

Kapitel 7. „Objektentwurf“

Teilweise nach „Brügge & Dutoit“, Kap. 8

Stand: 16.12.2010

Überblick

- Objektentwurf

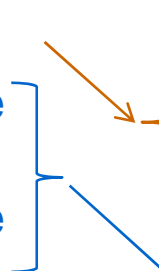
- ◆ Definition
- ◆ Aktivitäten

- Wichtige Aktivitäten

- ◆ Spezifikation von Schnittstellen
- ◆ Fortgeschrittene UML-Konzepte und ihre Umsetzung in bisher bekannte „Kern-UML“-Konzepte
- ◆ Umsetzung von „Kern-UML“ in implementierungsnahe, einfachere Designs

- Wichtige Hilfsmittel

- ◆ CRC-Karten
- ◆ „Design by Contract“
- ◆ Verhaltensprotokolle (Behaviour Protocols)
- ◆ „Split-Object“-Entwurfsmuster



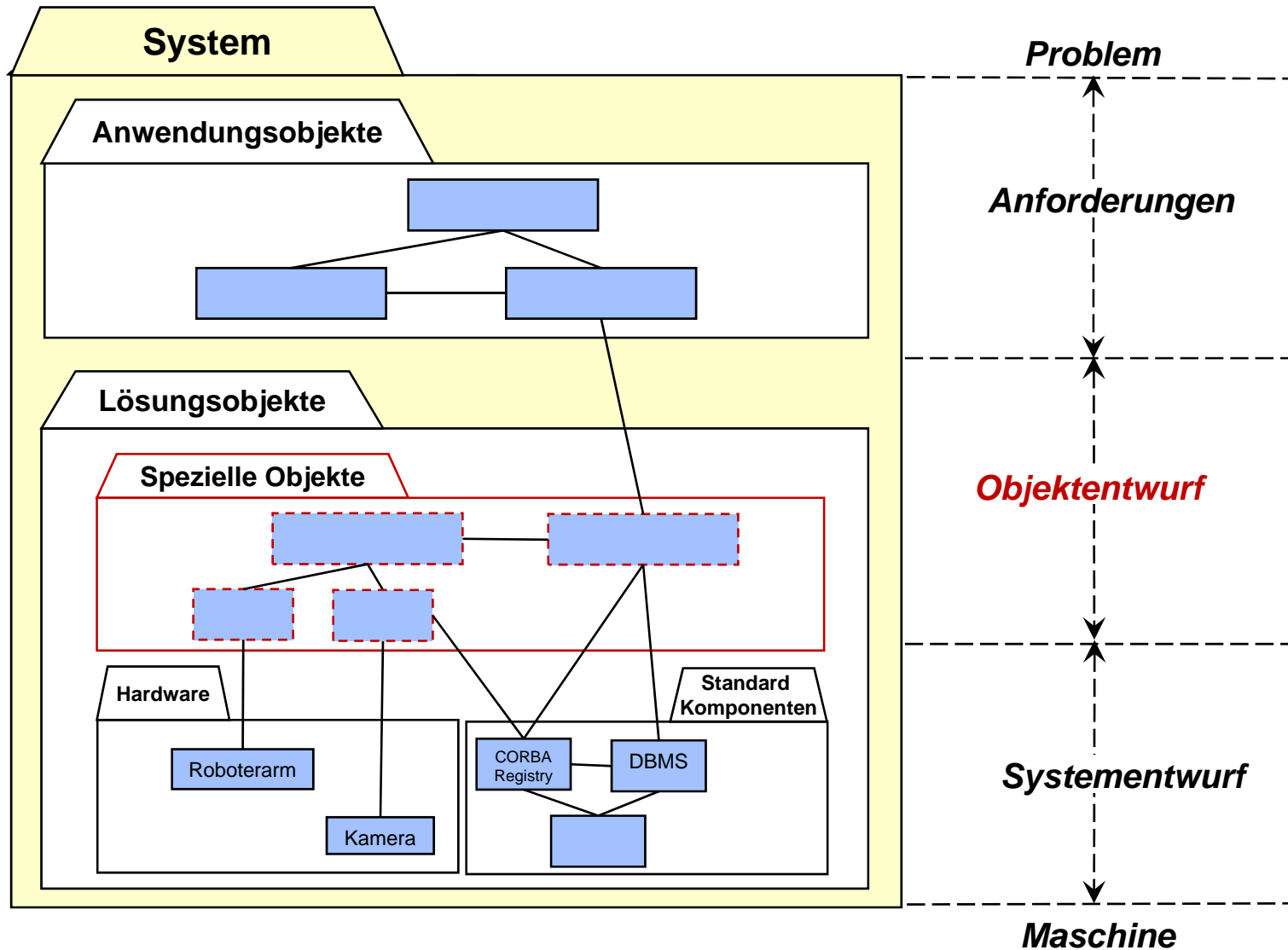
Der Objektentwurf als Produkt

- ... ist das komplette Modell des zu realisierenden Systems
 - ◆ Klassendiagramme und dynamische Diagramme
 - ◆ Verteilungs- und Komponentendiagramme
- ... dient als Basis für die Implementierung
 - ◆ Automatische Code-Generierung aus Klassendiagrammen
 - ⇒ Mittels CASE-Tools (CASE = Computer Aided Software Engineering = Computer-unterstützte Software Entwicklung)
 - ⇒ Beispiel: Together, RationalRose, EnterpriseArchitect, ArgoUML, ...
 - ◆ Manuelle Implementierung des „Restes“

Der Objektentwurf als Prozess

- ... bezeichnet die Ergänzung und Veränderung der Ergebnisse der Anforderungsanalyse und des Systementwurfs auf Basis von Implementierungsentscheidungen, die die gesetzten Anforderungen erfüllen.
- ... bezeichnet die Identifikation verschiedener Möglichkeiten, das Analyse- und Systemmodell zu implementieren sowie die *begründete* Auswahl zwischen den Alternativen (siehe auch Kapitel „Rationale Management“).
- Die Auswahl orientiert sich immer an den gesetzten Anforderungen und deren Prioritäten.
- Aber: Betreiben wir nicht schon in der Anforderungserfassung Objektentwurf?

Objektentwurf: Die Lücke schließen



Aufgaben des Objektentwurfs

- Vollständige Definition aller Klassen und Assoziationen
- Auswahl von Algorithmen und Datenstrukturen
- Bestimmung neuer Klassen, die unabhängig von der Anwendungsdomäne sind (z.B. *“Cache”*, *Abstraktionen*, *Entwurfsmuster*, ...)
- Überdenken der Typ- und Vererbungshierarchien
- Entscheidungen über den Kontrollfluss
- Optimierung

Aktivitäten während des Objektentwurfs

1. Schnittstellenspezifikation

- ◆ Spezifikation der Schnittstellen aller Typen (Klassen und Interfaces)

2. Auswahl vorhandener Komponenten

- ◆ Bestimmung von Standardkomponenten und zusätzlicher Objekte der Lösungsdomäne

3. Restrukturierung des Objektmodells

- ◆ Umformen des Modell des Objektentwurfs zur Verbesserung von Verständlichkeit und Erweiterbarkeit sowie Realisierung von UML-Konzepten, die keine Entsprechung in Ihrer Programmiersprache haben

4. Optimierung des Objektmodells

- ◆ Umformen des Objektmodells unter Berücksichtigung von Performanzkriterien, wie Reaktionszeit (z.B. der Benutzerschnittstelle) oder Speicherbedarf.

Fortsetzung aus dem
Systementwurf

Schnittstellenspezifikation

<u>Schnittstellenidentifikation:</u>	CRC-Karten
<u>Schnittstellenfestlegung:</u>	Signatur-Spezifikation
<u>Schnittstellenverfeinerung:</u>	Verhaltens-Spezifikation durch Design by Contract
<u>Schnittstellenverfeinerung:</u>	Interaktions-Spezifikation durch Behaviour Protocols
<u>Schnittstellenverfeinerung:</u>	Spezifikation der „ausgehenden“ Schnittstelle

Schnittstellen-Identifikation: CRC-Cards

OO Modellierung: Class-Responsibility-Collaboration (CRC) Karten

- Class
 - ◆ Welche Klasse betrachten wir?
- Responsibility
 - ◆ Beschreibt die Aufgaben der Klasse
- Collaboration
 - ◆ Welche anderen Klassen werden für die Aufgabe gebraucht?
- Nutzen
 - ◆ regt Diskussionen an
 - ◆ lenkt den Blick auf das Wesentliche
 - ◆ Hilft auf Schnittstelle statt Daten zu fokussieren
 - ◆ beugt Konzentration von zu vielen Verantwortlichkeiten an einer Stelle vor
- “Schreib nie mehr auf, als auf eine Karte paßt!”
 - ◆ eher die Klasse in zwei Klassen / Karten aufteilen!
 - ◆ 1 Karte = höchstens DIN A5 groß (halbes DIN A4 Blatt)

Bestellung	
Prüfe ob Artikel auf Lager	Lager
Bestimme Preis	Artikel
Prüfe Zahlungseingang	Kunde, Kasse
Ausliefern	Logistik

Class (Klassen-Name)

Responsibility
(Aufgaben)

Collaboration
(Zusammenarbeit)

Einsatz von CRC-Cards

- Wann
 - ◆ in Analyse und früher Design-Phase
- Wozu
 - ◆ Identifikation von Klassen, Operationen und „Kollaborations“-Beziehung
 - ◆ Einzig relevante Beziehung ist „Kollaboration“ mit anderen Klassen
 - ◆ Instanzvariablen werden weitgehend ignoriert
 - ◆ Fokus liegt auf Operationen und evtl. den Parametern und Ergebnissen, die sie im Rahmen einer Kollaboration brauchen

CRC-Cards sind sehr hilfreich für Anfänger in der OO Modellierung, da **verhaltenszentriertes Denken gefördert** wird!

Fokus auf Schnittstellen
statt auf Instanzvariablen, Aggregationen, Kardinalitäten, ...

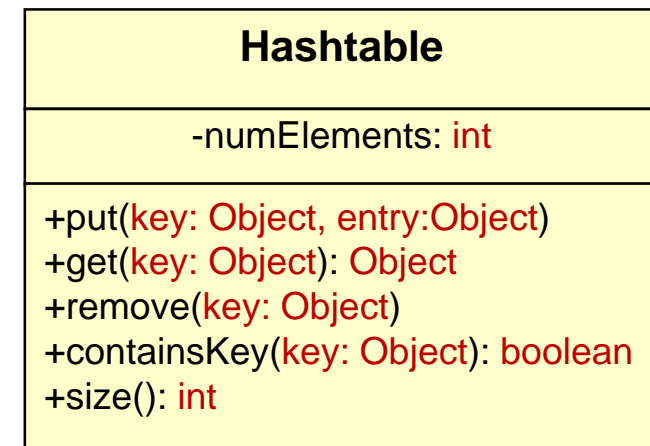
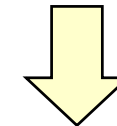
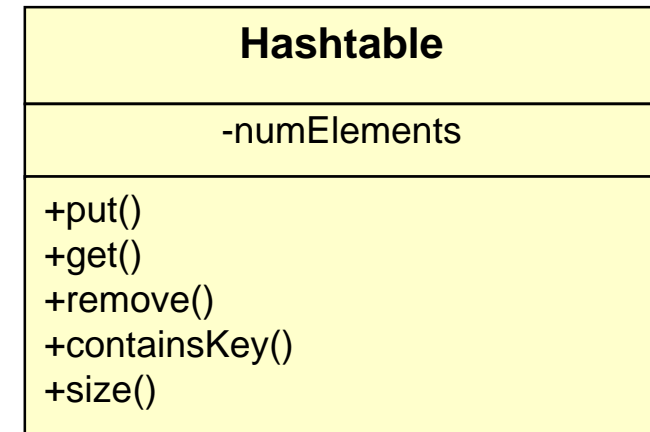
Was kommt nach CRC-Cards?

- Ausarbeitung der Beziehungen, Kardinalitäten, ...
 - ◆ → Herkömmliche Datenmodellierung (IS-Vorlesung)
- Verfeinerung des Verhaltens
 - ◆ → [Design by Contract](#)
- (Re)Strukturierung des Objektmodells
 - ◆ → [Objekt-Orientierte Modellierungs-Prinzipien](#)

Schnittstellen-Spezifikation: Signaturen = Schnittstellen-Syntax

Spezifikation der Schnittstellen

- In **Anforderungsanalyse**: Identifikation von
 - ◆ Attributen - ohne ihren Typ anzugeben
 - ◆ Operationen - ohne ihre Parameter(typen) anzugeben
- Im **Entwurf**: Hinzufügen von
 - ◆ Typsignaturen
 - ◆ ... weiteres später ...
- Im **Systementwurf** werden Typsignaturen für Dienste festgelegt.
- Im **Objektentwurf** werden Typsignaturen für alle Typen des Designs festgelegt.
 - ◆ Typen = Klassen und Interfaces



Warum reichen Typsignaturen nicht aus?

```
put(key: Object, entry:Object)
```

- Sie sagen nur etwas darüber, wie man eine Operation aufruft.
- Sie sagen nichts darüber, was die aufgerufene Operation macht.
 - ◆ keine Verhaltensspezifikation → *Verhaltenszusicherungen (Contracts)*
- Sie sagen nichts darüber, in welcher Reihenfolge verschiedene Operationen des gleichen Objektes aufgerufen werden müssen.
 - ◆ keine Interaktionsspezifikation → *Behaviour Protocols*
- Sie sagen nichts darüber, was der spezifizierte Typ selber von seiner Umgebung braucht, um die angebotenen Operationen realisieren zu können.
 - ◆ keine explizite Spezifikation von Abhängigkeiten → *Benutzte Schnittstellen (Required Interfaces)*
- Sie unterscheiden nicht verschiedene Clients
 - ◆ *Sichtbarkeitsinformationen*

Schnittstellen-Spezifikation: Verhaltensspezifikation durch „Design by Contract“

Design by Contract (DBC)

- Behauptung (Assertion)
 - ◆ Logische Aussage, die wahr sein muss
 - ◆ Macht die Annahmen explizit, unter denen ein Design funktioniert
 - ◆ Lässt sich automatisch überprüfen
- Vertrag (Contract)
 - ◆ Menge aller Assertions, die festlegen, wie zwei Partner interagieren
 - ◆ Auftraggeber = aufrufende Operation / Klasse
 - ◆ Auftragnehmer = aufgerufene Operation / benutzte Klasse

Design by Contract: Vorbedingungen („Preconditions“)

- Definition
 - ◆ Eine Precondition ist eine Voraussetzungen dafür, dass eine Operation korrekt ausgeführt werden kann
- Technische Realisierung
 - ◆ Assertion die wahr sein muss, bevor eine Operation ausgeführt wird.
- Beispiel: Operation “Ziehe die Wurzel einer Zahl”
 - ◆ Signatur: `squareRoot(input int)`
 - ◆ Pre-condition: `input >= 0`
- Verantwortlichkeiten
 - ◆ Der aufgerufene Code formuliert die Vorbedingung
 - ◆ Der aufrufende Code muss die Einhaltung der Vorbedingung sicherstellen

Design by Contract: Nachbedingung (Postcondition)

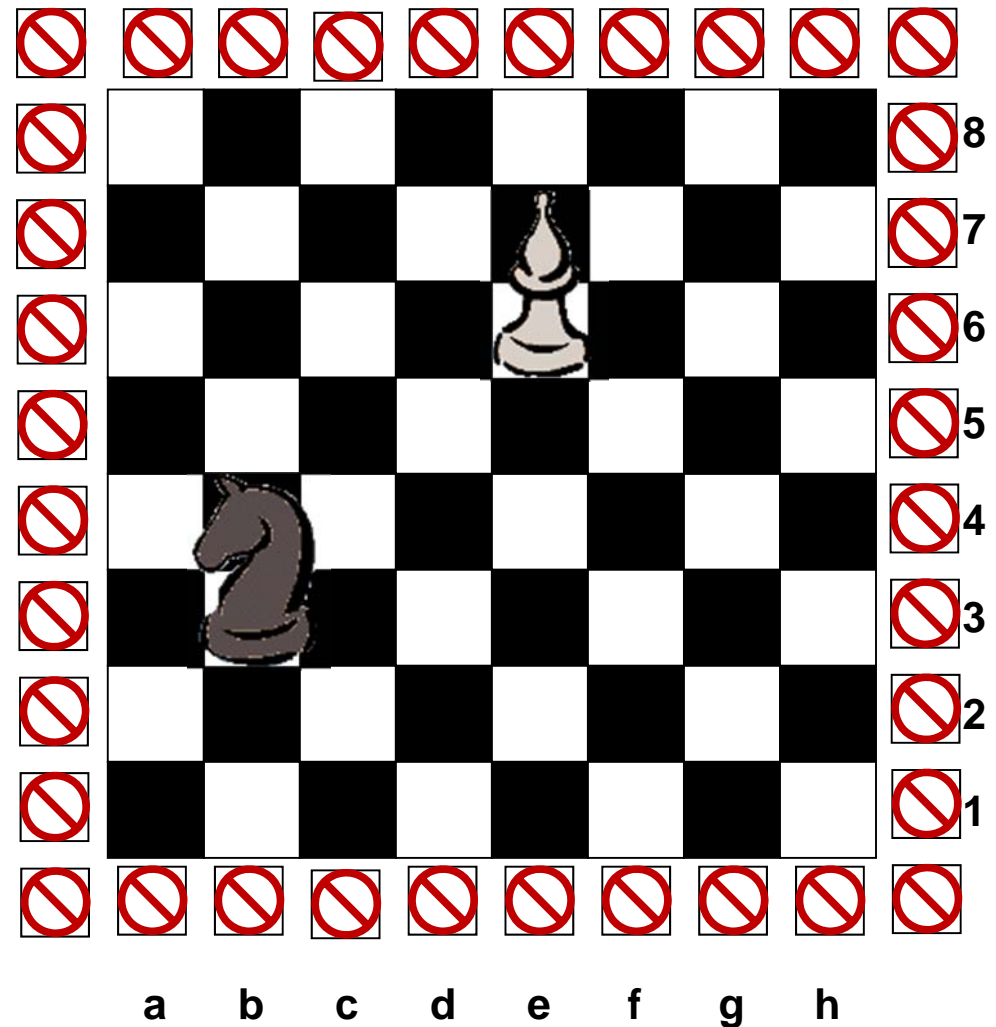
- Definition
 - ◆ Postcondition beschreibt **deklarativ** das Ergebnis eines korrekten Aufrufs
 - ◆ Sagt aus, **was** getan wird, nicht **wie** es getan wird
- Technische Realisierung
 - ◆ Assertion die wahr sein muss, *nachdem* eine Operation ausgeführt wird.
- Beispiel: Operation “Ziehe die Wurzel einer Zahl”
 - ◆ Signatur: `squareRoot(input i)`
 - ◆ Post-condition: `input = result * result`
- Verantwortlichkeiten
 - ◆ Der aufgerufene Code formuliert die Nachbedingung
 - ◆ Der aufgerufene Code garantiert die Einhaltung der Nachbedingung ...
aber nur wenn die Vorbedingung wahr ist!

Design by Contract: Klasseninvariante (Class Invariant)

- Definition
 - ◆ Invariante beschreibt deklarativ legale Zustände von Instanzen einer Klasse
- Technische Realisierung
 - ◆ Assertion die für alle Instanzen einer Klasse immer wahr ist.
- Beispiel: Klasse eines Benutzerkontos
 - ◆ Invariante: Der Kontostand ist immer die Summe aller Buchungen
 - ◆ `kontostand == summe(buchungen.betrag ())`
- Verwendung
 - ◆ Invarianten werden verwendet, um Konsistenzbedingungen zwischen Attributen zu formulieren.
- Verantwortlichkeiten
 - ◆ Diese Bedingungen einzuhalten liegt in der gemeinsamen Verantwortlichkeit aller Operationen einer Klasse.

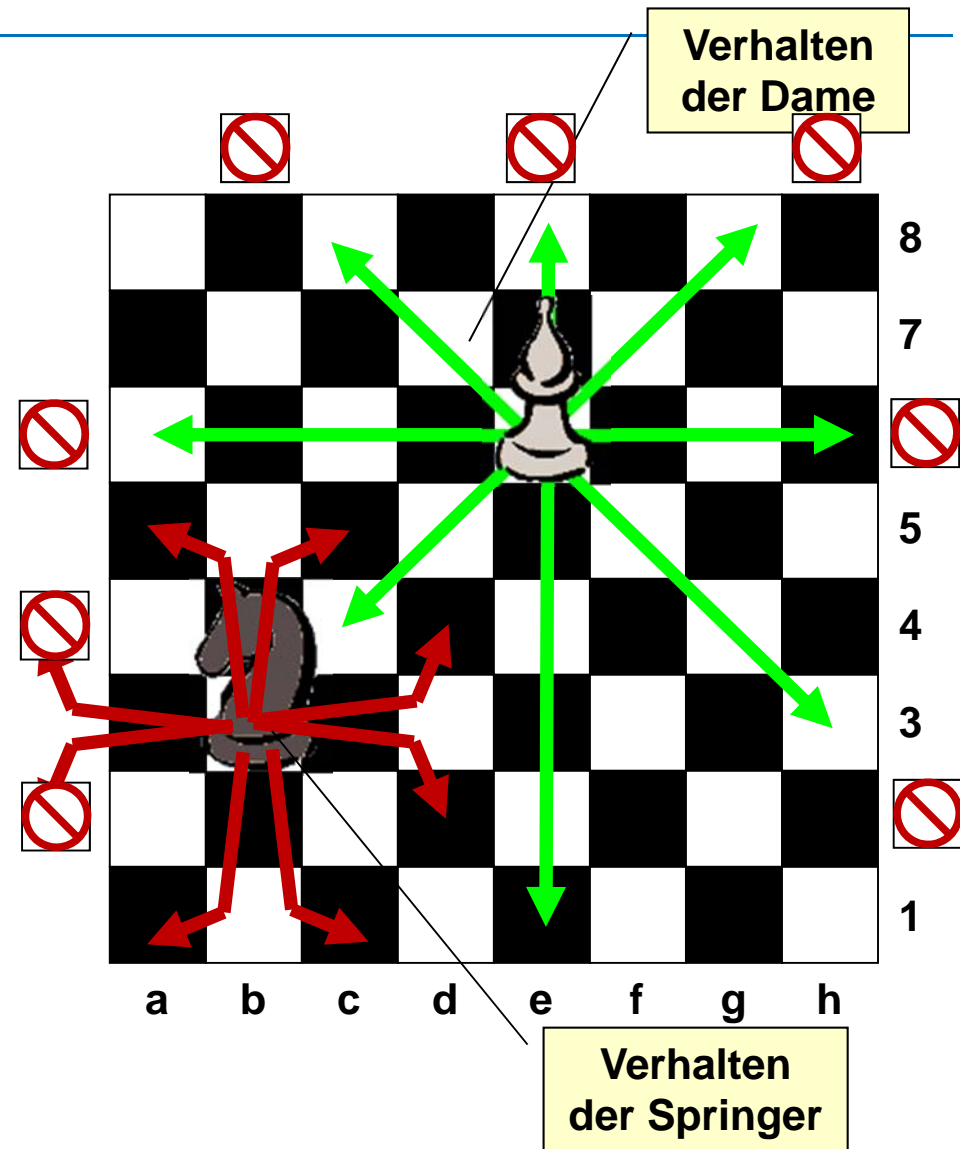
DBC spezifiziert legale Zustände

- **Zustand** zu einem Zeitpunkt = Die Werte aller Instanzvariablen
 - ◆ Mindestens der eigenen Var.
 - ◆ Konzeptionell ist auch der Zustand aller aggregierten Teil-Objekte eigener Zustand
- **legale Zustände** werden durch **Klasseninvarianten** definiert
 - ◆ **Dame** und **Springer** haben gleichen Zustandsraum (jedes Schachbrett-Feld)
 - ◆ Figuren dürfen Spielfeld nicht verlassen
 - ◆ Legale **Schachbrett-Zustände**: pro Feld max. eine Figur



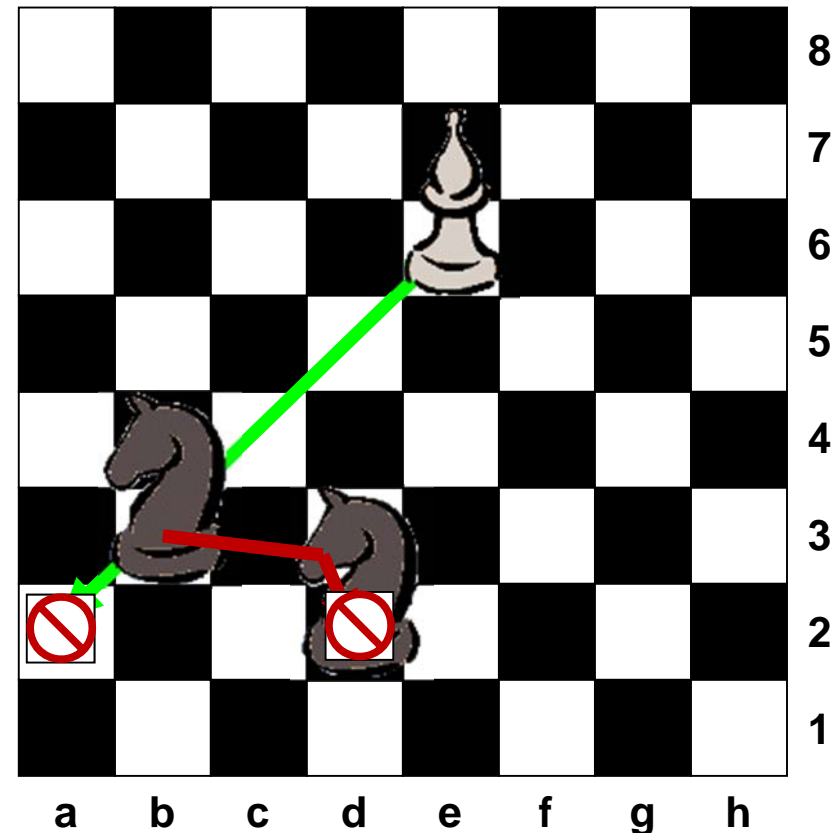
DBC spezifiziert legales Verhalten

- **Verhalten** = Zustandsübergänge und daran gekoppelte Aktionen
 - ◆ **Dame** bewegt sich horizontal und diagonal beliebig weit
 - ◆ **Springer** bewegt sich L-förmig
- **Legales Verhalten**
 - ◆ Mindestens: Zustandsübergänge die nicht in illegale Zustände führen
 - ◆ Meist gibt es zusätzliche applikationsspezifische Bedingungen („Constraints“) → s. nächste Folie



DBC spezifiziert legales Verhalten

- DDBC definiert legale Zustandsübergänge durch Vor- und Nachbedingungen
- Beispiel: Eigene Figuren schlagen ist verboten
 - ◆ Vorbedingung der Operation “zieheNach(Feld)”:
Feld nicht von eigener Figur belegt
- Beispiel: Keine Figur ausser dem Springer kann über andere hinüberspringen
 - ◆ Weitere Vorbedingung der Operation “zieheNach(Feld)”



Formulierung von Kontrakten in UML: → OCL (Object Constraint Language)

- OCL erlaubt die formale Spezifikation von Bedingungen (Constraints) für Werte von Modellelementen oder Gruppen von Modellelementen
 - ◆ Noch keine Bedingungen an den Kontrollfluss möglich.
- Ein Constraint wird in Form eines OCL Ausdrucks angegeben, der entweder den Wert wahr oder falsch hat.
 - ◆ „Constraint“ in OCL = „Assertion“ in DBC

● Beispiel: OCL Constraints für Hashtabellen

◆ Invariante:

Gültigkeitsbereich

⇒ `context Hashtable inv: numElements >= 0`

⇒ Bedeutet: „Für den Typ ‚Hashtable‘ gilt die Invariante ‚numElements >= 0‘.“

Dargestellte Art von Assertion

OCL Ausdruck:
Namen beziehen sich auf Elemente des Modells

◆ Vorbedingung:

⇒ `context Hashtable::put(key, entry) pre: !containsKey(key)`

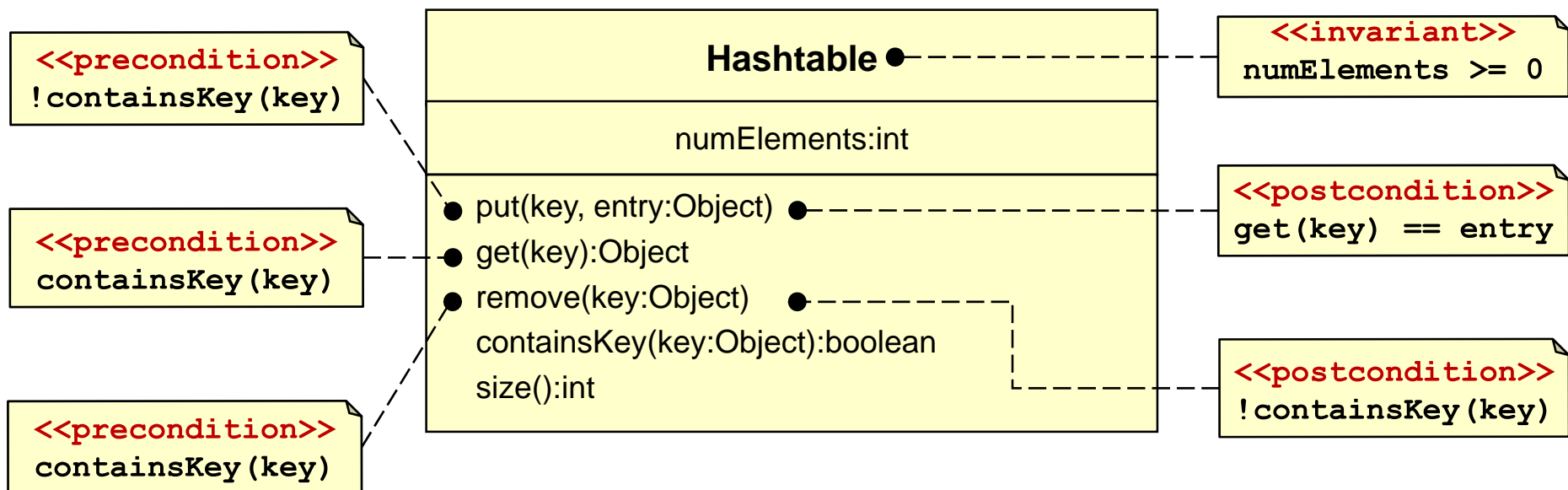
◆ Nachbedingung:

⇒ `context Hashtable::put(key, entry) post: containsKey(key) and get(key) = entry`

Notation für "Methode 'put' aus Typ 'Hashtable' "

Formulierung von Kontrakten in UML

- Jede Assertion kann auch als Notiz dargestellt und an des jeweilige UML Element "angehängt" werden.
 - ◆ Die Art der Assertion wird als Stereotyp angegeben



Besondere Schlüsselworte in Nachbedingungen („postconditions“)

- Wenn nichts explizit angegeben ist bezieht man sich in Nachbedingungen auf den Nachzustand und in Vorbedingungen auf den Vorzustand.
- In Nachbedingungen muss aber manchmal explizit der **Zustand vor und nach** der Ausführung der Operation in Beziehung gesetzt werden.
- Syntax
 - ◆ **expr@pre** = Der Wert des Ausdrucks **expr** **vor** Ausführung der **Operation** auf die sich die Nachbedingung bezieht.
 - ◆ **expr@post** = Der Wert des Ausdrucks **expr** **nach** Ausführung der **Operation** auf die sich die Nachbedingung bezieht.
- Beispiele
 - ◆ **context** **Person::birthdayHappens()** **post:** age = **age@pre** + 1
 - ◆ **age@pre** bezieht sich hier auf den Wert des Feldes **age** vor Beginn der Ausführung der Operation **birthdayHappens()** → „Alter Wert“
 - ◆ **a.b@pre.c** – Alter Wert des Feldes b von a. Darin der neue Wert des Feldes c.
 - ◆ **a.b@pre.c@pre** – Alter Wert des Feldes b von a. Darin der alte Wert des Feldes c.

Weitere OCL-Schlüsselworte

- result
 - ◆ Bezug auf den Ergebniswert einer Operation (in Nachbedingungen).
- self
 - ◆ Bezug auf das ausführende Objekt (wie „this“ in Java).
- Literatur
 - ◆ OCL 2.0 Specification, Version 2.0, Date: 06/06/2005,
<http://www.omg.org/docs/ptc/05-06-06.pdf>

DBC-Unterstützung in Java

- Assertions
 - ◆ Seit Java 1.4. Zur Ausführung des Java-Bytecodes mit Assertions wird eine JVM 1.4 oder höher vorausgesetzt.
 - ◆ Werden auch in den API-Klassen eingesetzt
 - ◆ Können per Option des java-Aufrufs eingeschaltet (enabled)...
 - ◆ ...und abgeschaltet (disabled: Default) werden
- Vorbedingungen, Nachbedingung und Invarianten
 - ◆ ...werden als boolesche Ausdrücke in den Quelltext geschrieben.
- Vorteile von Assertions
 - ◆ Schnelle, effektive Möglichkeit Programmierfehler zu finden.
 - ◆ Bieten die Möglichkeit, Annahmen knapp und lesbar im Programm zu beschreiben.
 - ◆ Die Nutzung von Assertions zur Entwicklungszeit erlaubt es zu zeigen, dass alle Annahmen richtig sind. Die Qualität des Codes erhöht sich somit.

DBC-Unterstützung in Java

- Anwendung

- ◆ Man kann Assertions an beliebigen Stellen des Quellcodes verwenden um logische Ausdrücke zu überprüfen.

- ⇒ Liefert ein solcher Ausdruck `false` zurück, wird ein `AssertionError` ausgelöst und die Ausführung des Programms stoppt.
- ⇒ Die ausgelösten Fehler sind vom Typ `Error` und nicht vom Typ `Exception`
- ⇒ Sie müssen daher nicht abgefangen werden!

- Syntax

```
assert <boolean expression>;
```

```
assert (c != null);
```

oder:

```
assert <boolean expression> : <String>;
```

```
assert (c != null) : "Customer is null";
```

Warum DBC-Unterstützung in Java?

- Der Effekt des `assert` Statements könnte auch mit einer `if`-Anweisung und explizitem Werfen von ungeprüften Ausnahmen realisiert werden.
- Die Vorteile von sprachunterstützten Assertions sind
 - ◆ Lesbarkeit
 - ⇒ Der Quellcode wird klarer und kürzer
 - ⇒ Auf den ersten Blick ist zu erkennen, dass es sich um eine Korrektheitsüberprüfung handelt und nicht um eine Verzweigung im Programmablauf
 - ◆ Effizienz
 - ⇒ Sie lassen sich für die Laufzeit wahlweise an- oder ausschalten
 - ⇒ Somit verursachen sie praktisch keine Verschlechterung des Laufzeitverhaltens.

Design by Contract: Sprachunterstützung

- Java
 - ◆ Assertions werden ab JDK 1.4 unterstützt
 - ◆ Formulierung von pre -und, postconditions und invariants ist damit möglich
- Contract4J
 - ◆ Contracts als assertions für JDK 1.5 (Java 5) formuliert
 - ◆ <http://www.contract4j.org>
- Eiffel
 - ◆ Kontrakte voll unterstützt (preconditions, postconditions und invariants)
 - ◆ [http://en.wikipedia.org/wiki/Eiffel_\(programming_language\)](http://en.wikipedia.org/wiki/Eiffel_(programming_language))
- Spec#
 - ◆ C# mit voller Kontraktunterstützung und vielen anderen Erweiterungen
 - ◆ <http://research.microsoft.com/specsharp/>
- Andere Sprachen
 - ◆ Kontrakte zumindestens in der Dokumentation explizit machen
- In allen Sprachen
 - ◆ Kontrakte als wichtiges Kriterium beim Entwurf mit beachten → Ersetzbarkeit

Subtyping / Ersetzbarkeit und Kontrakte

B ist ein Subtyp von **A**



Instanzen von **B** sind immer für Instanzen von **A** einsetzbar



Instanzen von **B** bieten mindestens und fordern höchstens
das gleiche wie Instanzen von **A**



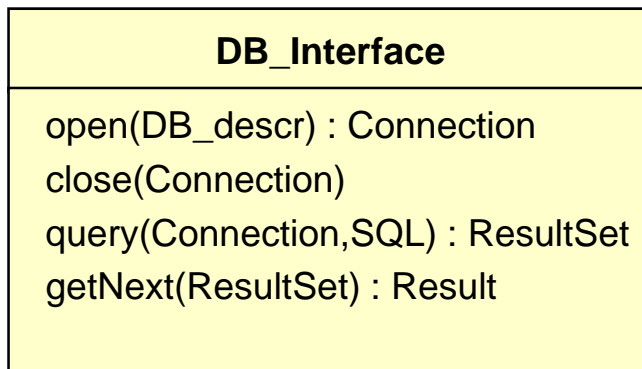
Instanzen von **B** haben mindestens alle Methoden von **A**,
und zwar mit (gleichen oder) stärkeren Nachbedingungen
und (gleichen oder) schwächeren Vorbedingungen

Schnittstellen-Spezifikation: Interaktionsspezifikation durch „Behaviour Protocols“

Interaktionsspezifikation durch „Behaviour Protocoll“

- Gegeben

- ◆ Folgende Typspezifikation
- ◆ ... und sogar zugehörige Contracts



- Problem

- ◆ Wir wüssten trotzdem nicht, wie das beabsichtigte Zusammenspiel der einzelnen Methoden ist.
- ◆ Kann man jede in jeder beliebigen Reihenfolge aufrufen, unabhängig von dem was man vorher aufgerufen hat?

- Lösung

- ◆ Zu jedem Typ wird sein „Verhaltensprotokoll“ mit angegeben
- ◆ Es ist ein regulärer Ausdruck der legale Aufrufsequenzen und Wiederholungen spezifiziert

- Beispiel

- ◆ „Erst Verbindung zuer Datenbank erstellen, dann beliebig oft anfragen und in jedem Anfrageergebnis beliebig oft Teilergebnisse abfragen, dann Verbindung wieder schließen.“

```
protocoll DB_Interface_Use =  
  open(DB_descr) ,  
  ( query(Connection,SQL) : ResultSet ,  
    ( getNext(resultSet) : Result )* ,  
  )* ,  
  close(Connection)
```

Schnittstellen-Spezifikation: Sichtbarkeitsinformationen

Hinzufügen von Sichtbarkeitsinformationen

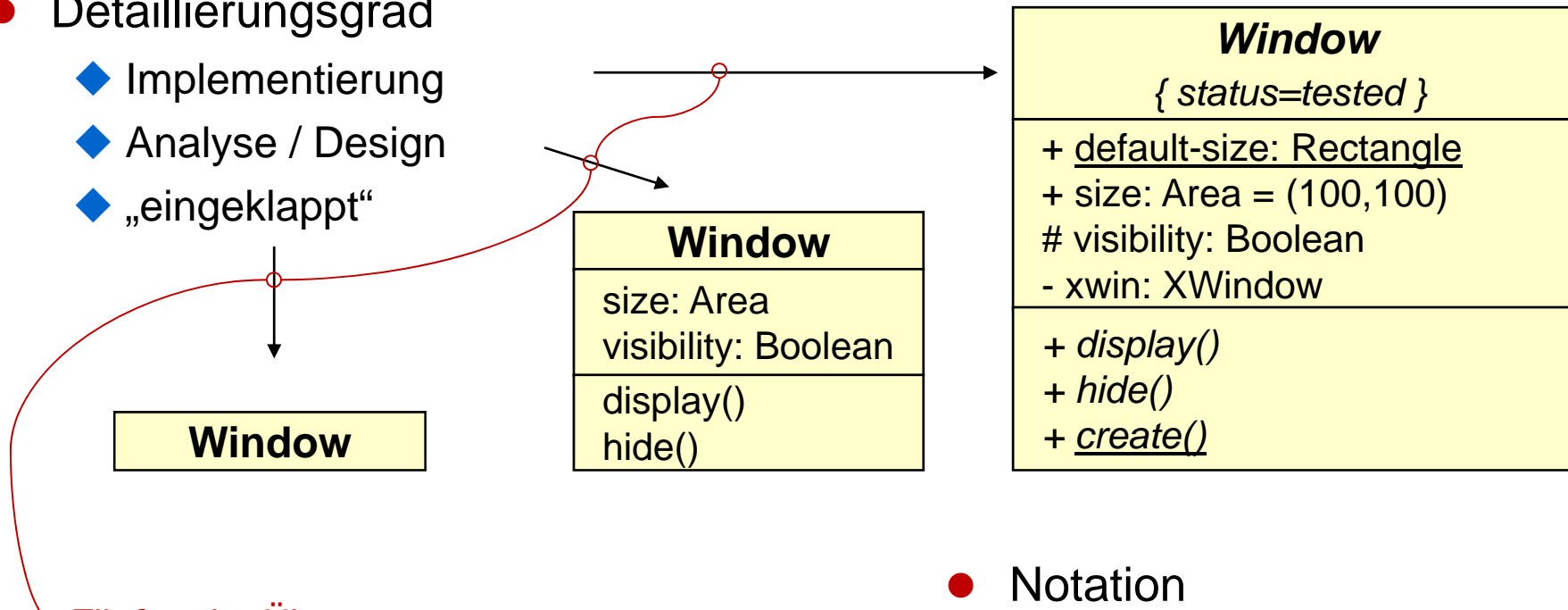
UML definiert drei Sichtbarkeitsgrade:

- Private (-)
 - ◆ Auf diese Attribute/Operationen kann nur aus der sie definierenden Klasse zugegriffen werden.
- Protected (#)
 - ◆ Auf diese Attribute/Operationen kann nur aus der sie definierenden Klasse und deren Subklassen zugegriffen werden.
- Public (+)
 - ◆ Auf diese Attribute/Operationen kann aus jeder Klasse zugegriffen werden.
- Die in Java bekannte „package Sichtbarkeit“ gibt es in der UML nicht. Wenn gewünscht, entsprechenden Stereotyp benutzen, z.B. <<package_visible>>.

Verschiedene Sichten von Klassen

- Detaillierungsgrad

- ◆ Implementierung
- ◆ Analyse / Design
- ◆ „eingeklappt“



Fließender Übergang

CASE-Tools unterstützen einen fließenden Übergang, indem verschiedene Sichten definiert bzw. verschiedene Detail-Level ein- und ausgeblendet werden können.

- Notation

- ◆ public: +
- ◆ protected: #
- ◆ private: -
- ◆ Klassenvariable / -methode
- ◆ *Abstrakte Klasse / Methode*

Sichtbarkeiten Zuletzt

- Festlegung der Sichtbarkeiten ist schon sehr implementierungsnah.
- Daher ist das der letzte Schritt, nach all den verschiedenen Formen der Festlegung der Typen über Signatur, Kontrakte und Protokolle.

Schnittstellen-Spezifikation: Explizite Spezifikation von „Benutzten Schnittstellen“

Siehe Abschnitt über „Komponenten“ im Kapitel „Systementwurf“

Aktivitäten während des Objektentwurfs

1. Schnittstellenspezifikation

- ◆ Spezifikation der Schnittstellen aller Typen (Klassen und Interfaces)

2. Auswahl vorhandener Komponenten

- ◆ Bestimmung von Standardkomponenten

3. Restrukturierung des Objektmodells

- ◆ Umformen des Objektmodell zur Verbesserung von Verständlichkeit und Erweiterbarkeit sowie Realisierung von UML-Konzepten, die keine Entsprechung in Ihrer Programmiersprache haben

4. Optimierung des Objektmodells

- ◆ Umformen des Objektmodells unter Berücksichtigung von Performanzkriterien, wie Reaktionszeit (z.B. der Benutzerschnittstelle) oder Speicherbedarf.

Wiederverwendung

Wiederverwendung

- Wähle passende Datenstrukturen zu den jeweiligen Algorithmen
 - ◆ Container-Klassen
 - ◆ Arrays, Listen, Queues, Stacks, Mengen, Bäume
- Suche nach vorhandenen Klassen in Klassenbibliotheken
 - ◆ JSAPI, JTAPI, ...
- Definiere neue interne Klassen und Operationen nur wenn nötig.
 - ◆ Komplexe Operationen erfordern eventuell Zerlegung
 - ⇒ in neue Teiloperationen
 - ⇒ in neue Klassen

Nutzung bestehender Software

- Komponenten-Suchmaschinen
 - ◆ Bsp: MeroBase, CodeConjurer → Prof. Atkinson, Universität Mannheim
- Auswahl existierender Standardklassenbibliotheken, Frameworks oder Komponenten
 - ◆ Eigene oder von Drittanbietern (open source oder kommerziell)
- Anpassung der Standardklassenbibliotheken, Frameworks oder Komponenten
 - ◆ Nutzung vorgesehener Anpassungsmöglichkeiten
 - ⇒ Parametrisierung
 - ⇒ Bildung von Unterklassen
 - ⇒ Konfiguration per „deployment descriptor“
 - ◆ Unvorhergesehene Anpassung
 - ⇒ Änderungen der API, falls der Quellcode verfügbar ist
 - ⇒ Sonst: Adapter oder Bridge Pattern
 - ⇒ Sonst: Aspektorientierte Programmierung

Erweiterung und Restrukturierung des Objektmodells

Umsetzung von fortgeschrittenen UML-Konzepten in „Kern-UML“
Umsetzung von Kern-UML in implementierungsnahes UML
Verbesserung von Verständlichkeit und Erweiterbarkeit

Aktivitäten während des Objektentwurfs

1. Schnittstellenspezifikation

- ◆ Spezifikation der Schnittstellen aller Typen (Klassen und Interfaces)

2. Auswahl vorhandener Komponenten

- ◆ Bestimmung von Standardkomponenten zusätzliche Objekte der Lösungsdomäne

3. Ergänzung und Restrukturierung des Objektmodells

- ◆ Umformen des Objektmodell zur Umsetzung von UML-Konzepten, die nicht direkt von der Realisierungsumgebung unterstützt werden
- ◆ ... sowie zur Verbesserung von Verständlichkeit und Erweiterbarkeit

4. Optimierung des Objektmodells

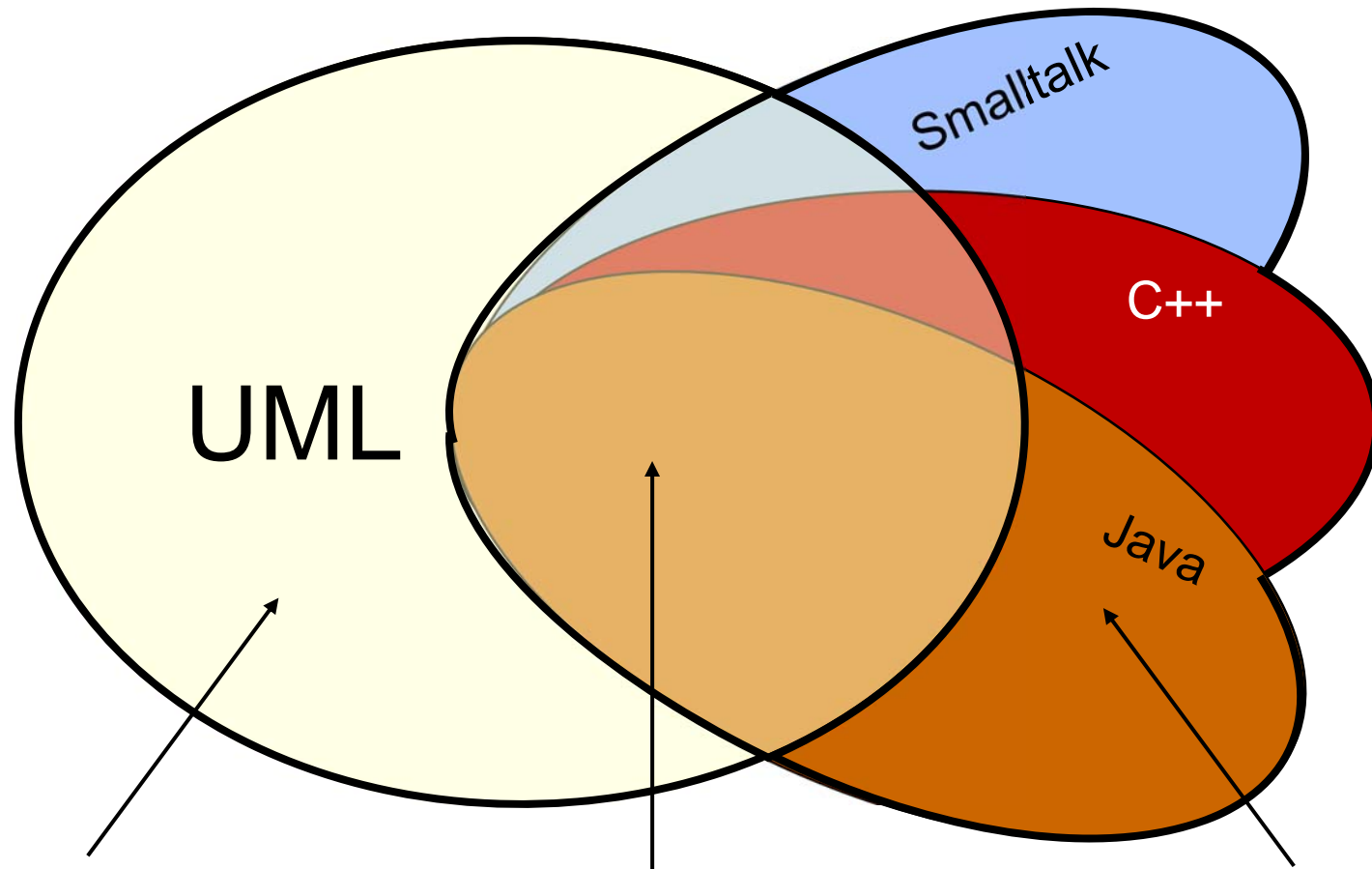
- ◆ Umformen des Objektmodells unter Berücksichtigung von Performanzkriterien, wie Reaktionszeit (z.B. der Benutzerschnittstelle) oder Speicherbedarf.

Erweiterung und Restrukturierung

► Fortgeschrittene UML-Konzepte

Abgeleitete Attribute
Assoziationen als Klassen
Assoziationsklassen
Qualifizierte Assoziationen

Bezug von UML zu gängigen OO Sprachen



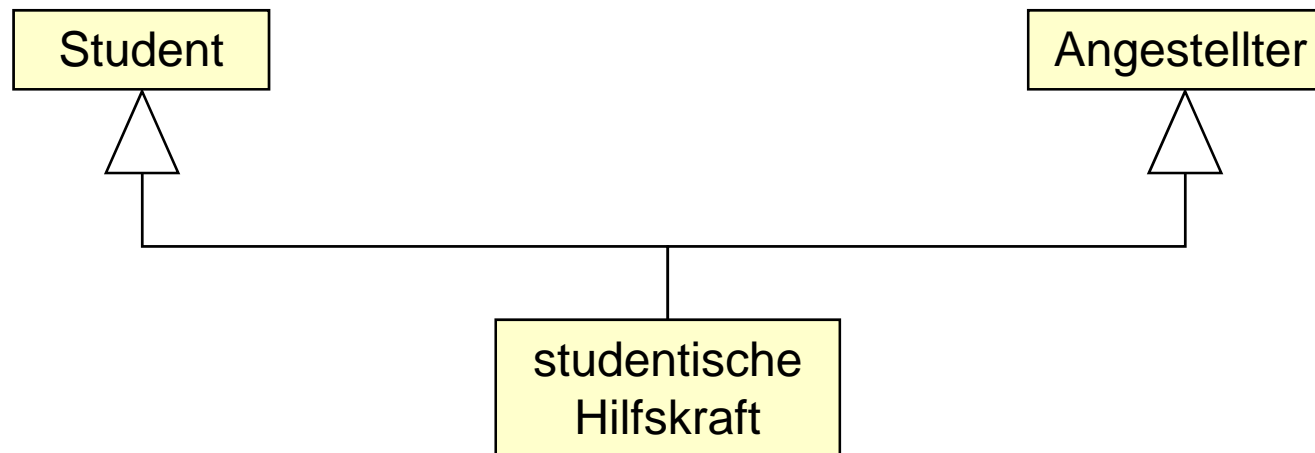
Konzepte ohne direkter
Entsprechung in
gängigen Sprachen

direkt ineinander
abbildbarer
gemeinsamer „Kern“

sprachspezifische
Details

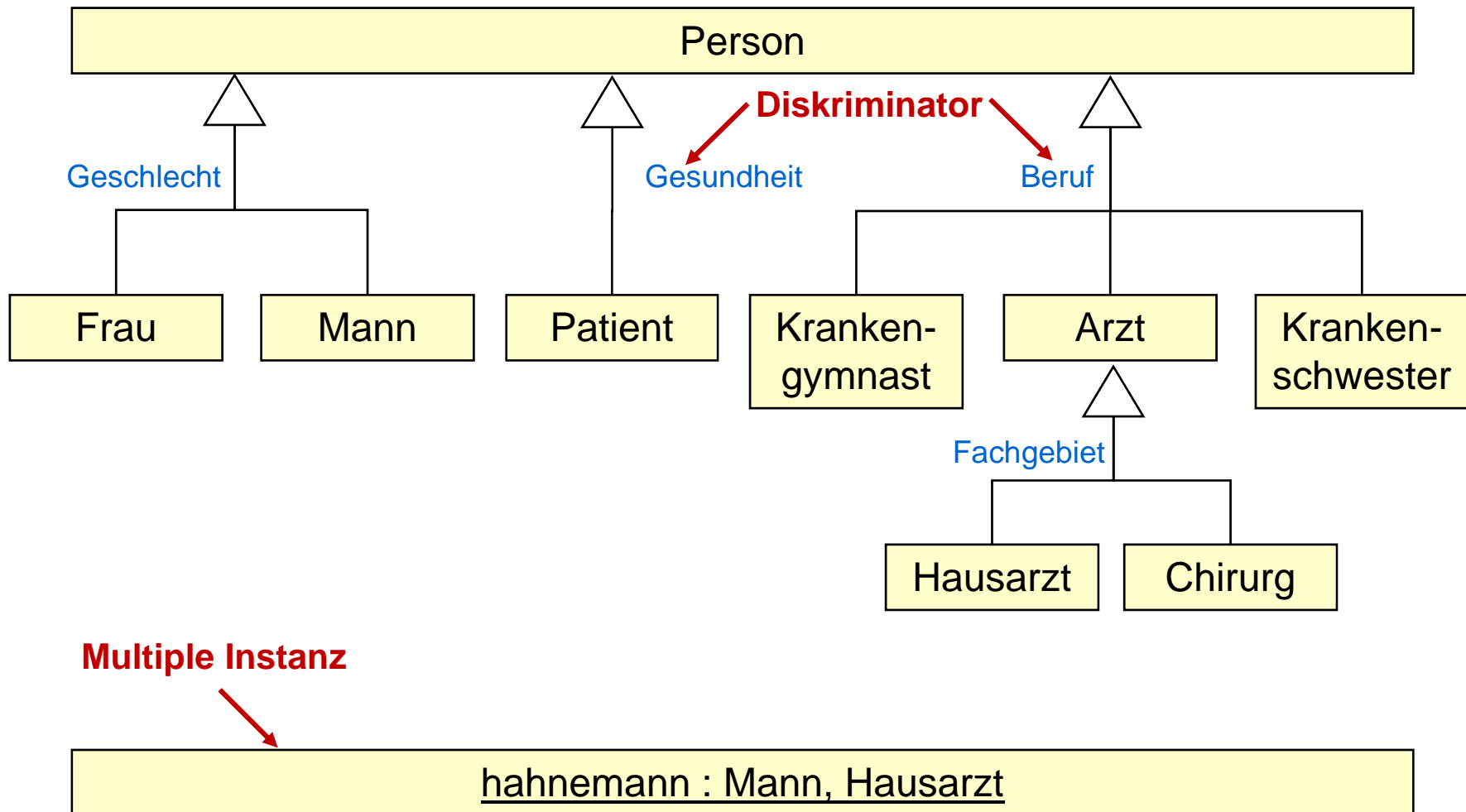
UML, statisches Modell: Multiple Vererbung

- Eine Klasse mit mehreren Oberklassen



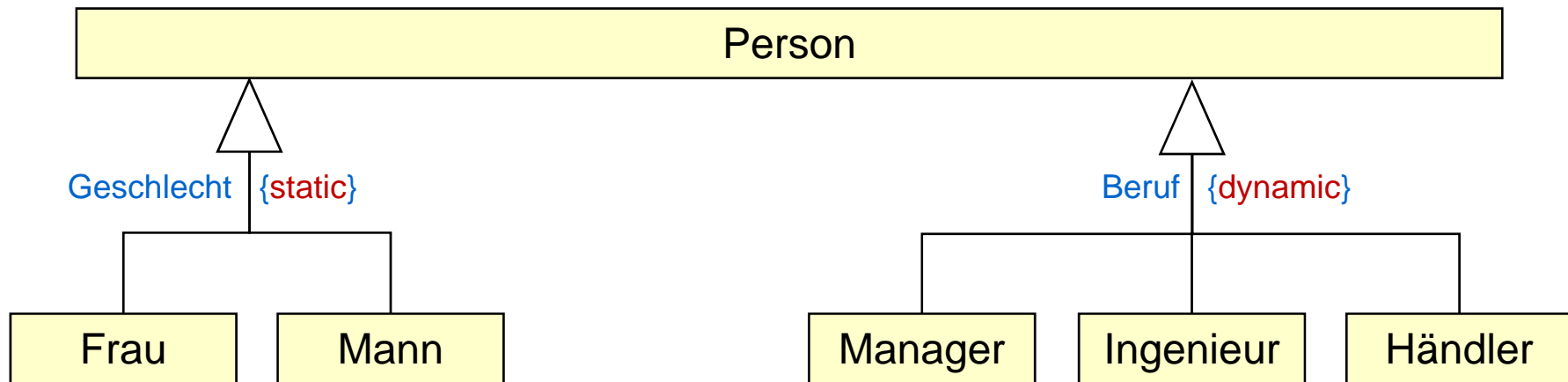
UML, statisches Modell: Multiple Klassifikation

- Ein Objekt kann Instanz mehrerer Klassen sein



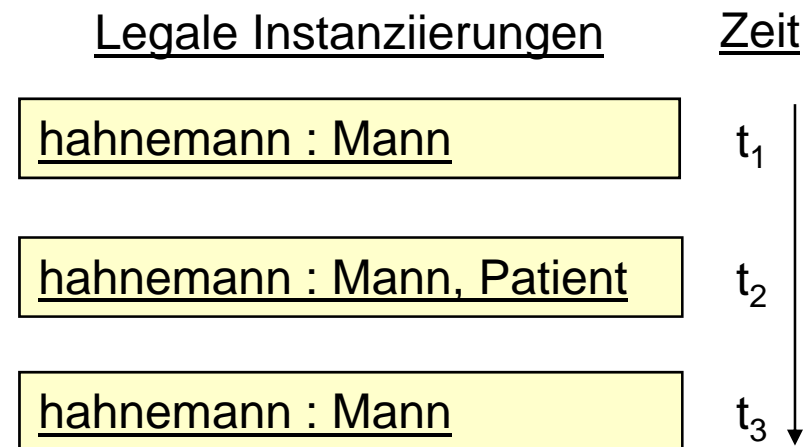
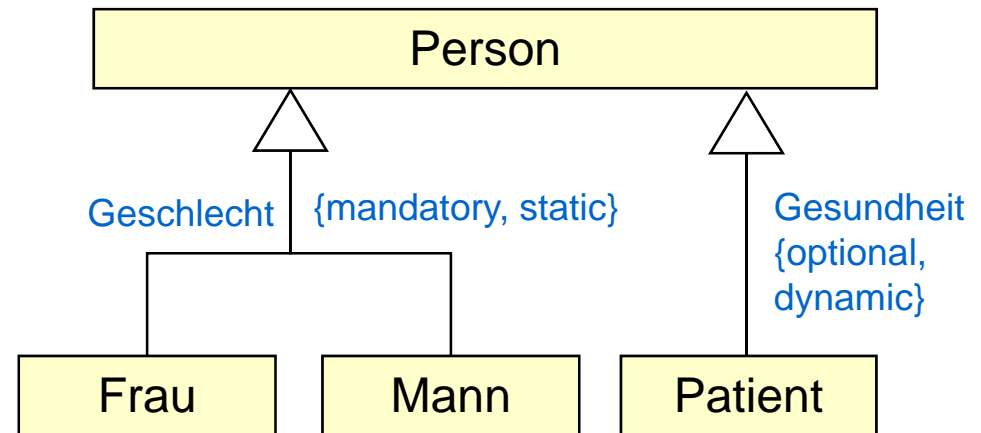
UML, statisches Modell: Dynamische Klassifikation

- Eine Objekt kann Instanz mehrerer Klassen sein
- ... und seine Klassenzugehörigkeit ändern



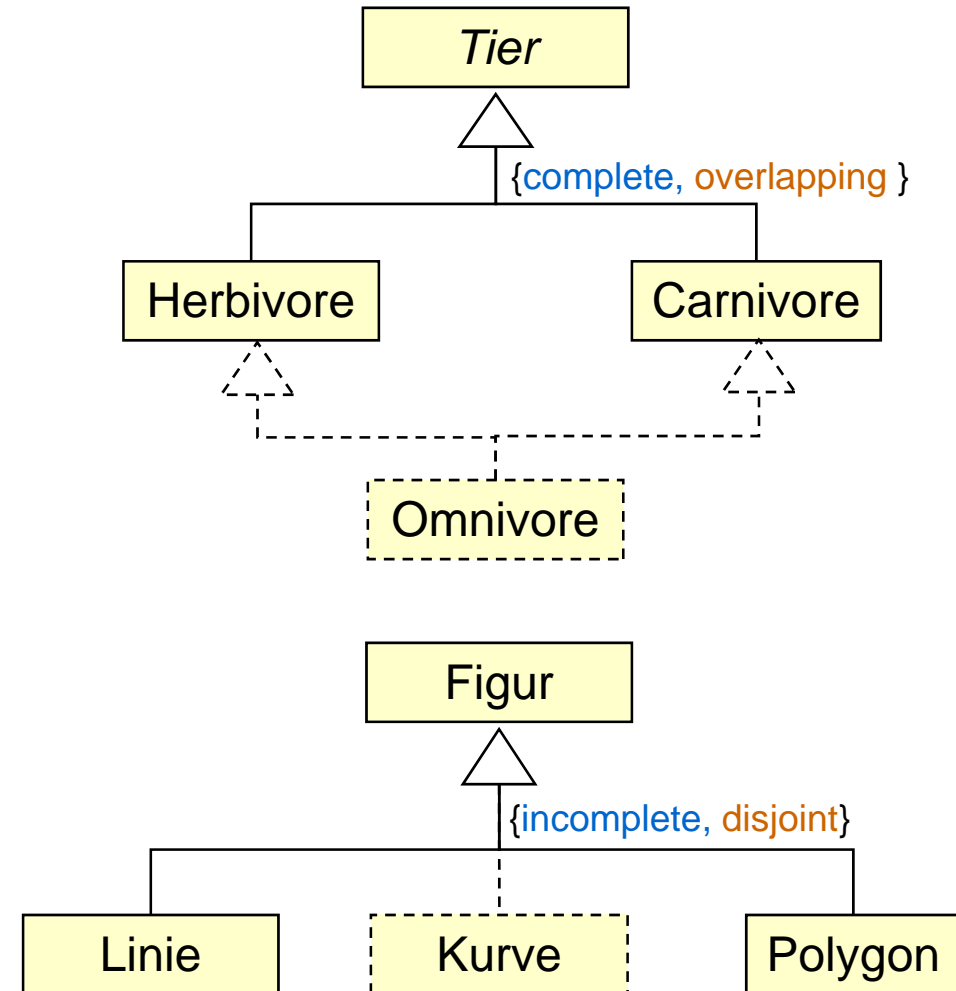
UML: Partitionierung von Unterklassen

- obligatorisch (mandatory)
 - ◆ Objekte müssen einer Klasse aus dieser Partition angehören
- optional (optional) (default)
 - ◆ Objekte müssen nicht ...
- statisch (static) (default)
 - ◆ Objekte bleiben lebenslang in einer Klasse
- dynamisch (dynamic)
 - ◆ Objekte können Klasse wechseln
 - ◆ impliziert "optional"



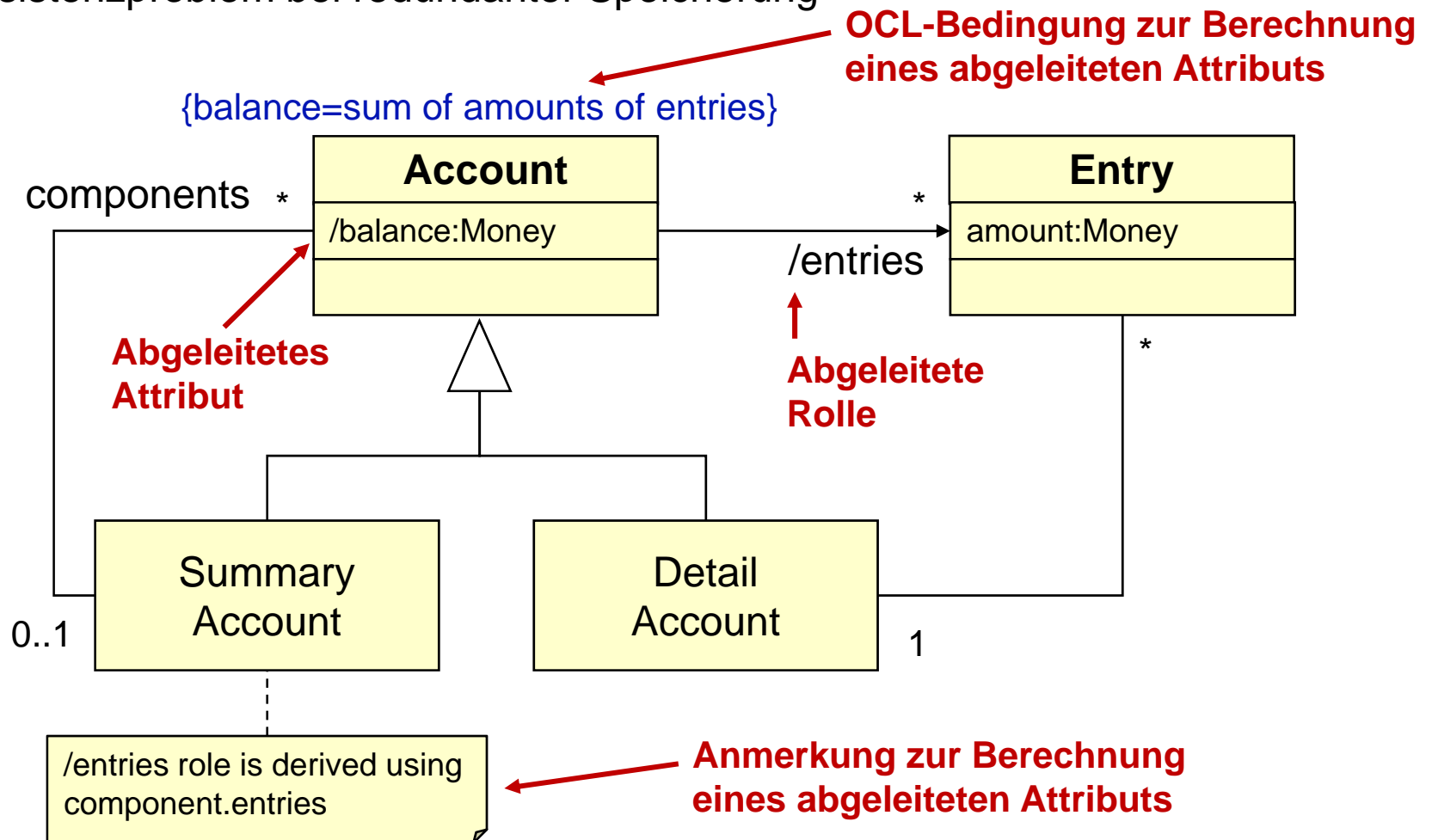
UML: Partitionierung von Unterklassen

- vollständig (**complete**) (default)
 - ◆ impliziert Abstraktheit der Oberklasse (falls es keine andere unvollständige Partition gibt)
- Überlappung (**overlapping**)
 - ◆ impliziert Existenz gemeinsamer Subtypen der Unterklassen
- unvollständig (**incomplete**)
 - ◆ hat oft konkrete Oberklasse für alle nicht explizit gemachten weiteren Alternativen
- disjunkt (**disjoint**) (default)
 - ◆ impliziert Fehlen gemeinsamer Subtypen der Unterklassen



UML: Abgeleitete Attribute

- Können aus anderen Attributen berechnet werden
- Geben Hinweis auf Abwägung zwischen Neuberechnungsaufwand und Konsistenzproblem bei redundanter Speicherung



Erweiterung und Restrukturierung

- ▶ **Umsetzung fortgeschrittener UML-Konzepte in „Kern-UML“**
 - ▶ **„Split Object“-Entwurfsmuster**

Strategy Pattern als motivierendes Beispiel

Essenz von Split Objects

Weitere wichtige „Split-Object“-Patterns: State, Multiple Vererbung, Decorator

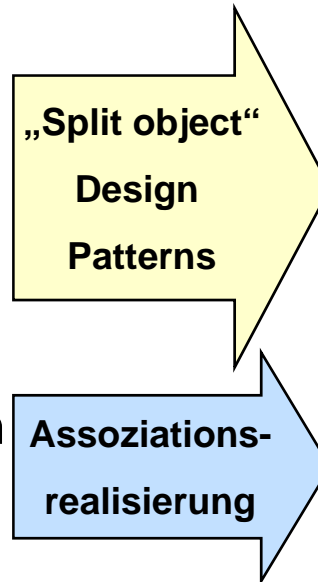
Restrukturierung des Objektmodells:

UML_{High} → UML_{Core}

UML_{High}

- Dynamische Klassifikation
- Multiple Instanziierung
- Multiple Vererbung

- Bidirektionale Assoziationen
- Qualifizierte Assoziationen



UML_{Core}

- Statische Klassifikation
- Einfache Instanziierung
- Einfache Vererbung

- Unidirektionale Assoziationen (Felder)

In diesem Abschnitt

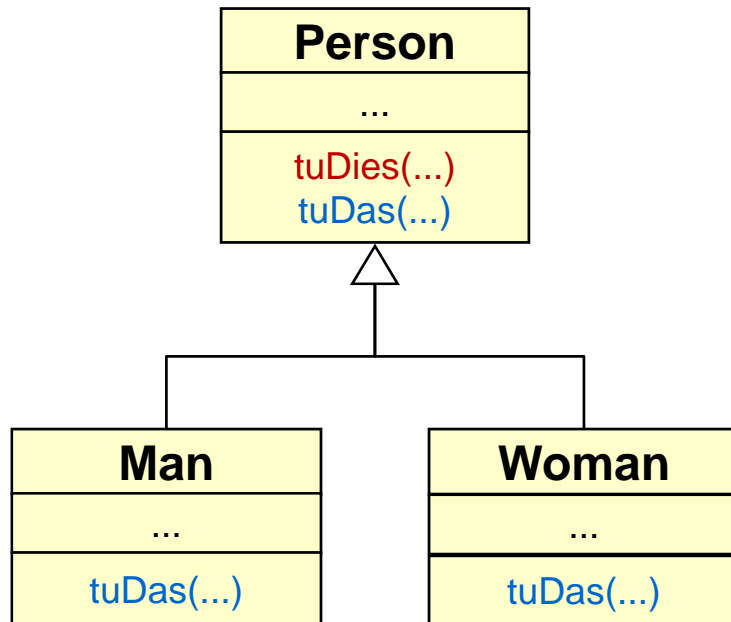
- Transformation von konzeptionellem Entwurf in „UML_{High}“ in implementierungsnahen Entwurf in „Kern-UML“
- Umsetzung in verfügbare Zielsprache danach einfach, da die Kern-UML-Konzepte 1:1 in Programmiersprachen unterstützt sind

Restrukturierung des Objektmodells: UML_{High} → UML_{Low}

Typische Umsetzungsbeispiele

- Multiple Vererbung
 - ◆ Wiederverwendung: durch Aggregation und Forwarding
 - ◆ Subtyping: durch Implementation eines gemeinsamen Interfaces
 - ◆ Overriding: durch „Rückreferenzen“ (als Parameter oder Feld)
- Multiple Instanziierung / Multiple Sichten
 - ◆ Decorator (evtl. mit obiger Simulation von Subtyping und Overriding)
- Dynamische Klassifikation / Dynamische Änderung der Klassenzugehörigkeit
 - ◆ Strategy

Motivation: Ein Beispiel



Wie modelliert man

- objektspezifisches
- zustandsspezifisches
- dynamisch veränderbares

Verhalten?

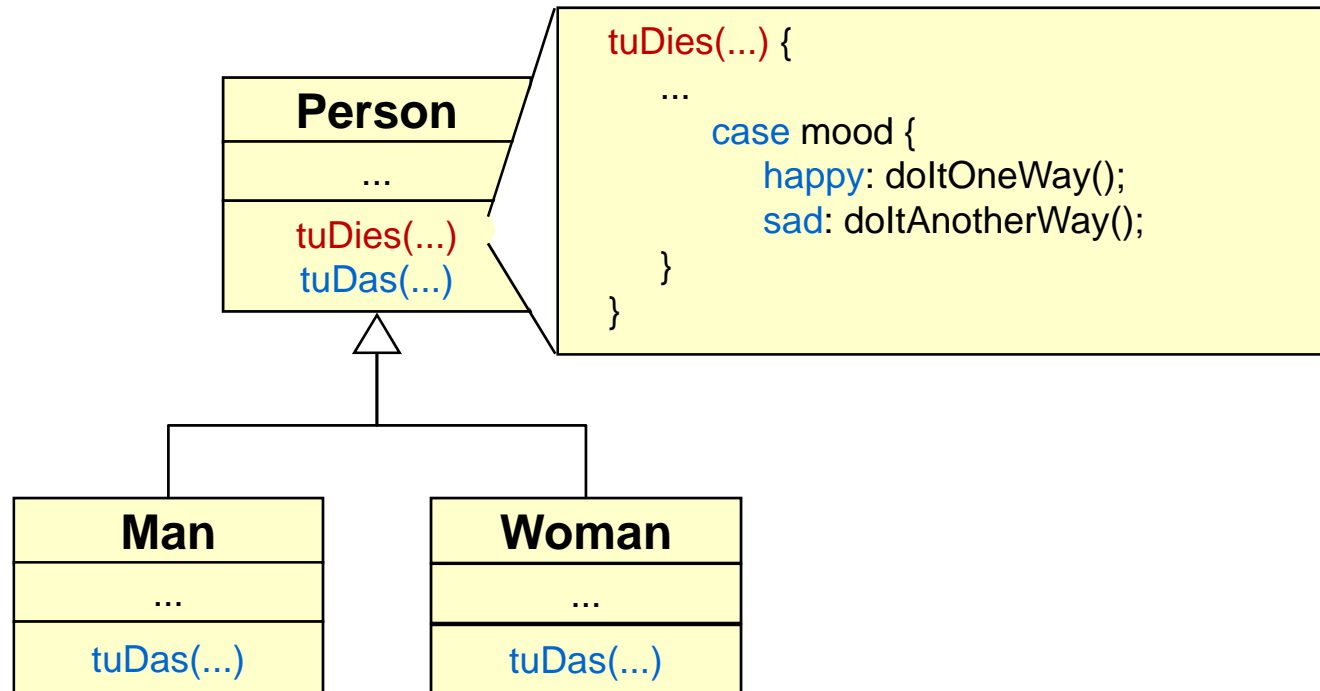
⇒ **tuDas()** ist geschlechtsspezifisch einheitlich



⇒ **tuDies()** ist personenspezifisch verschieden

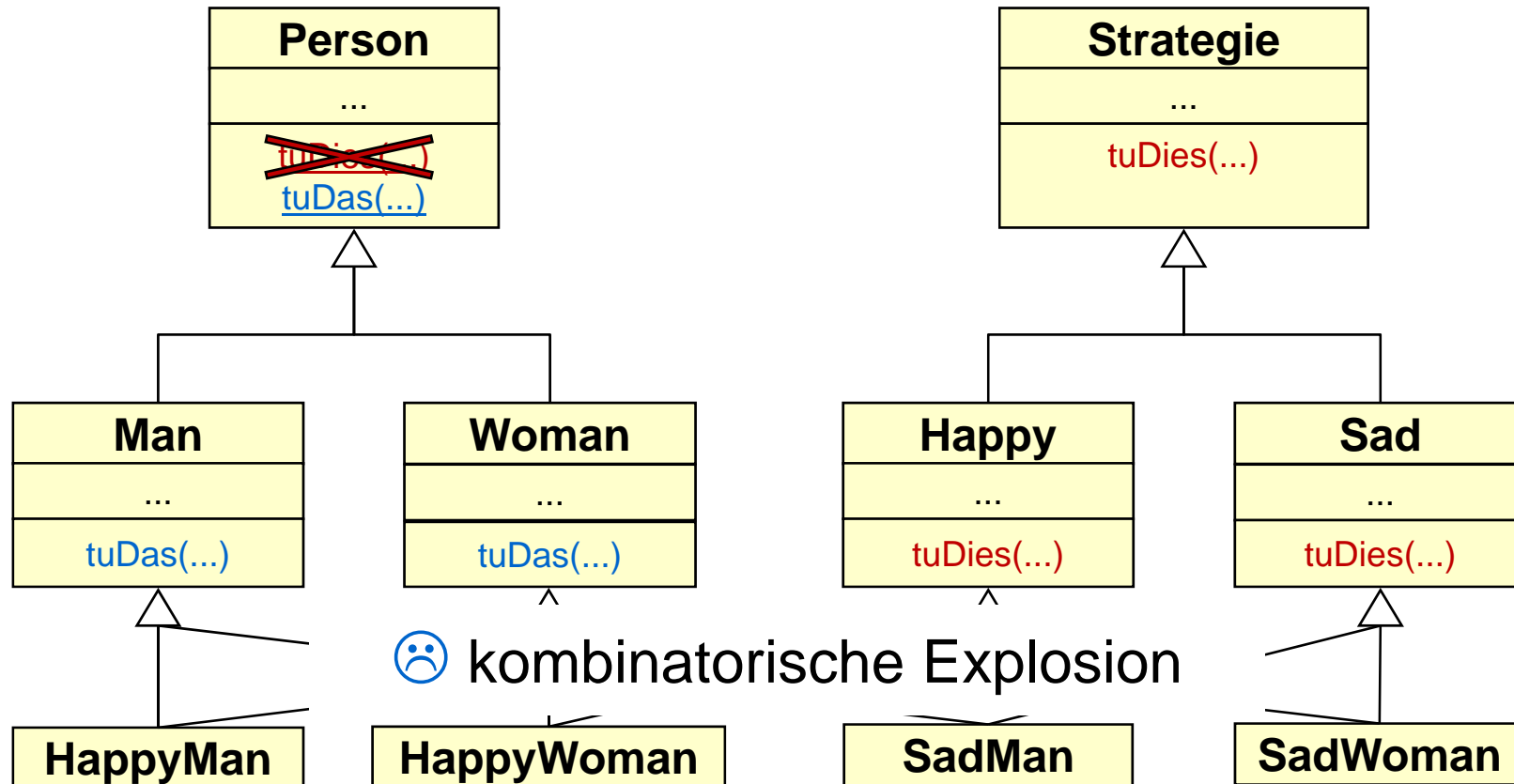


1. Versuch: "Fest Kodieren"



- Schlecht, denn jede neue Alternative („begeistert“, „depressiv“, ...) erfordert Änderung einer jeden mit Stimmungen befassten „case“-Anweisung
- Typischerweise ist ja nicht nur eine Aktion (hier: `tuDies`) stimmungsabhängig...

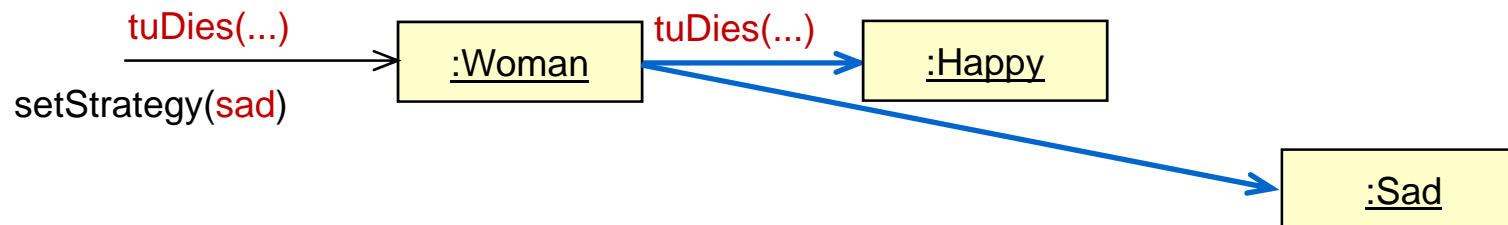
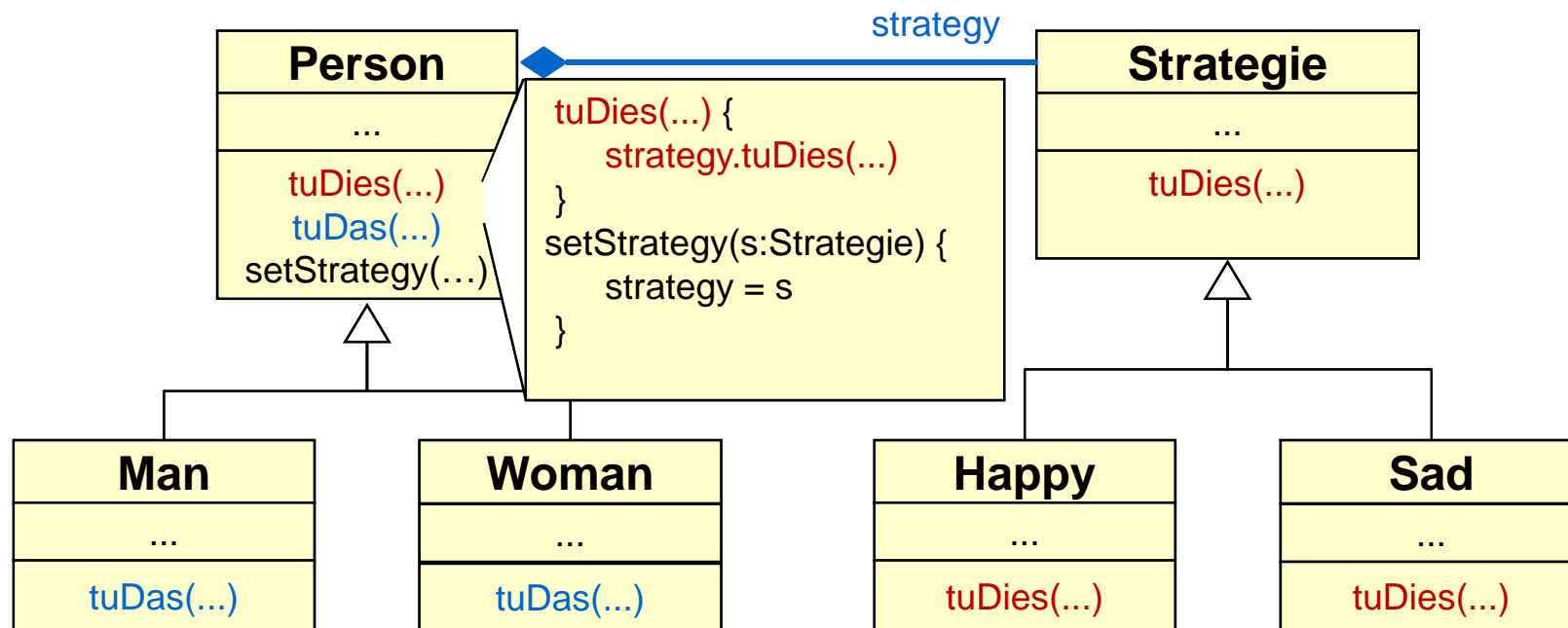
2. Versuch: "Multiple Vererbung"



☹ klassenspezifisch festes Verhalten

:SadWoman

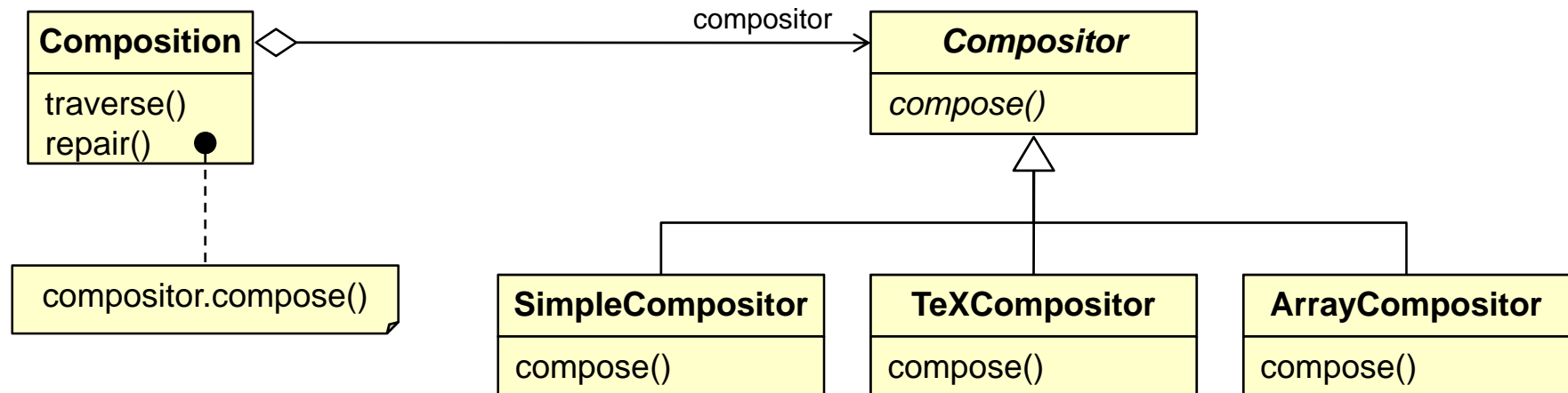
3. Versuch: "Strategy Pattern"



Das Strategy Pattern

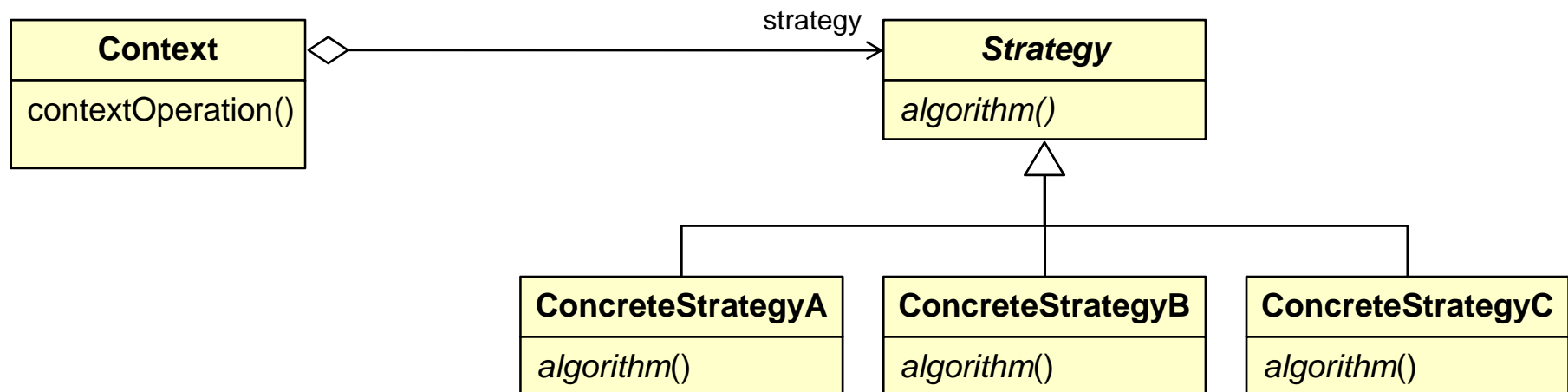
Das Strategy Pattern: Einführung

- Absicht
 - ◆ Kapselung einer Familie von Algorithmen mit der Möglichkeit, sie beliebig auszutauschen.
- Motivation
 - ◆ Berechnung von Zeilenumbrüchen
 - ⇒ mehrere Algorithmen können eingesetzt werden
 - ⇒ neue Varianten sollen später hinzugefügt werden können
- Struktur (für obiges Beispiel)



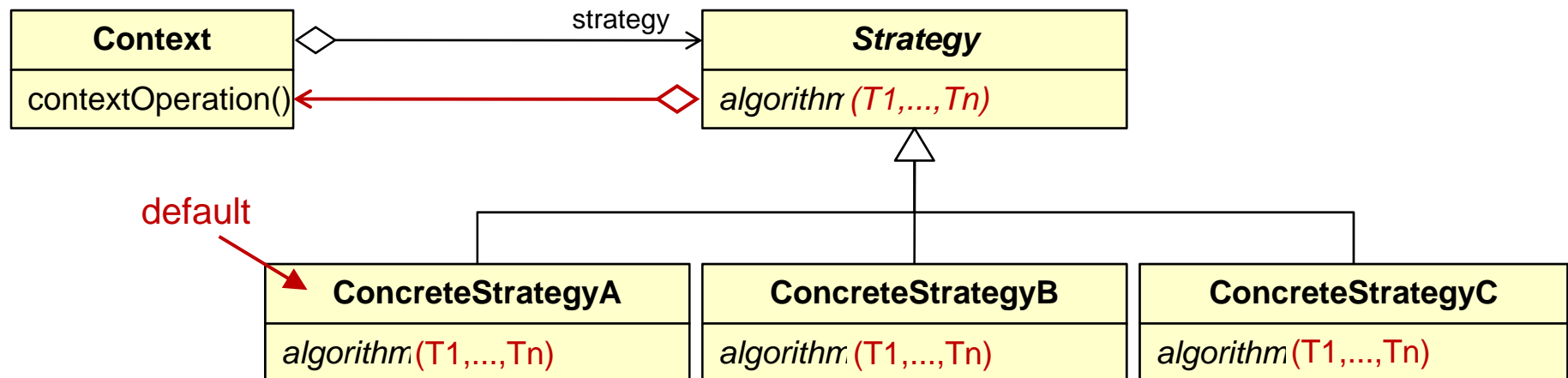
Das Strategy Pattern: Anwendbarkeit und Struktur

- Anwendbar in folgendem Kontext
 - ◆ Einige ähnliche Klassen unterscheiden sich nur in gewissen Aspekten des Verhaltens. Diese können in ein Strategie-Objekt ausgelagert werden.
 - ◆ Verhalten ist abhängig von äußeren Randbedingungen
 - ◆ Verschiedene Varianten eines Algorithmus werden benötigt
 - ⇒ z.B. mit unterschiedlicher Zeit-/Platzkomplexität.
 - ◆ Kapselung von Daten eines komplexen Algorithmus
- Struktur (allgemein)

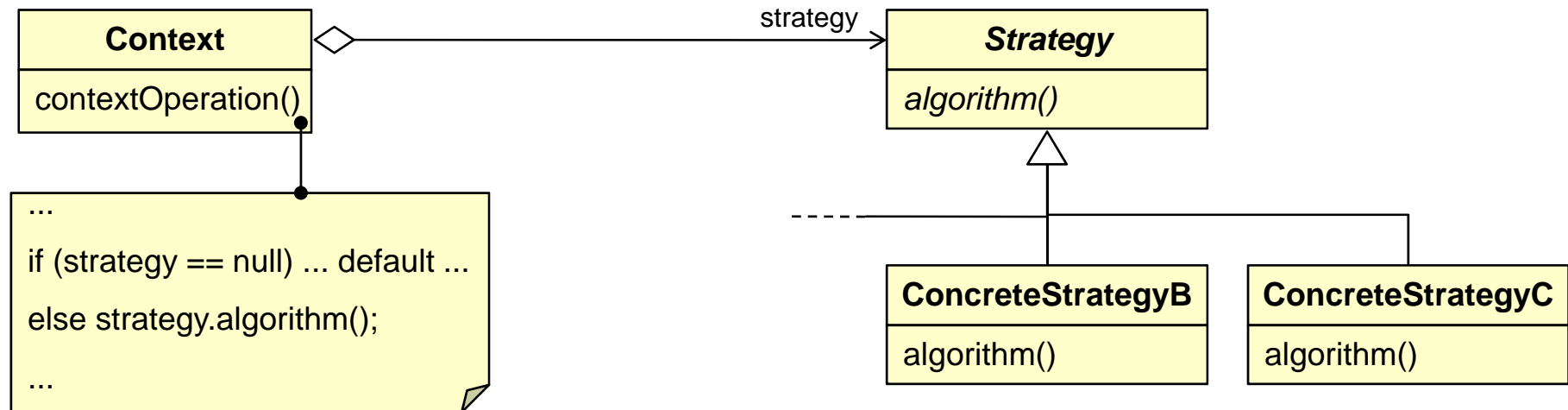


Das Strategy Pattern: Implementierung

- Alternative Schnittstellen zwischen Kontext und Strategien
 1. Kontext übergibt alle relevanten Daten an die Strategie-Methode
 2. Kontext übergibt nur **this** an Strategie-Methode → flexibelste Lösung
 3. Strategie-Objekt speichert bei Initialisierung feste Referenz auf Kontext
- Implementierung von Default-Verhalten möglich
 - ◆ In der Kontext-Klasse wird ein Default-Verhalten verwendet, wenn kein Strategie-Objekt gesetzt ist.

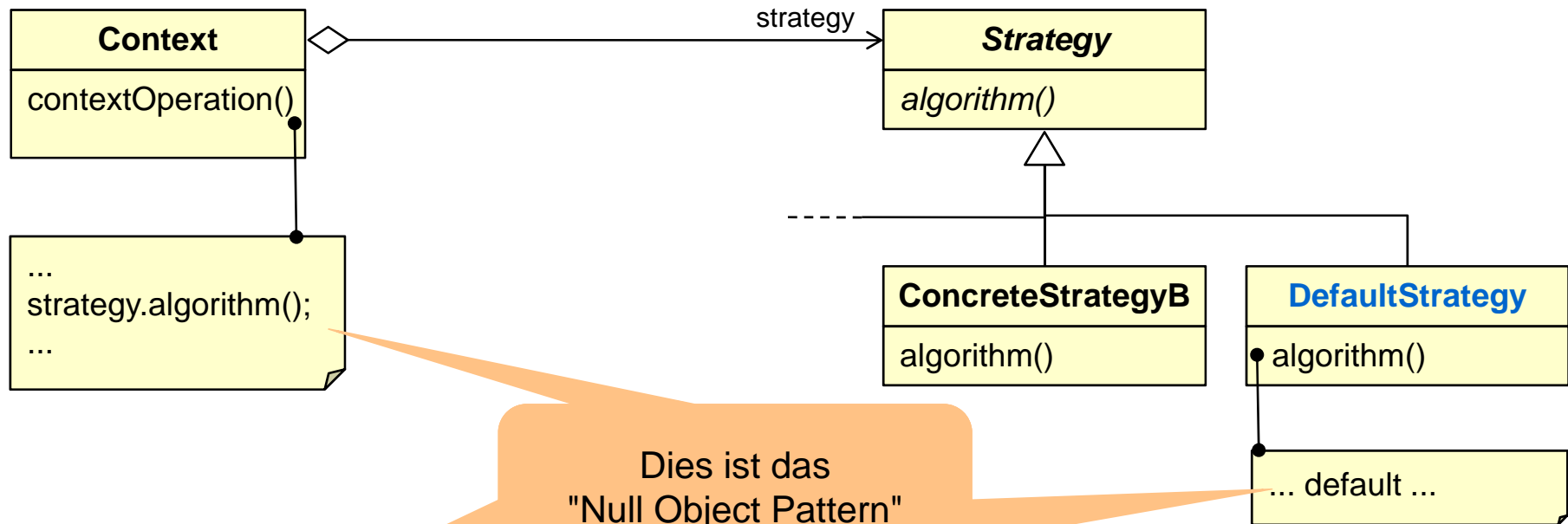


Implementierung: Fallunterscheidung in Kontext



- Vorteile
 - ◆ ???
- Nachteile
 - ◆ Uneinheitliche Lösung: Kontext muss Default-Strategie kennen

Implementierung: „Default-Strategie“-Klasse



- Idee

- ◆ Jede Zuweisung "`Strategy s = null;`" ersetzen durch "`Strategy s = new DefaultStrategy();`"
- ◆ Abfragen auf `null` und entsprechende Fallunterscheidungen löschen

- Vorteile

- ◆ lesbarer Code
- ◆ einheitliche Lösung, klare Trennung von Kontext und Strategien

Das Strategy Pattern: Konsequenzen

- Konzeptuell

- ◆ Familie von zusammengehörigen Algorithmen
- ◆ Auswahl verschiedener Implementierungen desselben Verhaltens
- ◆ dynamische Alternative zu Unterklassenbeziehung
- ◆ Polymorphismus statt Fallunterscheidungen (if-then-else, switch-case)
- ◆ leichtere Erweiterbarkeit

- Konsequenzen aus Implementierung

- ◆ Kontext übergibt evtl. Parameter, die nicht jedes Strategie-Objekt benötigt
 - ⇒ this zu übergeben ist allgemeiner
- ◆ Zusätzliche Nachrichten zwischen Kontext und Strategie
- ◆ Erhöhte Anzahl an Objekten
 - ⇒ Möglicherweise können aber Strategie-Objekte gemeinsam verwendet werden
 - ⇒ Flyweight-Pattern

Strategy Pattern: Bewertung

- Prinzip
 - ◆ Dekomposition: 2 Objekte
 - ◆ Weiterleitung von Anfragen

- Vorteile
 - ◆ Dynamik
 - ◆ Multiplizität
 - ◆ Erweiterbarkeit

Split Objects: Prinzipien

Was sind also "Split Objects"?

- Definition

- ◆ verschiedene Objekte die konzeptuell als eine Einheit agieren
- ◆ (schieubar) gemeinsame Identität
- ◆ (schieubar) gemeinsamer Zustand
- ◆ (schieubar) gemeinsames Verhalten
- ◆ Clients glauben mit einem einzigen Objekt zu interagieren

- Motivation

- ◆ **Vorausschauende Dekomposition**


- ⇒ Aus konzeptuellem Objekt Teilaspekte (Zustand / Verhalten) extrahieren, die austauschbar sein sollen

- ◆ **Unvorhergesehene Komposition**

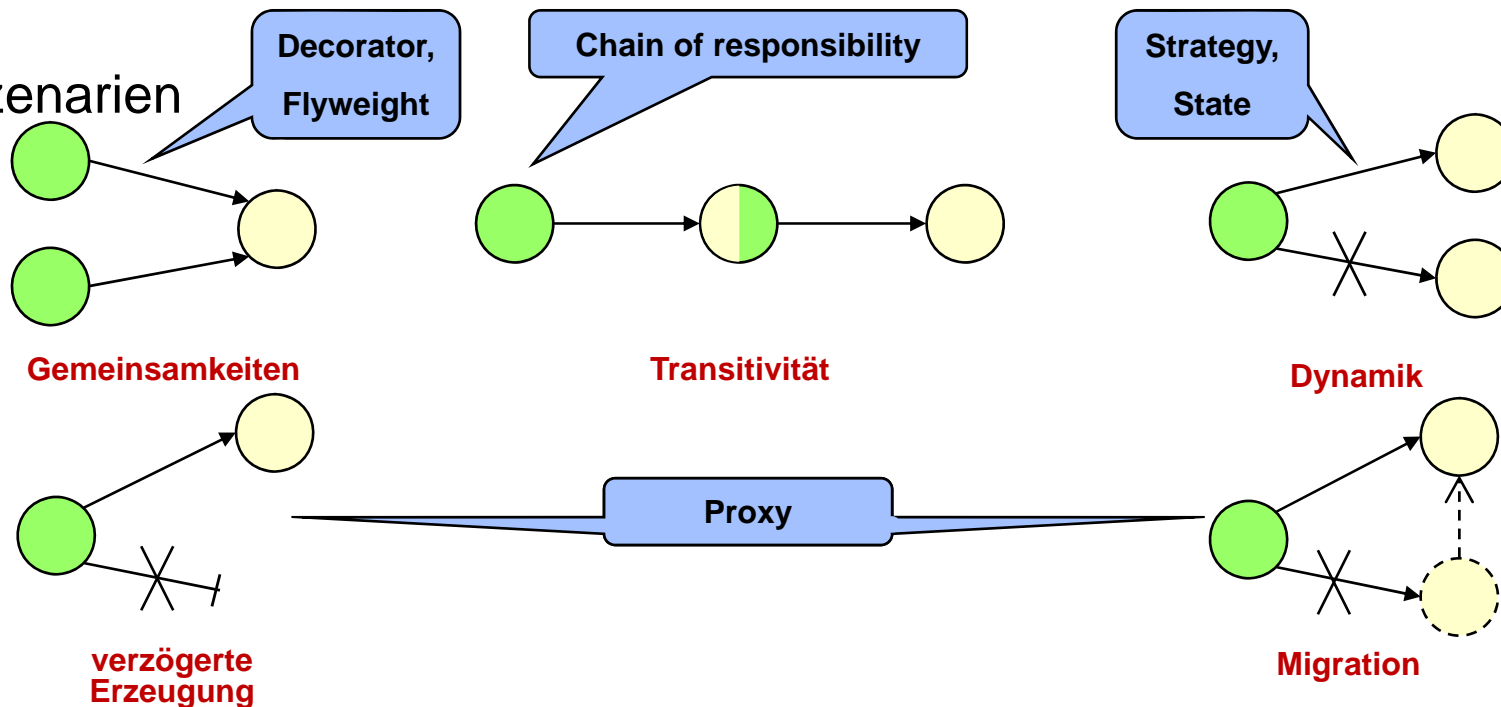
- ⇒ Zu existierendem Objekt nachträglich Teilaspekte (Zustand / Verhalten) hinzufügen, die konzeptuell dazugehören

Split Objects

- Technik

- ◆ Mehrere physikalische Objekte
- ◆ Nur eines davon ist nach außen hin sichtbar 
- ◆ Es stellt das Interface des konzeptuellen Gesamtobjektes zur Verfügung
- ◆ ... indem es die Fähigkeiten der anderen mit benutzt

- Szenarien



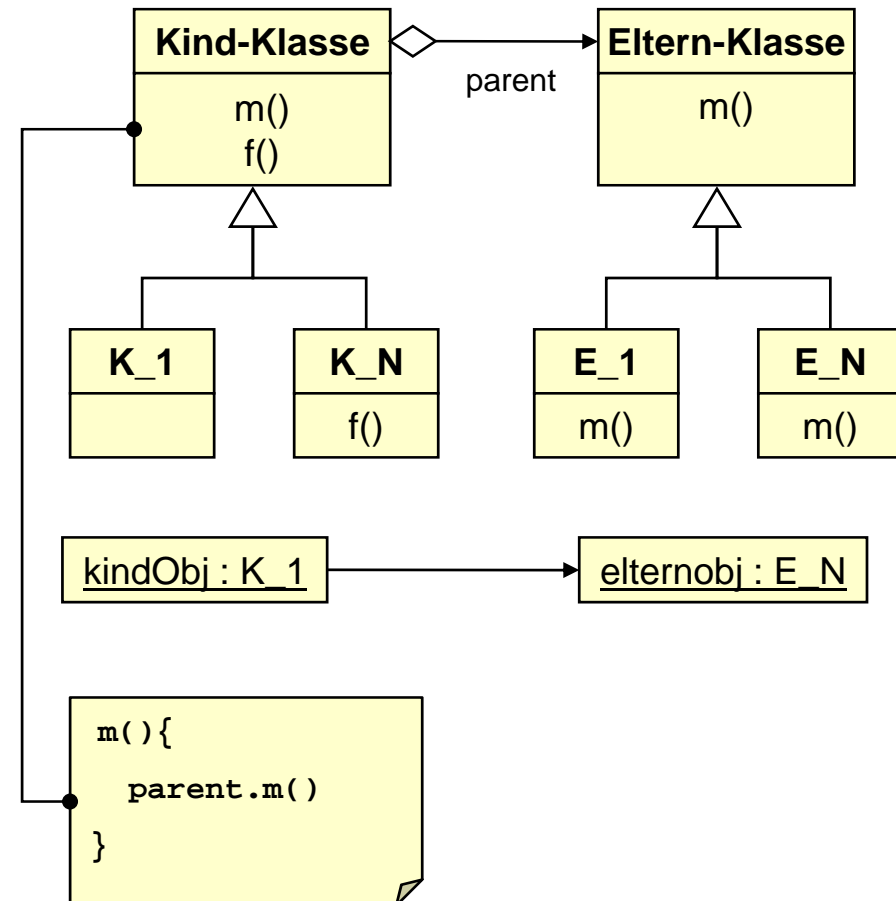
Split Objects sind die Grundlage vieler Design Patterns

- Vorausschauende Dekomposition
 - ◆ Proxy
 - ◆ Strategy
 - ◆ State
 - ◆ Flyweight

- Unvorhergesehene Komposition
 - ◆ Adapter
 - ◆ Decorator
 - ◆ Chain of Responsibility

Gemeinsame Struktur

- Aggregation
 - ◆ Kind-Klasse ist "Ganzes"
 - ◆ Eltern-Klasse ist "Teil"
 - ◆ modellieren zusammen den prinzipiellen Ablauf der Interaktion
- evtl. Unterklassen
 - ◆ modellieren Variabilität
 - ◆ beliebige Kombination der Instanzen
- Forwarding
 - ◆ Kind leitet empfangene Nachricht an Eltern-Objekt weiter
 - ◆ Grundlage für Code-Wiederverwendung



Beziehung zwischen Kind- und Elternobjekt

Elementare Beziehungen

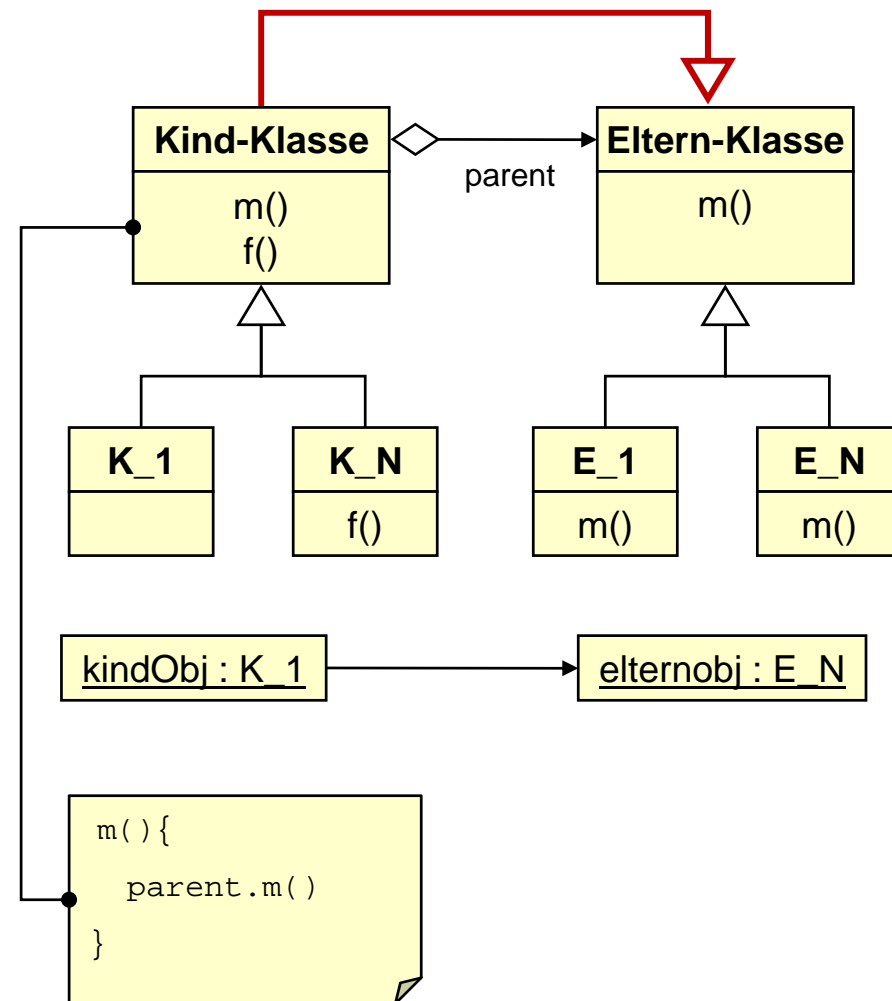
- Forwarding
 - ◆ Kind leitet empfangene Nachricht an Eltern-Objekt weiter
- Subtyping
 - ◆ Kind bietet volles Eltern-Interface
- Overriding
 - ◆ im Kontext weitergeleiteter Nachrichten werden Methoden des Kindobjekts benutzt
 - ◆ ... anstelle entsprechender Methoden des Elternobjekts

Zusammengesetzte Beziehungen

- Resending
 - ◆ forwarding
 - ◆ ... ohne subtyping
 - ◆ ... ohne overriding
- Consultation
 - ◆ forwarding
 - ◆ ... mit subtyping
 - ◆ ... ohne overriding
- Delegation („Objektvererbung“)
 - ◆ forwarding
 - ◆ ... mit subtyping
 - ◆ ... mit overriding

Implementierung der Subtypbeziehung: Variante 1

- Idee
 - ◆ Kind-Klasse ist Unterklasse
- Vorteil
 - ◆ allgemein anwendbar
- Nachteil
 - ◆ Kindklasse erbt auch die Variablen der Elternklasse
 - ◆ Duplizierung von Daten
 - ⇒ Speicherverschwendung
 - ⇒ Konsistenzprobleme



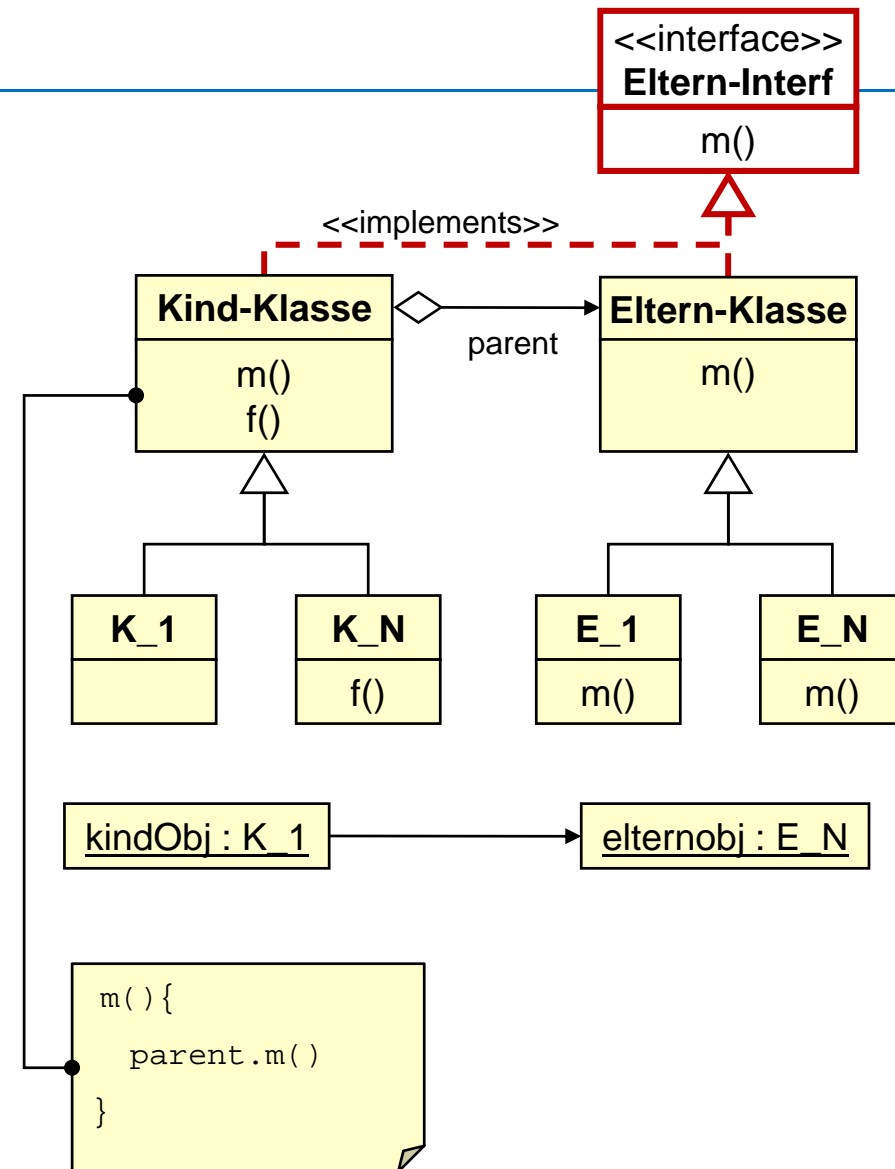
Implementierung der Subtypbeziehung: Variante 2

- Idee

- ◆ Kind implementiert gleiches Interface wie die Elternklasse
- ◆ Eltern-Interface anstatt Eltern-Klasse in Typdeklarationen verwenden

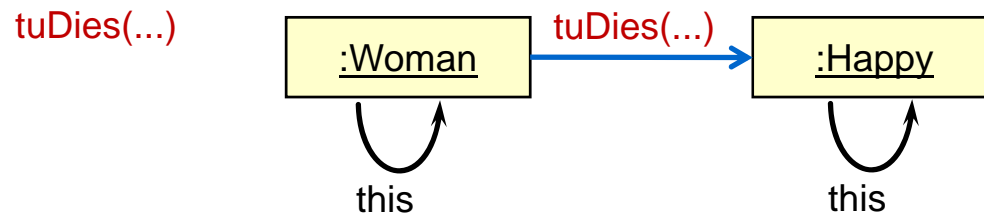
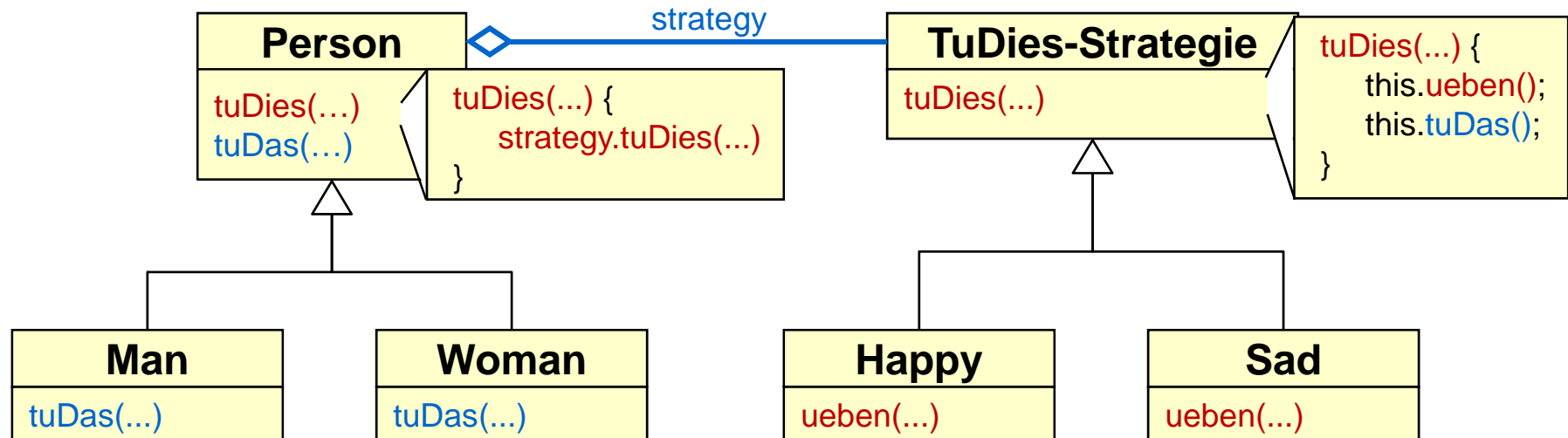
- Vorteil

- ◆ keine Datenduplizierung
- ◆ saubere Trennung
 - ⇒ Subtyping
 - ⇒ Code Wiederverwendung



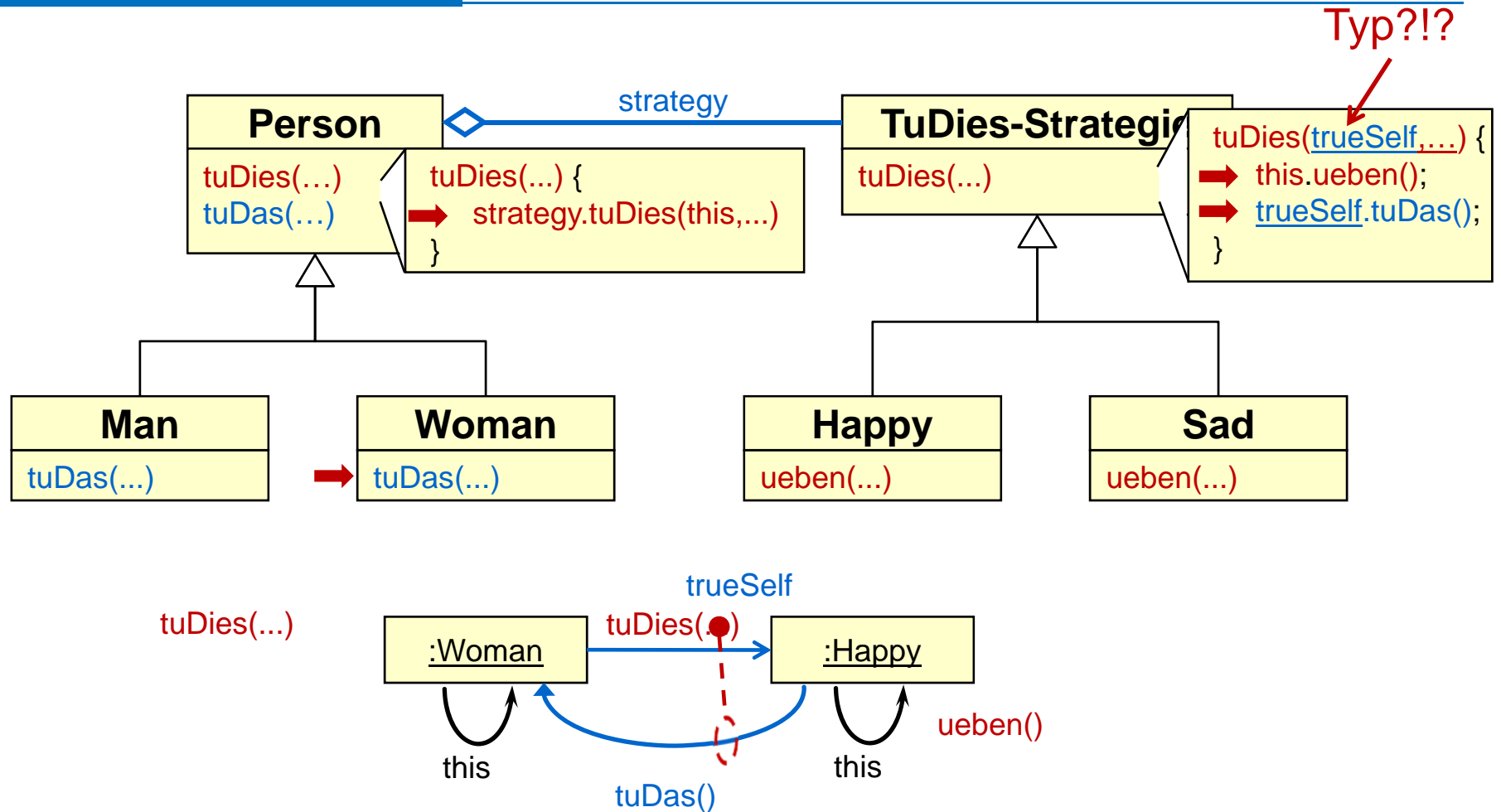
Wie modelliert man Overriding?

Strategy Pattern als Beispiel des Overriding-Problems



→ **“Schizophrene Objekte”**: verschiedenes “this” in **ursprünglicher Nachricht** und **weitergeleiteter Nachricht**, obwohl beide das gleiche konzeptionelle ein Objekt „fröhliche Frau“ ansprechen.

Overriding-Simulation: "this" explizit machen



Heureka!

Heureka?

Overriding-Simulation: Schwachpunkte

- Festlegung des Typs von **trueSelf**
 - ◆ “Strategy”-Klasse kann nur von “Personen” benutzt werden
 - ◆ eingeschränkte Wiederverwendbarkeit
- Festlegung welche Nachricht an **trueSelf** und welche an **this** geschickt wird
 - ◆ Festlegung was in Unterklassen und was in Kindklassen redefinierbar ist
 - ◆ eingeschränkte Wiederverwendbarkeit
- manuelle Weiterleitung von Anfragen
 - ◆ Fehleranfälligkeit
 - ◆ hoher Programmieraufwand
 - ◆ Erweiterung der Elternklassen erfordert Änderung aller Kindklassen
- Änderung der Schnittstelle der Elternklasse erforderlich
 - ◆ Zusätzlicher **trueSelf** Parameter erfordert Anpassung der Aufrufe in allen Clients der Elternklasse

Alternative: Overriding-Simulation mit Instanzvariable

- Idee
 - ◆ trueSelf dem Elternobjekt im Konstruktor übergeben
 - ◆ ... in Instanzvariable speichern
- Vorteil
 - ◆ Schnittstelle der Elternklasse bleibt unverändert
 - ◆ Keine Folgeänderungen in Clients der Elternklasse
- Nachteile
 - ◆ Nur anwendbar wenn jedes Elternobjekt nur ein Kindobjekt hat
 - ◆ Nicht anwendbar, wenn Nachrichten auch direkt an das Elternobjekt geschickt werden (statt via dem gespeicherten Kindobjekt)
 - ◆ Nicht rekursiv anwendbar
- Anwendbarkeit z.B. für
 - ◆ Simulation multipler Vererbung durch "split objects"

Split Objects: Zwischen-Fazit

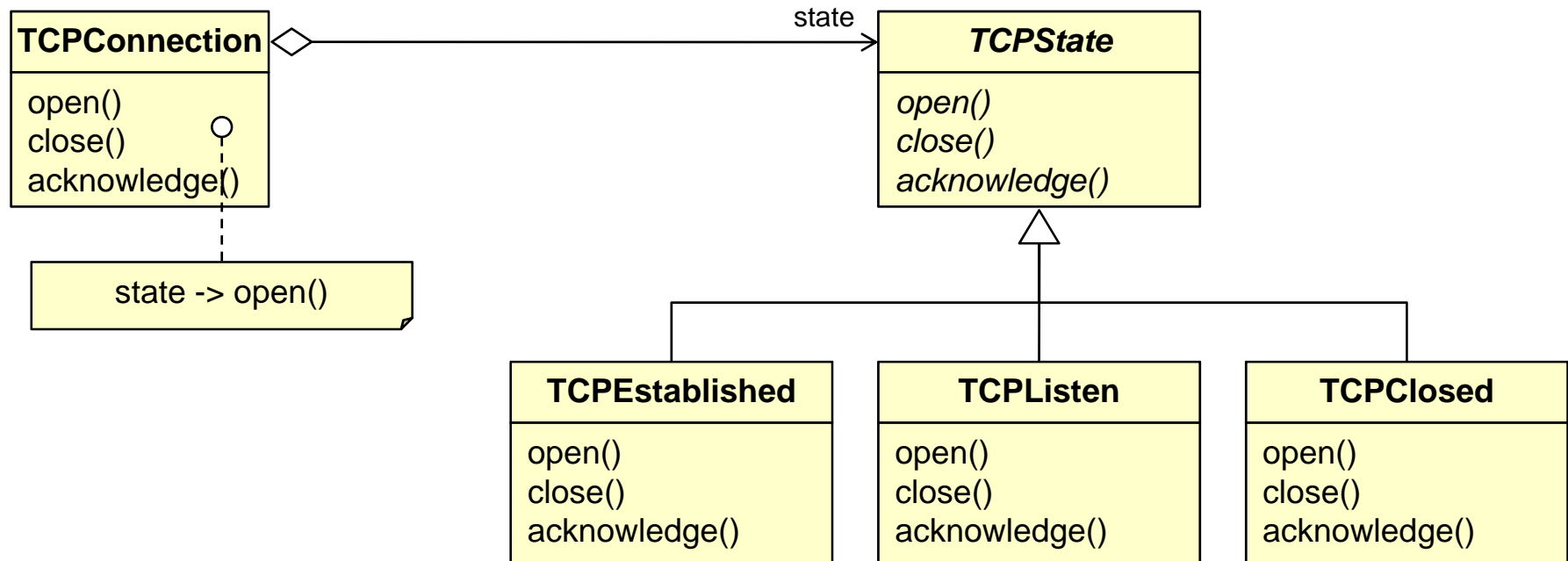
- Grundidee
 - ◆ Zerlegung in zwei Objekte
- Techniken
 - ◆ **Code-Wiederverwendung** mittels Forwarding
 - ◆ **Subtypbeziehung** mittels Interfaces
 - ◆ **Overriding** mittels explizitem "this" (als Parameter oder gespeichert)
 - ◆ Je anspruchsvoller die Beziehung zwischen split objects
 - ⇒ Resending
 - ⇒ Consultation
 - ⇒ Delegation
 - ◆ ... um so komplexer die Implementierung
- Diese Techniken bilden eine „Pattern Language“ zur Simulation von (multipler) Vererbung auf Objektebene

Auf "Split Objects"-Idee basierende Patterns

2. Das State Pattern

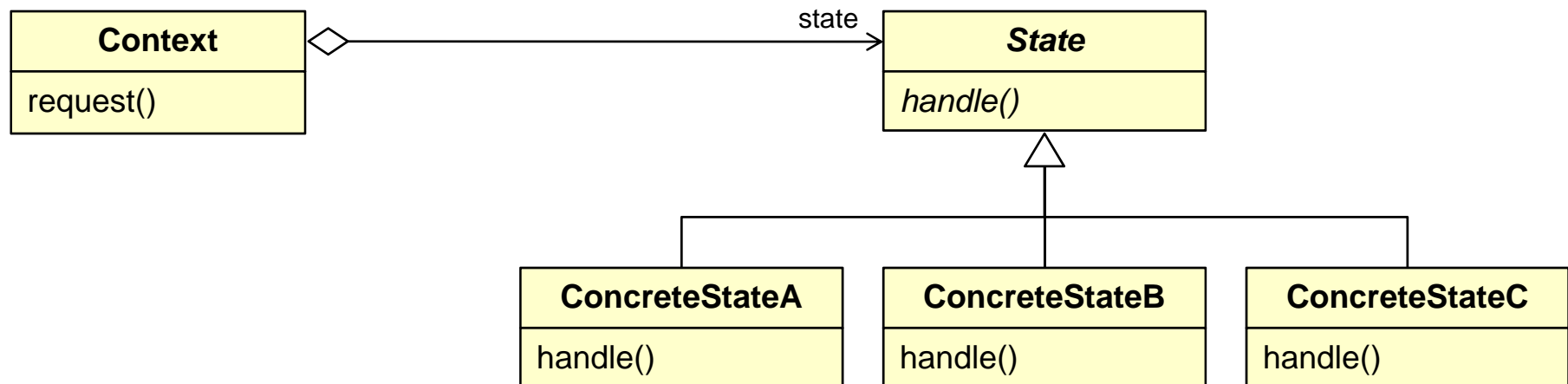
Das State Pattern

- Absicht
 - ◆ Ein Objekt soll sein Verhalten ändern können, wenn sein Zustand sich ändert.
- Motivation
 - ◆ Beispiel: Implementation von TCP/IP



Das State Pattern

- Anwendbarkeit
 - ◆ Das Verhalten eines Objekts hängt von seinem Zustand ab
 - ◆ viele Fallunterscheidungen, die zustandsabhängig ein Verhalten auswählen
- Struktur



Implementation des State Patterns

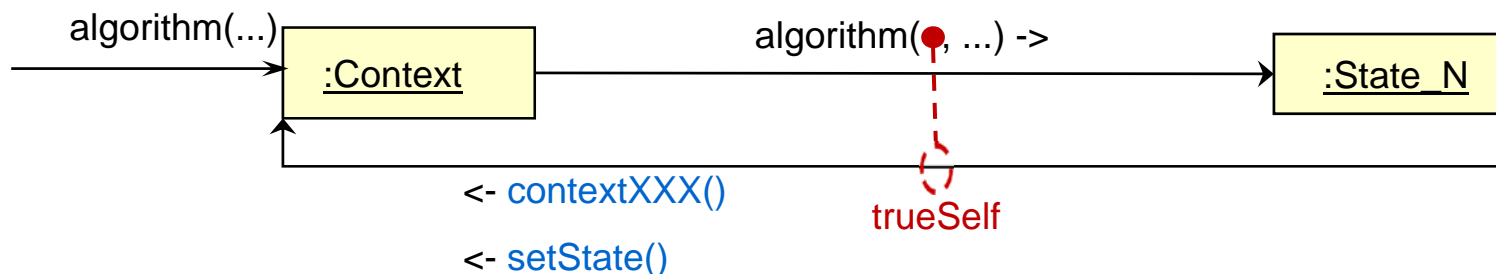
- Definition der Zustandsänderungen
 - ◆ Zustandsänderungen werden entweder im Kontext definiert
 - ◆ ... oder (flexibler) in den Zustandsobjekten
- Erzeugung von Zustandsobjekten
 - ◆ Zustandsobjekte werden entweder einmal für den gesamten Programmlauf erzeugt,
 - ◆ oder jedesmal bei Bedarf
- Verwendung von Delegation oder dynamischen Klassenänderungen
 - ◆ in Self und Lava kann das State Pattern direkt ausgedrückt werden

Das State Pattern: Konsequenzen

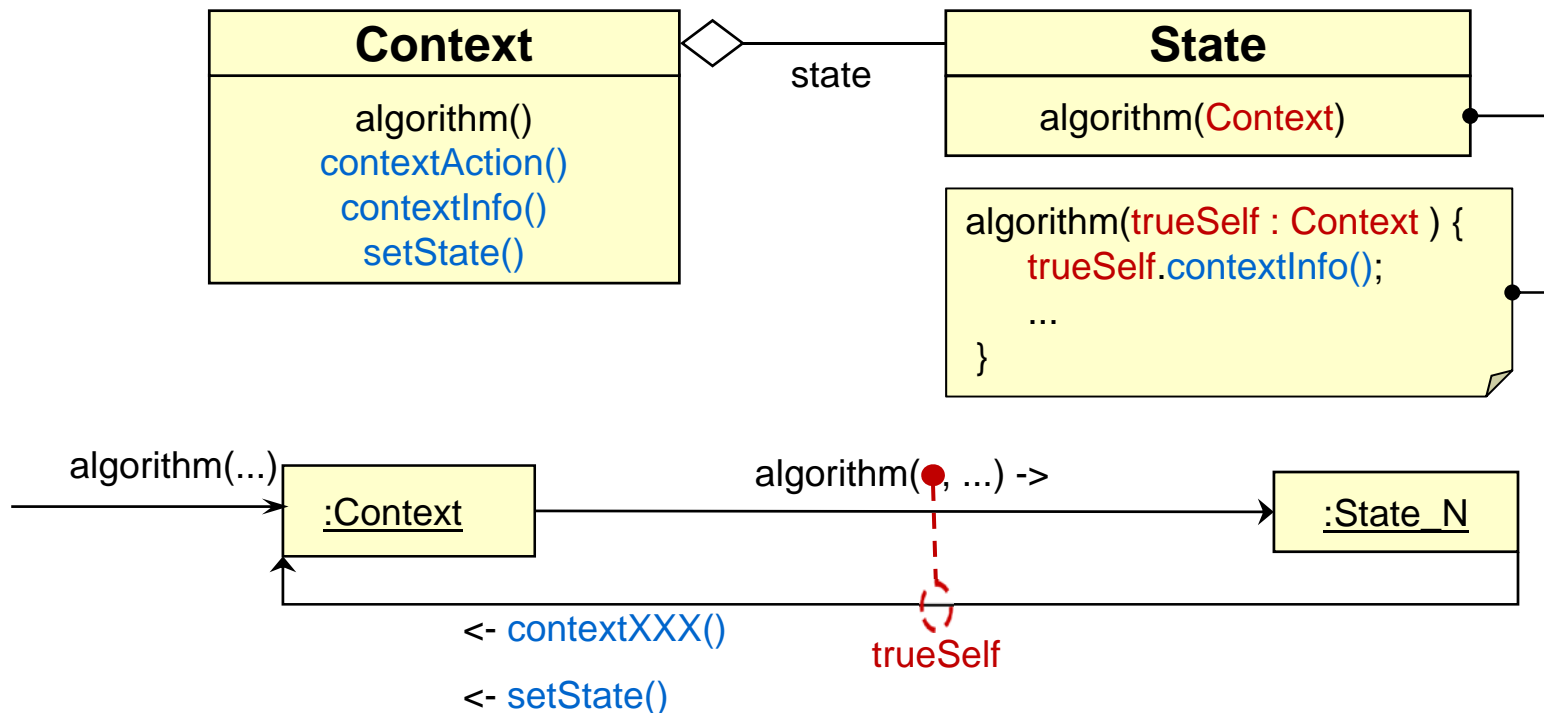
- Modularisierung
 - ◆ Zustandabhängiges Verhalten in eigene Klassenhierarchie ausgelagert
 - ◆ zusammengehörige Methoden werden nach Zuständen getrennt
- Explizitheit
 - ◆ Zustandsänderungen werden explizit gemacht
- Thread-Safety
 - ◆ Zustandsänderungen sind atomar (eine Zuweisung)
- Wiederverwendung
 - ◆ Zustandsobjekte können evtl. von verschiedenen Kontexten verwendet werden
- Erweiterbarkeit
 - ◆ neue Zustände erfordern keine / wenig Änderungen des Kontexts

Implementation des Strategy und State Patterns

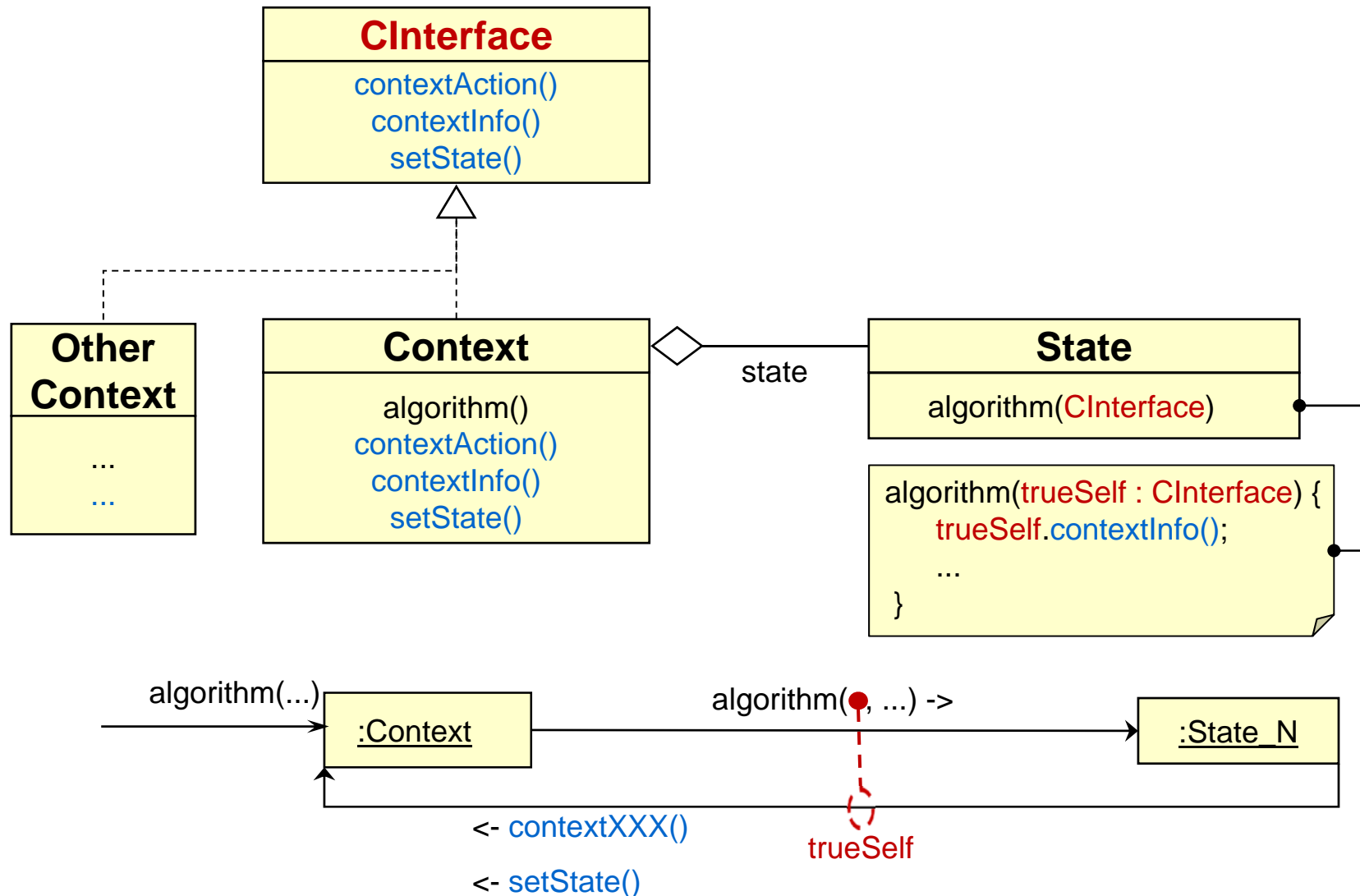
- Alternativen Schnittstellen zwischen Kontext und Strategien / States
 - ◆ Kontext übergibt alle relevanten Daten an die Strategie-Methode
 - ◆ Kontext übergibt this an Strategie-Methode
 - ◆ Strategie-Objekt speichert Referenz auf Kontext
- Die letzten beiden Varianten
 - ◆ sind flexibler
 - ◆ sind ähnlich der vorgestellten Simulation von Overriding
 - ◆ werfen ähnliche Fragen auf
 - ⇒ Typ des übergebenen / gespeicherten "this"
 - ⇒ Schnittstelle von Kontext und Strategie bzw. State



Simulation von Overriding: Typ von "trueSelf" = Context



Simulation von Overriding: Typ von "trueSelf" = CInterface



Vergleich

- trueSelf : Context
 - ◆ keine Verwendung der Elternhierarchie (States / Strategies / etc.) mit anderen Context-Typen möglich
 - ◆ das ist oft ausreichend
- trueSelf : ContextInterface
 - ◆ verschiedene Context-Typen die das ContextInterface implementieren sind möglich
 - ◆ flexibler
 - ◆ Umstellung Variante 1 → Variante 2 ist leicht:
 - ⇒ begrenzte Änderung:
 - ⇒ trueSelf ist nur in der Elternhierarchie und dem Context bekannt
 - ⇒ mechanische Änderung:
 - ⇒ suchen und ersetzen
 - ⇒ Korrektheit:
 - ⇒ Compiler meldet, falls Ersetzung "Context → ContextInterface" zu Typfehlern führt
 - ◆ "Erst einfache Lösung, ändern kann man immer noch." (siehe "Refactoring")

Abgrenzung Strategy / State: Änderung des Elternobjektes

- Bei Strategy-Pattern
 - ◆ meist extern veranlasst
 - ◆ Z.B. Anwendung bekommt mit, dass Verbindungsqualität schlecht ist, und weist den Media-Player an, einen schnelleren aber niedrig-qualitativen Video-Rendering-Algorithmus zu nutzen.
- Bei State-Pattern
 - ◆ meistens intern veranlasst
 - ◆ als Teil einer anderen Aktion, möglichst erst am Ende der Aktion!
 - ◆ Merke: State-Pattern modelliert oft einen Zustandsautomat. Daher ist klar, dass hier die Logik der Zustandsübergänge nicht extern bestimmt ist, sondern mit in den Zustands-Klassen modelliert wird
 - ⇒ „Wenn im Zustand ... das Ereignis ... auftritt und die Bedingung ... wahr ist, wird die Aktion ... durchgeführt und in den Zustand ... übergegangen.“
 - ⇒ Siehe auch „Event [Condition] / Action“ –Notation in Zustandsdiagrammen

Auf "Split Objects"-Idee basierende Patterns

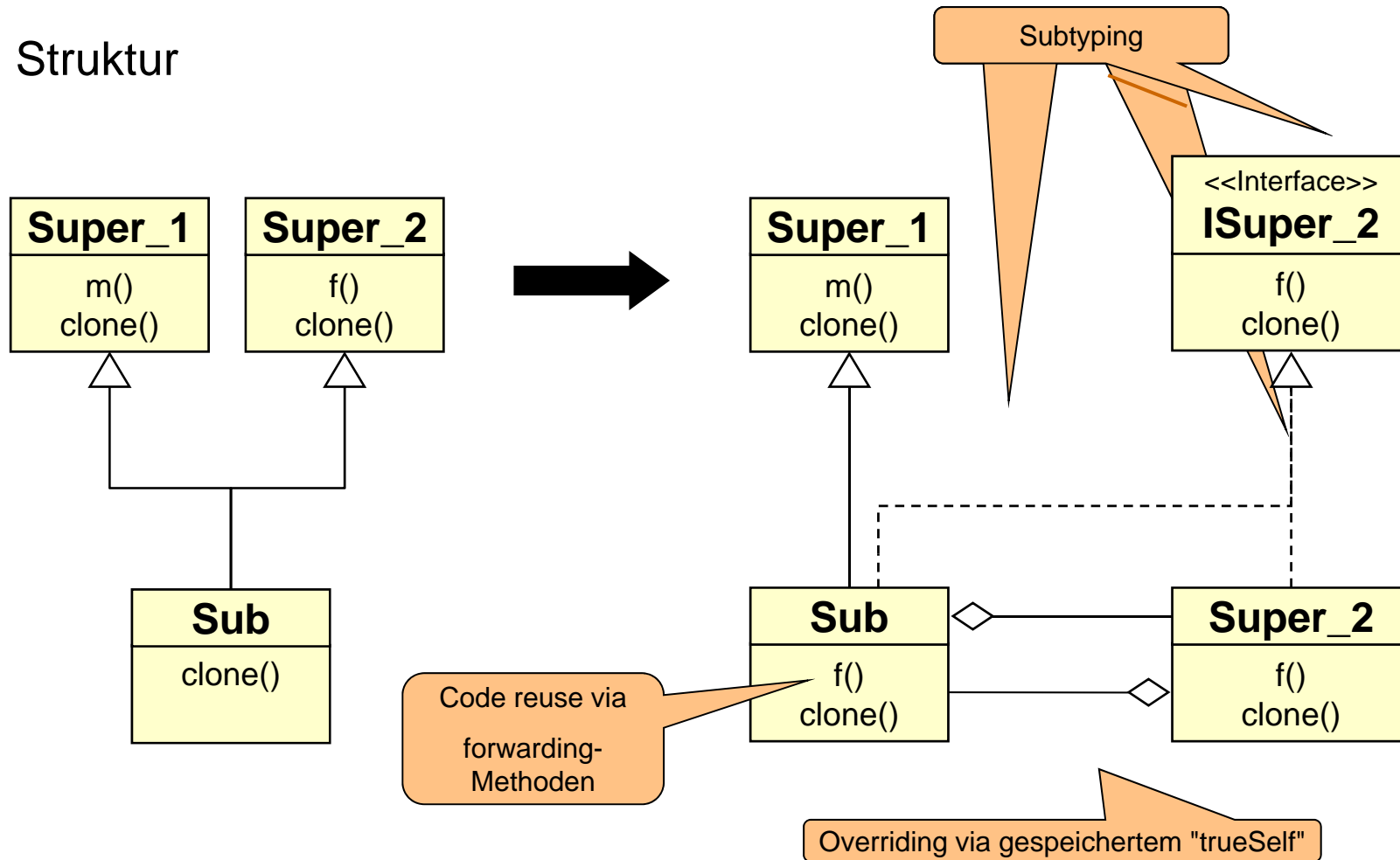
3. Simulation von Multipler Vererbung in Java

Multiple Vererbung in Java

- Absicht
 - ◆ Interface und Code mehrerer "Oberklassen" wiederverwenden
 - ◆ ... obwohl Java nur Einfachvererbung erlaubt
- Motivation
 - ◆ komplexe Klassen nicht reinimplementieren
- Anwendbarkeit
 - ◆ Interface und Code mehrerer "Oberklassen" wiederverwenden
 - ◆ keine semantischen Konflikte zwischen "Oberklassen"-Methoden

Multiple Vererbung in Java

- Struktur



Implementation

- Code-Reuse
 - ◆ Aggregation und Forwarding
 - ◆ `protected` Methoden der „Oberklasse“ müssen `public` deklariert werden
 - ◆ `protected` Variablen der „Oberklasse“ brauchen `public` Zugriffsmethoden damit sie aus der „Unterklasse“ aufrufbar sind.
- Subtyping
 - ◆ explizites "Oberklassen-Interface"
 - ◆ ... enthält alles was in der Simulation `public` ist
 - ◆ ... wird implementiert von "Oberklasse" und "Unterklasse"
- Overriding
 - ◆ gespeichertes "`trueSelf`„ (keine Schnittstellen-Änderung erforderlich)
 - ⇒ im Konstruktor übergeben
 - ◆ gespeichertes "`trueSelf`„ ist vom Typ "Oberklassen-Interface"
 - ⇒ verschiedene Unterklassen möglich

Implementation: Probleme

- Code-Reuse

- ◆ keine automatische Propagierung von Änderungen des Oberklassen-Interfaces
- ◆ Schutz von `protected`-Variablen und -Methoden aufgehoben

- Subtyping

- ◆ Ersetzbarkeit von "Unterklasse" und "Oberklasse" für "Oberklassen-Interface,,", aber nicht von "Unterklasse" für "Oberklasse"!

- ◆ globale Änderung von Typdeklarationen erforderlich:

```
⇒ Oberklasse expr;
```



```
⇒ OberklassenInterface expr;
```

- ◆ globale Änderung von Zugriffen auf `public`-Variablen der "Oberklasse" erforderlich:

```
⇒ Oberklasse expr;          expr.var;
```

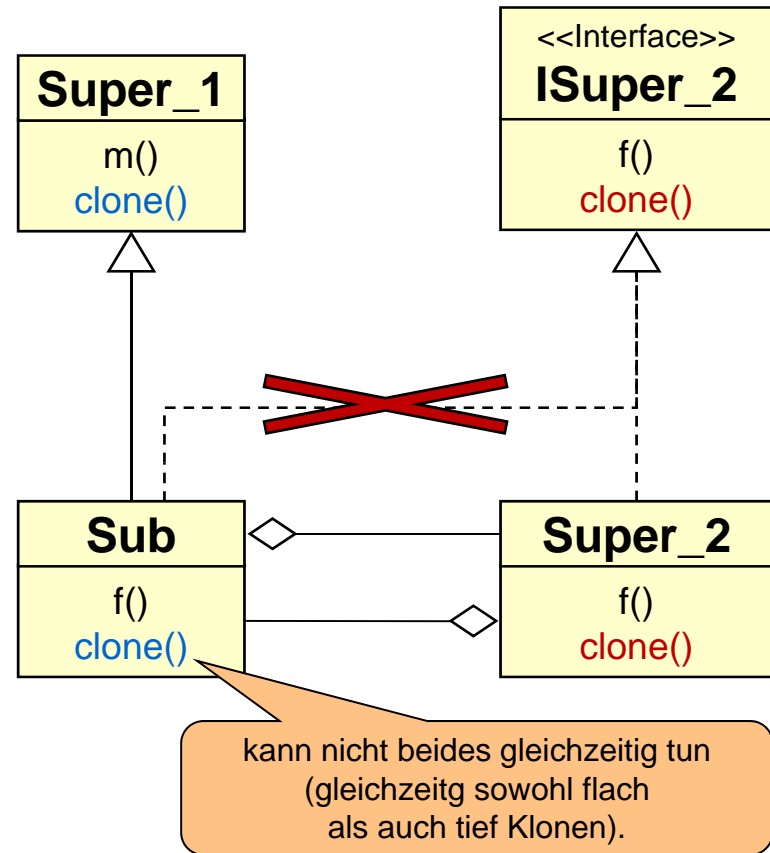
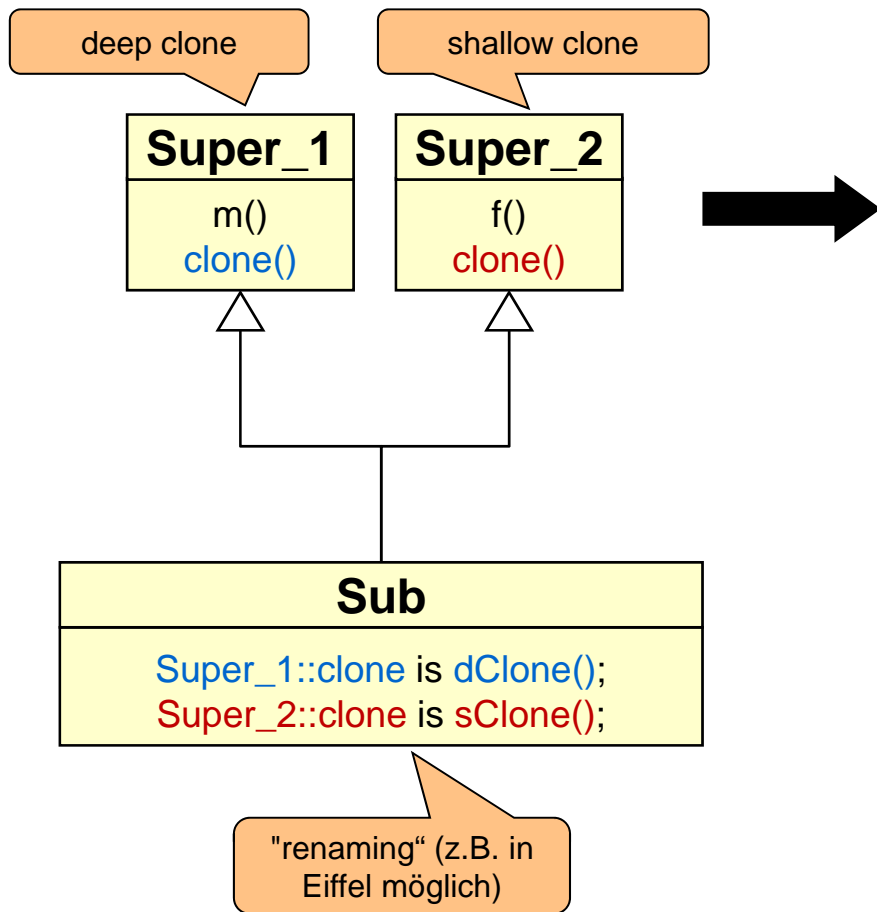


```
⇒ OberklassenInterface expr; expr.getVar();
```

Multiple Vererbung: Semantische Konflikte

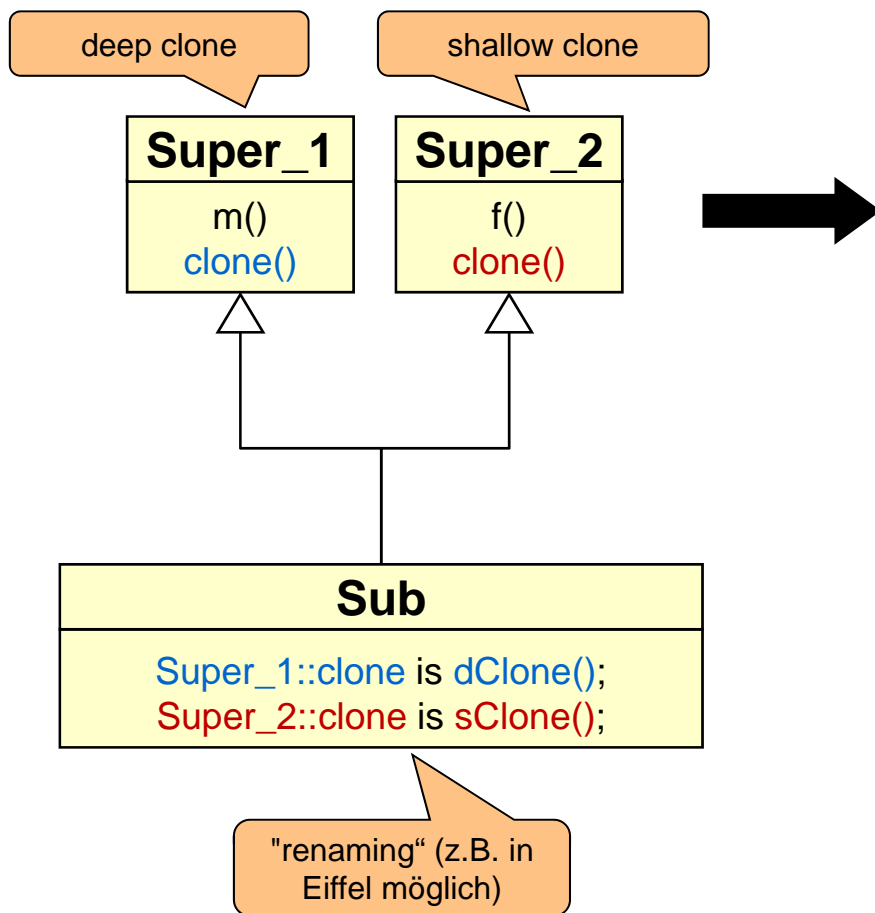
Eiffel: Lösung semantischer Konflikte durch renaming

Java: Semantische Konflikte verhindern Subtyping

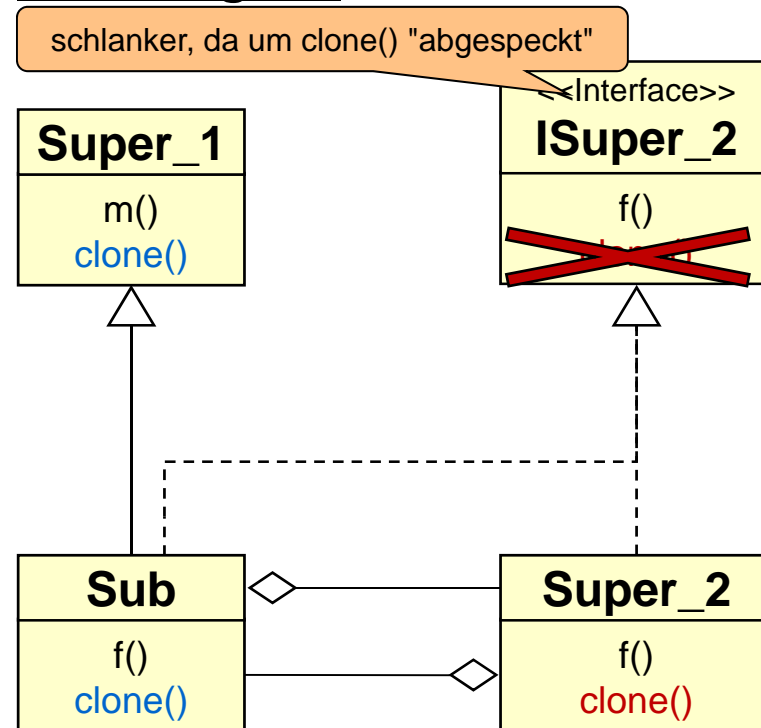


Multiple Vererbung: Semantische Konflikte

Eiffel: Lösung semantischer Konflikte durch renaming



Java: Semantische Konflikte durch "schlanke interfaces" vorbeugen!



Implementation: Fazit

- Simulation von „Multipler Vererbung“ ist am leichtesten bei neu entworfenen Programmen
- „Multiples Erben“ von bestehenden „Oberklassen“ ist
 - ◆ unmöglich, wenn nicht vorbereitet
 - ◆ aufwendig, wenn es Clients der "Oberklassen" gibt
- Vorausschauender Entwurf lohnt sich
 - ◆ keine public-Variablen
 - ◆ Interfaces statt Klassen in Typdeklarationen
 - ◆ "schlanke Interfaces"
 - ◆ trueSelf-Parameter / -Variable vorsehen

Beispiel: Klasse "Thread"

Background: Erzeugung von Threads durch Implementierung des Interface Runnable

- Thread-Definition

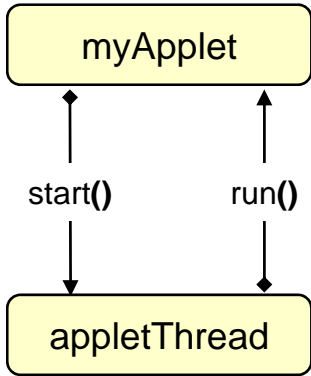
- ◆ eigene Klasse implementiert Methode run() des Interface Runnable

```
class MyApplet extends Applet implements Runnable {  
    Thread appletThread;  
    public void run() {  
        ...  
    }  
}
```

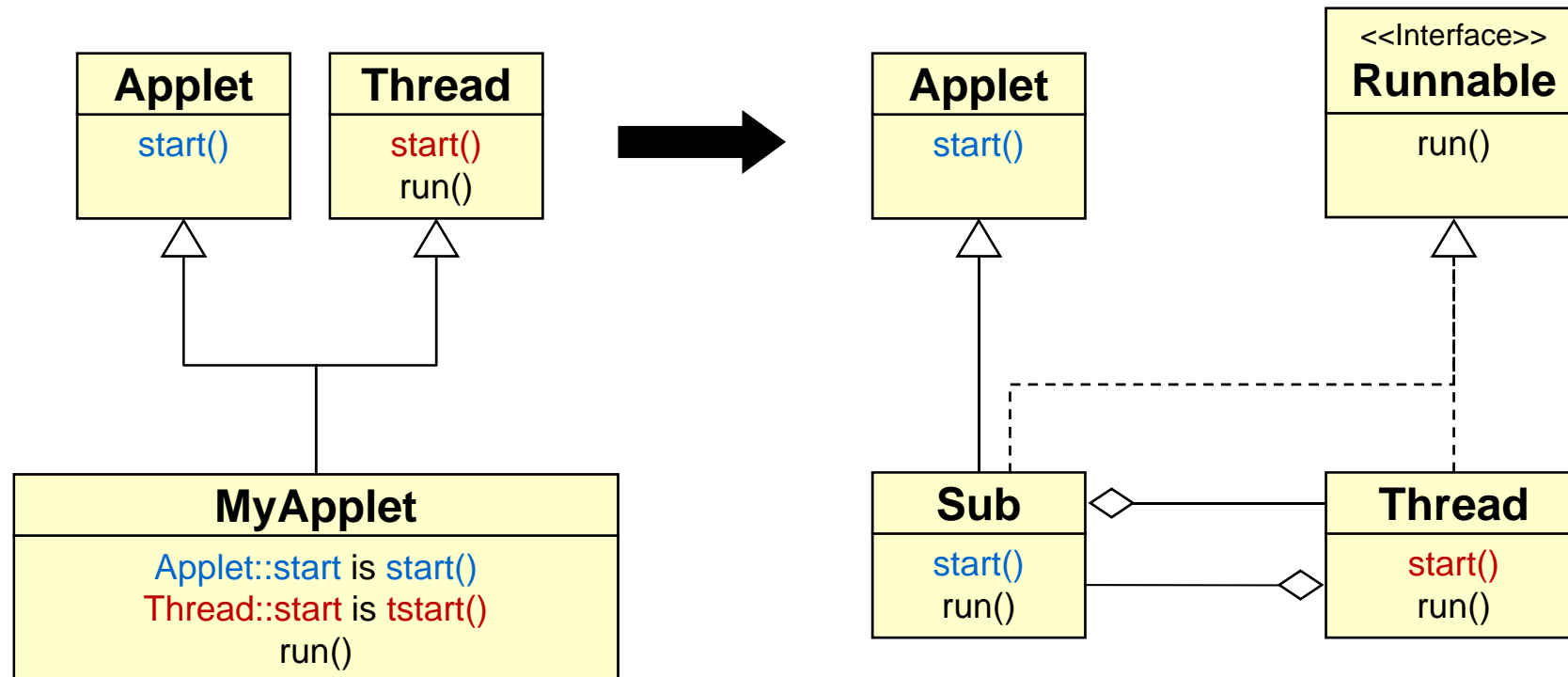
- kannThread-Erzeugung

- ◆ Übergeben einer Runnable-Instanz an Thread-Konstruktor
- ◆ ihre run() -Methode wird vom Thread aufgerufen
- ◆ von beliebiger Oberklasse ≠ Thread erben

```
public void start() {  
    appletThread = new Thread(this);  
    appletThread.start();  
}
```



Thread-Design als antizipierte multiple Vererbung mit schlankem Interface



Wann ist multiple Vererbung überflüssig?

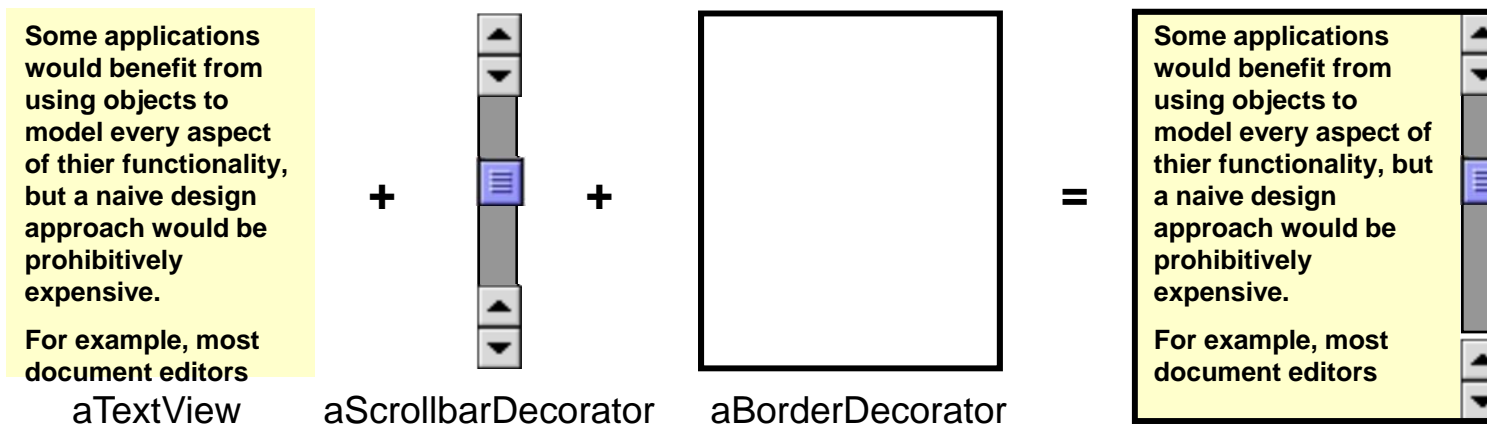
- Wenn nur multiples Subtyping erwünscht (keine Code-Vererbung)
 - ◆ lässt sich durch Interfaces erreichen
- Wenn kein Subtyping erwünscht
 - ◆ z.B. Stack erbt nicht das Listen-Interface
 - ◆ ... sondern benutzt eine Liste zur internen Implementierung
- Wenn Subtyping von einem schlankeren interface ausreicht
 - ◆ z.B. Thread

Auf "Split Objects"-Idee basierende Patterns

4. Das Decorator Pattern

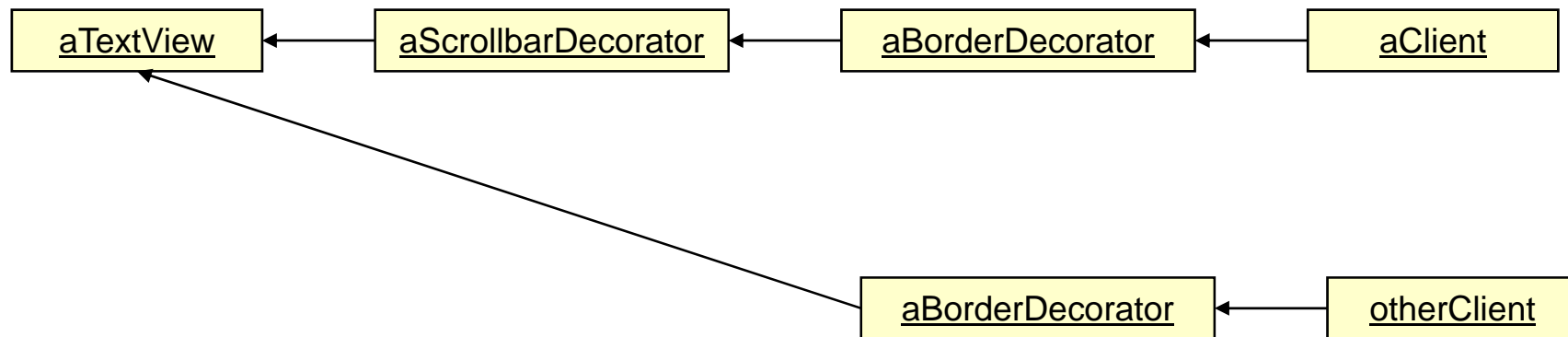
Das Decorator Pattern: Motivation

- Absicht
 - ◆ vorhandenen Objekten zusätzliche Fähigkeiten geben (laut Gamme & al)
 - ◆ Fähigkeiten vorhandener Objekte verändern
- Motivation
 - ◆ objekt-spezifische Eigenschaften
 - ◆ kontext-spezifische Eigenschaften
 - ◆ modulare / unvorhergesehene Erweiterung
 - ◆ Beispiel: GUI-Elemente
 - ⇒ Scrollbars, Rahmen, etc. nur bei Bedarf zu TextView hinzufügen



Das Decorator Pattern: Idee

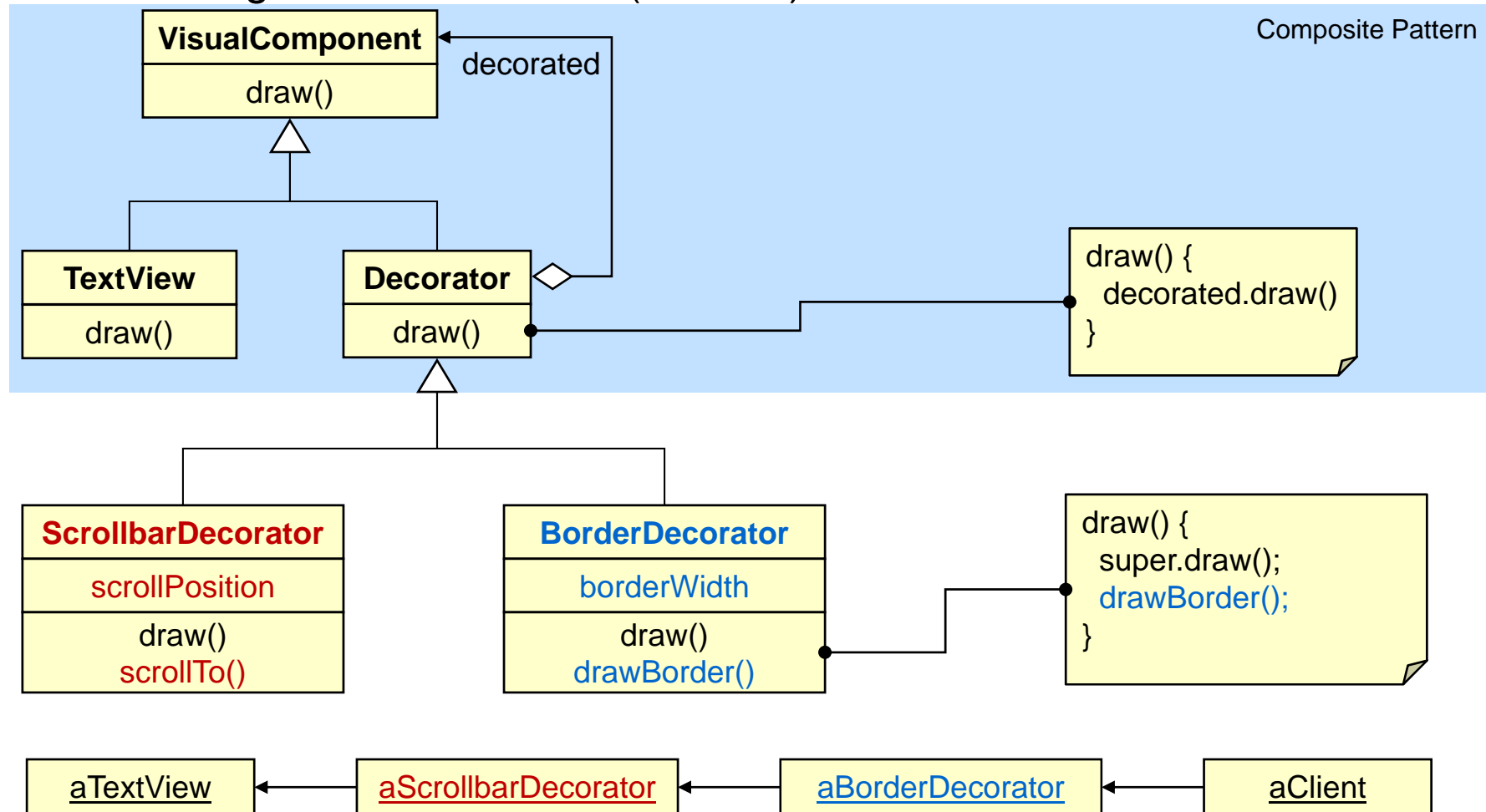
- **Veränderte oder zusätzliche Fähigkeiten**
 - ◆ ... sind zusätzliche Objekte
 - ◆ ... mit erweiterter Schnittstelle
 - ◆ ... zwischen ursprünglichem Objekt und Client



- **Context-spezifische Sicht**
 - ◆ ... entsteht durch Zugriff über verschiedene Decorators des gleichen Objekts

Das Decorator Pattern: Klassenhierarchie

- Vorschlag aus Gamma & al (für C++)



Das Decorator Pattern: Anwendbarkeit

- objekt-spezifische Eigenschaften
- kontext-spezifische Eigenschaften

- vorhandenen Objekten zusätzliche Fähigkeiten geben (laut Gamme & al)
 - ◆ wie gesehen
- vorhandene Fähigkeiten verändern (schwieriger)
 - ◆ zusätzlich objektspezifisches Overriding realisieren

- modulare / unvorhergesehene Erweiterung
 - ◆ keine Änderung des "Hauptobjekts" erforderlich

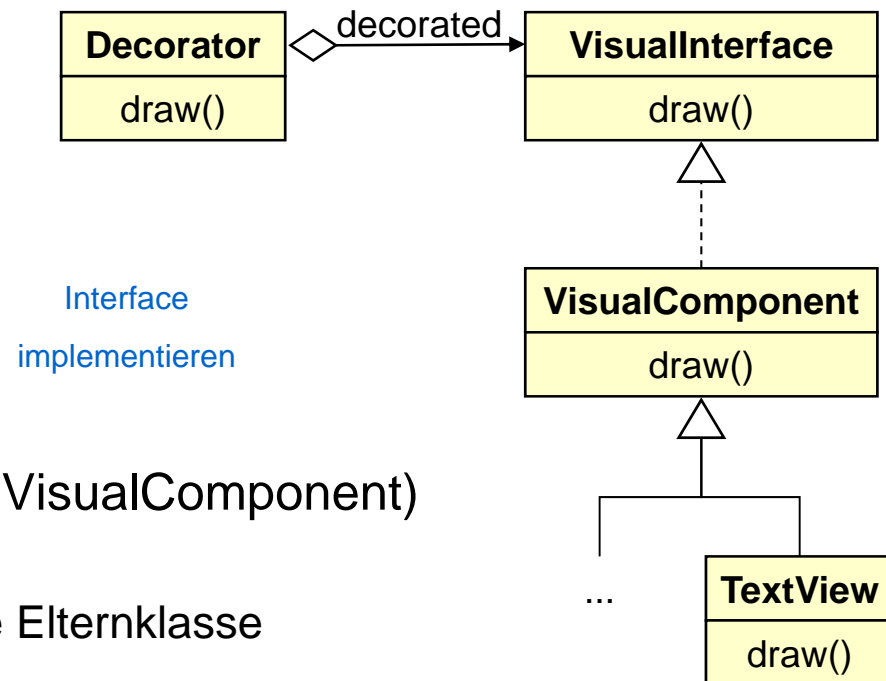
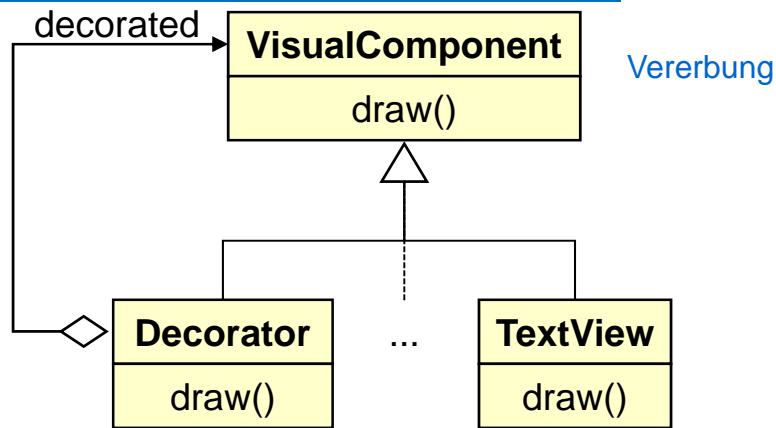
- wenn Vererbung nicht anwendbar ist

- wenn die Identität des "Hauptobjekts" nicht wichtig ist
 - ◆ siehe nächste Folie

Das Decorator Pattern: Konsequenzen

- Dynamik
 - ◆ unvorhergesehene nachträgliche Erweiterung
 - ◆ antizipierte nachträgliche Verhaltensänderung
- Kleine Klassen
 - ◆ Funktionalität wird incrementell hinzugefügt
 - ◆ ... wenn man sie braucht!
 - ◆ Kombinationen entstehen dynamisch statt statisch durch Vererbung
- Verschiedene Objektidentität
 - ◆ Identitäts-Tests (==) durch equals-Methode ersetzen
- Viele Objekte
 - ◆ leicht zu konfigurieren / anpassen
 - ◆ Gesamtverhalten evtl. schwieriger zu durchschauen

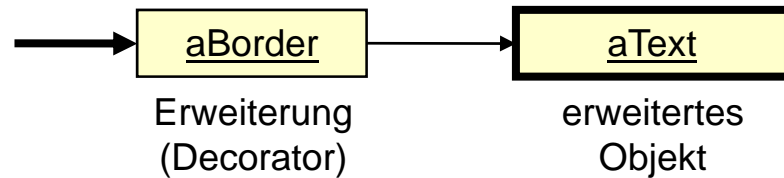
Das Decorator Pattern: Implementation



- Erweiterung der Eltern-Schnittstelle (VisualComponent)
 - ◆ von Elternklasse erben oder ...
 - ◆ gleiches Interface implementieren wie Elternklasse
- Eltern-Klasse (VisualComponent)
 - ◆ "schlankes" Interface
 - ⇒ nur was wirklich für alle Unterklassen gilt
 - ◆ keine (wenig) Variablen
 - ⇒ sonst werden Decorators mit irrelevantem Zustand überfrachtet (via Vererbung)
- Abstrakte Decorator-Klasse (Decorator)
 - ◆ kann weggelassen werden, wenn es nur einen konkreten Dekorator gibt

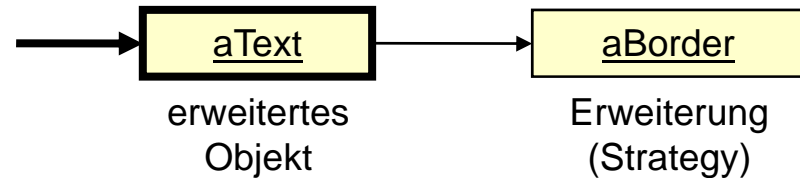
Decorator versus Strategy

Decorator



- Identität aus Client-Sicht
 - ◆ konzeptuell: Eltern-Objekt
 - ◆ real: Kind-Objekt
- Erweiterung
 - ◆ Kind-Objekt erweitert Eltern-Objekt
 - ◆ erweiterte Klasse unverändert
- Typ-Konformität
 - ◆ Erweiterung muss Subtyp sein

Strategy

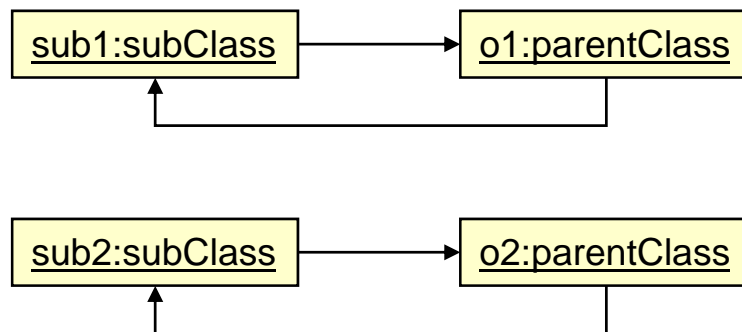


- Identität aus Client-Sicht
 - ◆ konzeptuell: Kind-Objekt
 - ◆ real: Kind-Objekt
- Erweiterung
 - ◆ Eltern-Objekt erweitert Kind-Objekt
 - ◆ erweiterte Klasse verändert
- Typ-Konformität
 - ◆ Erweiterung hat beliebigen Typ

Simulation multipler Vererbung versus Decorator

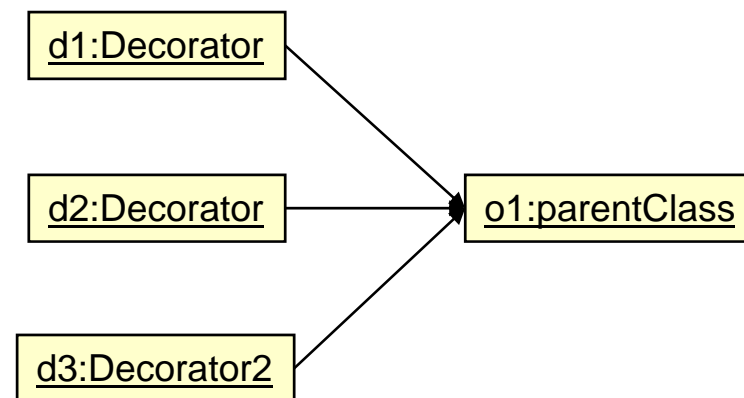
Simulation Multipler Vererbung

- statisch
 - ◆ Kindobjekt hat nach Initialisierung immer gleiches Elternobjekt
- „unshared“
 - ◆ jedes Elternobjekt hat ein einziges Kindobjekt



Decorator Pattern

- statisch
 - ◆ Kindobjekt hat nach Initialisierung immer gleiches Elternobjekt
- „shared“
 - ◆ Kindobjekte teilen sich oft ein Elternobjekt



Erweiterung und Restrukturierung

- ▶ **Umsetzung fortgeschrittener UML-Konzepte in „Kern-UML“**
- ▶ **Assoziationen**

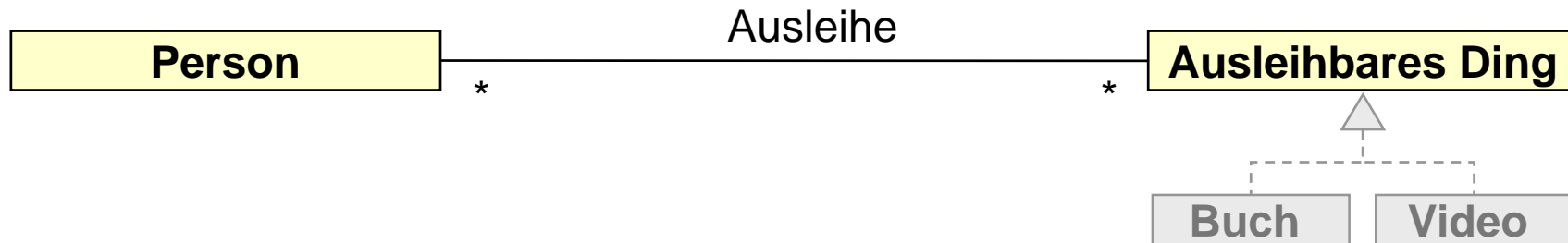
Assoziationen und Assoziationsklassen

Umsetzung von Assoziationen je nach ihrer Multiplizität und Navigierbarkeit

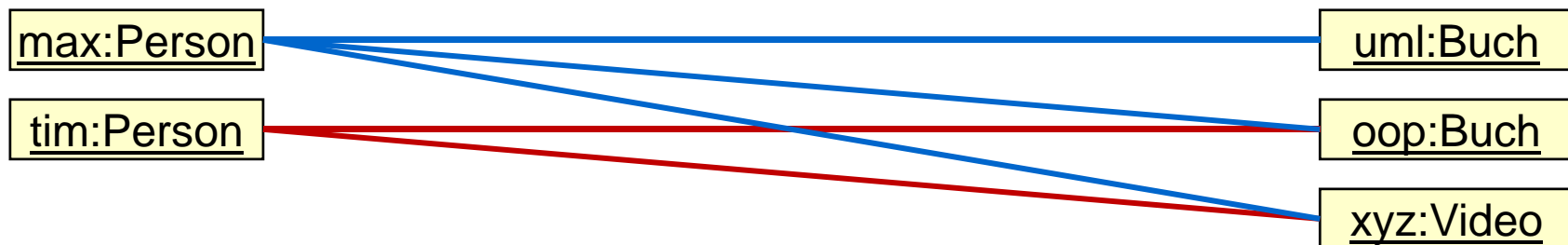
Umsetzung qualifizierter Assoziationen

Assoziationen sind implizite Klassen!

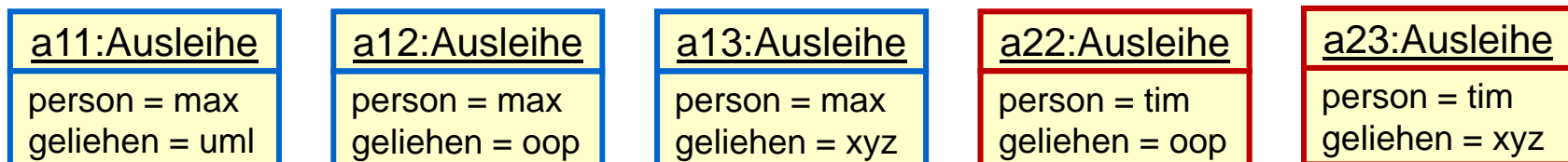
- Eine Assoziation



- ... ihre Elemente

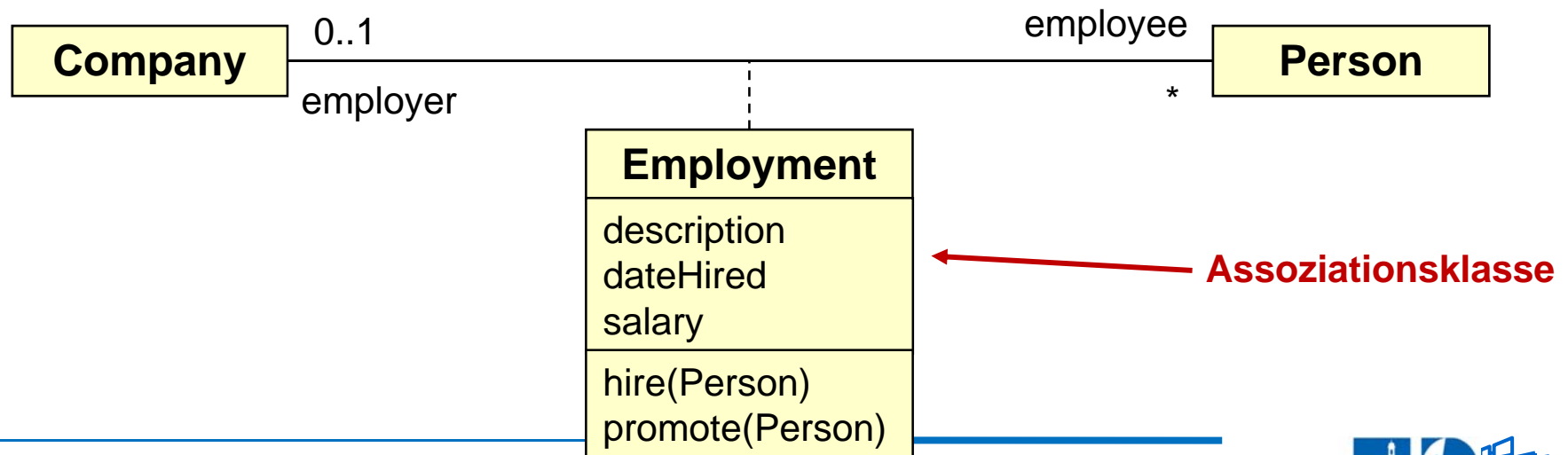


- ... können aufgefasst werden als Instanzen einer Klasse "Ausleihe"



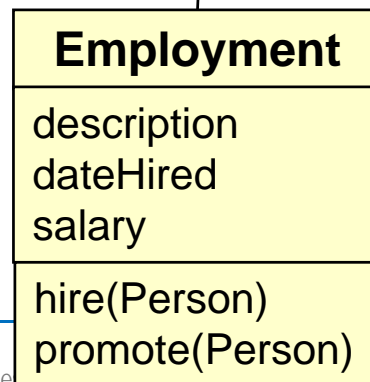
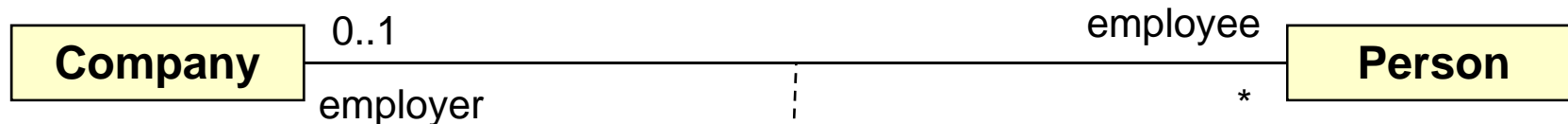
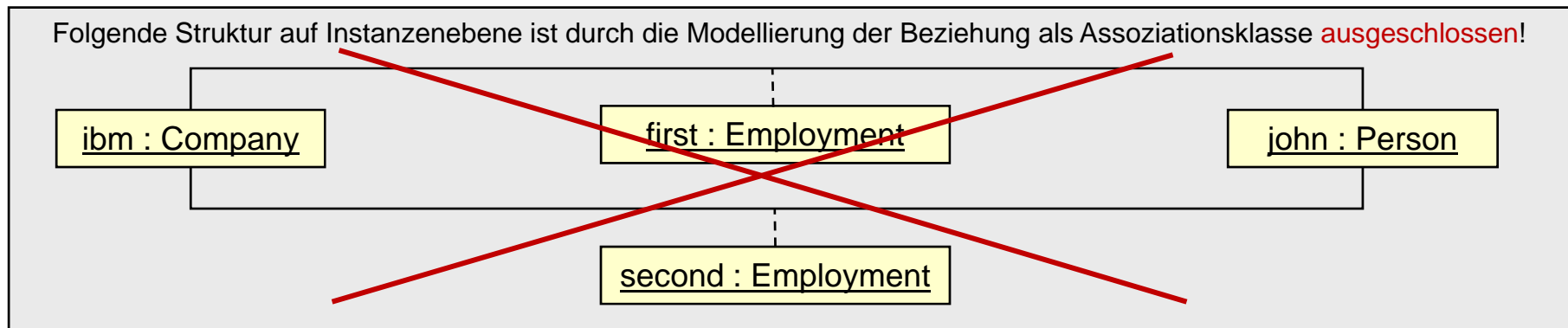
UML, statisches Modell: Assoziationsklassen (1)

- Modellieren komplexe Assoziationen zwischen Klassen
 - ◆ Machen die implizite Klasse explizit wenn zusätzliche Attribute und Operationen der Beziehung modelliert werden müssen.
- Beispiel
 - ◆ Das Datum der Einstellung (Attribut "dateHired")
 - ◆ Der Vorgang der Einstellung (Operation "hire")
 - ... sind Eigenschaften der Employment-Beziehung



UML, statisches Modell: Assoziationsklassen-Constraint

- Es darf nur eine einzige Ausprägung der Beziehung zwischen zwei Objekten geben!
 - ◆ Z.B.: Eine Person dürfte nicht mehrere Anstellungen bei der gleichen Firma haben

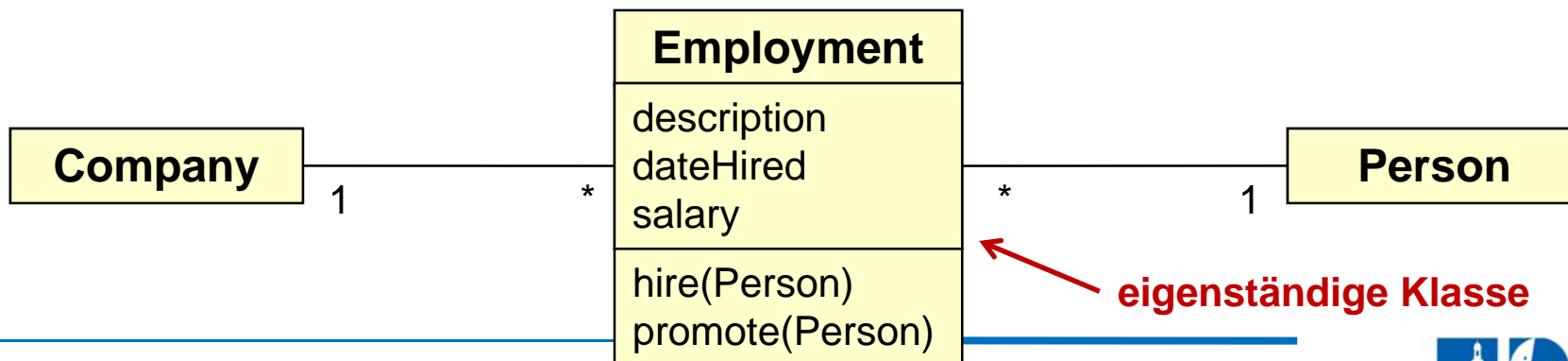
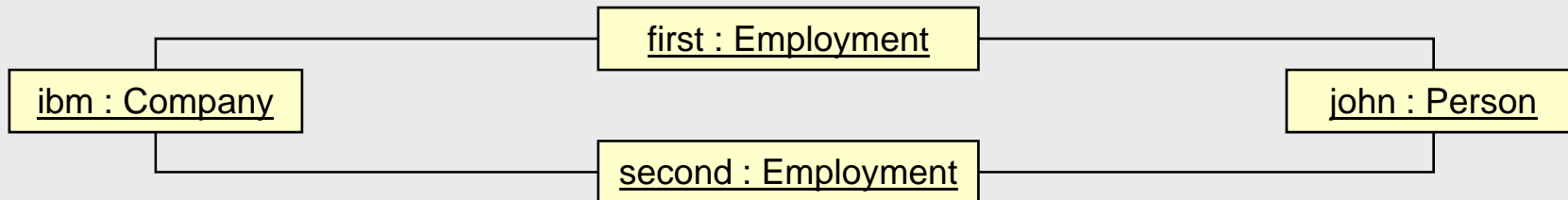


← **Assoziationsklasse**

UML: Assoziation als eigenständige Klasse

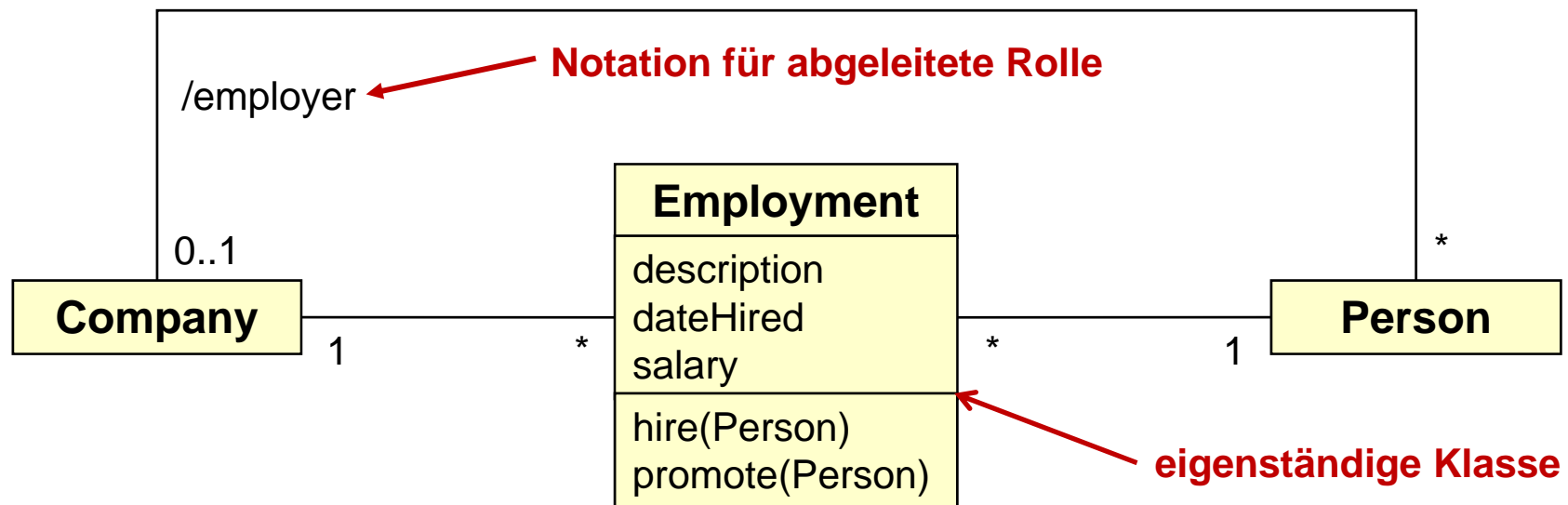
- Alternative Modellierung: eigenständige Klasse statt Assoziationsklasse
 - ◆ Es kann nun beliebig viele Employment-Instanzen zwischen einer Person-Instanz und einer Company-Instanz geben
 - ◆ Mit anderen Worten: Jede Person kann beliebig oft in der gleichen Firma arbeiten

Folgende Struktur auf Instanzenebene ist durch die Modellierung der Beziehung als eigenständige Klasse **möglich!**



UML: Assoziation als eigenständige Klasse (2)

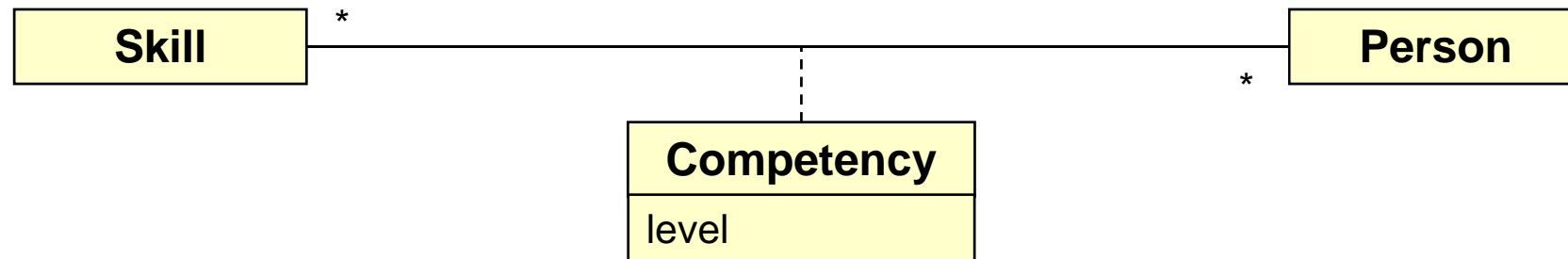
- Alternative Modellierung: eigenständige Klasse statt Assoziationsklasse
- Neben-Effekt
 - ◆ "employer"-Rolle kann nun abgeleitet werden
 - ◆ ... müsste eigentlich nicht mehr explizit angegeben werden
 - ◆ ... außer zur Festlegung des Rollen-Namens "employer"



UML: Vergleich der Alternativen

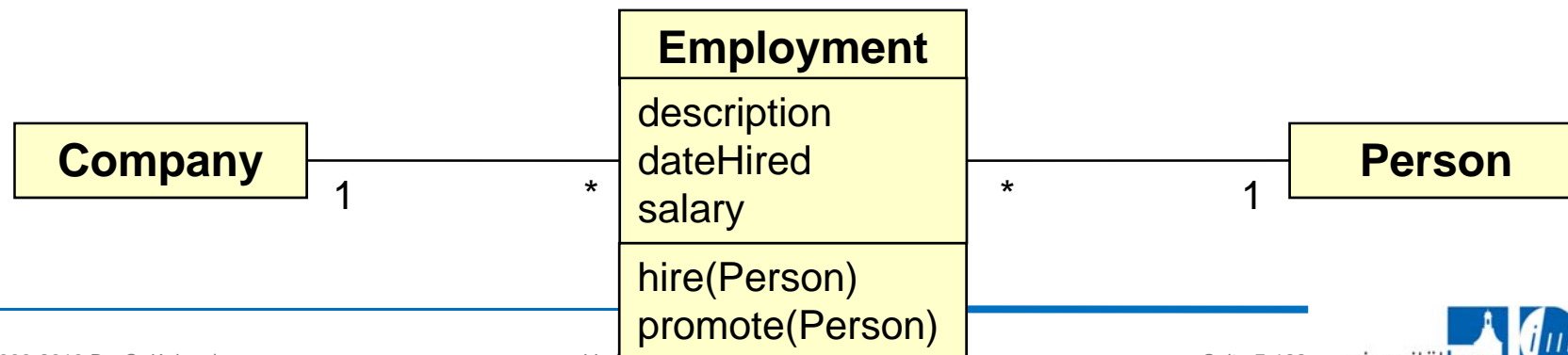
- Fallbeispiel 1:

- ◆ Jede Person hat pro Fähigkeit nur **eine** Kompetenz
- ➔ “Competency” als **Assoziationsklasse**



- Fallbeispiel 2:

- ◆ Eine Person hat in verschiedenen Arbeitsperioden **unterschiedliche** Jobs.
- ➔ “Employment” als **eigenständige** Klasse



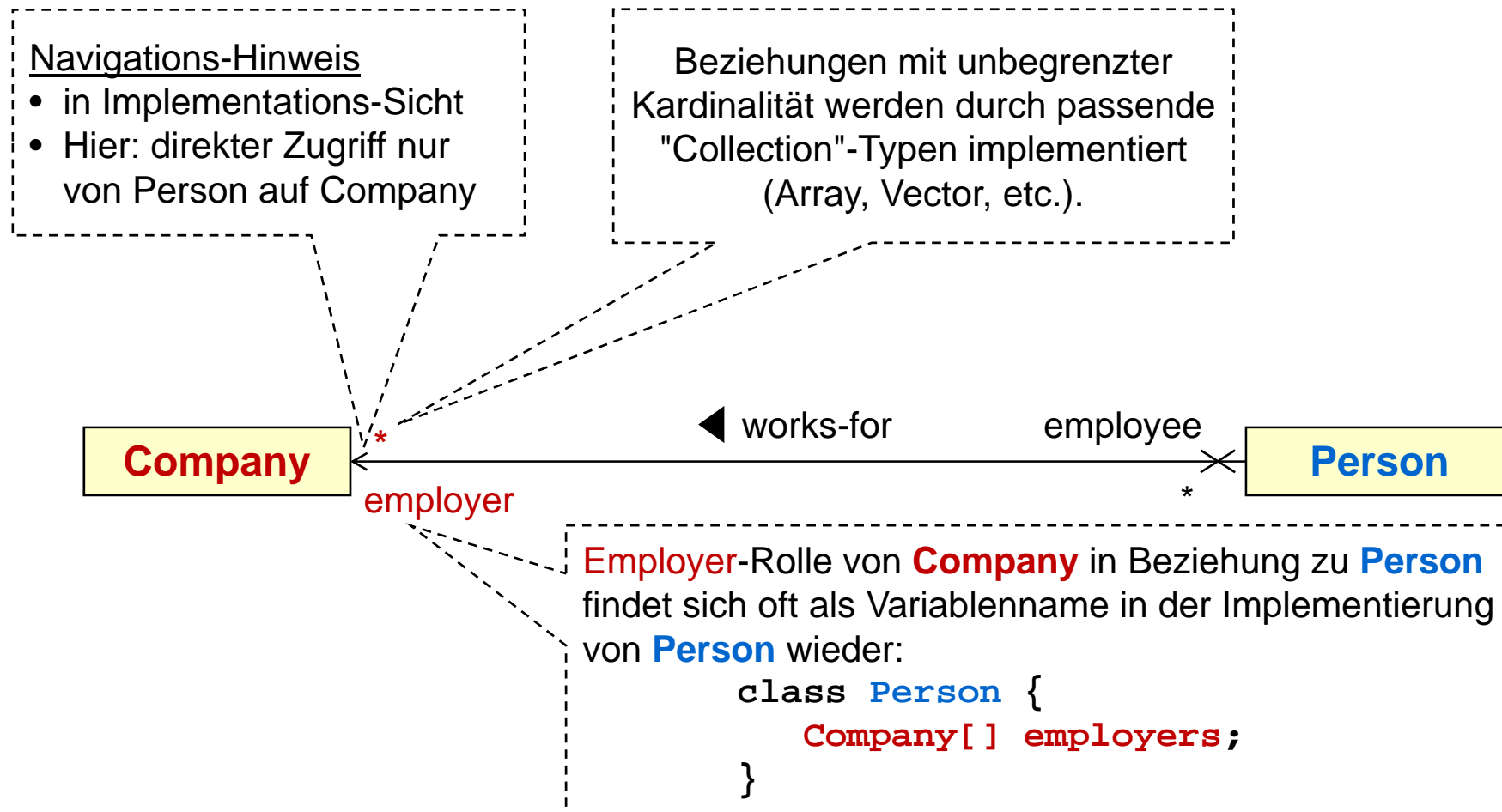
Implementierung von Assoziationen

1:1, unidirektional oder bidirektional

1:N, unidirektional oder bidirektional

N:M bidirektional

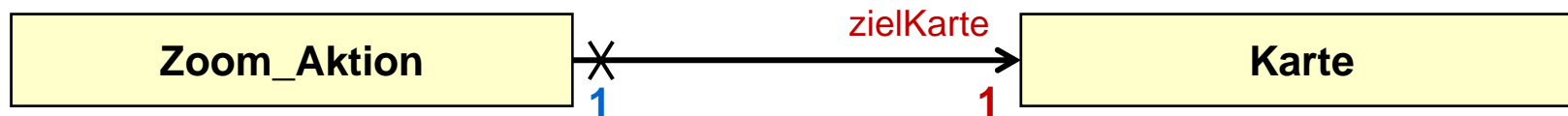
UML: Assoziationen (Implementations-Sicht)



Varianten je nach Navigierbarkeit (uni-/bidirektional) und Kardinalität (1/*)
→ s. nächste Folien.

Unidirektionale Assoziation (1:1, N:1, 1:N)

Objektentwurfsmodell vor der Transformation



Objektentwurfsmodell nach der Transformation



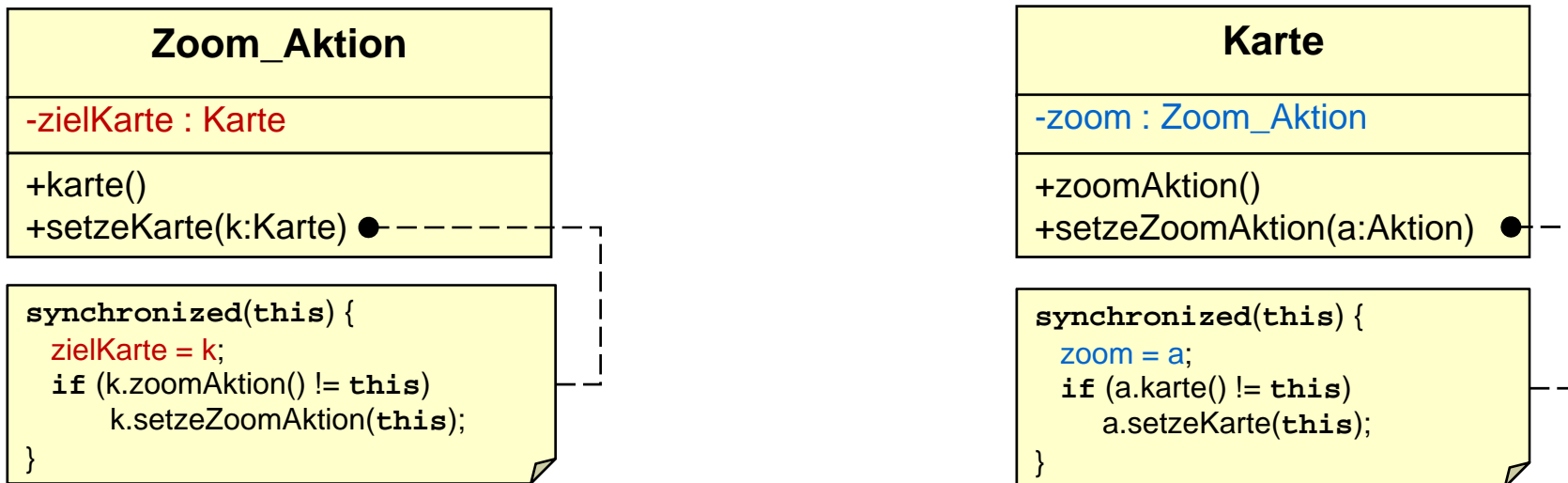
- Unidirektionale Assoziationen sind einfach:
 - ◆ Assoziation durch Instanzvariable in der „referenzierenden“ Klasse ersetzen.
 - ◆ Falls die Kardinalität auf der „referenzierten“ Seite $N > 1$ ist, sollte die Instanzvariable eine Collection sein.
 - ◆ Falls die Kardinalität auf der „referenzierten“ Seite 1 ist braucht man keine Collection (unabhängig von der Kardinalität der „referenzierenden“ Seite!)

Bidirektionale 1:1 Assoziation

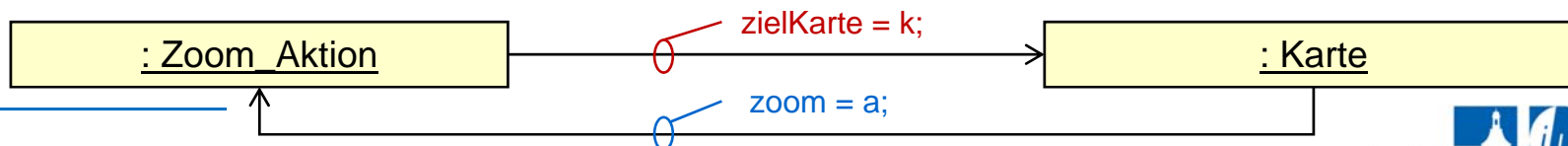
Objektentwurfsmodell vor der Transformation



Objektentwurfsmodell nach der Transformation



Beispiel-Instanziierung nach der Transformation



Bidirektionale 1:N Assoziation

Objektentwurfsmodell vor der Transformation



Objektentwurfsmodell nach der Transformation



Beispiel-Instanziierung nach der Transformation



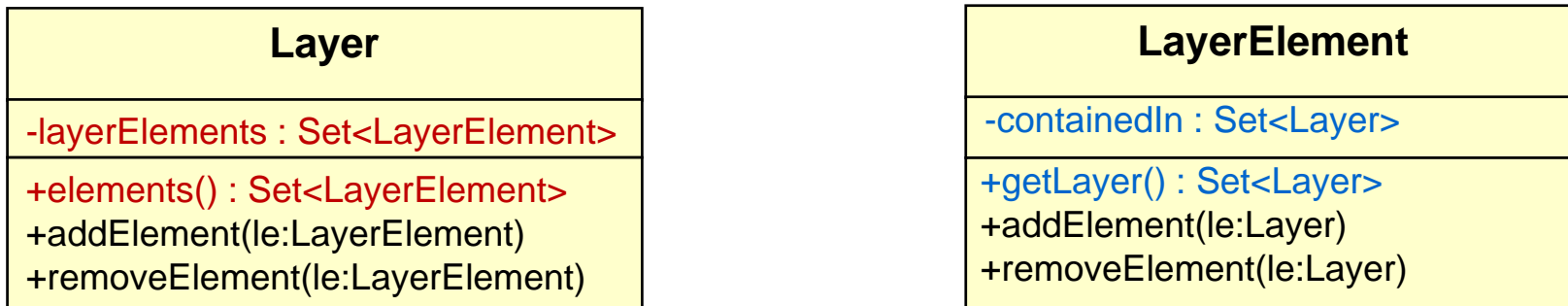
- Zuweisung der Werte für **layerElements** und **containedIn** wie bei bidirektionaler 1:1 Assoziation

Bidirektionale N:M Assoziation (Naive Variante)

Objektentwurfsmodell vor der Transformation



Objektentwurfsmodell nach der Transformation



Beispiel-Instanziierung nach der Transformation



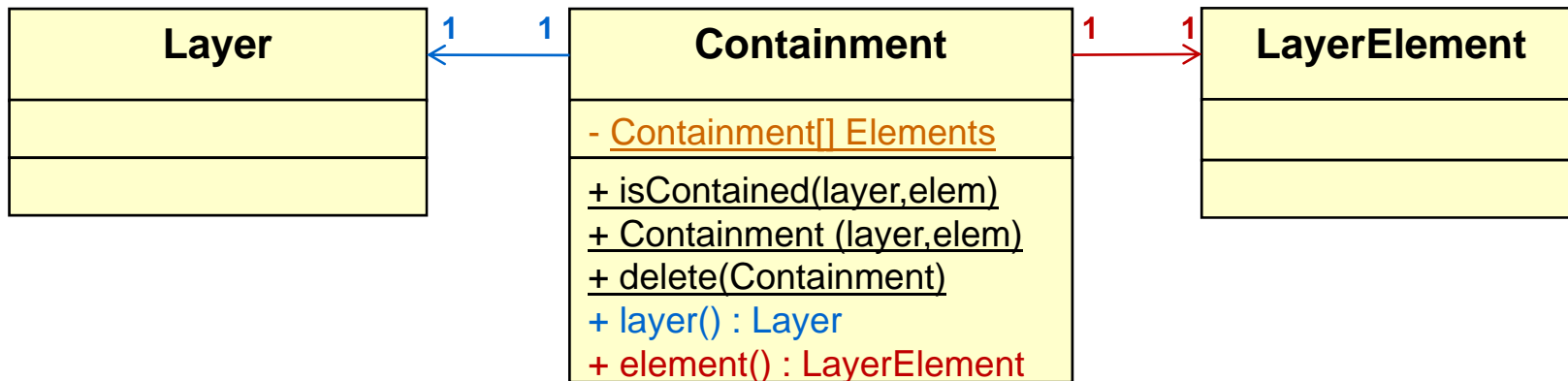
- Problem 1: Deadlockfreies setzen von Rückreferenzen nicht mehr trivial.
- Problem 2 (aller bisherigen Varianten): Die Beziehung wird fest in den Klassen verdrahtet. Das schafft zusätzliche Abhängigkeiten.

Bidirektionale N:M Assoziation (Assoziation als Klasse)

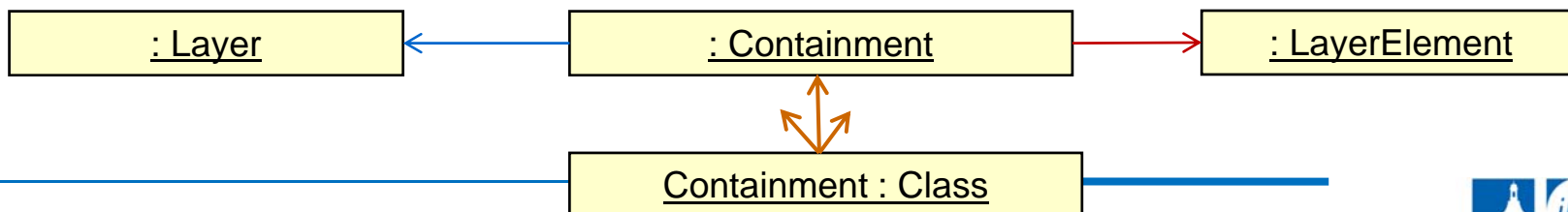
Objektentwurfsmodell vor der Transformation



Objektentwurfsmodell nach der Transformation



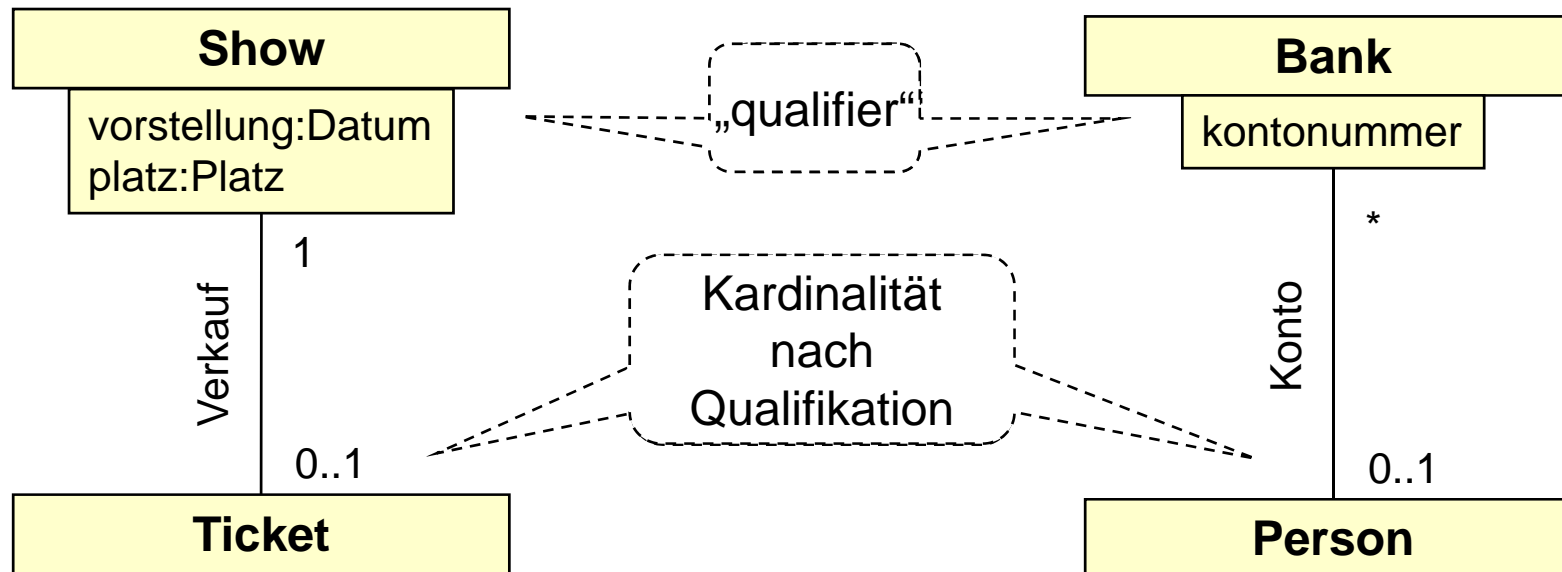
Beispiel-Instanziierung nach der Transformation



Qualifizierte Assoziation

- Qualifier

- ◆ gehört zur Assoziationen, nicht zu den Partner-Klassen
- ◆ modelliert Indizierung aus Sicht der einen Klasse
- ◆ Effekt: Kardinalität „am anderen Ende“ der Beziehung ist immer 0,1
 - ⇒ Entweder gibt es kein passendes Objekt oder es ist durch den Qualifier eindeutig bestimmt

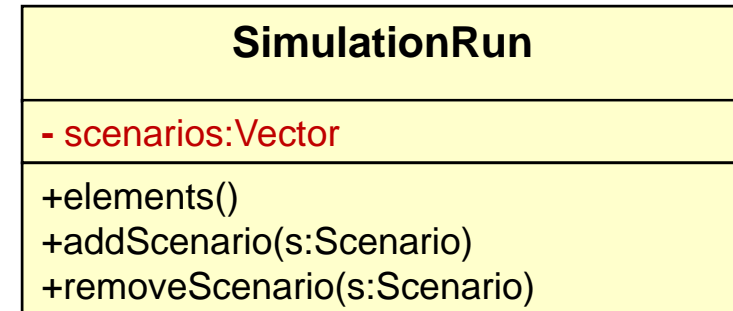
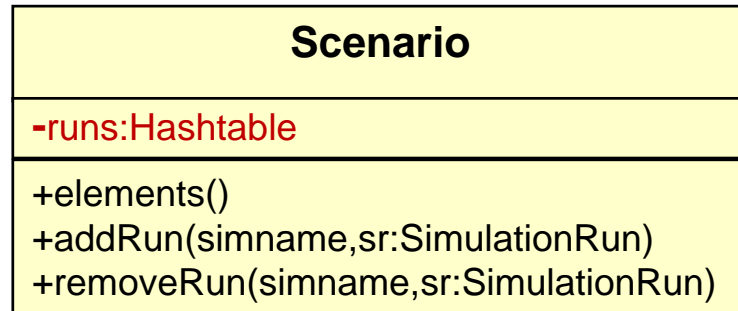


Qualifizierte Assoziation: Umsetzung

Objektentwurfsmodell vor der Transformation



Objektentwurfsmodell nach der Transformation



- Der Qualifier „name“ wird zum Schlüssel („key“) in der Hashtabelle.

Optimierung des Objektmodells

Aktivitäten während des Objektentwurfs

1. Schnittstellenspezifikation

- ◆ Spezifikation der Schnittstellen aller Typen (Klassen und Interfaces)

2. Auswahl vorhandener Komponenten

- ◆ Bestimmung von Standardkomponenten zusätzliche Objekte der Lösungsdomäne

3. Restrukturierung des Objektmodells

- ◆ Umformen des Objektmodell zur Verbesserung von Verständlichkeit und Erweiterbarkeit

4. Optimierung des Objektmodells

- ◆ Umformen des Objektmodells unter Berücksichtigung von Performanzkriterien, wie Reaktionszeit (z.B. der Benutzerschnittstelle) oder Speicherbedarf.

Entwurfsoptimierung

- Ein Objektdesigner muss einen Ausgleich zwischen Effizienz, Klarheit und Allgemeinheit schaffen.
 - ◆ Optimierungen machen das Modell undurchsichtiger.
 - ◆ Optimierungen machen das Modell weniger erweiterbar, da sie bestimmte Annahmen voraussetzen.

Aktivitäten während der Entwurfsoptimierung

1. Umordnen der Ausführungsreihenfolge

- ◆ Eliminiere „tote Pfade“ so früh wie möglich. (Verwende Wissen über Verteilung, Frequenz der Pfadtraversierung)
- ◆ Grenze Suche so früh wie möglich ein
- ◆ Überprüfe ob die Ausführungsreihenfolge von Schleifen umgekehrt werden sollte

2. Hinzufügen redundanter Assoziationen

- ◆ Was sind die häufigsten Operationen? (Abfrage von Sensordaten?)
- ◆ Wie häufig werden diese Operationen aufgerufen? (30 mal pro Monat, alle 30 Millisekunden)

3. Speichern abgeleiteter Attribute um Rechenzeit zu sparen

- ◆ Achtung, Redundanz → Konsistenzerhaltung erforderlich (z.B. Observer)

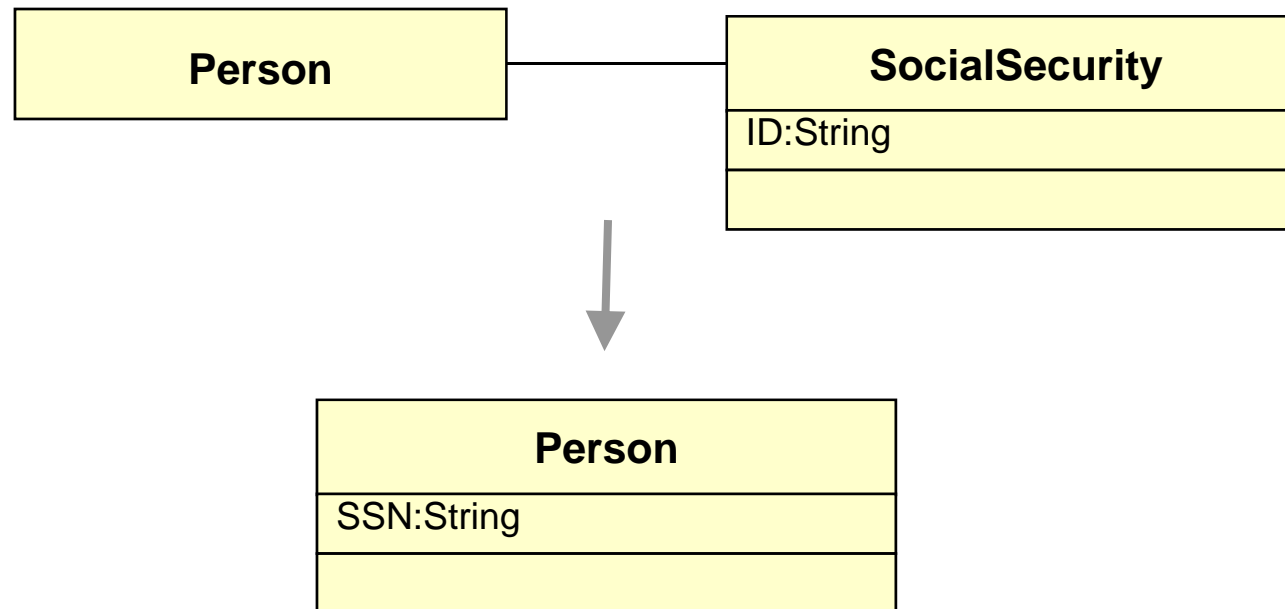
4. Transformation von Klassen zu Attributen

Implementierung von Klassen der Anwendungsdomäne

- Attribut oder Assoziation?
 - ◆ Assoziationen durch Attribute ersetzen?
- Mögliche Entwurfsentscheidungen
 - ◆ Implementiere Entität als eingebettetes Attribut
 - ◆ Implementiere Entität als separate Klasse mit Assoziationen zu anderen Klassen
- Assoziationen sind flexibler als Attribute, führen aber häufig zu unnötigen Indirektionen

Aktivitäten während der Optimierung: Reduzieren von Objekten zu Attributen

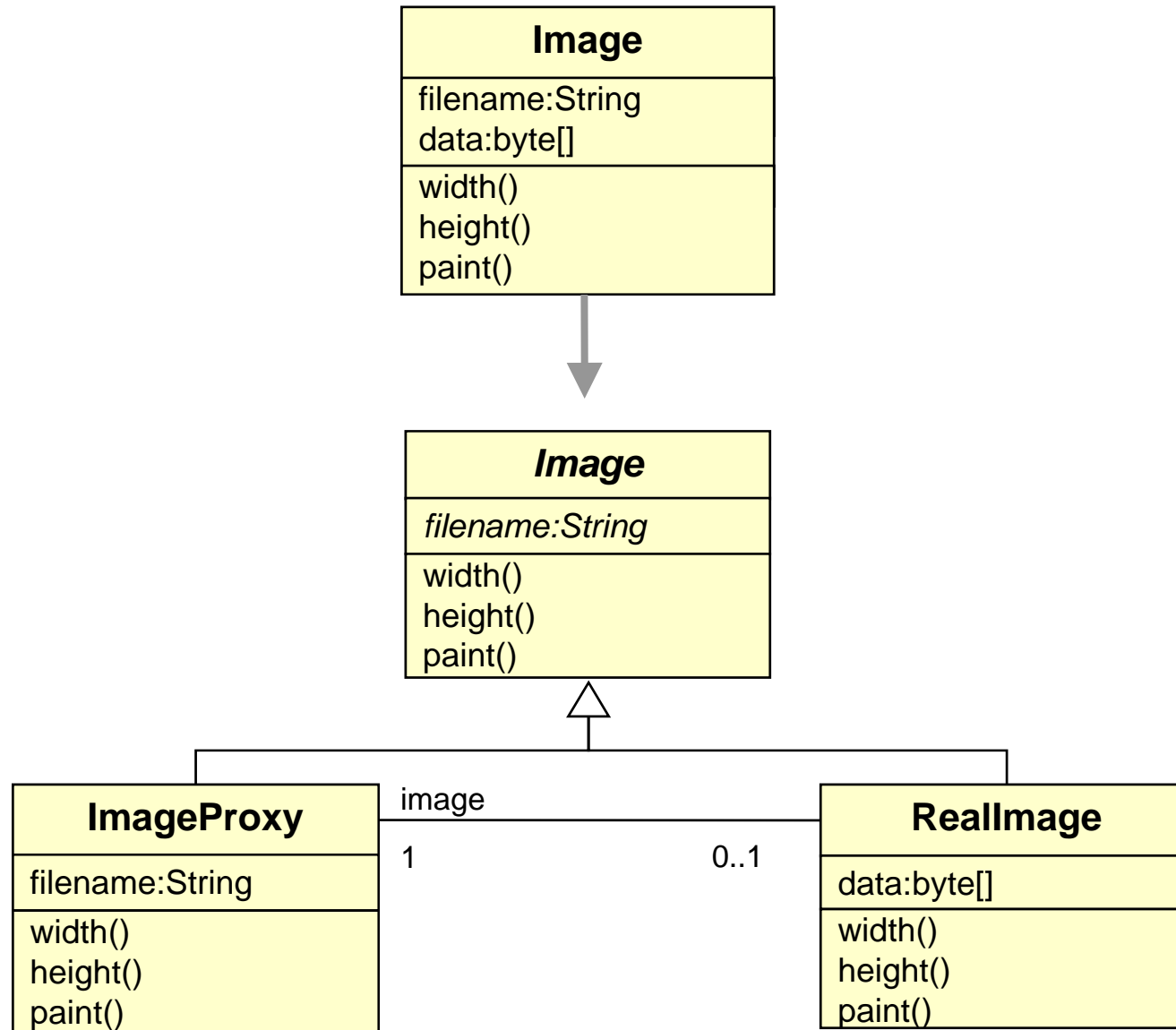
- Verwandle eine Klasse in ein Attribut, wenn get() und set() die einzigen Methoden sind, die von ihr definiert werden.



Entwurfsoptimierungen: Speichern von abgeleiteten Attributen

- (Zwischen-)speichern abgeleiteter Attribute
 - ◆ z.B.: Definition einer neuen Klasse um Daten lokal zu speichern (Datenbankcache)
- Problem von abgeleiteten Attributen
 - ◆ Abgeleitete Attribute müssen auf den neusten Stand gebracht werden, wenn sich der Basiswert ändert.
 - ◆ Es gibt drei Möglichkeiten, mit diesem Problem umzugehen:
 - ⇒ Expliziter Code: Der Programmierer bestimmt die betroffenen abgeleiteten Attribute (*push*)
 - ⇒ Regelmäßige Neuberechnung: Abgeleitete Attribute werden gelegentlich neu berechnet (*pull*)
 - ⇒ Active value: Ein Attribut kann eine Menge von abhängigen Attributen bestimmen, die automatisch auf den neusten Stand gebracht werden wenn sich der „aktive Wert“ (*active value*) ändert. (*event notification, data trigger*)

Aktivitäten während der Optimierung: Teure Operationen erst bei Bedarf



Zusammenfassung

- Der Objektentwurf schließt die Lücke zwischen Anforderungen und bestehendem System.
- Der Objektentwurf bezeichnet den Prozess, in dem dem Ergebnis der Anforderungsanalyse und des Systementwurfs Details hinzugefügt und Implementierungsentscheidungen getroffen werden.
- Der Objektentwurf beinhaltet
 1. Die Spezifikation von Schnittstellen (Signaturen, DBC, Behav. Protocols)
 2. Die Auswahl von Komponenten
 3. Die Restrukturierung des Objektmodells
 4. Die Optimierung des Objektmodells