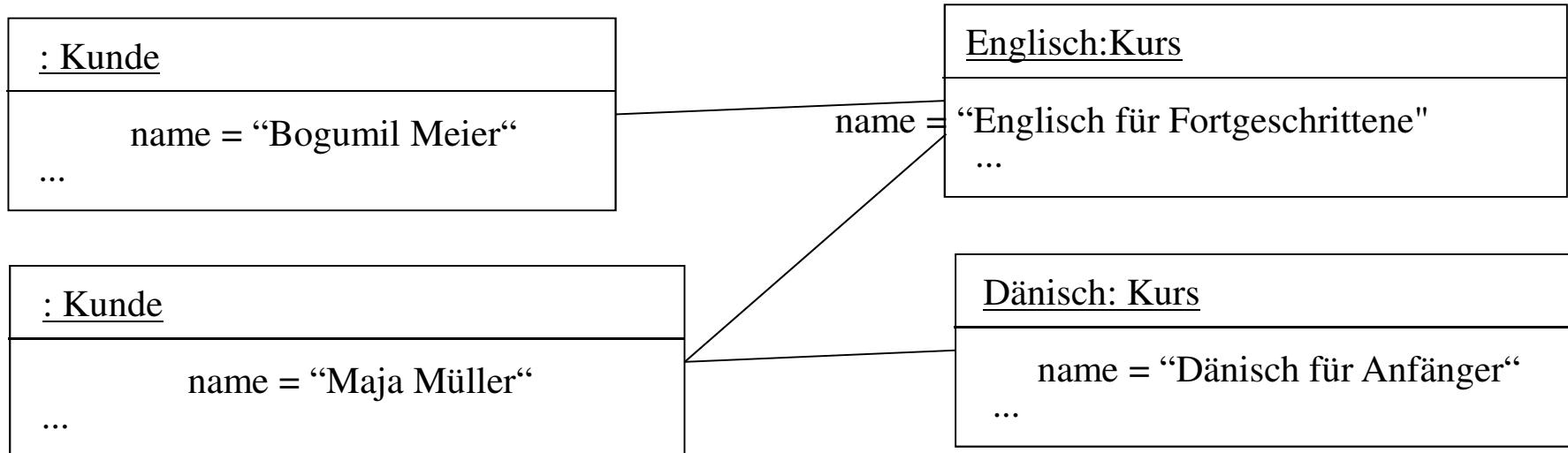
Fremdschlüssel:  
verweist auf id-Spalte der Tabelle Kunde

Fremdschlüssel:  
verweist auf id-Spalte der Tabelle Kurs

Bemerkung: Die Tabelle Buchung ist die Assoziationsstabelle!

# Beispiel (Fortsetzung)

## Objektdiagramm:



## Zugehörige Tabellen:

Tabelle Kunde

| <b>id</b> | <b>name</b>   | ... |
|-----------|---------------|-----|
| 12        | Bogumil Meier | ... |
| 14        | Maja Müller   | ... |

Tabelle Buchung

| <b>kunde</b> | <b>kurs</b> |
|--------------|-------------|
| 12           | 57          |
| 14           | 57          |
| 14           | 77          |

Tabelle Kurs

| <b>id</b> | <b>name</b>                   | ... |
|-----------|-------------------------------|-----|
| 57        | Englisch für Fortgeschrittene | ... |
| 77        | Dänisch für Anfänger          | ... |

# Aufgaben beim Abbilden eines Objektmodells in ein Datenbankschema

- 
- 
- 
- Abbilden von Vererbung
  - Diskutieren zwei Möglichkeiten:
    - Separate Tabellen
    - Duplizierte Spalten

# Realisierung einer Vererbungshierarchie durch separate Tabellen

## Technik:

- Für jede Klasse wird eine Tabelle angelegt, die genau die (persistenten) Attribute dieser Klasse speichert
- Attribute von Oberklassen werden in den Oberklassen-Tabellen gespeichert
- Beziehung der Einträge wird über Objektidentitätsspalten herbeigeführt

## Vorteil:

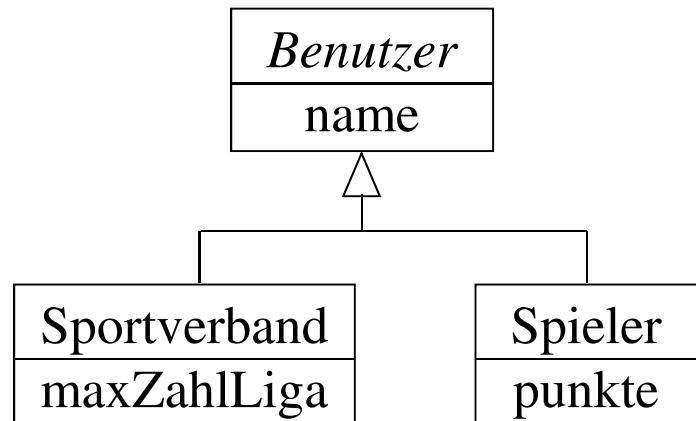
- Stabile allgemeine Lösung, auch bei Änderungen von Klassen

## Nachteil:

- Zugriff auf Subklassen kann ineffizient sein, da mehrere Tabellen beteiligt sind

# Realisierung einer Vererbungshierarchie durch separate Tabellen: Beispiel

Klassendiagramm:



Objektdiagramm:



Datenbankschema und Tabellen:

Tabelle Sportverband

| <b>id</b> | <b>maxZahlLiga</b> |
|-----------|--------------------|
| 99        | 12                 |

Tabelle Benutzer

| <b>id</b> | <b>name</b>            |
|-----------|------------------------|
| 99        | Deutscher Handballbund |
| 42        | Hans Meier             |
| 31        | Grete Schulz           |

Tabelle Spieler

| <b>id</b> | <b>punkte</b> |
|-----------|---------------|
| 42        | 56            |
| 31        | 66            |

# Realisierung einer Vererbungshierarchie durch duplizierte Spalten

## Technik:

- Alle Attribute der Oberklasse(n) einer (Unter-)Klasse werden mit in die Tabelle für diese (Unter-)Klasse aufgenommen

## Vorteil:

- Zugriff auf Subklassen effizienter als separate Tabellen, da keine Joins gebildet werden müssen
- Für abstrakte Oberklassen muss überhaupt keine Tabelle angelegt werden

## Nachteil:

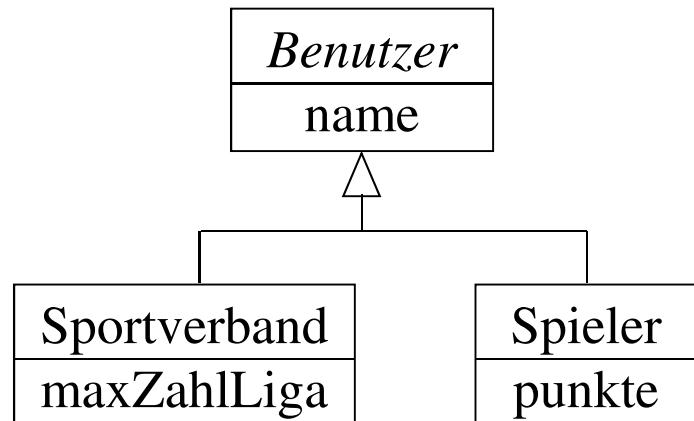
- Datenbankschema weniger stabil, denn:
- Änderungen an einer Klasse haben Auswirkungen auf Schema für alle Subklassen

## Bemerkung:

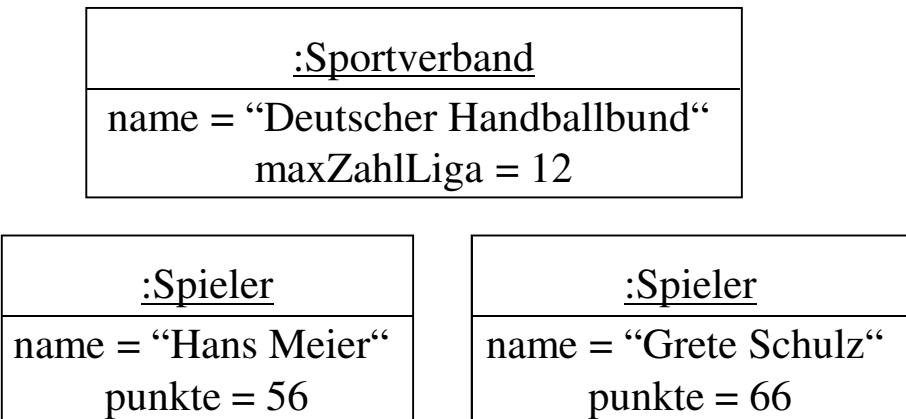
- In der Praxis mischt man die beiden Realisierungsvarianten für Vererbung

# Realisierung einer Vererbungshierarchie durch duplizierte Spalten: Beispiel

Klassendiagramm:



Objektdiagramm:



Datenbankschema und Tabellen:

Tabelle Sportverband

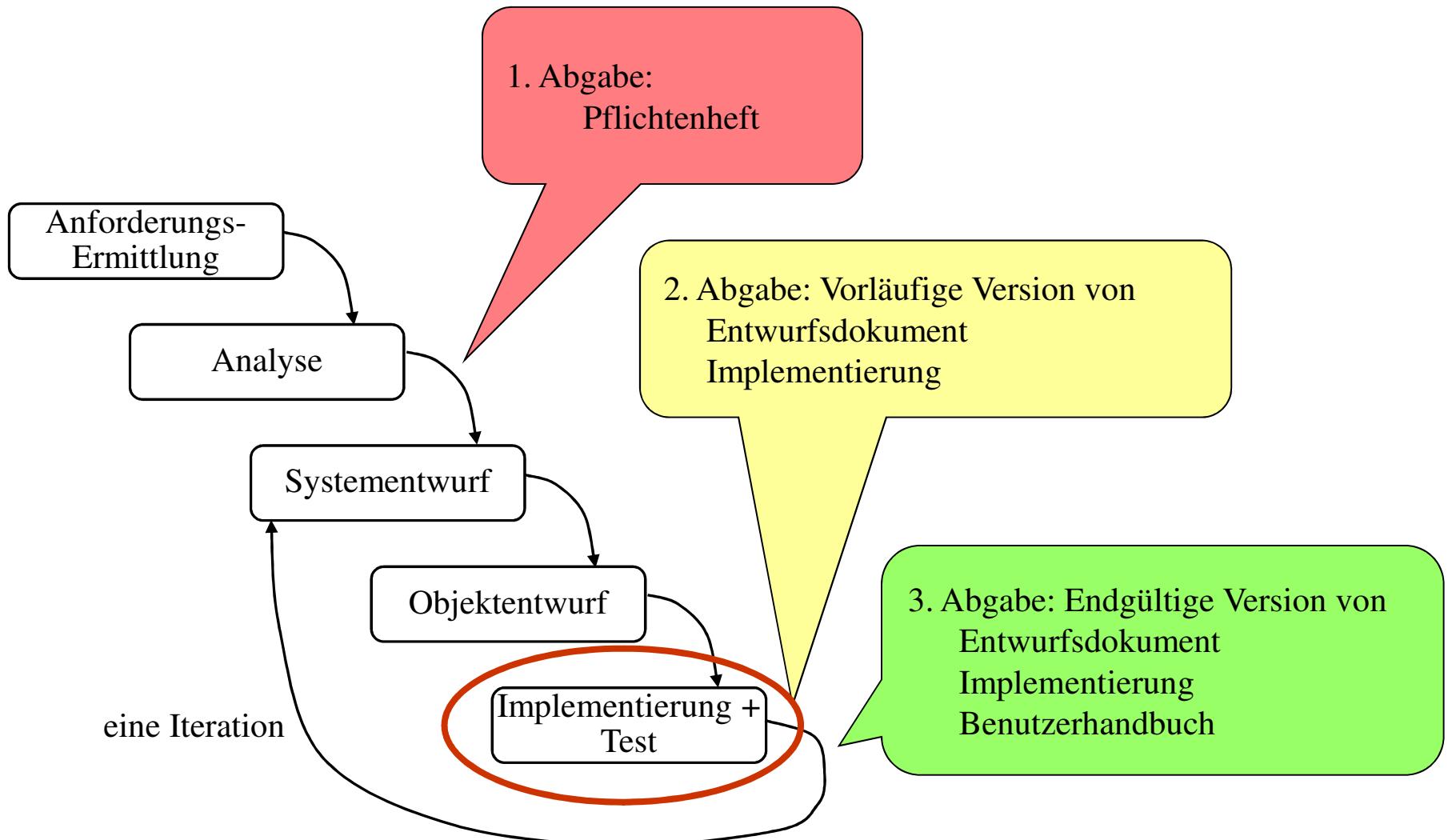
| <b>id</b> | <b>name</b>            | <b>maxZahlLiga</b> |
|-----------|------------------------|--------------------|
| 99        | Deutscher Handballbund | 12                 |

Tabelle Spieler

| <b>id</b> | <b>name</b>  | <b>punkte</b> |
|-----------|--------------|---------------|
| 42        | Hans Meier   | 56            |
| 31        | Grete Schulz | 66            |

**Beachte:** Im Beispiel keine Tabelle für Klasse Benutzer, weil sie abstrakt ist!

# SoPra-Vorgehensmodell



# Testen

## Testen:

- Programm oder Programmteil mit ausgewählten Eingaben ausführen und tatsächliches Ergebnis mit erwartetem Ergebnis vergleichen

## Ziel:

- Fehler finden, d.h.,
- Abweichungen des tatsächlichen vom erwarteten Verhalten aufdecken

## Bemerkungen:

Testen kann Fehler aufzeigen (in der Regel) aber keine Fehlerfreiheit, weil:

- Es ist unrealistisch oder unmöglich, alle Eingaben zu überprüfen
  - Nicht alle Randbedingungen sind vom Tester (leicht) zu beeinflussen, z.B.: Werte uninitialisierter Variablen, Compiler + Laufzeitsystem, Timing,...
- ⇒ Testfälle sorgfältig wählen, um dennoch Vertrauen zu gewinnen

# Erinnerung: Wichtige Heuristiken zur Wahl der Testfälle

## Äquivalenzteilung:

- Mögliche Testfälle in Äquivalenzklassen einteilen; aus jeder Klasse einen Fall testen
- z.B. auf Basis der Eingabe:
  - Methode mit Listenparameter für leere Liste, ein-elementige Liste, eine „typische“ mehr-elementige Liste testen als Repräsentant der Klasse aller mehr-elementigen Listen
  - Test des Gesamtsystems: Einen typischen Durchlauf eines Anwendungsfalls testen als Repräsentant der Äquivalenzklasse aller Instanzen dieses AFs
- z.B. auf Basis des Codes:
  - z.B. Anweisungsüberdeckung: Jede Anweisung des Programms sollte durch mindestens einen Testfall durchlaufen werden

## Randfälle testen:

- z.B. leere Liste
- z.B. leere Liste, sortierte Liste, absteigend sortierte Liste für Sortierverfahren

Guter Testfall sollte mit gewisser Wahrscheinlichkeit einen durch die übrigen Testfälle nicht aufgedeckten Fehler finden können

# Klassifikation von Tests nach Aufwand für Vorbereitung und Archivierung

nach [Ludewig/Lichter, 2007]

## Laufversuch

- Entwickler übersetzt Software; nach Abstürzen und Korrekturen keine offensichtlich falschen Resultate mehr

## Wegwerftest

- Programm mit ad-hoc gewählten Daten ausführen; dabei evtl. Fehler finden

## Systematischer Test

- Randbedingungen werden präzise erfasst
- Eingaben systematisch ausgewählt
- Ergebnisse werden dokumentiert und nach vor dem Test festgelegten Kriterien bewertet

**Bemerkung:** Systematische Tests zunächst aufwendiger aber auf Dauer effektiver:

- **objektive Aussage** über den Prüfling
- schnell und **mit geringen Kosten wiederholbar**
- **Verbesserung der Testdatenwahl möglich**, wenn später Fehler auftreten

# Entwicklung von Testtreibern mit JUnit

- JUnit: Framework zur Entwicklung von Testtreibern für Java
- Entwickelt im Kontext von *Extreme Programming* von Kent Beck und Erich Gamma
- Philosophie: **Test-getriebene Entwicklung** („*test-first development*“): siehe nächste Folie
- Integriert in Java IDEs, z.B. Netbeans und Eclipse

# Entwicklung von Testtreibern mit JUnit (Forts.)

Test-getriebene Entwicklung („*test-first development*“)

Grundidee:

Test für zu entwickelnden Code **vor** dem Code entwickeln,  
nicht danach!

Vorteile:

- Durch Testentwicklung klarere Vorstellung gewinnen, was der zu schreibende Code tun soll (Was vor Wie ;-))
- Entwicklung gut testbarer Programme fördern
- Code kann sofort nach Fertigstellung getestet werden
- Konsequenz: Programmierer profitiert direkt von der Entwicklung des Tests; dadurch höherer Anreiz, Tests zu entwickeln

# JUnit (Fortsetzung)

Idee (ab JUnit 4.0):

- Testfall ist Methode, die mit @org.junit.Test annotiert ist
- bei erfolgreichem Testlauf übergibt diese Methode true an die Methode org.junit.Assert.assertTrue(boolean b), sonst false
- Bemerkung: org.junit.Assert enthält weitere Methoden, um Erfolg oder Fehlschlagen von Tests anzuzeigen !

Ausführen aller Testfälle in den Klassen TestClass1, ...:

- org.junit.runner.JUnitCore.runClasses(TestClass1.class, ...); aufrufen
- führt alle Testfälle aus und gibt ein Testprotokoll auf der Konsole aus

Java IDEs unterstützen Ausführung von Testklassen,  
so dass runClasses(...) nicht explizit aufgerufen werden muss

## Beispiel: JUnit

Eine (sehr fehlerhafte) Klasse, die rationale Zahlen implementieren soll:

```
package rational;

public class Rational {
    private int zaehler = 0;
    private int nenner = 0;

    public Rational(int zaehler, int nenner) {
        this.zaehler = zaehler;
        this.nenner = nenner;
    }

    public void add(Rational r) {
        this.zaehler += r.zaehler;
        this.nenner += r.nenner;
    }

    public boolean equals (Rational r) {
        return (this.zaehler == r.zaehler)
            && (this.nenner == r.nenner);
    }

}
```

## Beispiel: JUnit

Testklasse mit 3 Tests für die Klasse Rational:  
test1() ist erfolgreich; test2() und test3() schlagen fehl

```
package rational;
import org.junit.Test;
import static org.junit.Assert.*;

public class RationalTest {
    @Test
    public void test1() {
        Rational third = new Rational(1,3);
        assertTrue(third.equals(new Rational(1,3)));
    }
    @Test
    public void test2() {
        Rational third = new Rational(1,3);
        assertTrue(third.equals(new Rational(2,6)));
    }
    @Test
    public void test3() {
        Rational third = new Rational(1,3);
        Rational sum = new Rational(1,6);
        sum.add(third);
        assertTrue(sum.equals(new Rational(1,2)));
    }
}
```

# JUnit: Testhintergründe (*test fixtures*)

- Oft möchte man vor und nach allen Tests, den gleichen Code durchlaufen, z.B. um eine für alle Test gemeinsame Anfangssituation (Testhintergrund) herbeizuführen
- Für derartige Fälle stehen spezielle JUnit-Annotationen zur Verfügung:
  - `@org.junit.Before`: führt markierte Methode vor jedem Testfall aus
  - `@org.junit.After`: führt markierte Methode nach jedem Testfall aus
  - `@org.junit.BeforeClass`: ... einmal vor allen Testfällen der Klasse
  - `@org.junit.AfterClass`: ... einmal nach allen Testfällen der Klasse
  - `@org.junit.Ignore` `@org.junit.Test`: ignoriert diesen Testfall

## Beispiel: JUnit

```
package rational;
import org.junit.Test;
import org.junit.Before;
import org.junit.Ignore;
import static org.junit.Assert.*;

public class RationalTest_1 {
    Rational third;
    @Before
    public void setup() {
        third = new Rational(1,3);
    }
    @Test
    public void test1() {
        assertTrue(third.equals(new Rational(1,3)));
    }
    @Ignore
    @Test
    public void test2() {
        assertTrue(third.equals(new Rational(2,6)));
    }
    @Test
    public void test3() {
        Rational sum = new Rational(1,6);
        sum.add(third);
        assertTrue(sum.equals(new Rational(1,2)));
    }
}
```

- setUp() initialisiert vor jedem Test das Attribut third;
- test2() wird nicht durchgeführt

# Abschließende Bemerkungen zum Testen im Softwarepraktikum

- Tests **vor** dem Code entwickeln, nicht danach !
- Tests auch für kleine zu entwickelnde Codeteile:  
Einzelne Methode, einzelne Klasse !
- Nie lange programmieren, ohne einen neuen Test zu schreiben !
- Sicherstellen, dass der Test auf die gewünschte Weise fehlschlägt,  
bevor die Funktionalität implementiert wird („Testen des Tests“) !

# Integrations-Strategien

- Nach Bau und Test aller Subsysteme: Integration zum Gesamtsystem
- Verschiedene Strategien für Integration und Test des Zusammenspiels der Subsysteme:
  - Urknall-Strategie
  - Top-Down-Strategie
  - Bottom-Up-Strategie
  - Integration auf Basis der Anwendungsfälle
- In der Praxis oft Kombination aus top-down-, bottom-up-, und AF-Strategie !

# Integrations-Strategien (Fortsetzung)

## Urknall-Strategie (*big-bang*):

- Alle (oder viele) Komponenten werden in einem Schritt integriert

Vorteil: keine (zusätzlichen) Teststümpfe und Testtreiber nötig

Nachteil:

- Fehler schwer lokalisierbar: Welche Komponente ist verantwortlich?
  - Fehler an Schnittstellen nicht von Fehlern in Komponenten zu unterscheiden
- ⇒ Nachteile so gravierend, dass (in der Regel) nicht praktikabel;  
daher besser: Inkrementelle Integration (top-down, bottom-up, AF-gesteuert, ...)

# Integrations-Strategien (Fortsetzung)

## Top-down-Strategie:

- Zu den Komponenten der obersten Schicht werden nach und nach Komponenten der darunterliegenden Schichten hinzugenommen

### Vorteile:

- früh vorzeigbarer Prototyp (→ Kunden, Manager)
- Entwurfsfehler werden früh gefunden

### Nachteil:

- erfordert viele Teststümpfe
- dadurch hoher Aufwand

# Integrations-Strategien (Fortsetzung)

## Bottom-up-Strategie:

- Zu den Komponenten der unteren Schicht werden schrittweise Komponenten höherer Schichten hinzugenommen

### Vorteil:

- **keine** (bzw. wenige) **Teststümpfe** benötigt

### Nachteil:

- **Behebung von Fehlern** in oberen Komponenten kann
  - umfangreiche Änderungen in unteren Komponenten erzwingen
  - dadurch **vorherige Integrations-Tests wertlos machen**
- **bis zum Schluss kein lauffähiger Prototyp**
- **Entwurfsfehler werden spät gefunden**

# Integrations-Strategien (Fortsetzung)

## Integration auf Basis der Anwendungsfälle:

- Zunächst nur Komponenten integrieren, die für einzelnen Anwendungsfall benötigt werden.  
Dann nach und nach Komponenten für weitere Anwendungsfälle hinzufügen.

### Bemerkung

- In der Praxis oft Kombination aus bottom-up-, top-down-, und AF-Strategie !

# Vorgehen bei der Implementierung im Sopra

## Pair Programming: Eine Technik aus dem Extreme Programming

- Jeweils zwei Personen entwickeln Code *gemeinsam*, d.h. sie schreiben, kommentieren und testen den Code gemeinsam an *einem* Computer
- Vorteil: „Vier Augen sehen mehr als zwei“ – Fehler vermeiden oder zumindest früh finden
- Die beiden Personen nehmen dabei wechselnde Rollen ein !

## Systematisches Testen

- Methoden, Klassen und Subsysteme einzeln testen !
- Dafür JUnit verwenden !
- Test und Programmierung gehen Hand in Hand !
- Mit Test nicht bis zur Systemintegration warten !

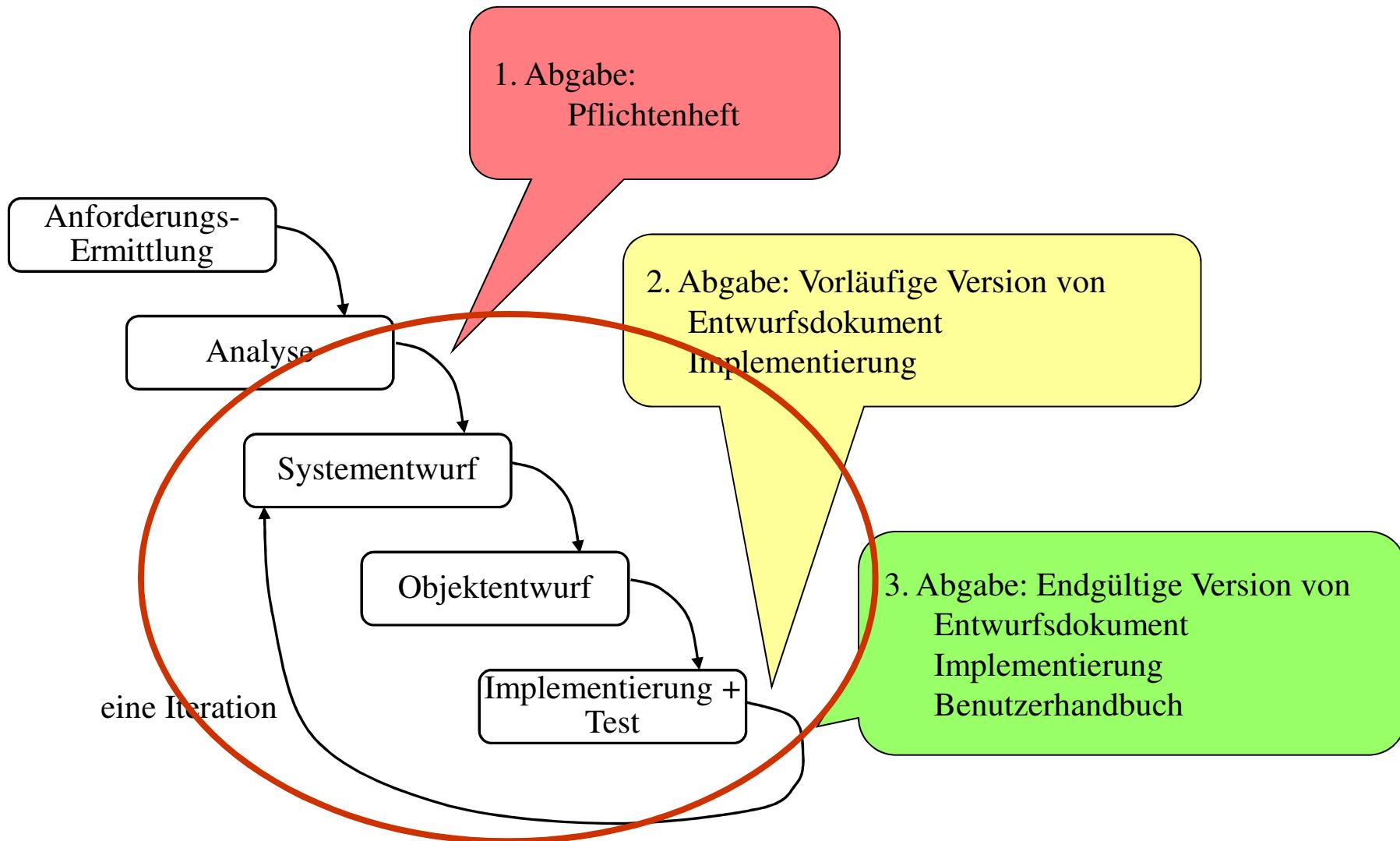
## Arbeitsteilung:

- Spezifizierte Klassen können unabhängig voneinander implementiert werden (jeweils durch Zweierteams, s.o. ...)
- **Jedes Gruppenmitglied programmiert einen Teil des Systems !!**

# Vorgaben für den Source-Code

- Dokumentation mit Javadoc  
<http://java.sun.com/j2se/javadoc>
- Autorenschaft jeder Klasse dokumentieren
  - mit @author-Javadoc-Kommentaren
- Informelle Verhaltens-Beschreibungen zu *jeder* Methode und *jeder* Klasse angeben (außer trivialen set- und get-Methoden)
- Spielregeln zur Dokumentation und zum Prüfen von Kontrakten beachten (s.o.)
- Erläuterung nicht-trivialer Algorithmen und Gültigkeit kritischer Vorbedingungen
- Exceptions nicht für den regulären Kontrollfluss verwenden:
  - Unchecked Exceptions (Errors oder RuntimeExceptions) nur zur Behandlung von Programmfehlern (verletzte Vorbedingungen, verletzte Klasseninvarianten, Division durch Null etc.),
  - Checked Exceptions nur zur Anzeige von Ausnahmefällen, die von der Anwendung behandelt werden müssen (File not found, ...)
- Ggf. Dokumentation verwendeter Entwurfsmuster

# SoPra-Vorgehensmodell



# Inhalt der Iterationen

## Erste Iteration:

- Entwicklung eines ersten Prototyps mit Basisfunktionalität

## Zweite Iteration:

- Entwicklung der endgültigen Version mit erweiterter Funktionalität