universitätbonn ROOTS

Kapitel 9. „Objektentwurf“

Stand: 08.01.2014

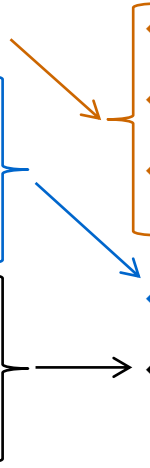
Teilweise nach „Brügge & Dutoit“, Kap. 8

Überblick

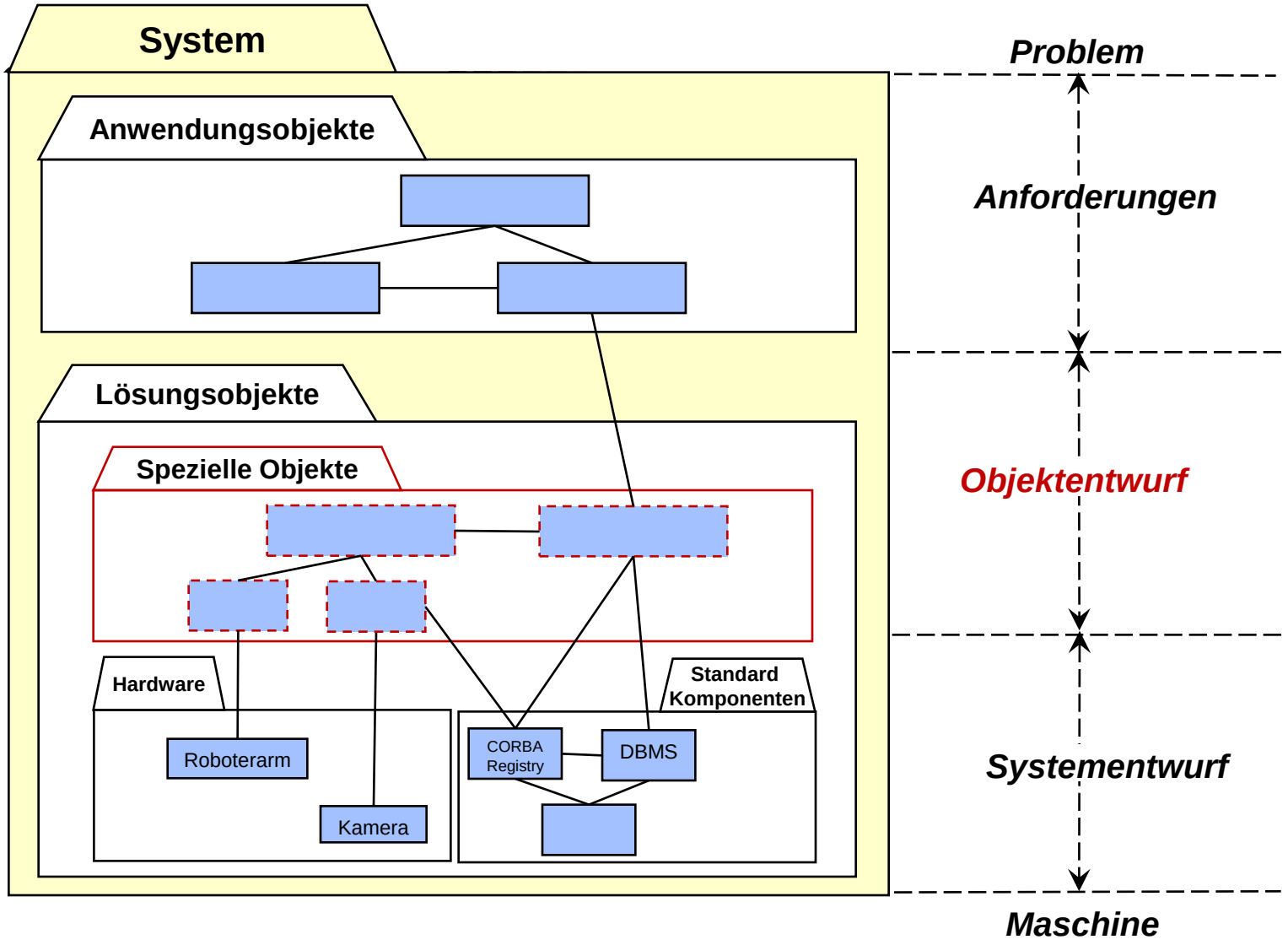
- Objektentwurf
  - Definition
  - Aktivitäten

- Wichtige Aktivitäten
  - Spezifikation von Schnittstellen
  - Fortgeschrittene UML-Konzepte
  - Umsetzung in „Kern-UML“-Konzepte
  - Umsetzung von „Kern-UML“ in implementierungsnahe Designs
  - Umsetzung in Code

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



Objektentwurf: Die Lücke schließen



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.

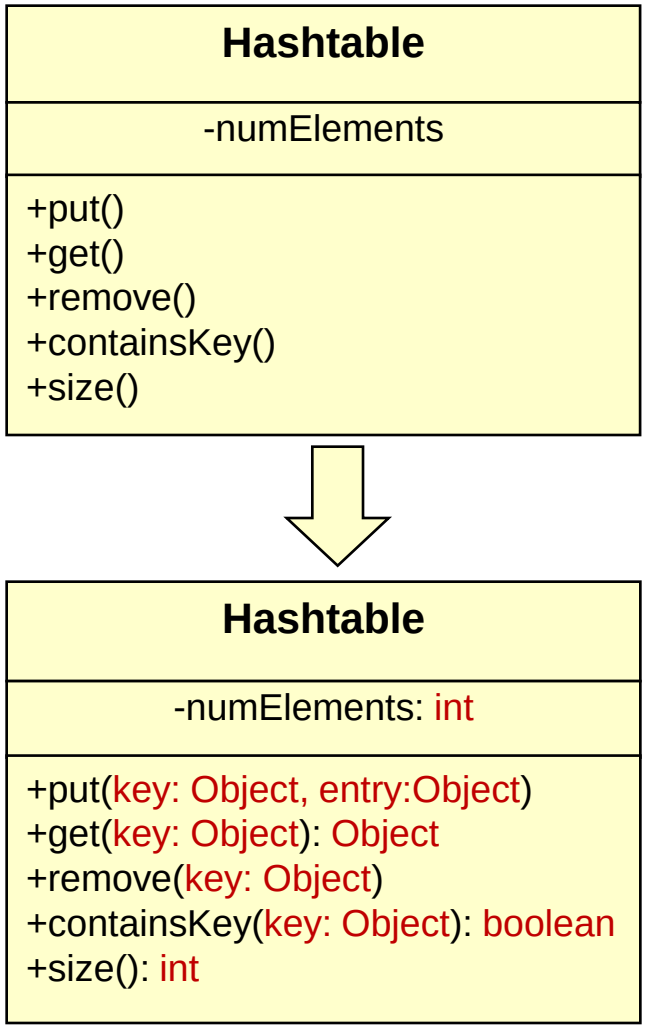
Fortsetzung aus dem Systementwurf

Schnittstellenspezifikation

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

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: Sichtbarkeitsinformationen

Hinzufügen von Sichtbarkeitsinformationen

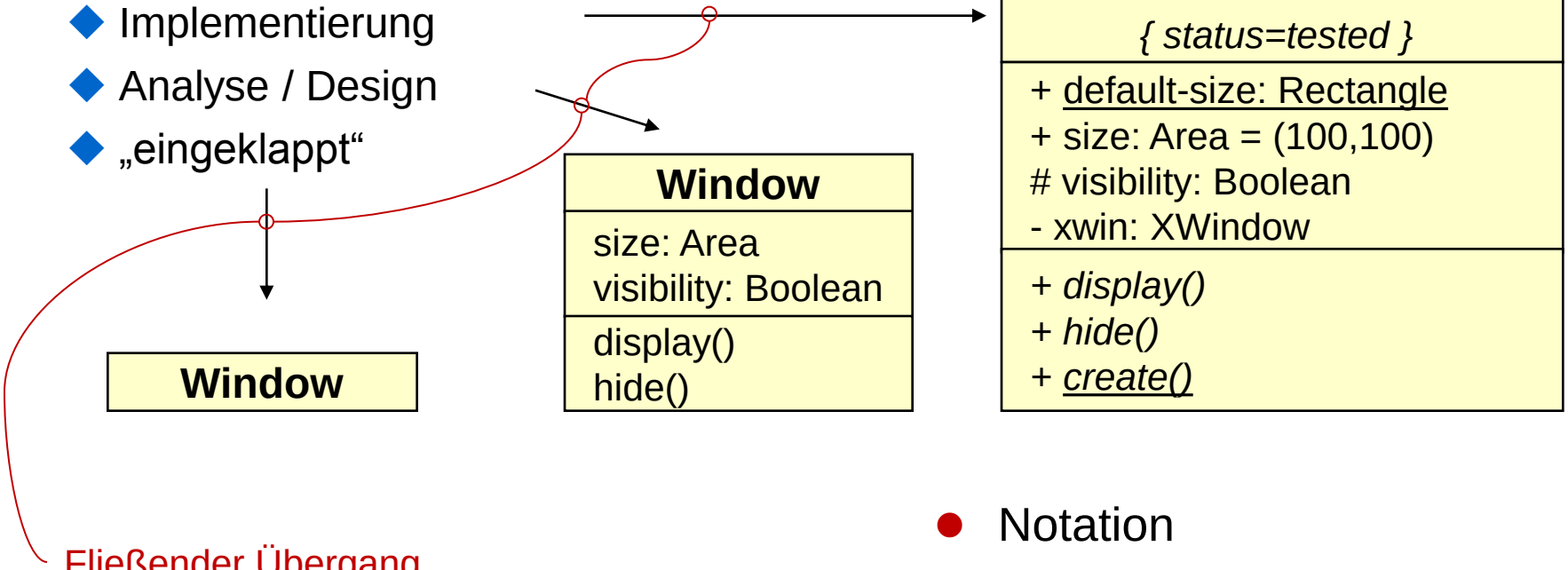
UML definiert drei Sichtbarkeitsgrade:

- 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 bekante „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.

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

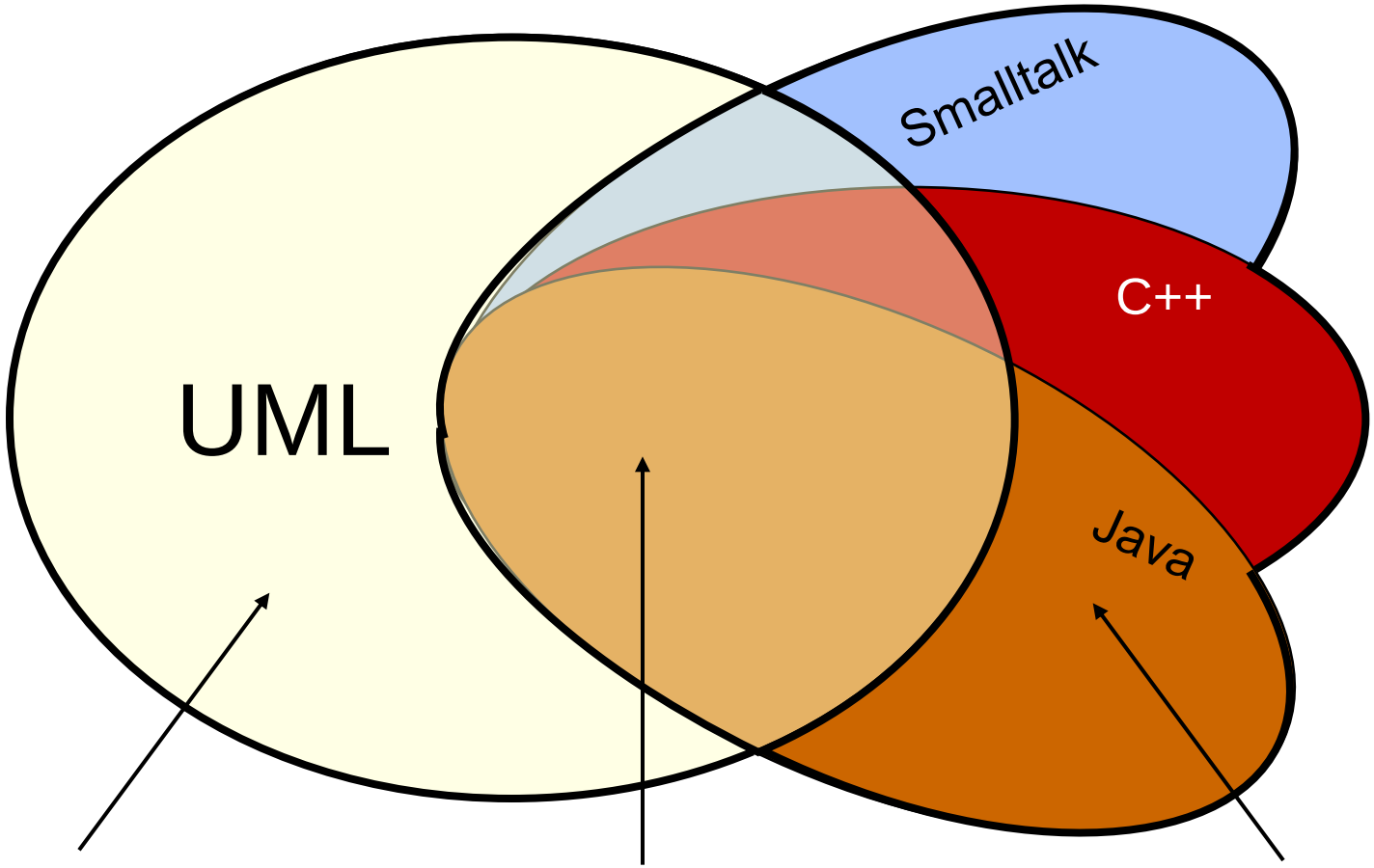
Aktivitäten während des Objektentwurfs

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



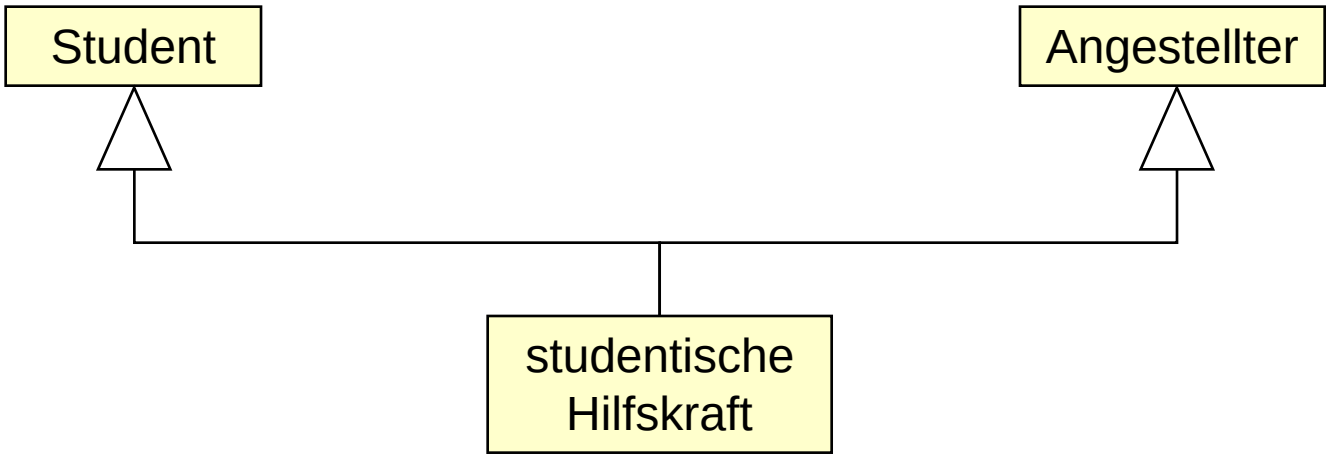
Konzepte ohne direkter Entsprechung in gängigen Sprachen

direkt ineinander abbildbarer gemeinsamer „Kern“

sprachspezifische Details

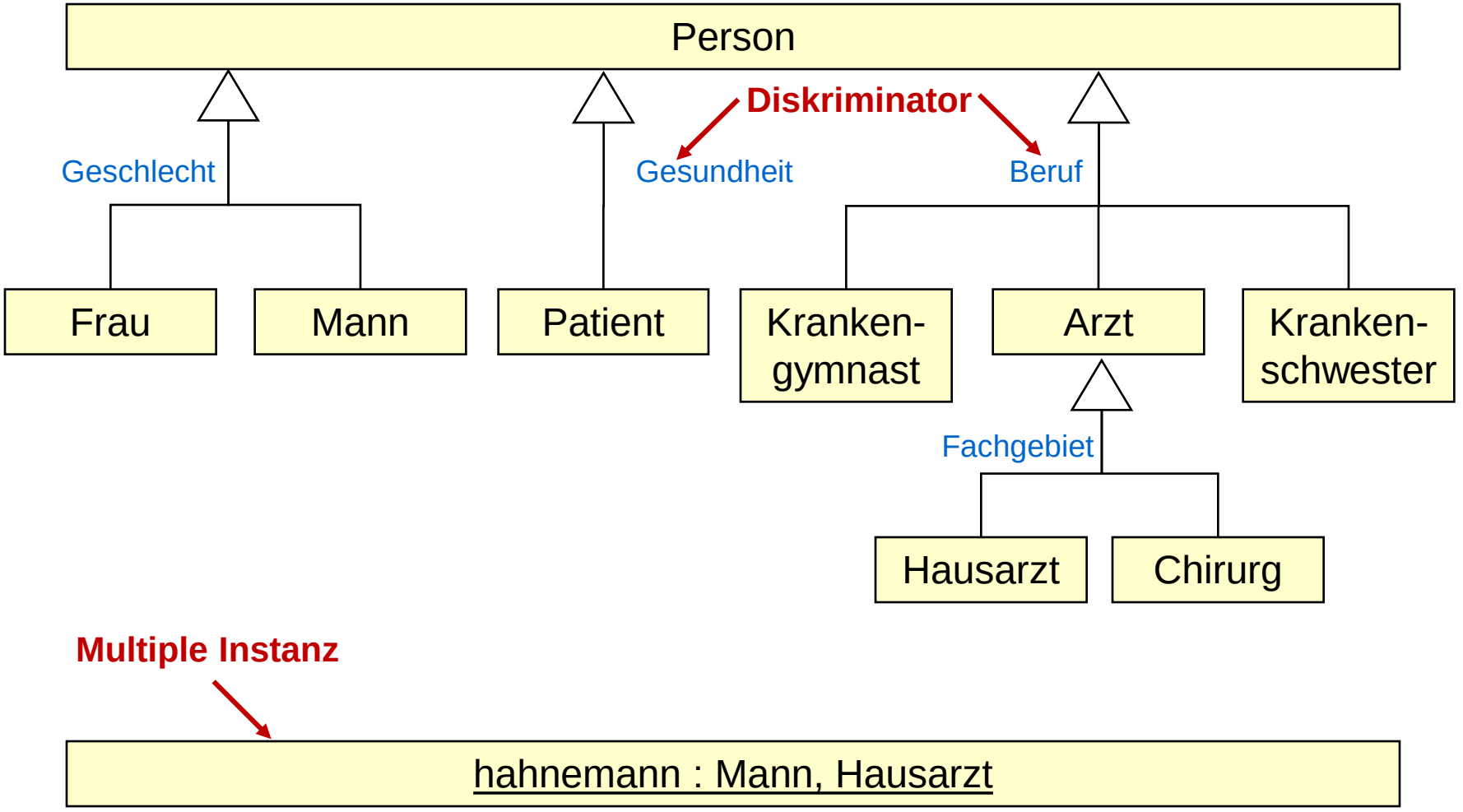
UML, statisches Modell: Multiple Vererbung

- Eine Klasse mit mehreren Oberklassen



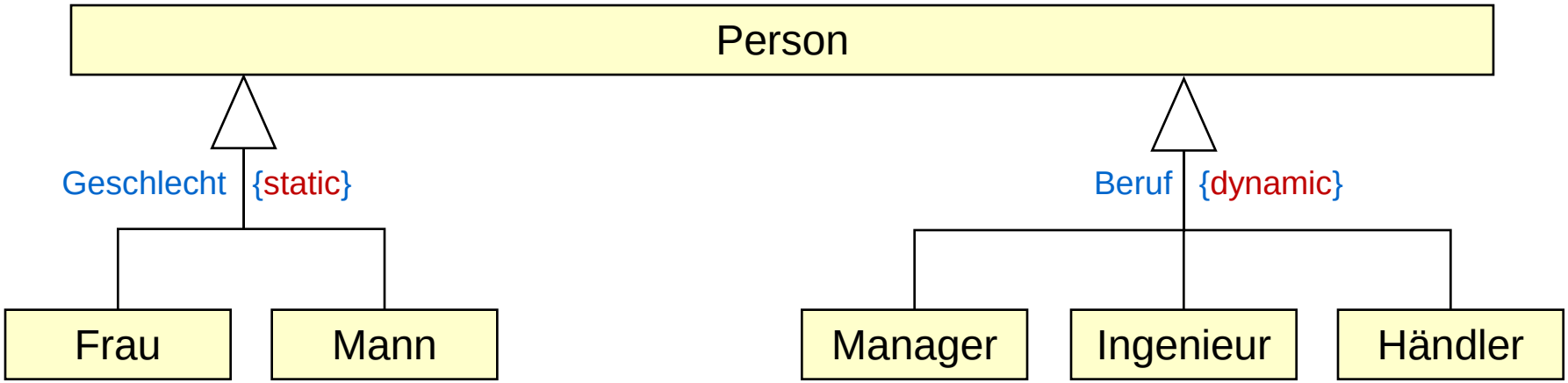
UML, statisches Modell: Multiple Klassifikation

- Ein Objekt kann Instanz mehrerer Klassen sein



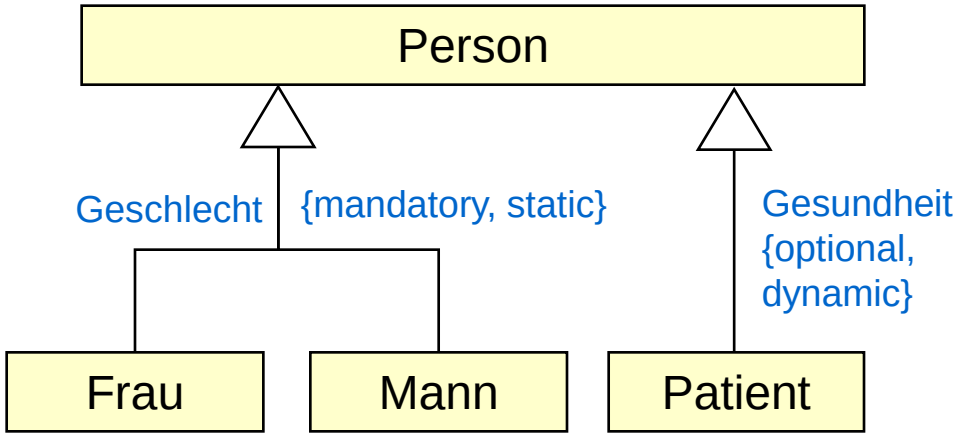
UML, statisches Modell: Dynamische Klassifikation

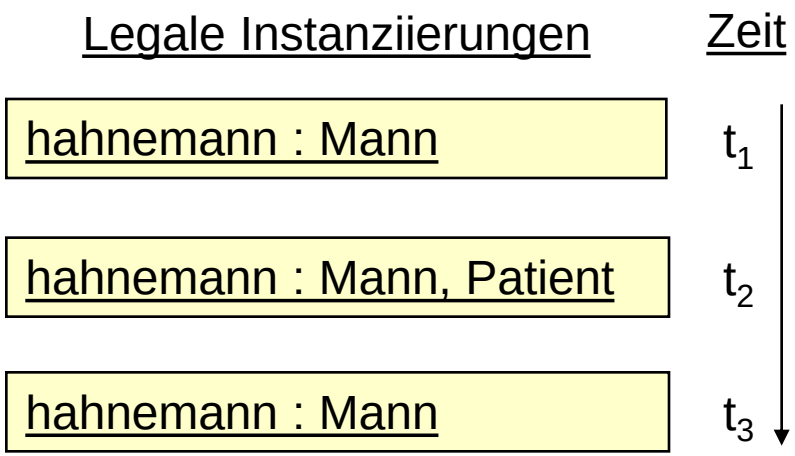
- Eine Objekt kann Instanz mehrerer Klassen sein
- ... und seine Klassenzugehörigkeit ändern



UML: Partitionierung von Unterklassen

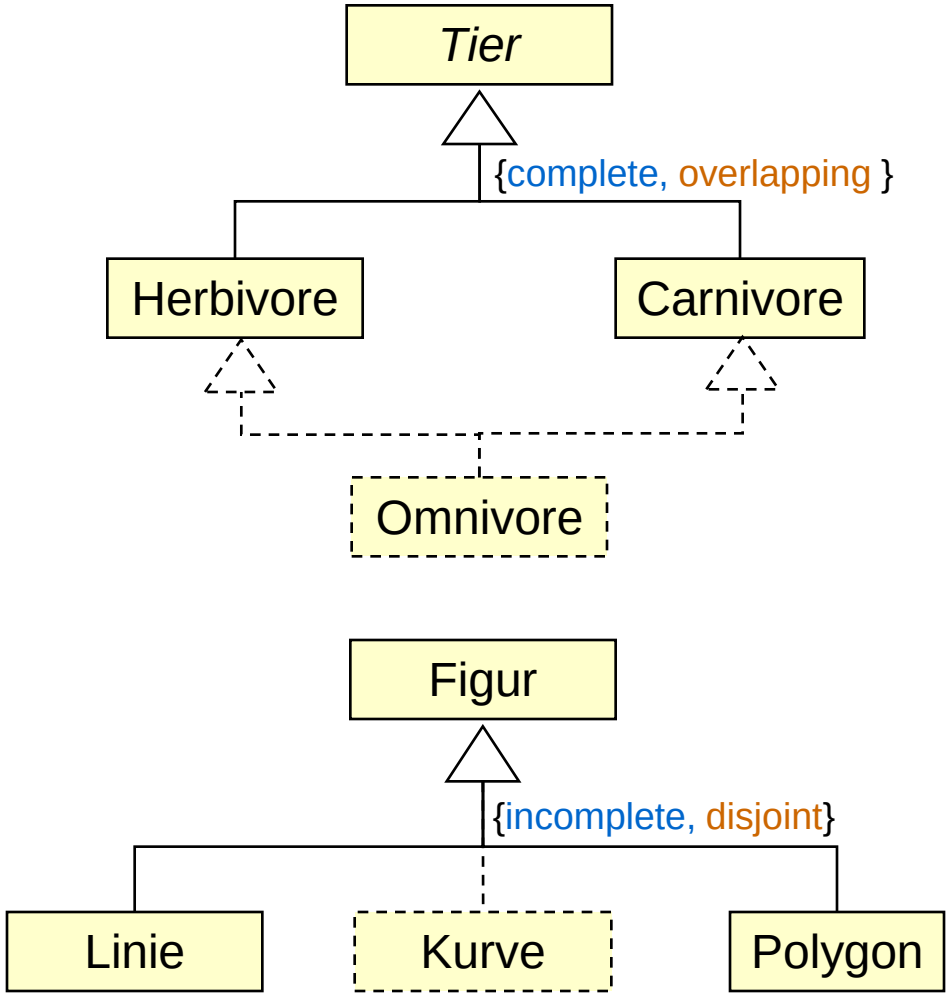
- obligatorisch (mandatory)
  - Objekte müssen einer Klasse aus dieser Partition angehören
- optional (optional) (default)
  - Objekte müssen nicht ...
- statisch (static) (default)
  - Objekte bleiben lebenslang in einer Klasse
- dynamisch (dynamic)
  - Objekte können Klasse wechseln
  - impliziert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





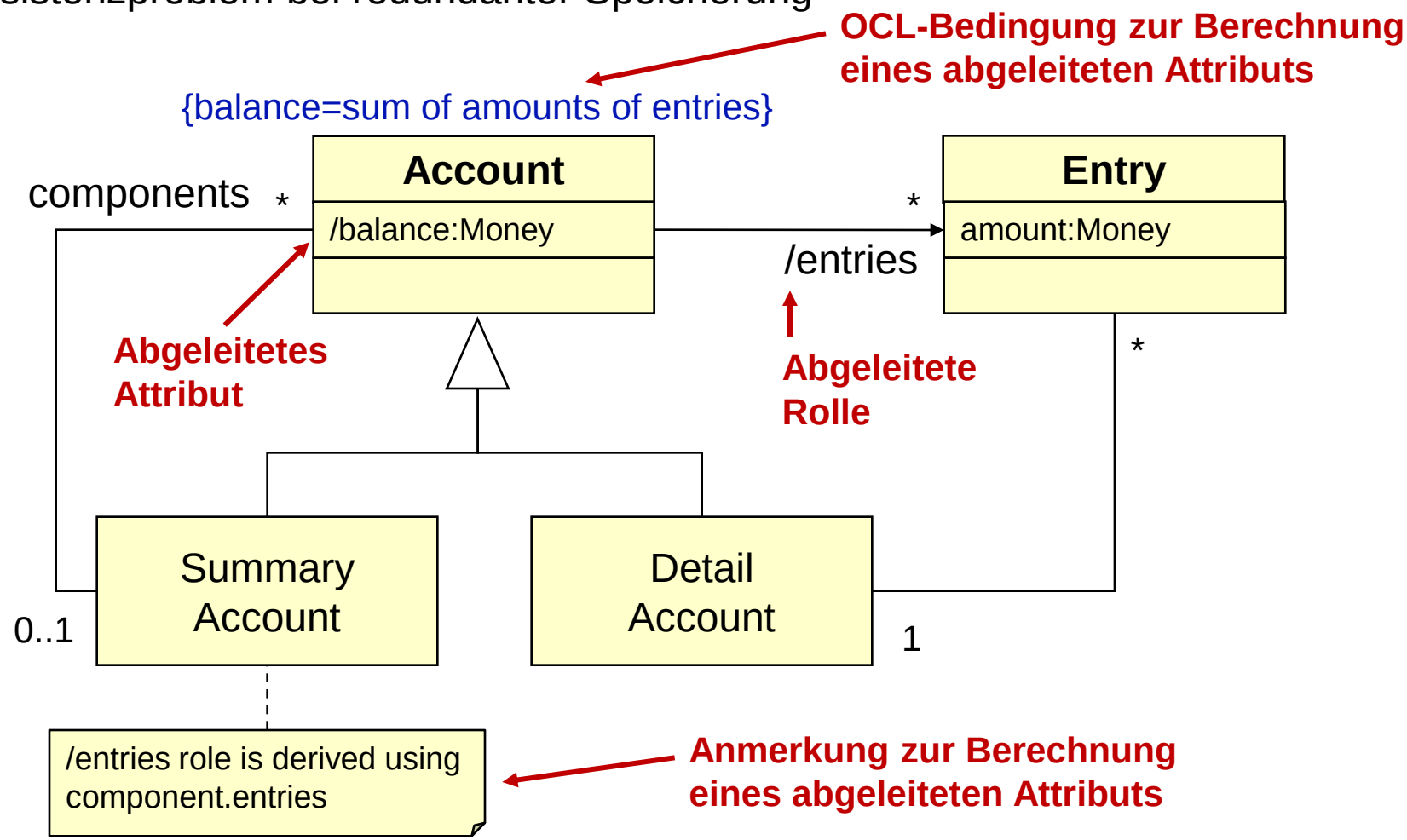
UML: Partitionierung von Unterklassen

- 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

„Split object“ Design Patterns

UML_{Core}

- Statische Klassifikation
- Einfache Instanziierung
- Einfache Vererbung

- Bidirektionale Assoziationen
- Qualifizierte Assoziationen

Assoziations-realisierung

- Unidirektionale Assoziationen (Felder)

In diesem Abschnitt

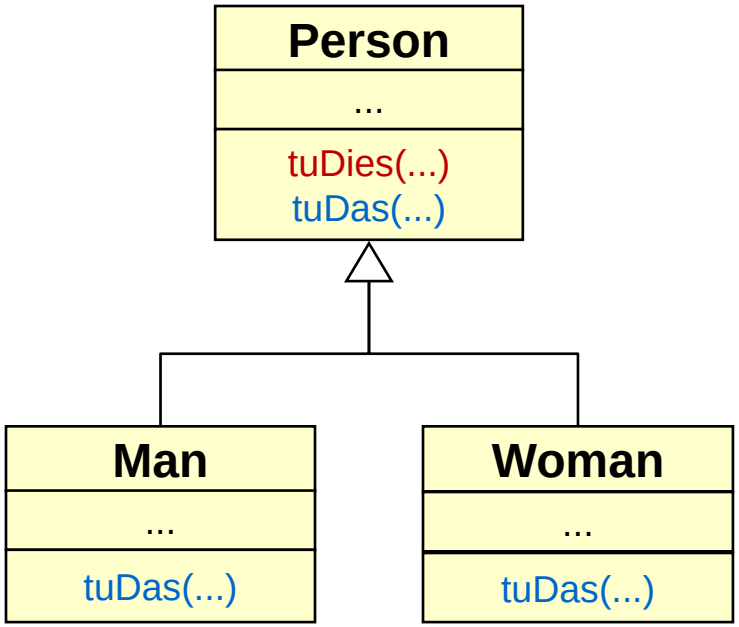
- Transformation von konzeptionellem Entwurf in „UML_{High}“ in implementierungsnahen Entwurf in „Kern-UML“
- Umsetzung in verfügbare Zielsprache danach einfach, da die Kern-UML-Konzepte 1:1 in Programmiersprachen unterstützt sind

Restrukturierung des Objektmodells: $UML_{High} \rightarrow UML_{Low}$

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

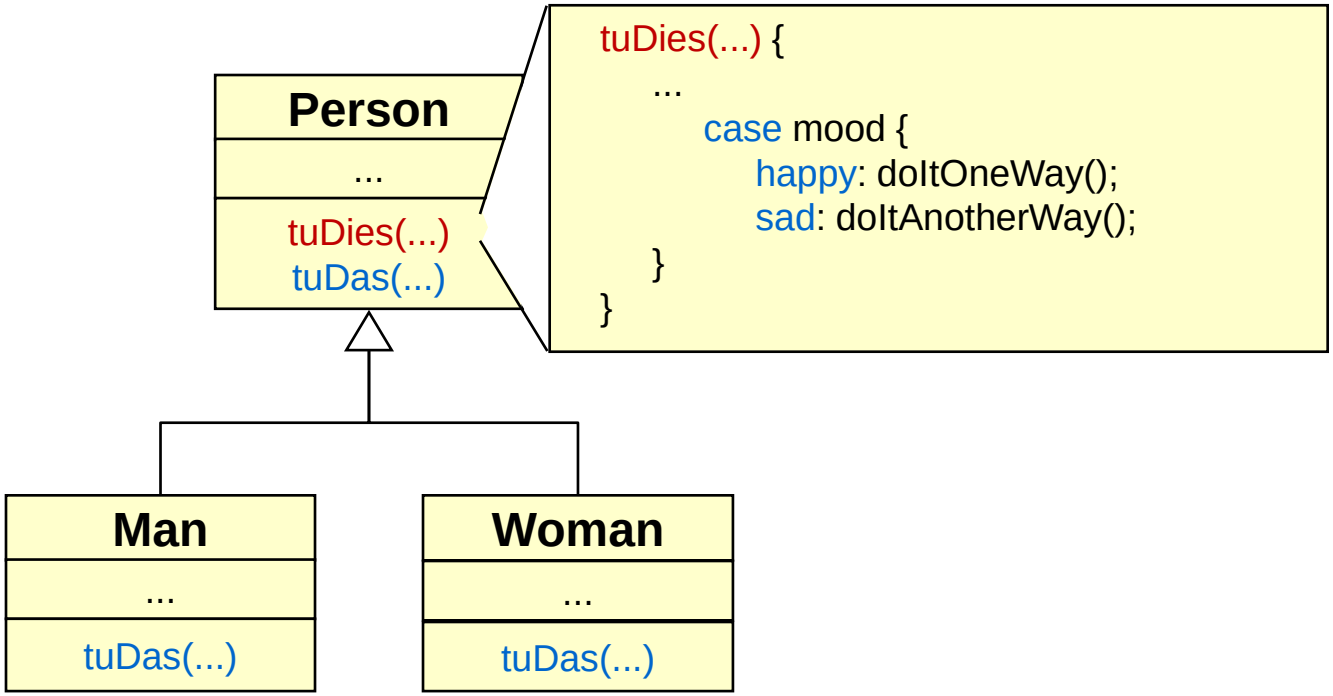
- objektspezifisches
- zustandsspezifisches
- dynamisch veränderbares

Verhalten?

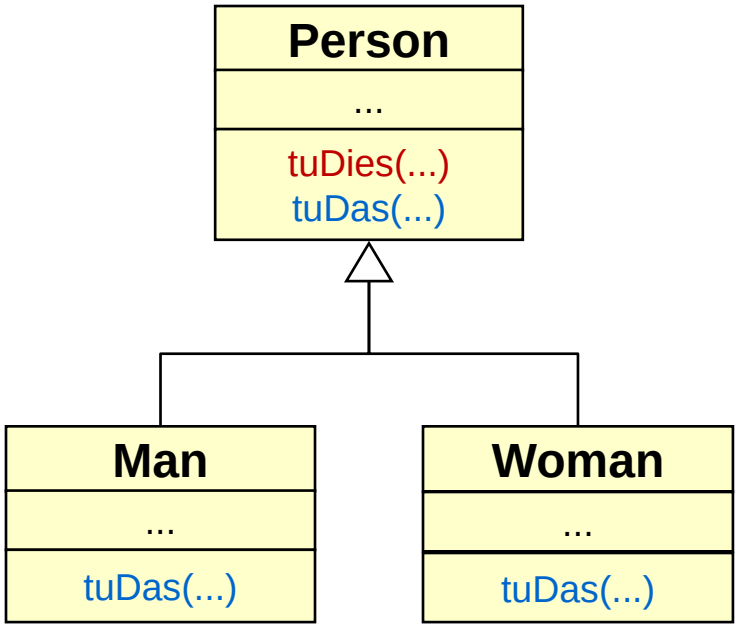
⇨ **tuDas()** ist geschlechtsspezifisch einheitlich ✓

⇨ **tuDies()** ist personenspezifisch verschieden ?

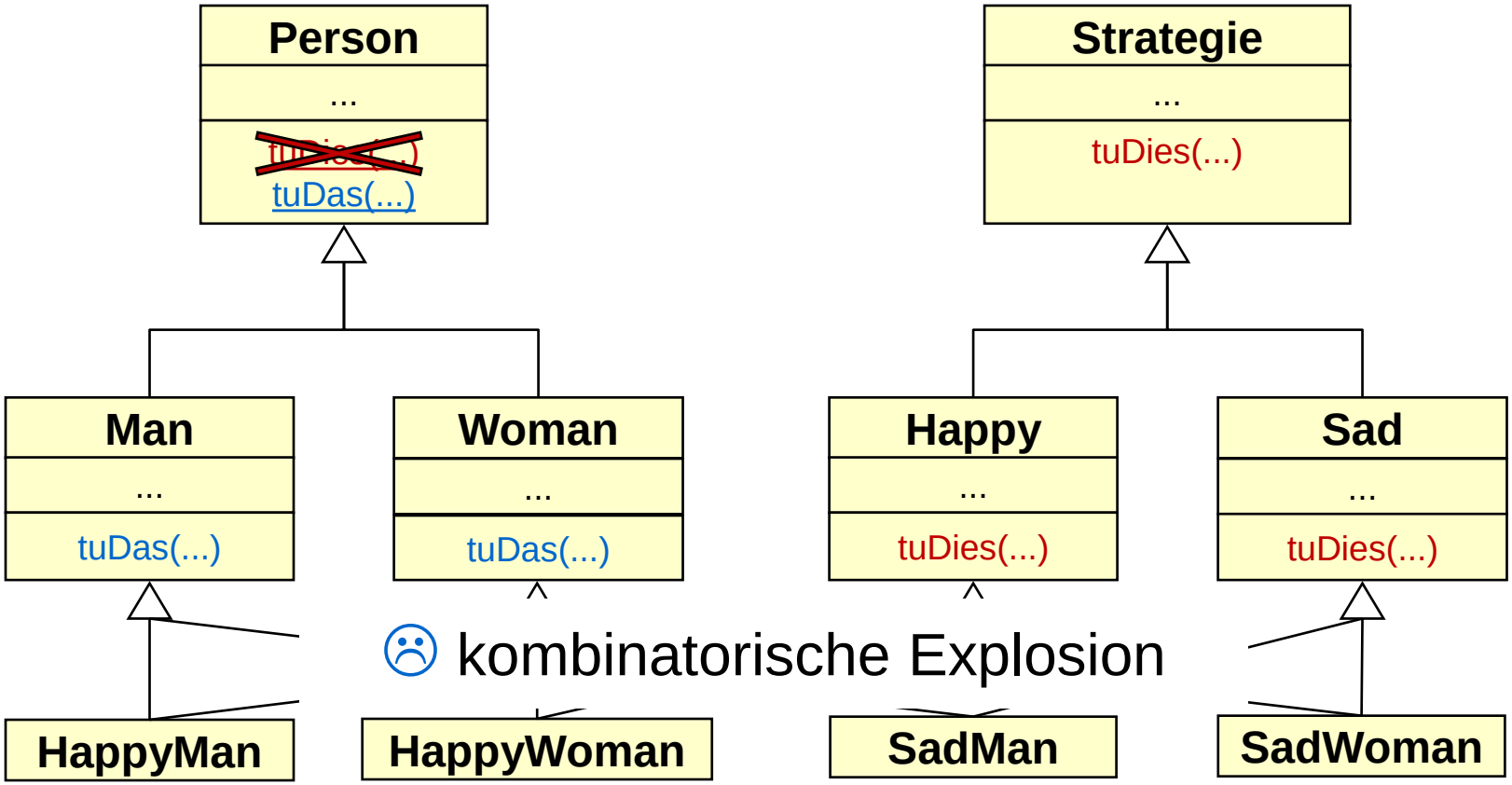
1. Versuch: “Fest Kodieren”



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


Person
...
tuDies(...)
tuDas(...)
Man
...
tuDas(...)
Woman
...
tuDas(...)

2. Versuch: “Multiple Vererbung”

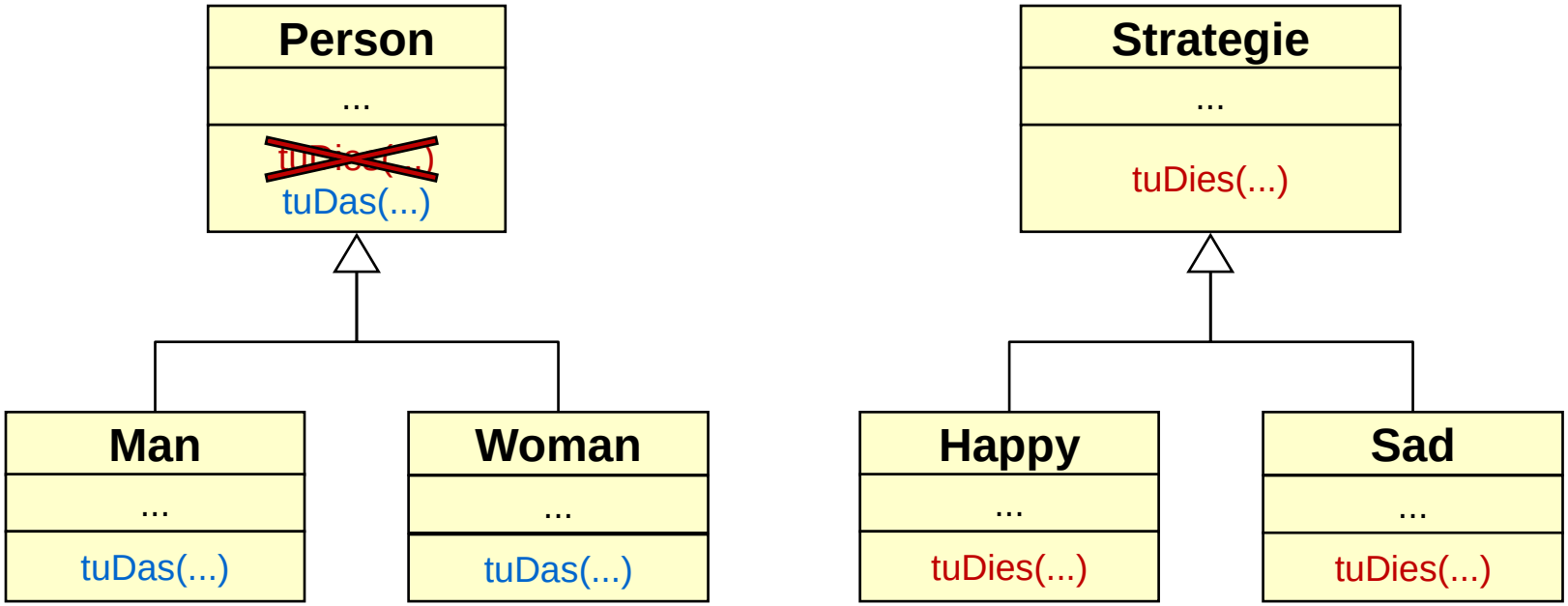


☹ kombinatorische Explosion

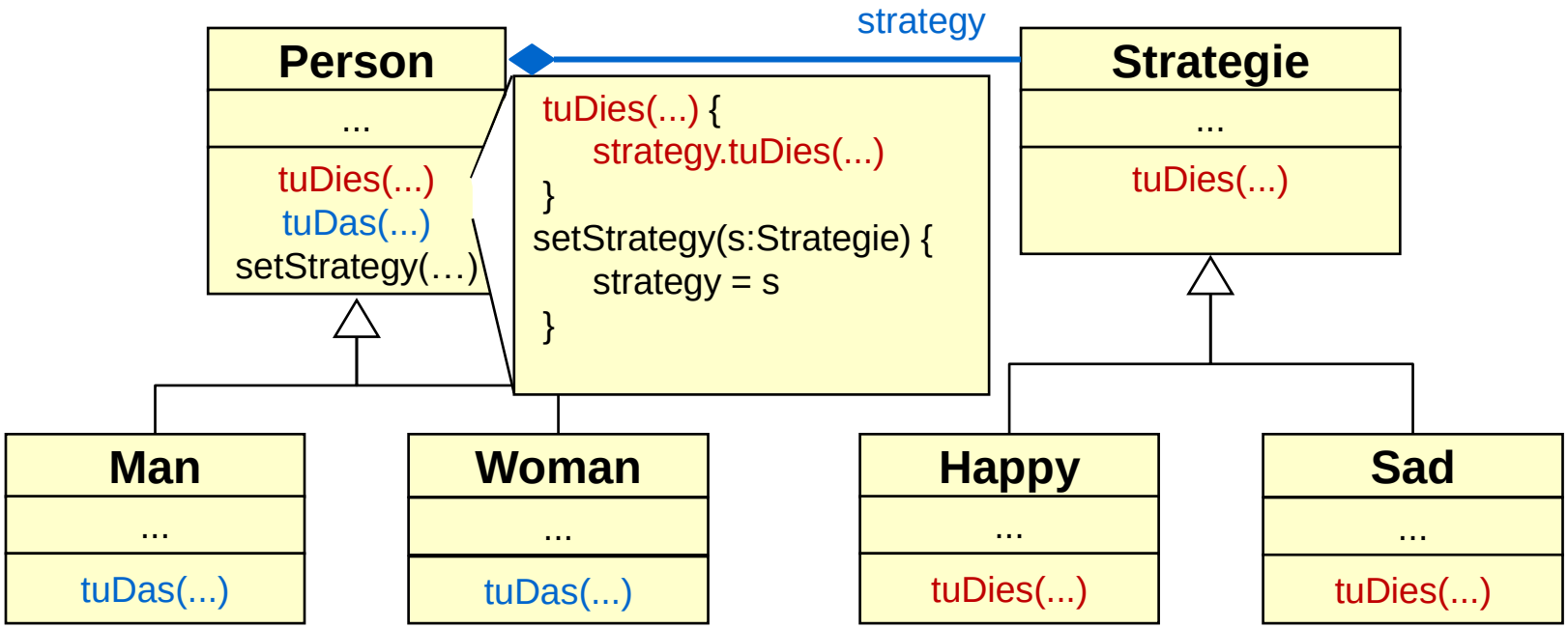
☹ klassenspezifisch festes Verhalten

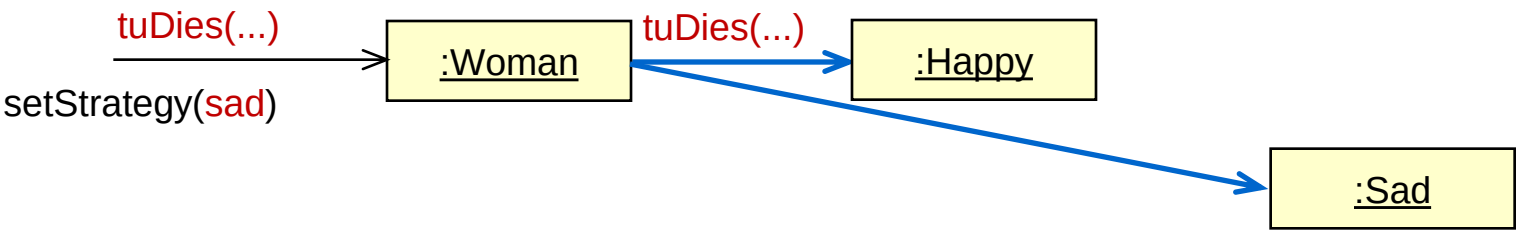
:SadWoman

3. Versuch: “Strategy Pattern”



3. Versuch: “Strategy Pattern”

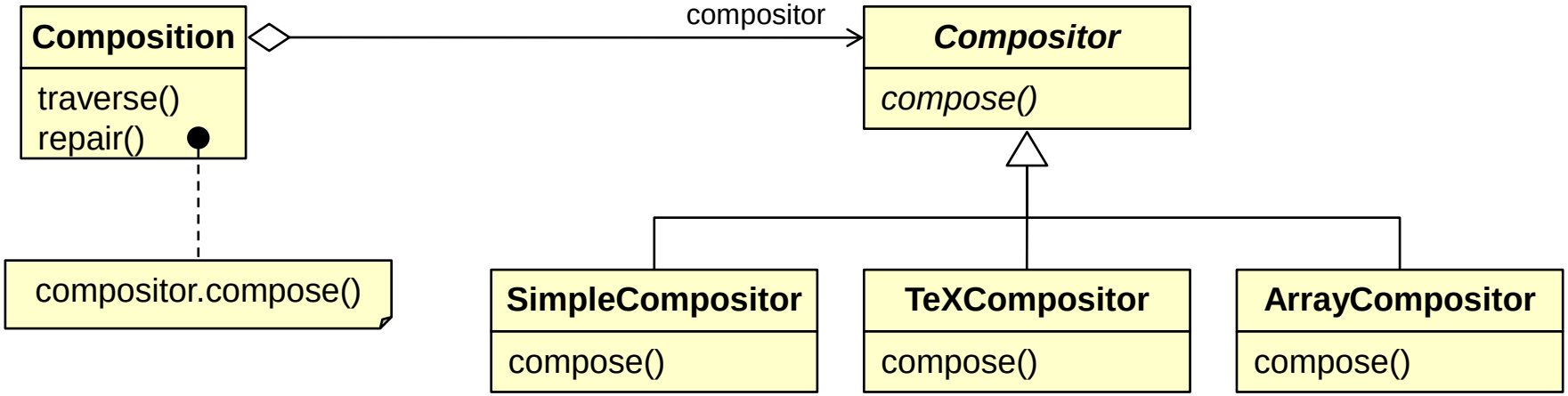




Das Strategy Pattern

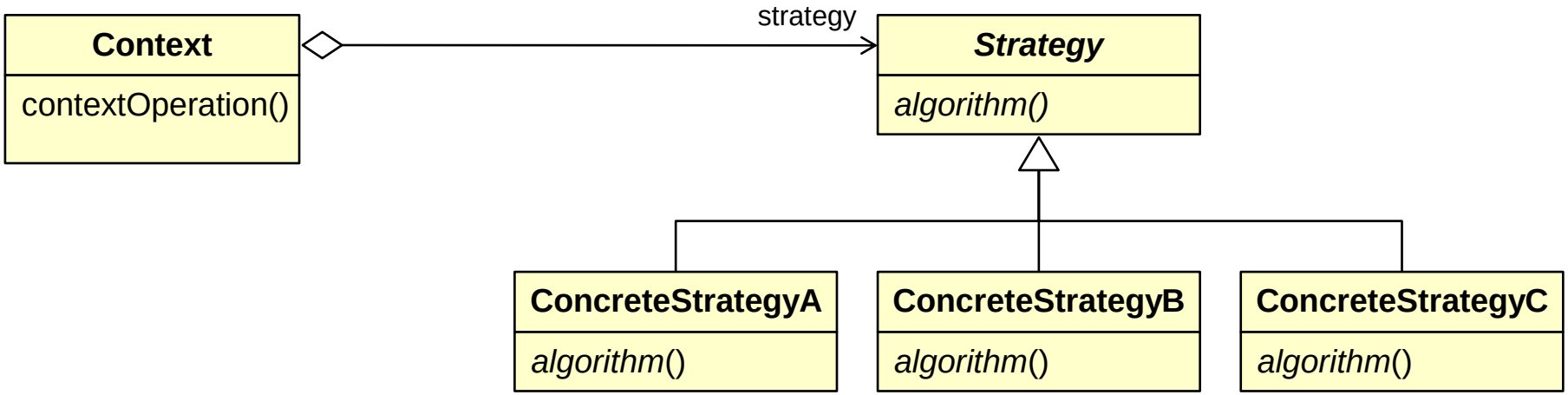
Das Strategy Pattern: Einführung

- 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

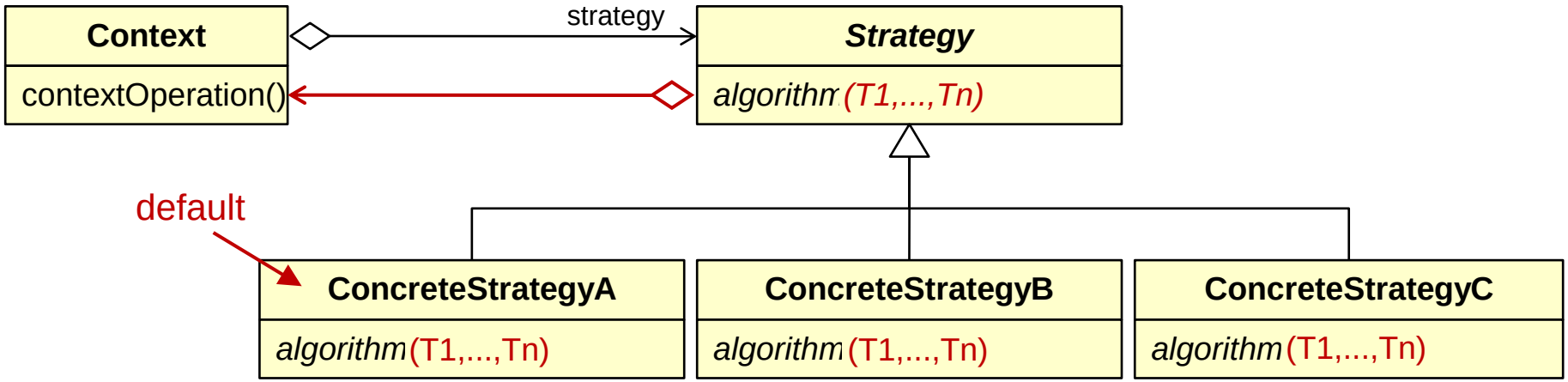
- Anwendbar in folgendem Kontext
  - Einige ähnliche Klassen unterscheiden sich nur in gewissen Aspekten des Verhaltens. Diese können in ein Strategie-Objekt ausgelagert werden.
  - Verhalten ist abhängig von äußeren Randbedingungen
  - Verschiedene Varianten eines Algorithmus werden benötigt
    - ⇨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



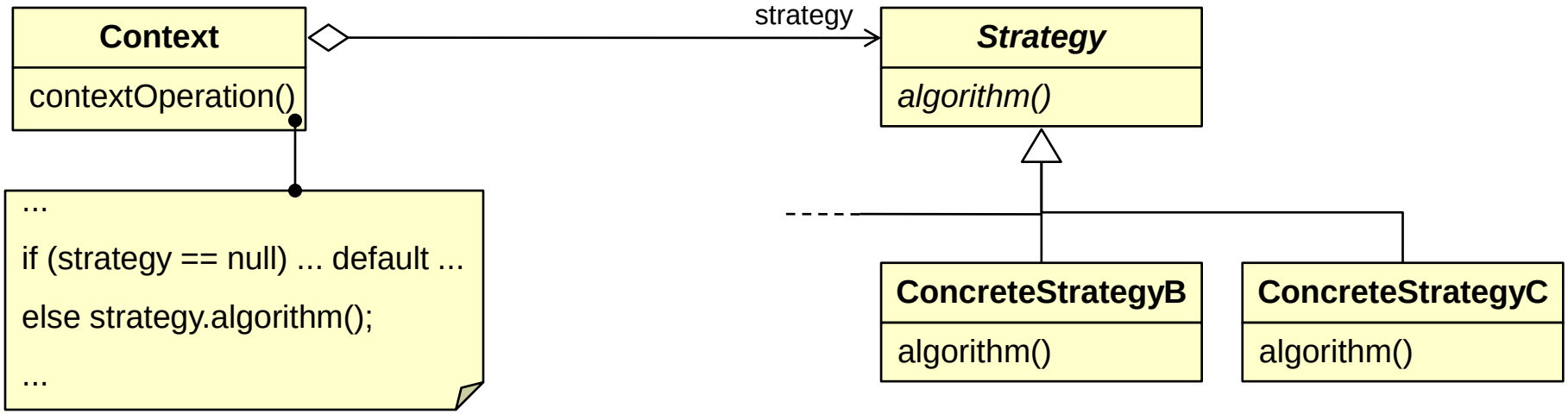
Das Strategy Pattern: Implementierung

- 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.

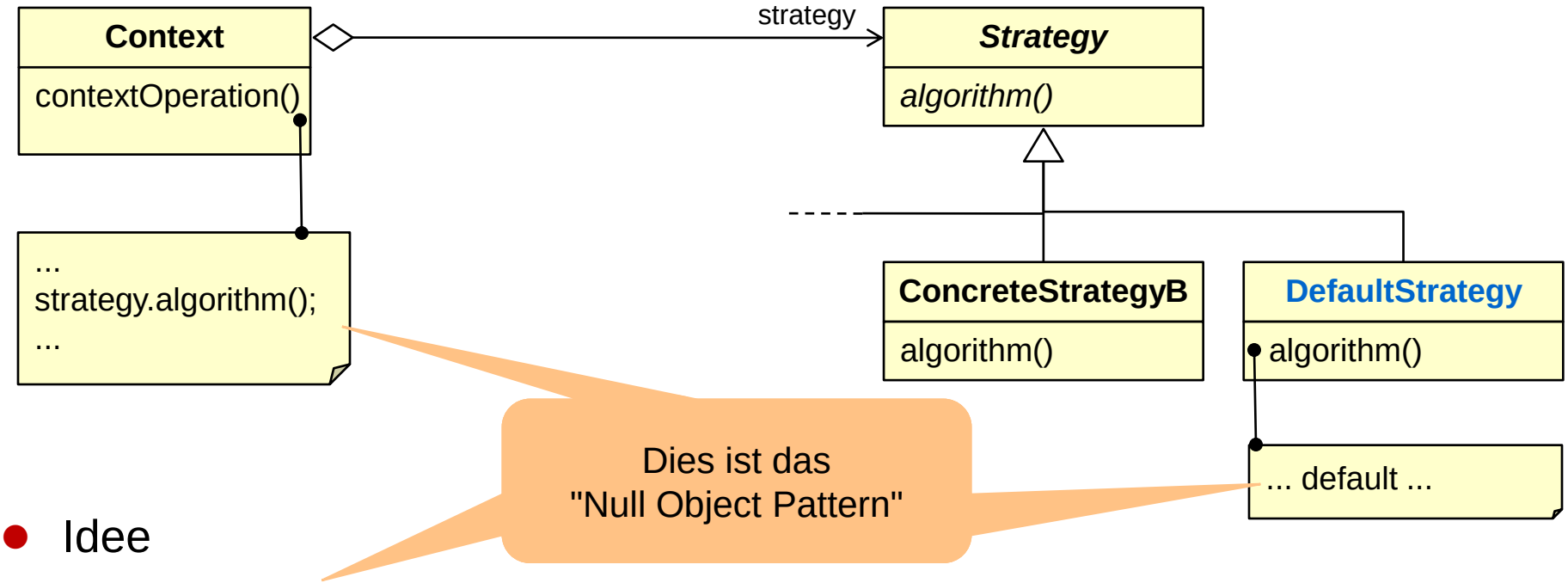


Implementierung: Fallunterscheidung in Kontext



- 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
  - lesbarerer 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

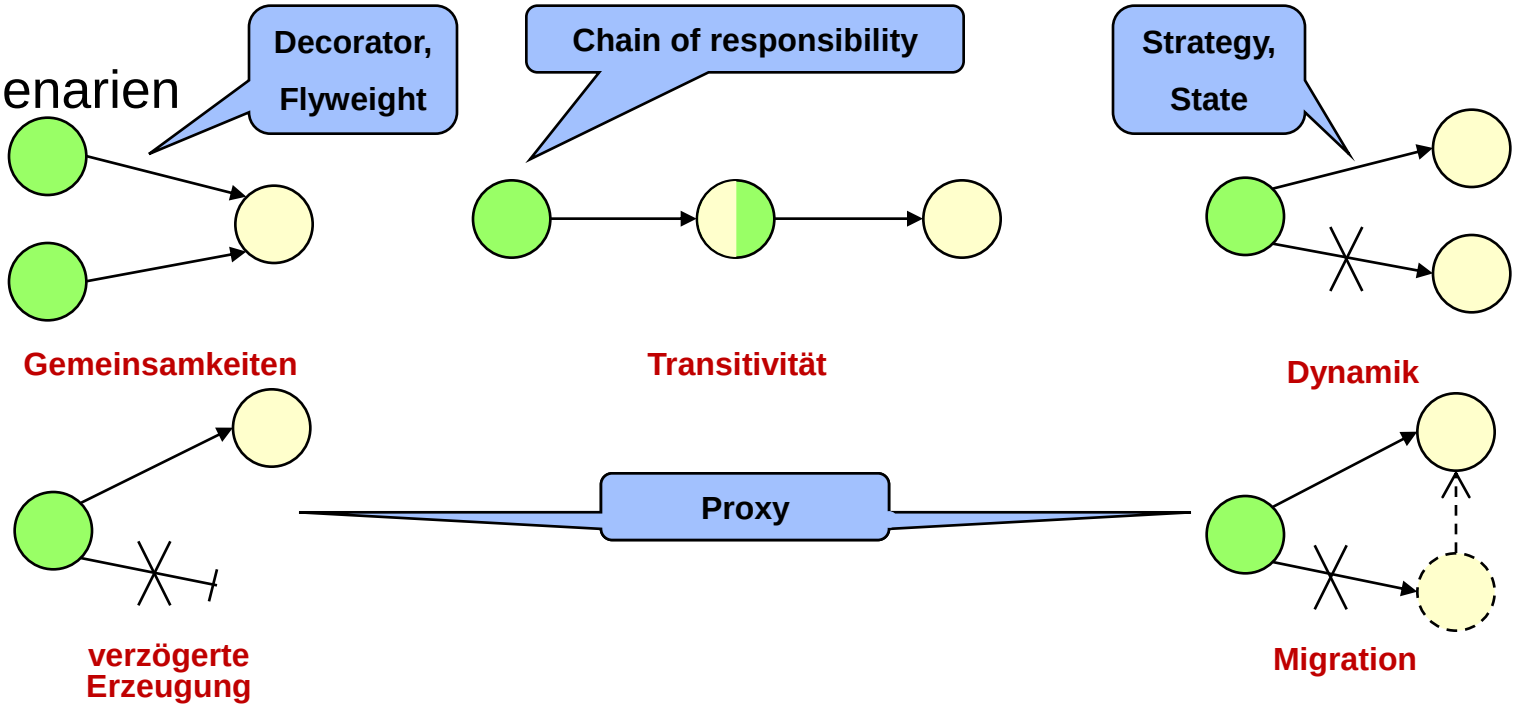
Was sind also "Split Objects"?

- Definition
  - verschiedene Objekte die konzeptuell als eine Einheit agieren
  - (schienbar) gemeinsame Identität
  - (schienbar) gemeinsamer Zustand
  - (schienbar) 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

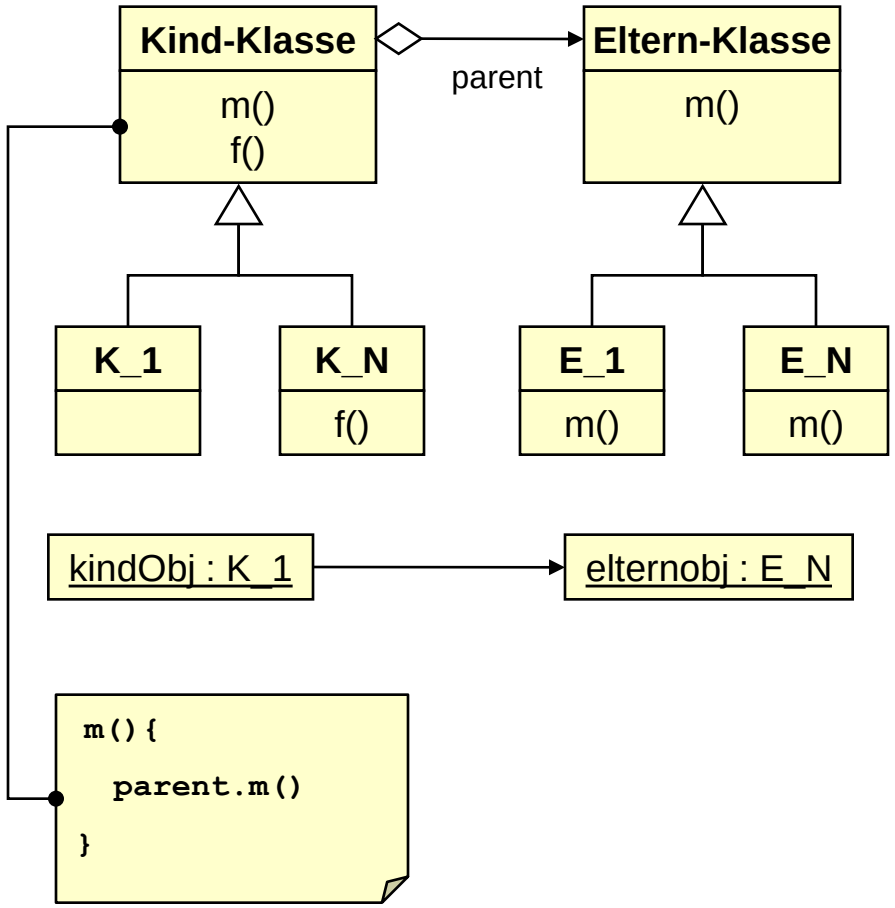


Split Objects sind die Grundlage vieler Design Patterns

- 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

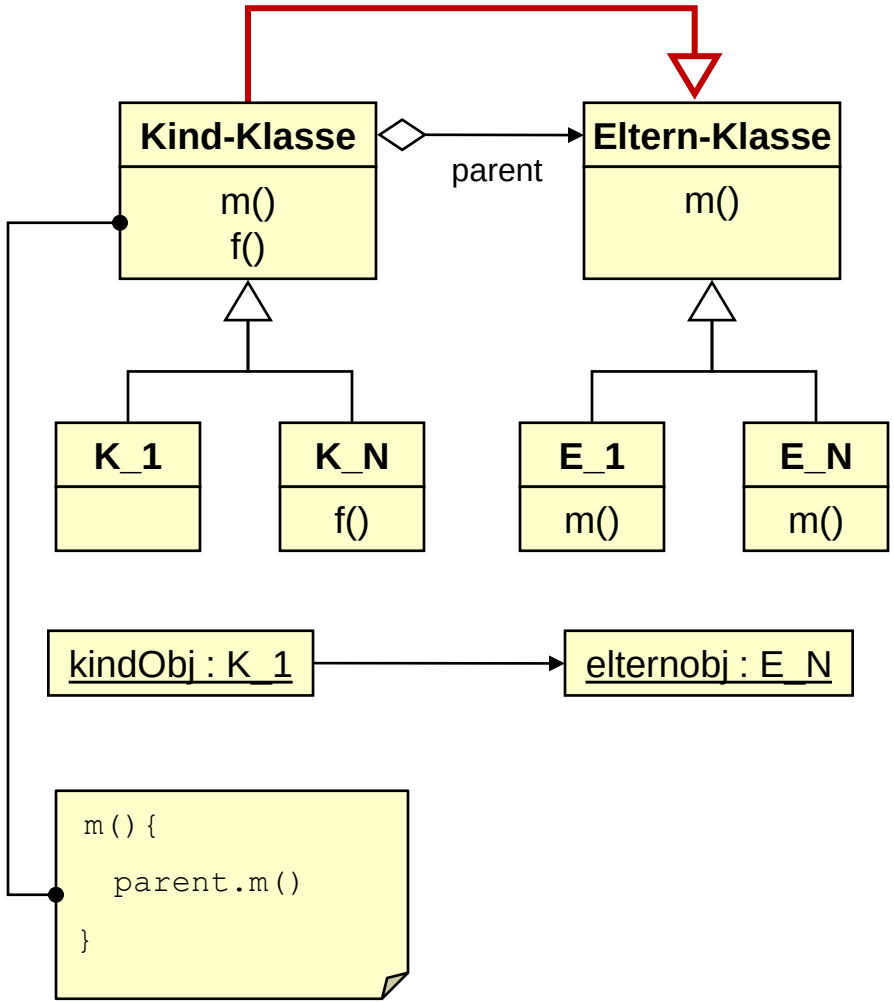
- Forwarding
  - Kind leitet empfangene Nachricht an Eltern-Objekt weiter
- Subtyping
  - Kind bietet volles Eltern-Interface
- Overriding
  - im Kontext weitergeleiteter Nachrichten werden Methoden des Kindobjekts benutzt
  - ... anstelle entsprechender Methoden des Elternobjekts

Zusammengesetzte Beziehungen

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

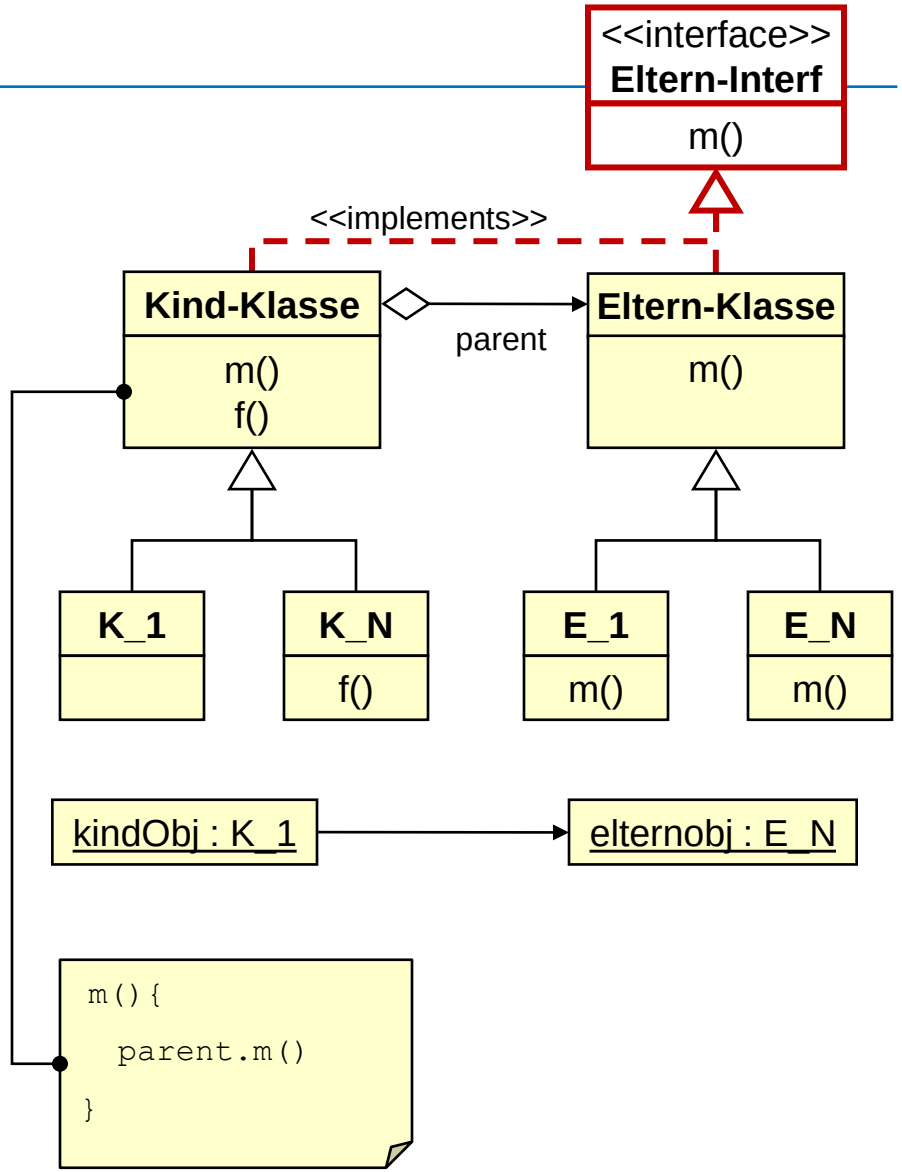
Implementierung der Subtypbeziehung: Variante 1

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



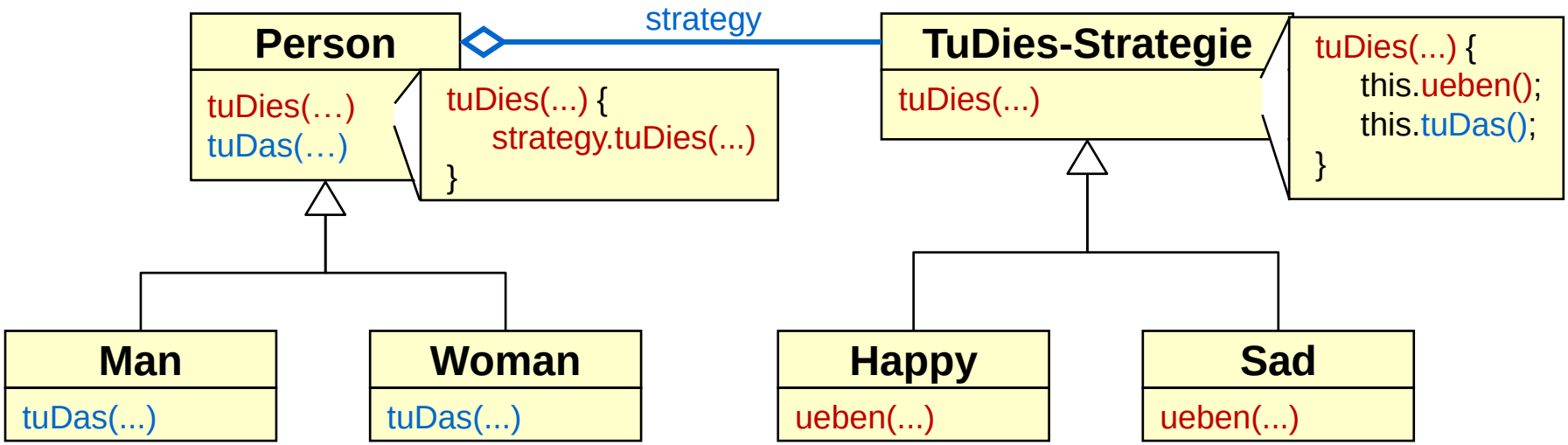
Implementierung der Subtypbeziehung: Variante 2

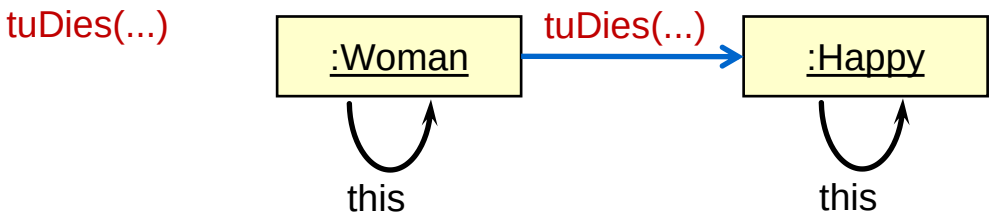
- Idee
  - Kind implementiert gleiches Interface wie die Elternklasse
  - Eltern-Interface anstatt Eltern-Klasse in Typdeklarationen verwenden
- Vorteil
  - keine Datenduplizierung
  - saubere Trennung
    - ⇨ Subtyping
    - ⇨ Code Wiederverwendung



Wie modelliert man Overriding?

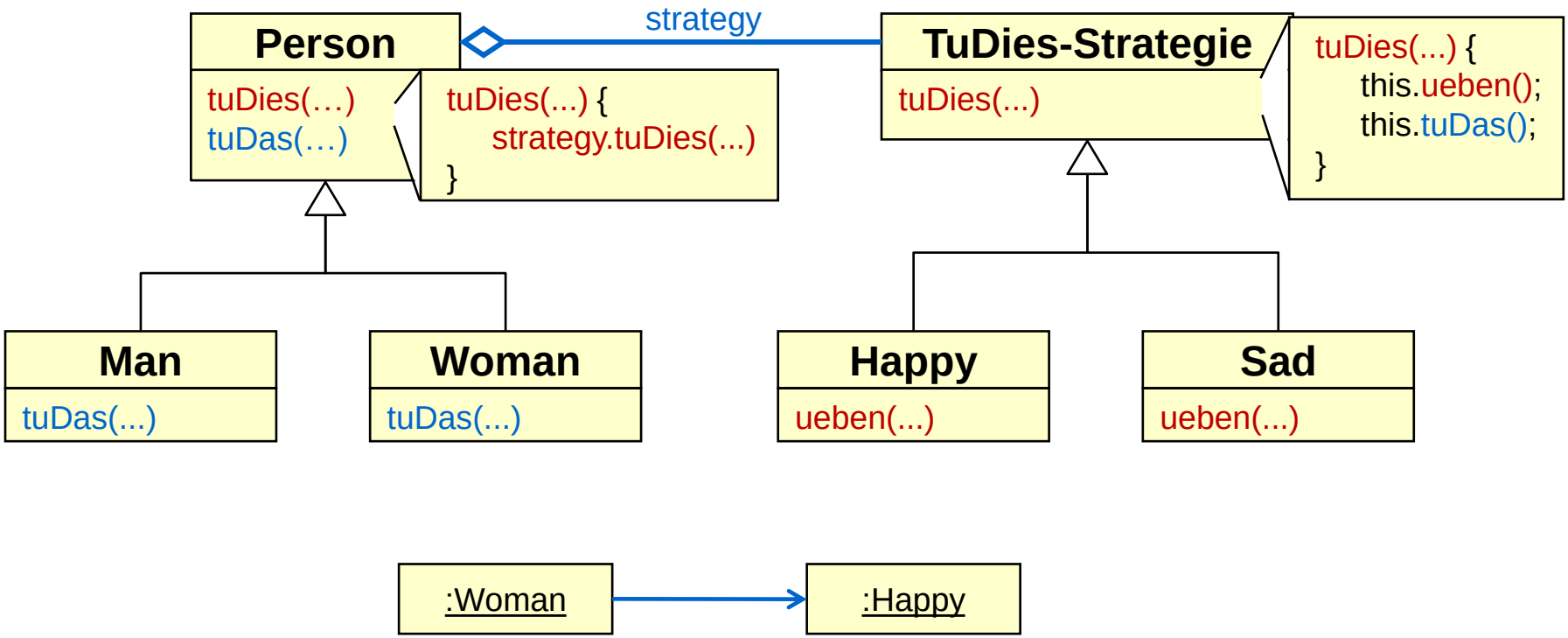
Strategy Pattern als Beispiel des Overriding-Problems



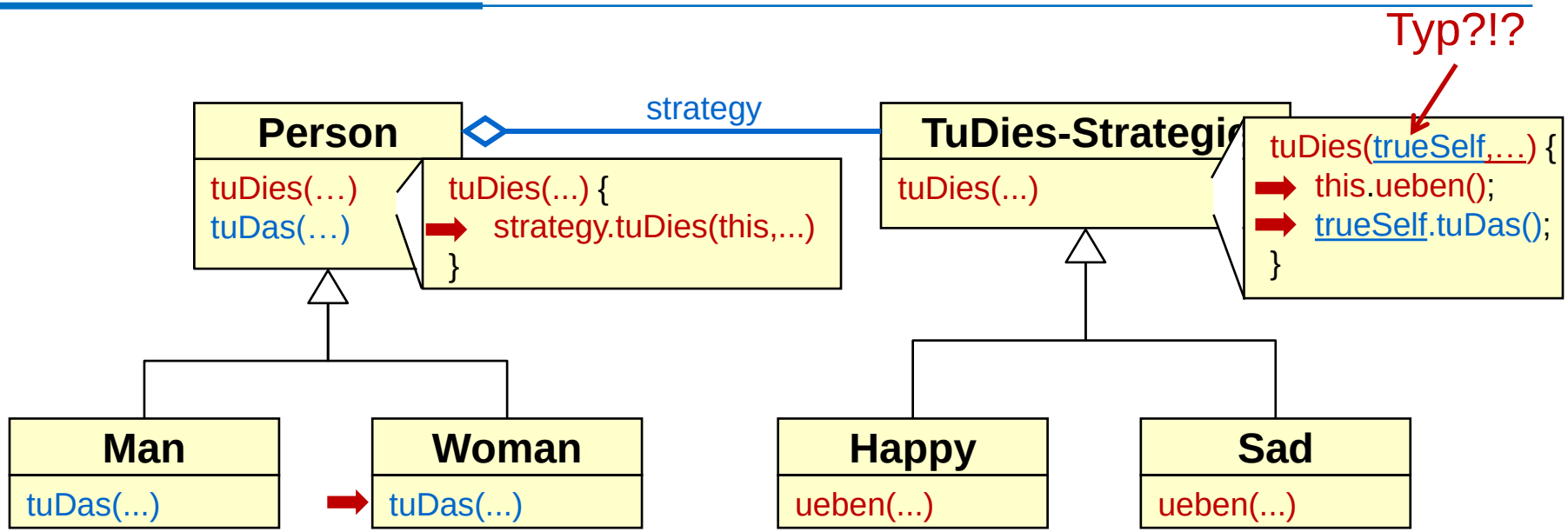


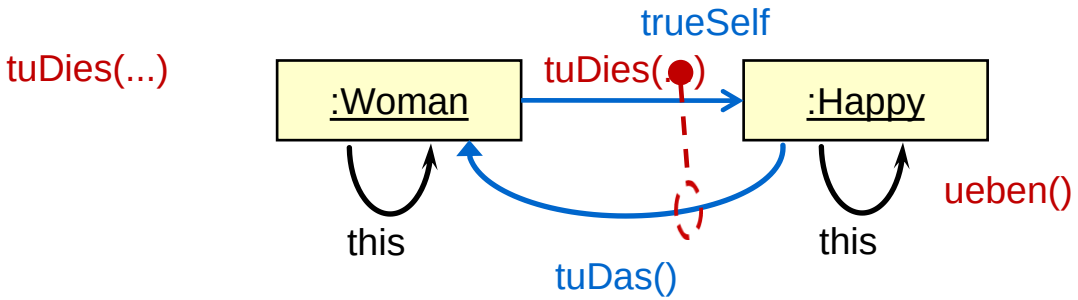
➔ **“Schizophrene Objekte”: verschiedenes “this” in ursprünglicher Nachricht und weitergeleiteter Nachricht, obwohl beide das gleiche konzeptionelle ein Objekt „fröhliche Frau“ ansprechen.**

Overriding-Simulation: "this" explizit machen



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 erfroderlich
  - Zusätzlicher trueSelf Parameter erfordert Anpassung der Aufrufe in allen Clients der Elternklasse

Alternative: Overriding-Simulation mit Instanzvarible

- 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

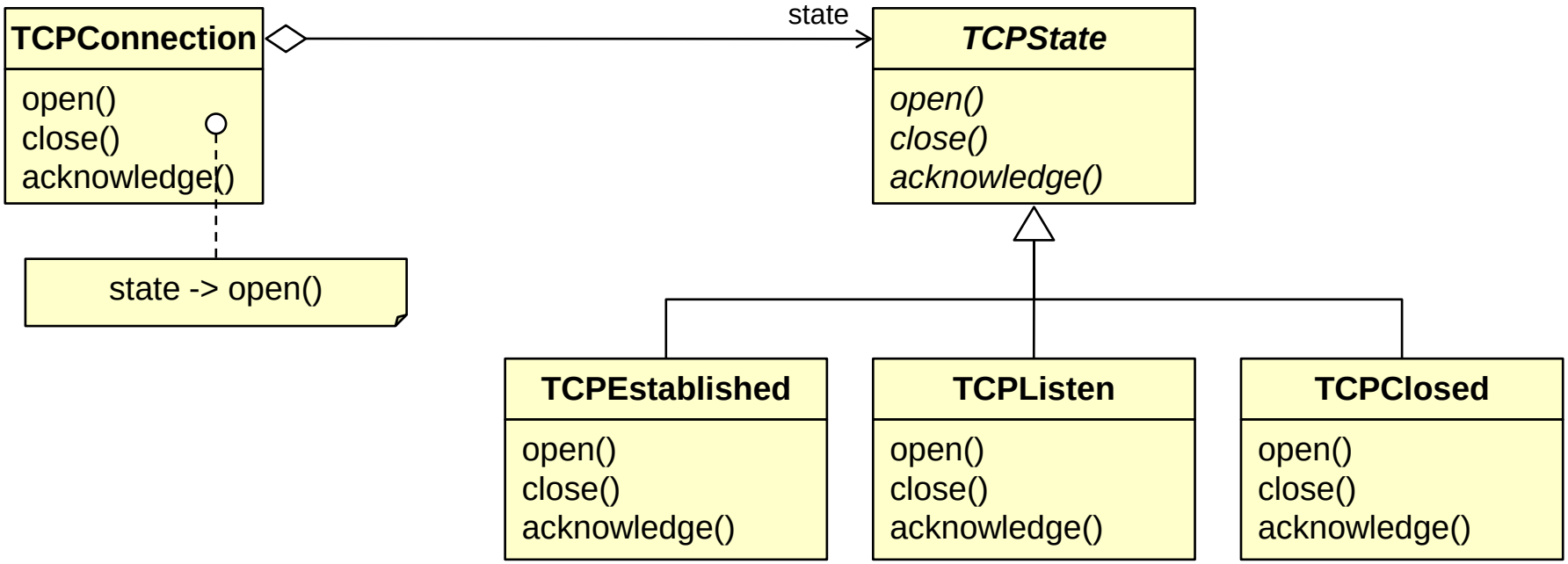
- Diese Techniken bilden eine „Pattern Language“ zur Simulation von (multipler) Vererbung auf Objektebene

Auf "Split Objects"-Idee basierende Patterns

2. Das State Pattern

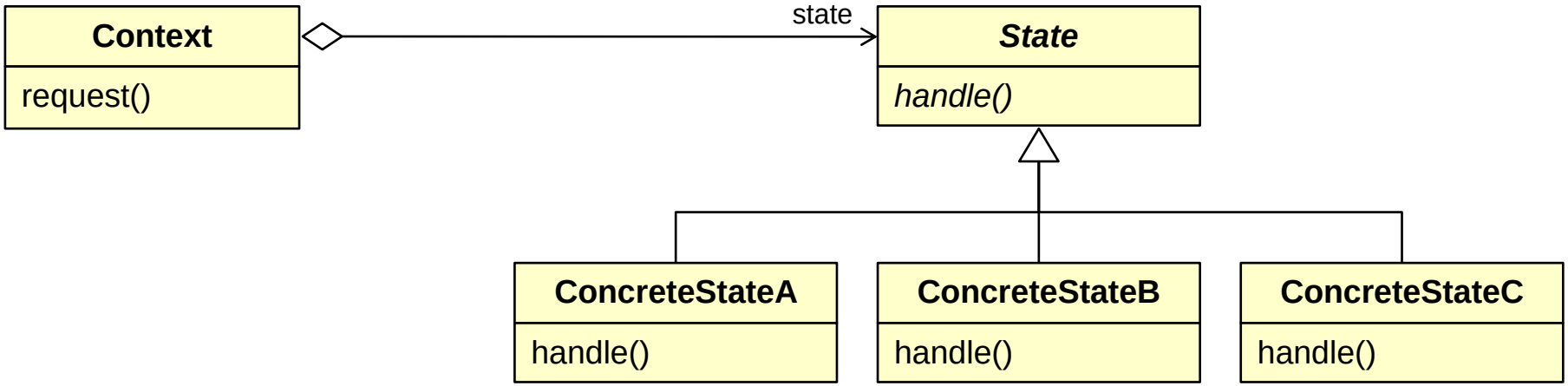
Das State Pattern

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

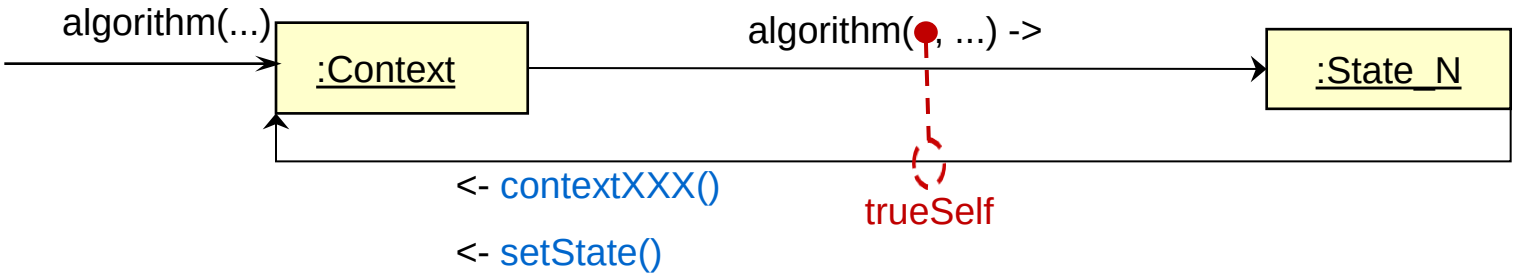
Das State Pattern: Konsequenzen

- 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

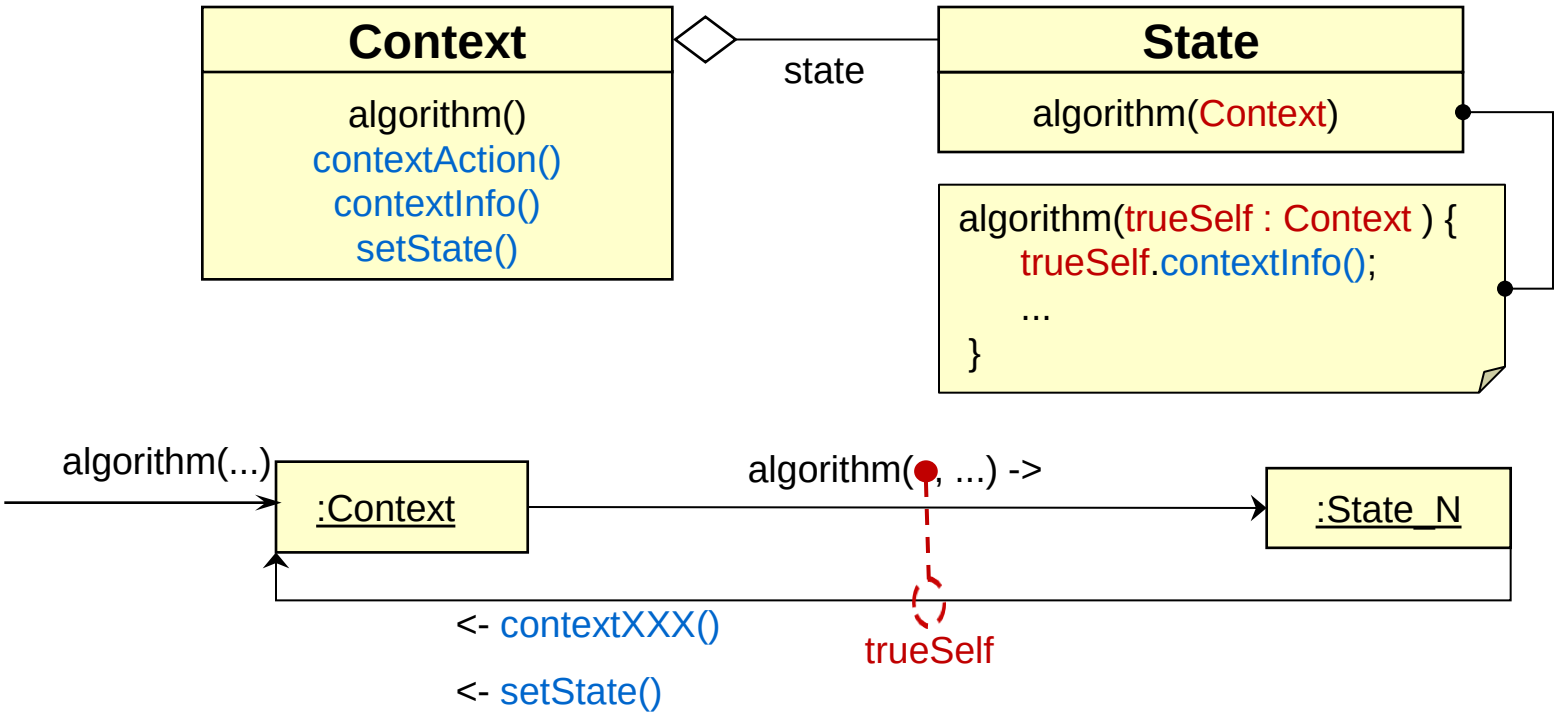
Implementation des Strategy und State Patterns

- 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

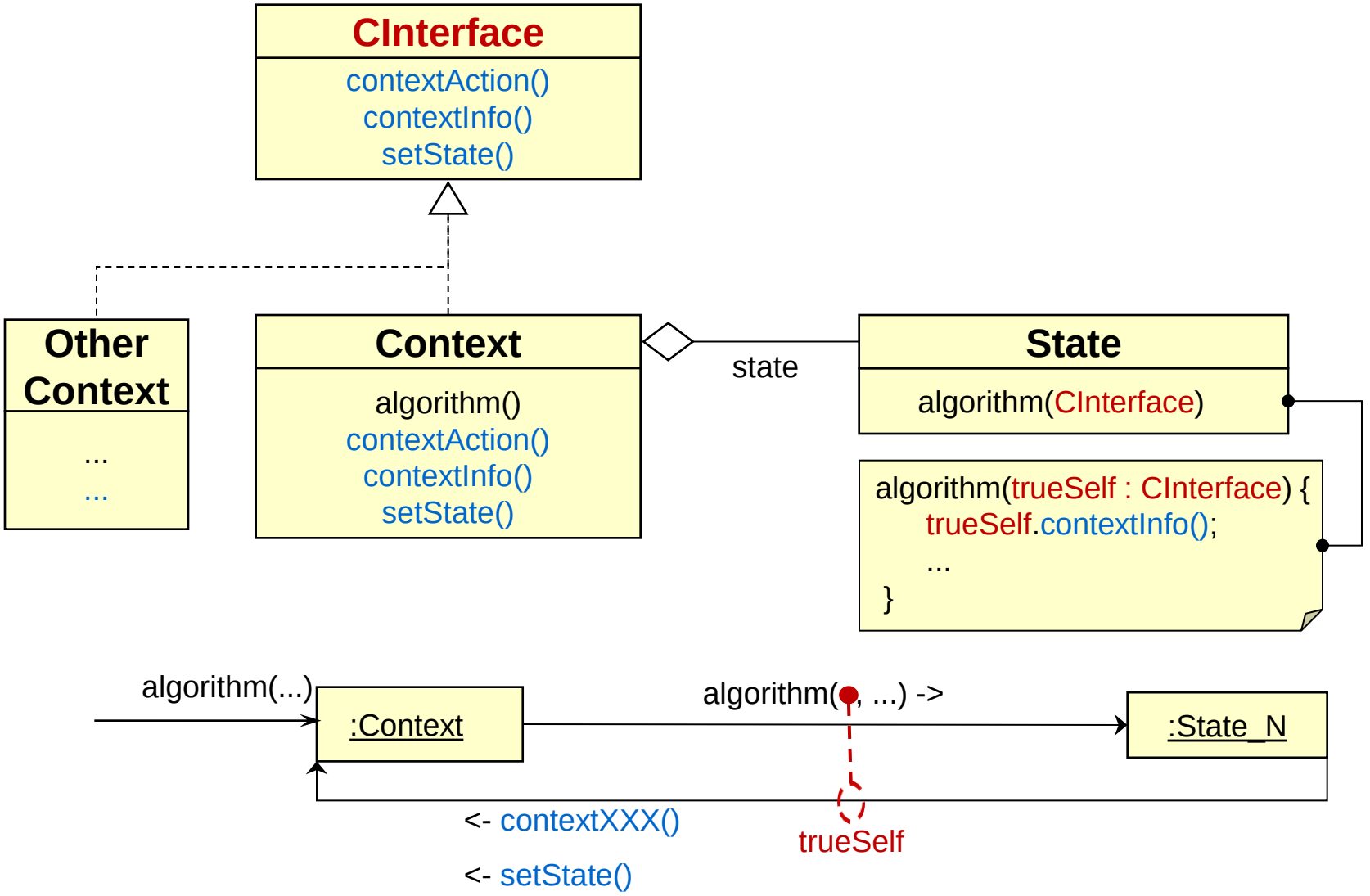
- Die letzten beiden Varianten
  - sind flexibler
  - sind ähnlich der vorgestellten Simulation von Overriding
  - werfen ähnliche Fragen auf
    - ⇨ Typ des übergebenen / gespeicherten "this"
    - ⇨ Schnittstelle von Kontext und Strategie bzw. State



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

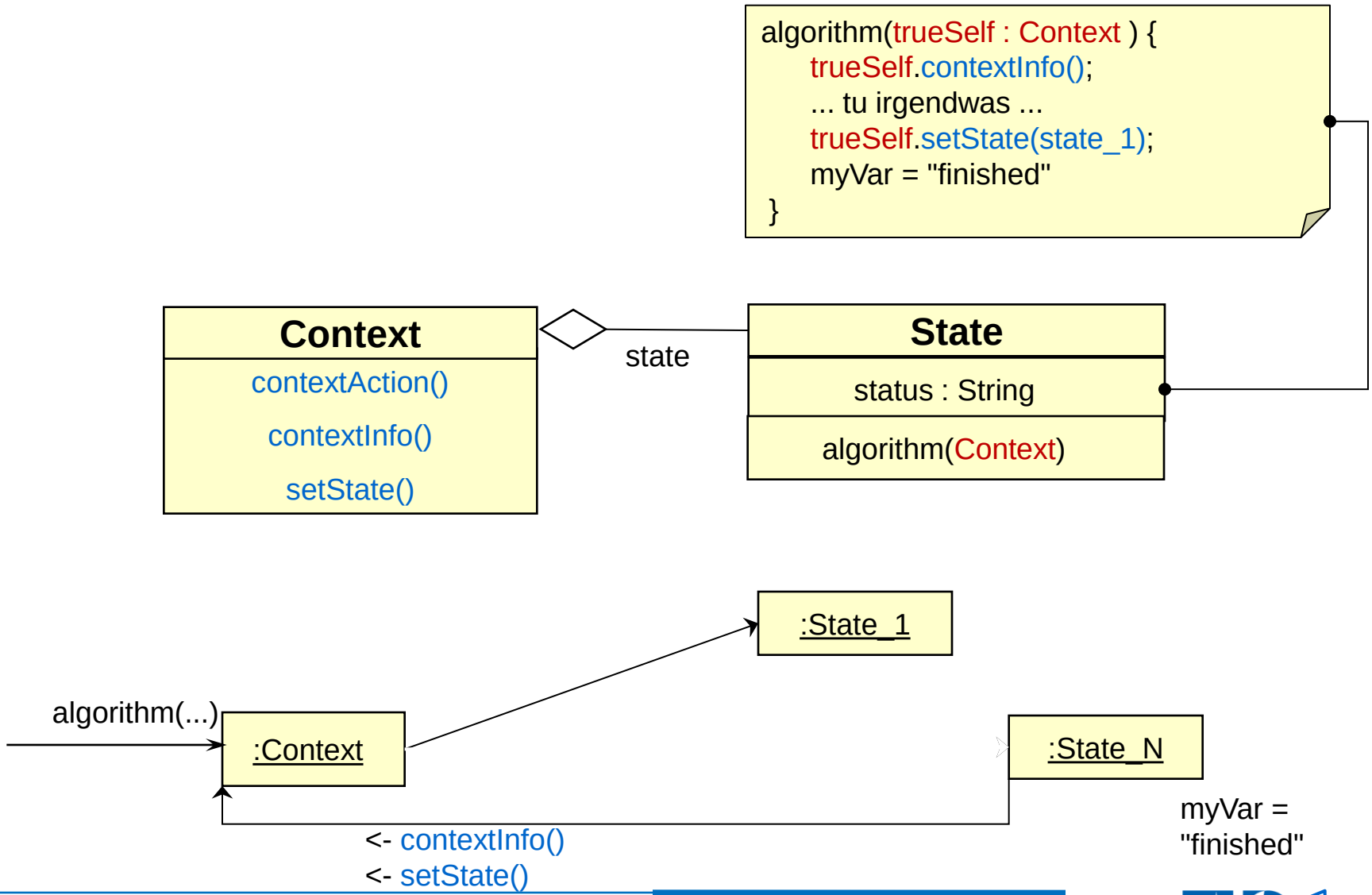
- trueSelf : ContextInterface
  - verschiedene Context-Typen die das ContextInterface implementieren sind möglich
  - flexibler
  - Umstellung Variante 1 → Variante 2 ist leicht:
    - ⇨ begrenzte Änderung:
    - ⇨ trueSelf ist nur in der Elternhierarchie und dem Context bekannt
    - ⇨ mechanische Änderung:
    - ⇨ 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

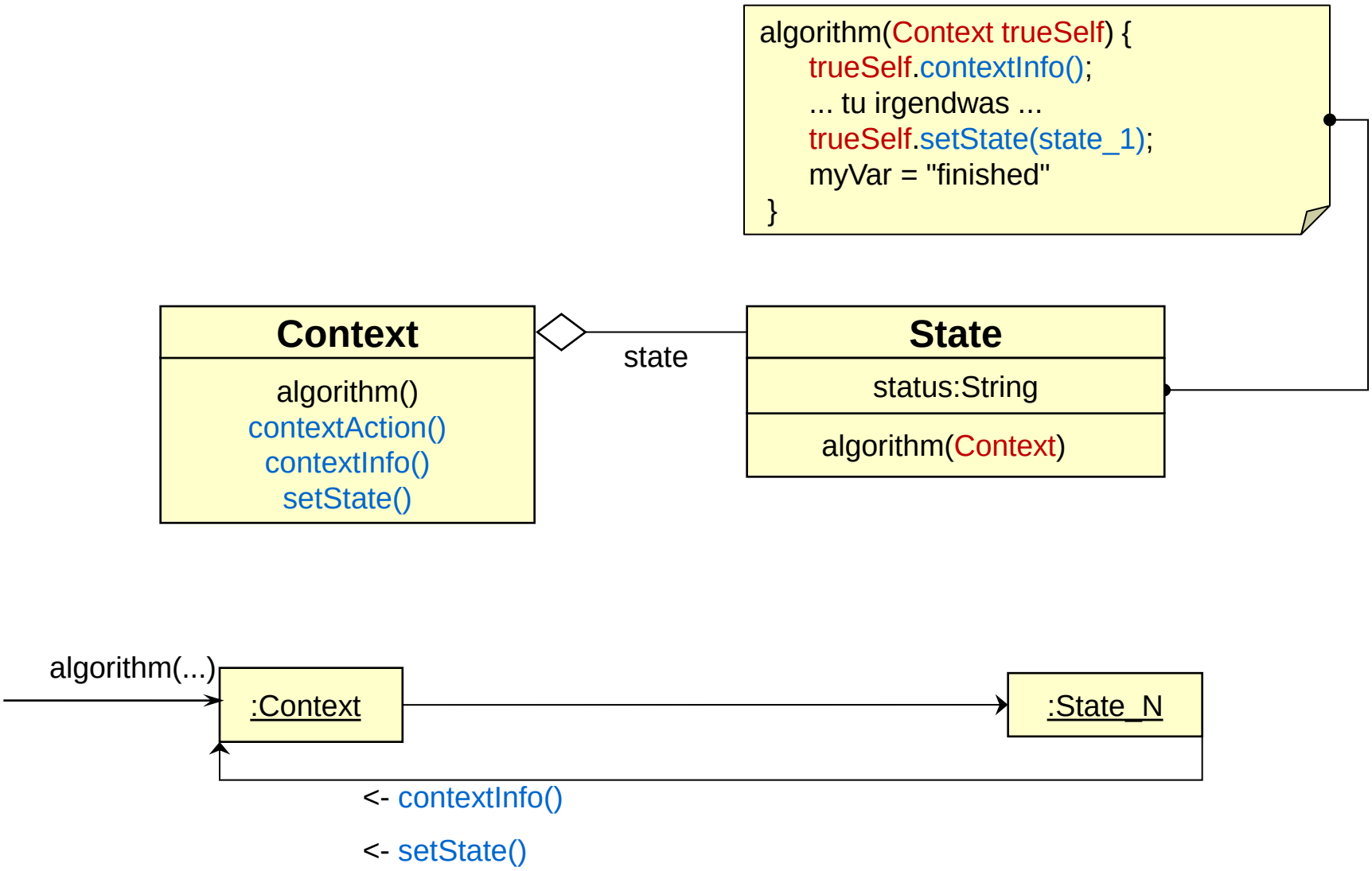
- Bei Strategy-Pattern
  - meist extern veranlasst
  - Z.B. Anwendung bekommt mit, dass Verbindungsqualität schlecht ist, und weist den Media-Player an, einen schnelleren aber niedrig-qualitativen Video-Rendering-Algorithmus zu nutzen.

- 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

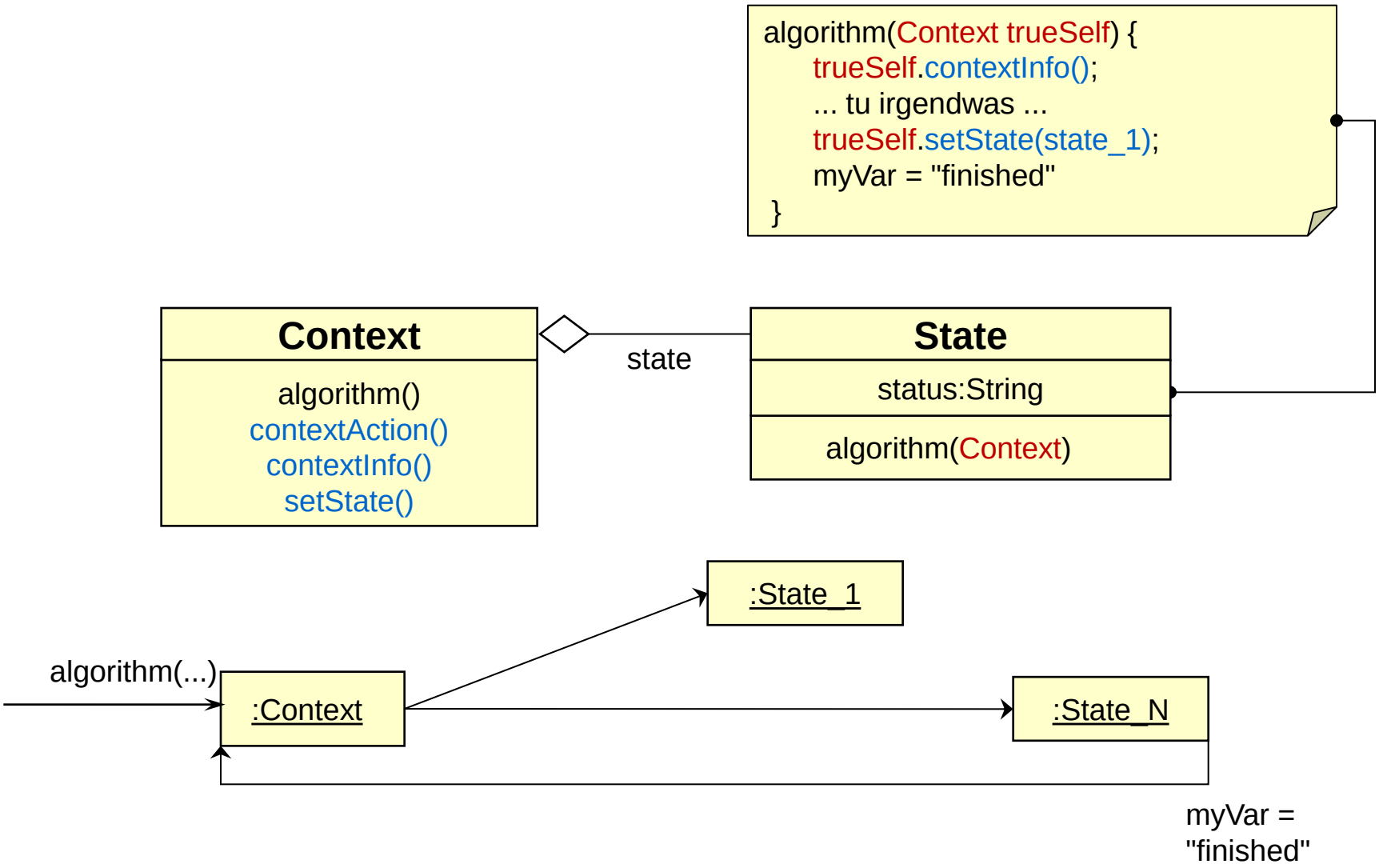
Beispiel: Verfrühtes Wechseln des Elternobjektes



Beispiel: Verfrühtes Wechseln des Elternobjektes (1)



Beispiel: Verfrühtes Wechseln des Elternobjektes (2)



Auf "Split Objects"-Idee basierende Patterns

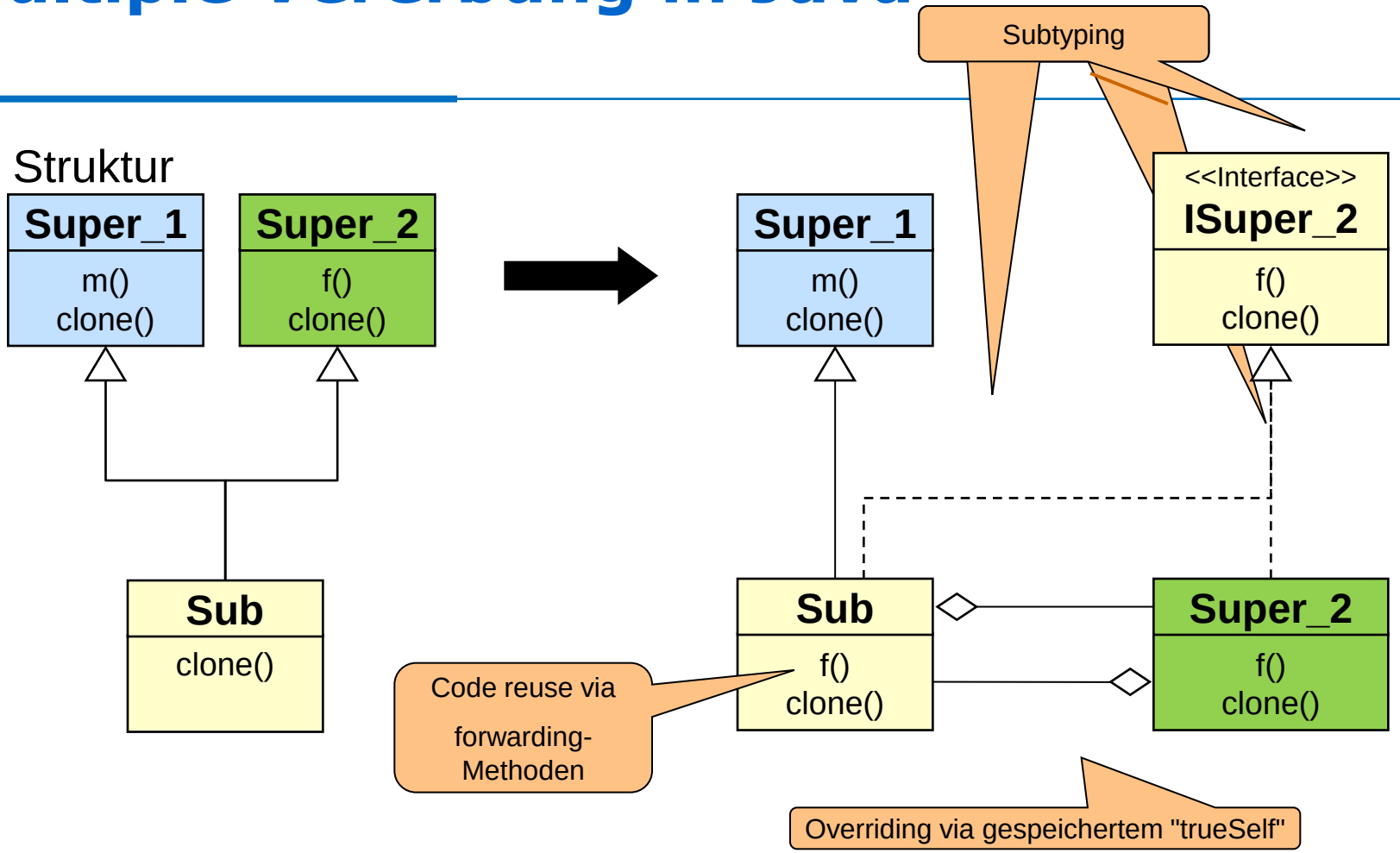
3. Simulation von Multipler Vererbung in Java

Multiple Vererbung in Java

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

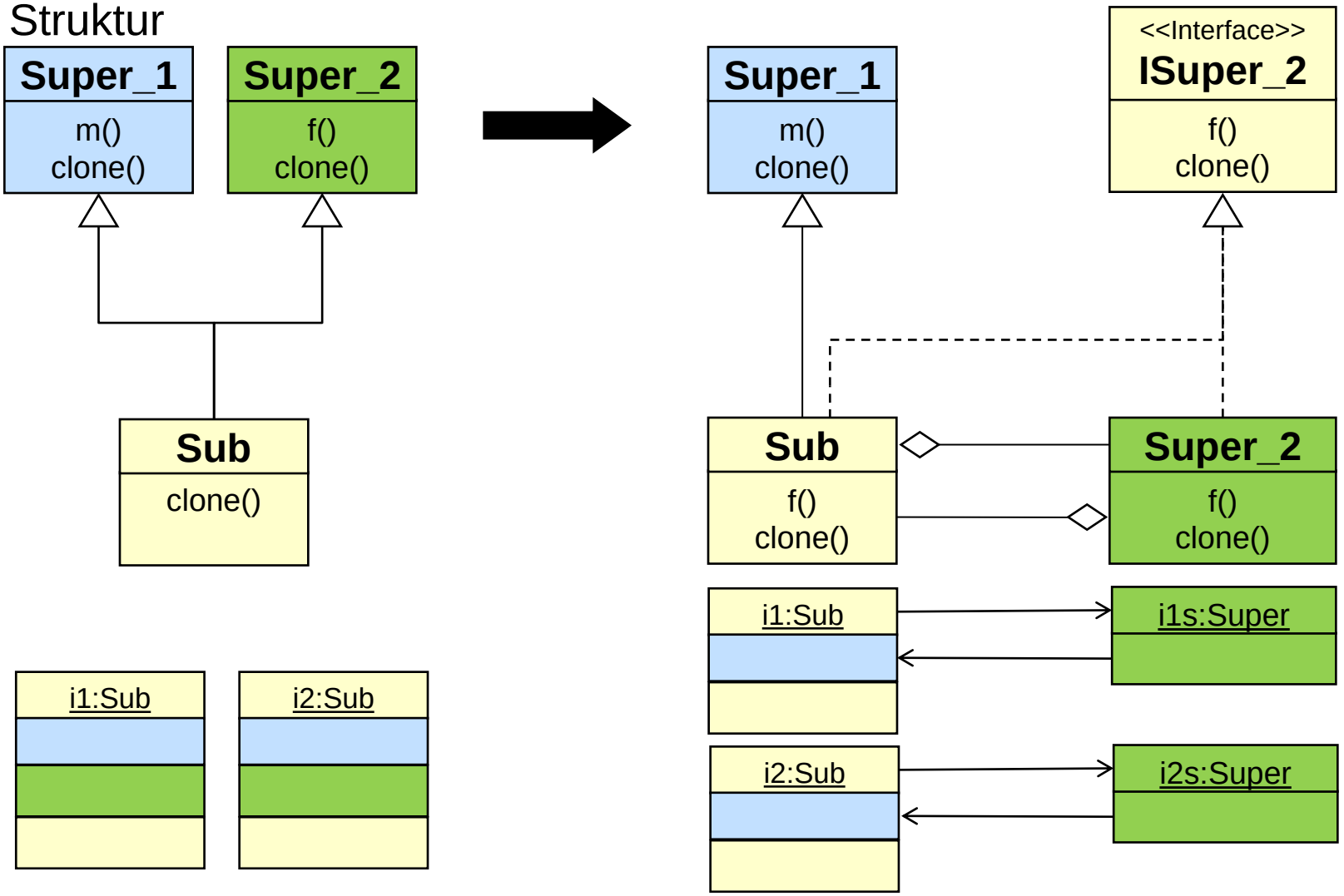
Multiple Vererbung in Java

- Struktur



Multiple Vererbung in Java

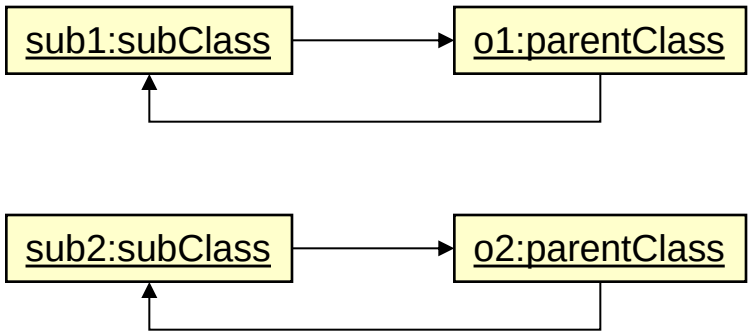
- Struktur



Simulation multipler Vererbung versus Decorator

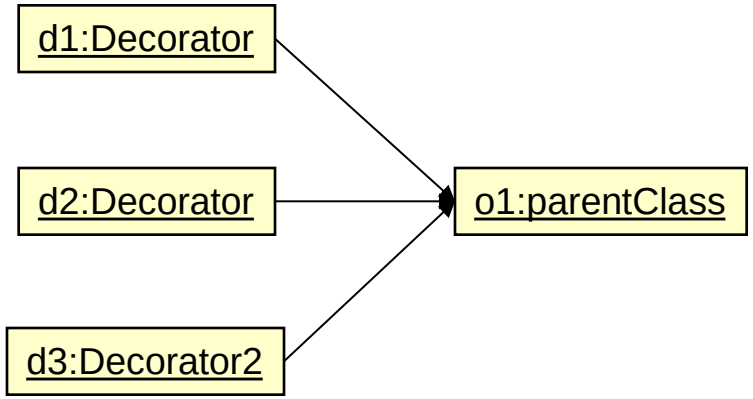
Simulation Multipler Vererbung

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



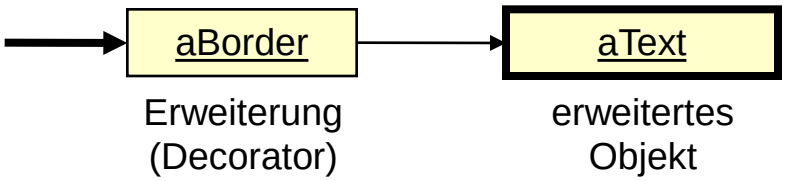
Decorator Pattern

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



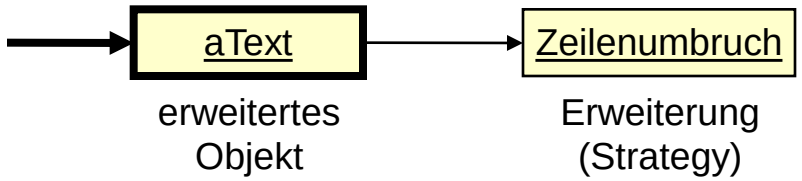
Decorator versus Strategy

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

Erweiterung und Restrukturierung

- Umsetzung fortgeschrittener UML-Konzepte in „Kern-UML“
  - Assotiationen

Assoziationen und Assoziationsklassen

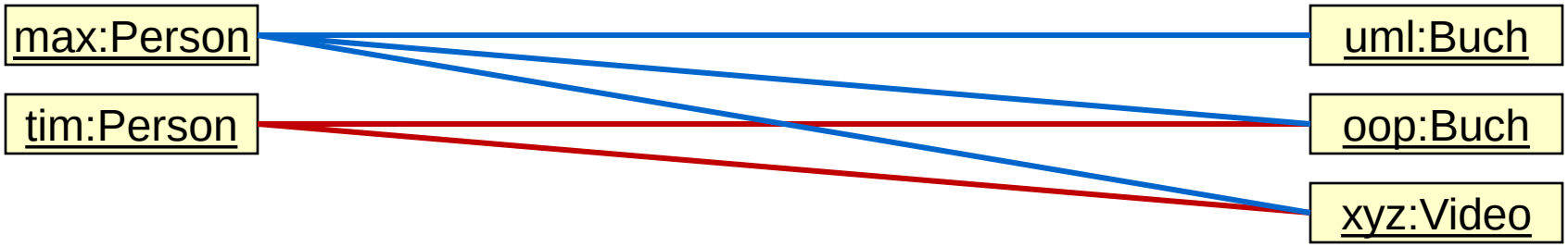
Umsetzung von Assoziationen je nach ihrer Multiplizität und Navigierbarkeit

Umsetzung qualifizierter Assoziationen

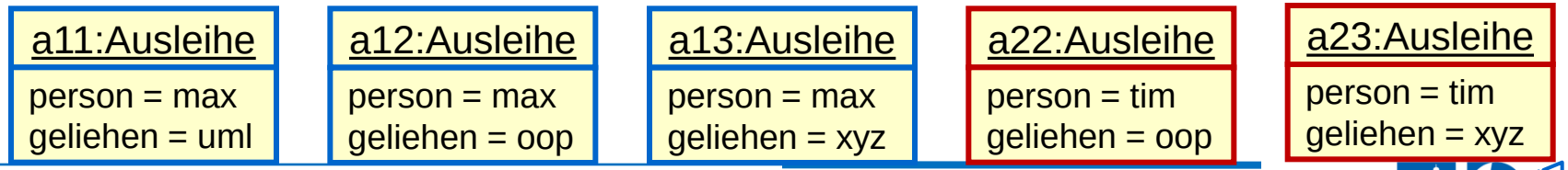
Assoziationen sind implizite Klassen!

- Eine Assoziation

- ... ihre Elemente



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

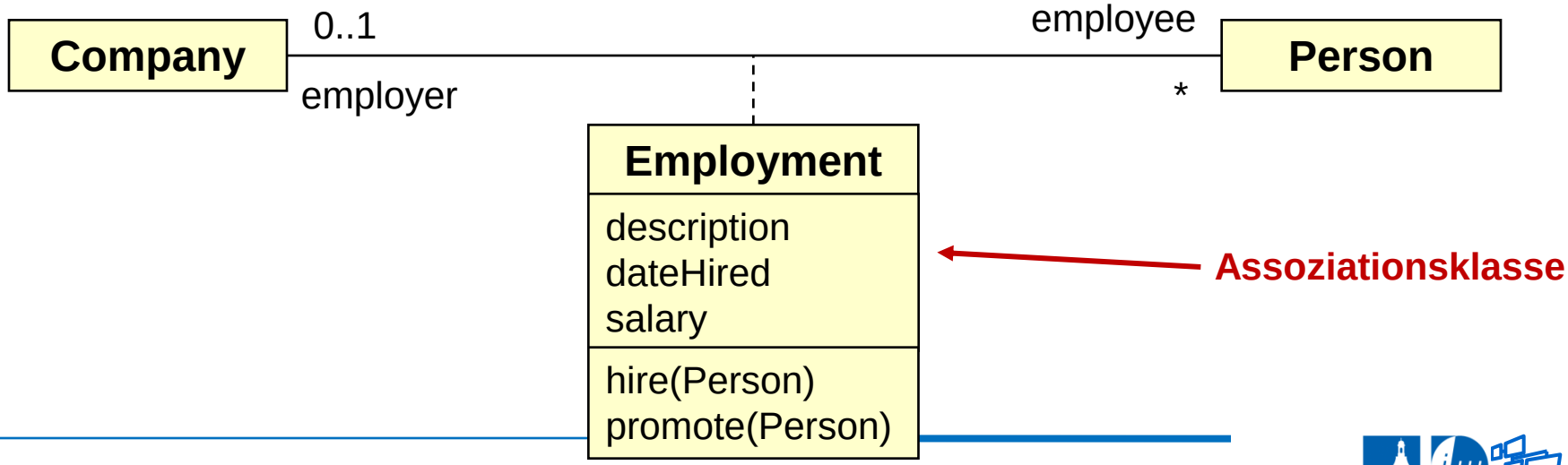


UML, statisches Modell: Assoziationsklassen (1)

- Modellieren komplexe Assoziationen zwischen Klassen
  - Machen die implizite Klasse explizit wenn zusätzliche Attribute und Operationen der Beziehung modelliert werden müssen.

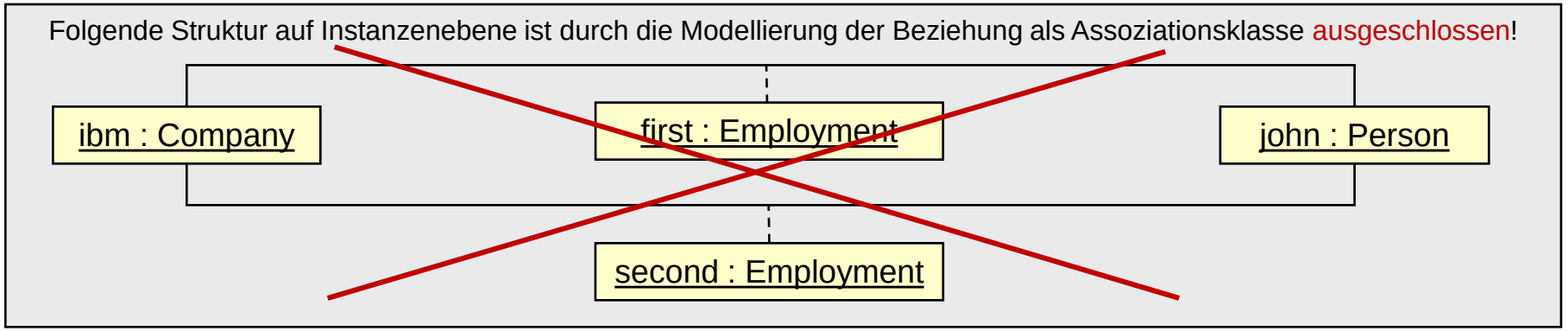
- Beispiel
  - Das Datum der Einstellung (Attribut "dateHired")
  - Der Vorgang der Einstellung (Operation "hire")

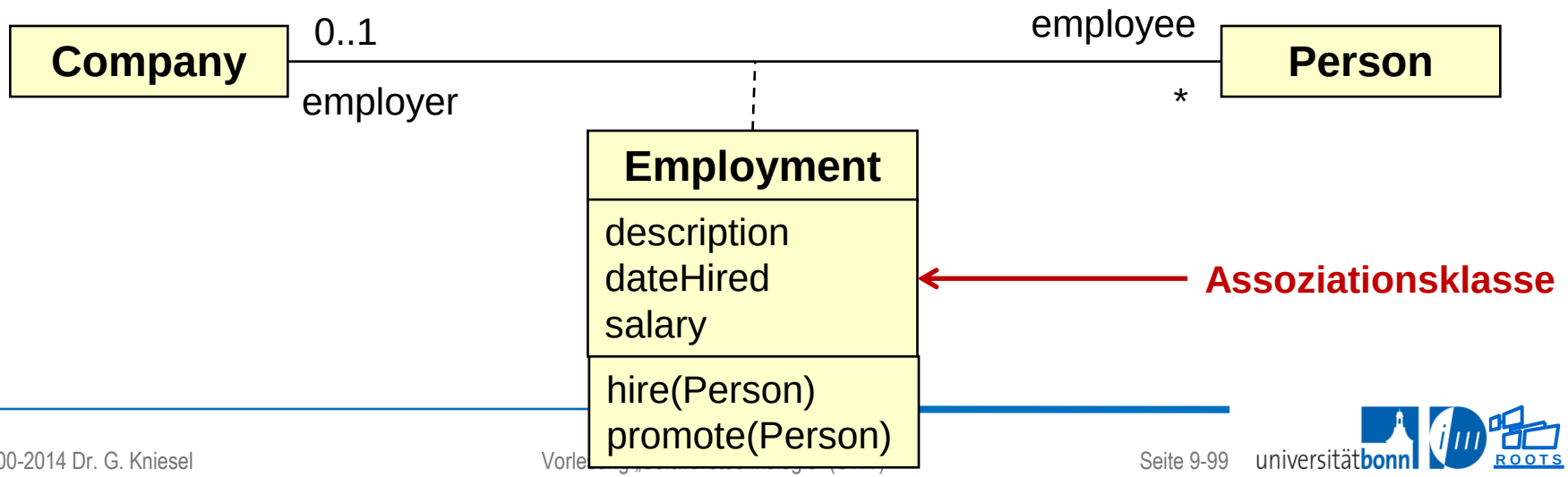
  ... sind Eigenschaften der Employment-Beziehung



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

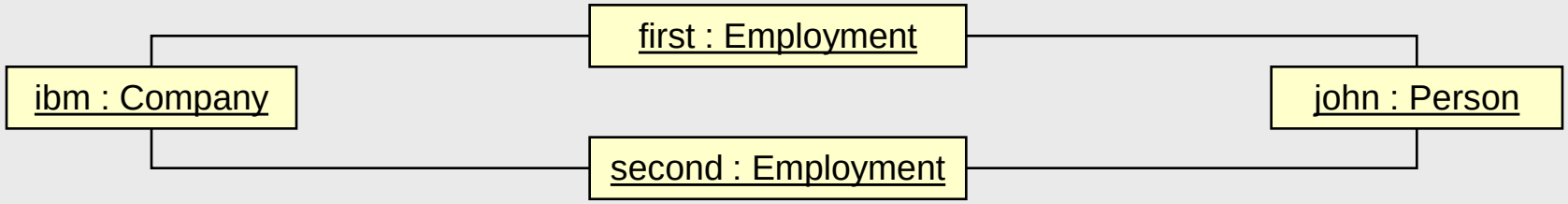


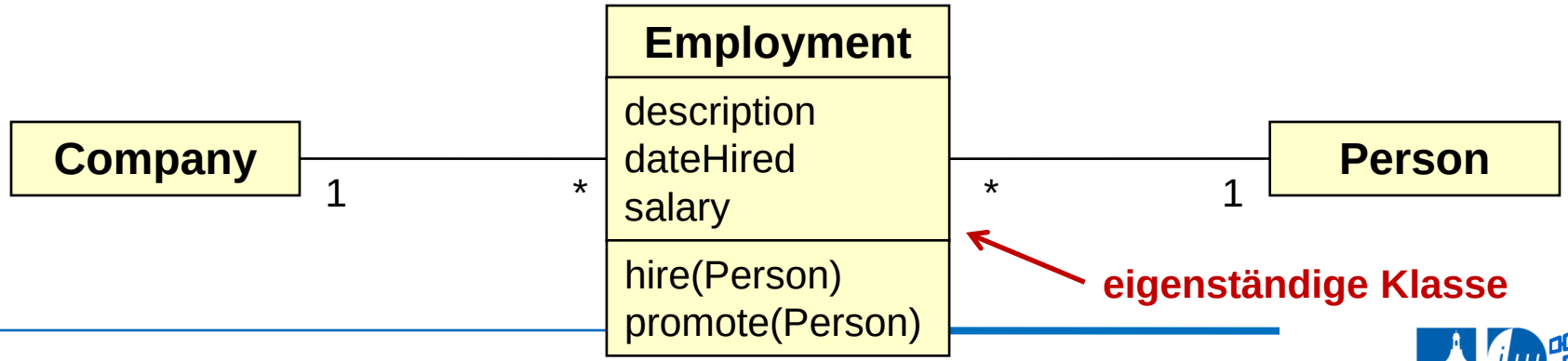


UML: Assoziation als eigenständige Klasse

- 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

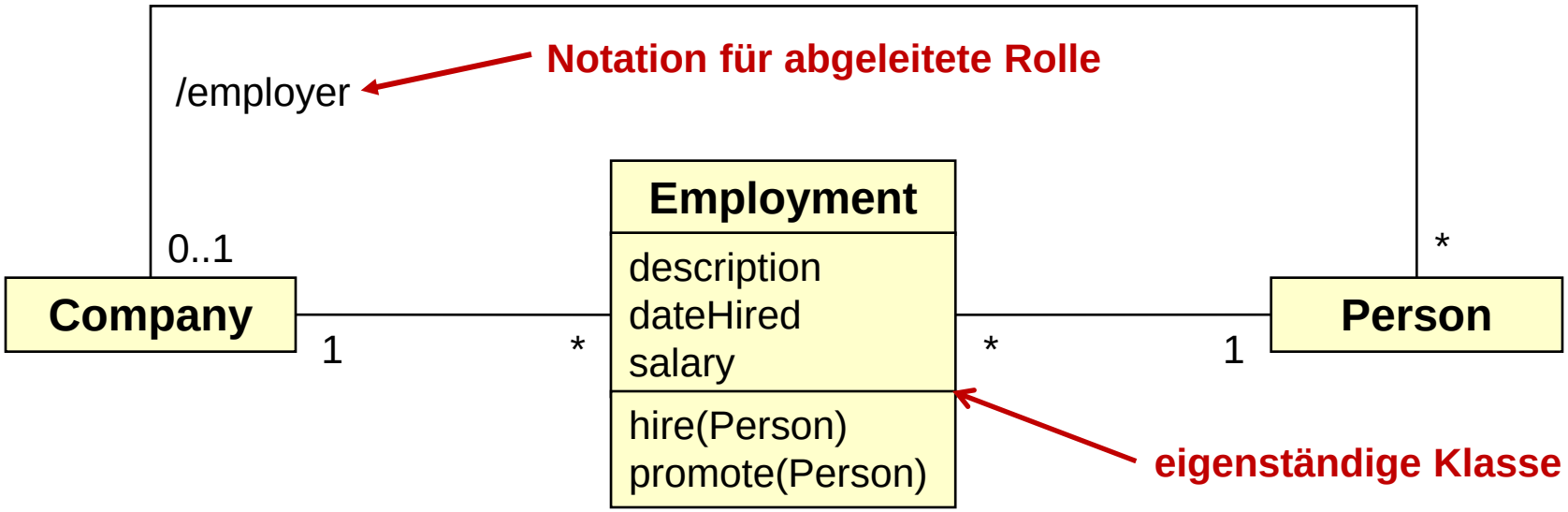
Folgende Struktur auf Instanzenebene ist durch die Modellierung der Beziehung als eigenständige Klasse möglich!





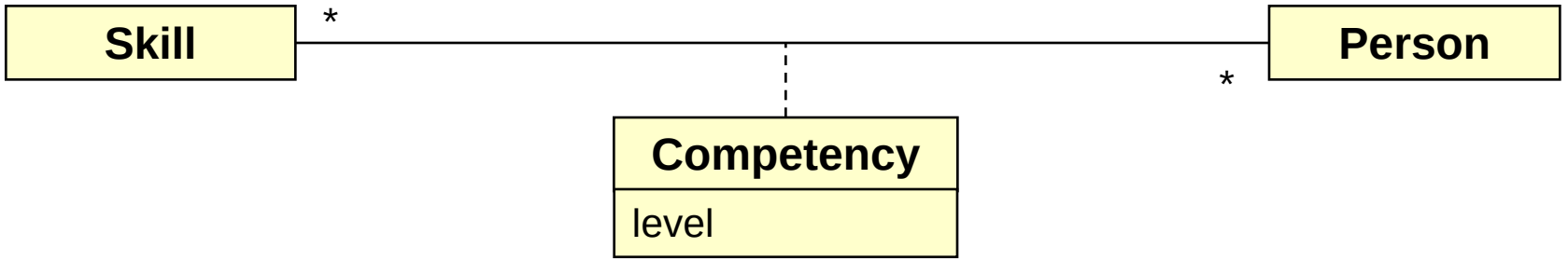
UML: Assoziation als eigenständige Klasse (2)

- 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"

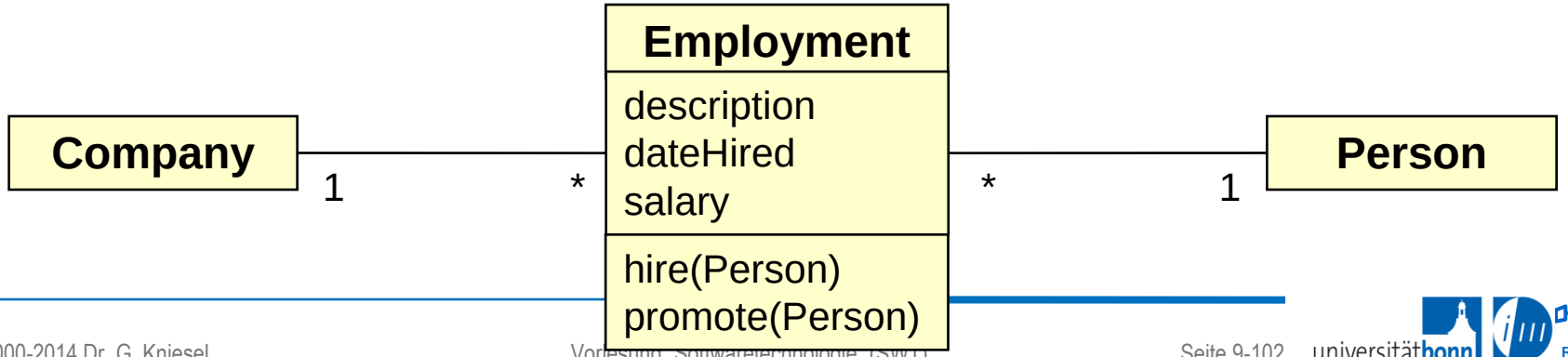


UML: Vergleich der Alternativen

- 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

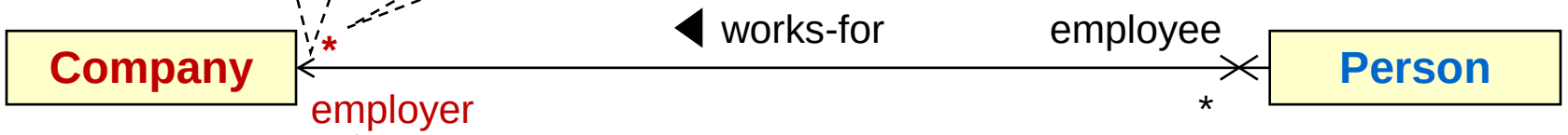
1:N, unidirektional oder bidirektional

N:M bidirektional

UML: Assoziationen (Implementations-Sicht)

Navigations-Hinweis
- in Implementations-Sicht
- Hier: direkter Zugriff nur von Person auf Company

Beziehungen mit unbegrenzter Kardinalität werden durch passende "Collection"-Typen implementiert (Array, Vector, etc.).



Employer-Rolle von **Company** in Beziehung zu **Person** findet sich oft als Variablenname in der Implementierung von **Person** wieder:

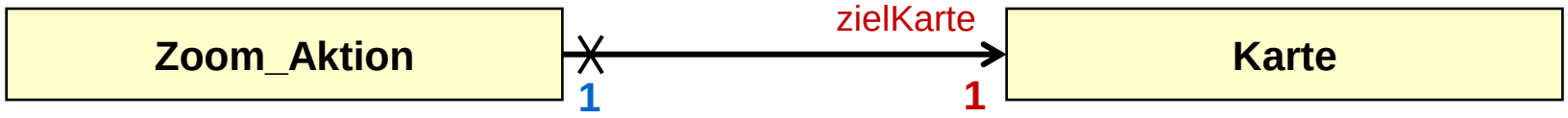
```
class Person {
    Company[] employers;
}
```

Varianten je nach Navigierbarkeit (uni-/bidirektional) und Kardinalität (1/*)

→ s. nächste Folien.

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

Objektentwurfsmodell vor der Transformation

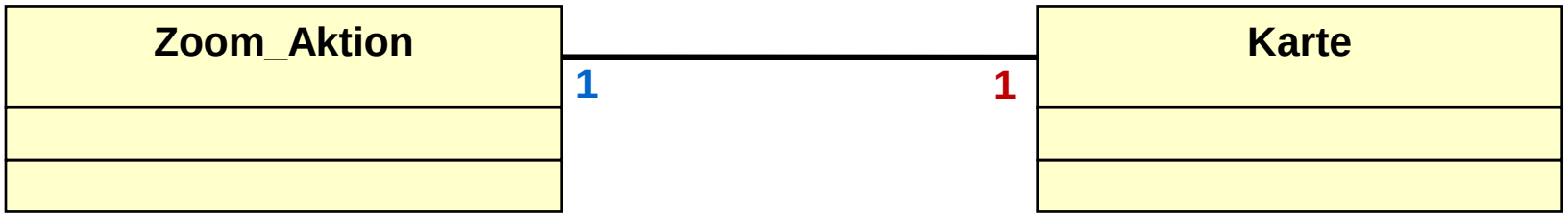


Objektentwurfsmodell nach der Transformation

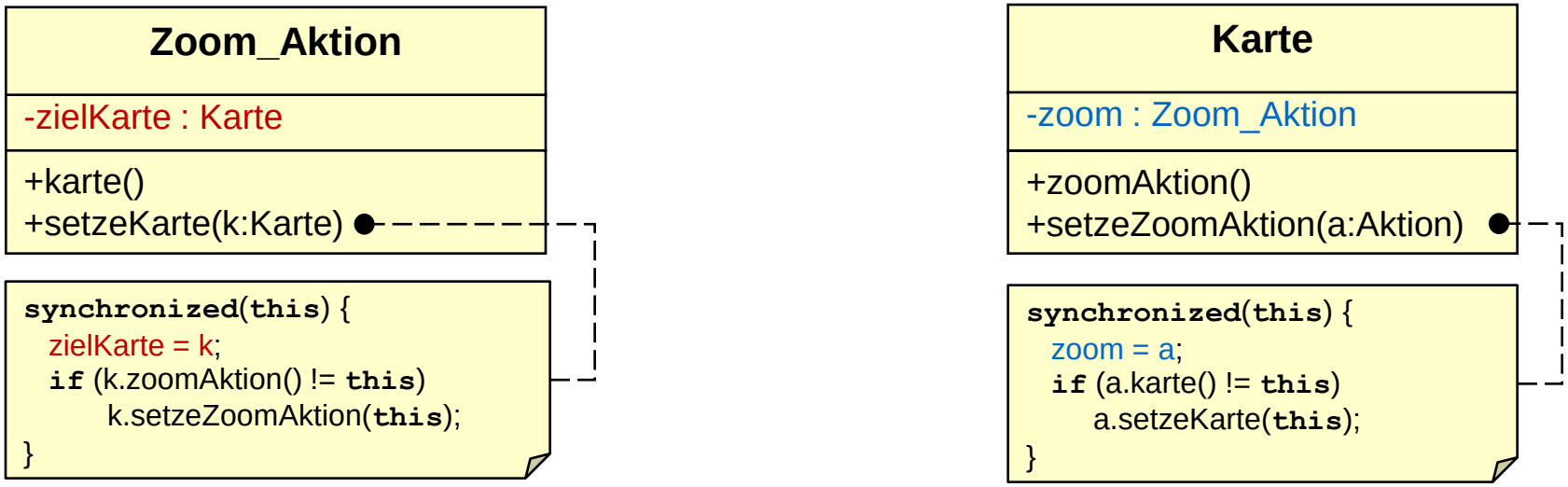
- Unidirektionale Assoziationen sind einfach:
  - Assoziation durch Instanzvariable in der „referenzierenden“ Klasse ersetzen.
  - Falls die Kardinalität auf der „referenzierten“ Seite N >1 ist, sollte die Instanzvariable eine Collection sein.
  - Falls die Kardinalität auf der „ referenzierten“ Seite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

Bidirektionale *1:1* Assoziation

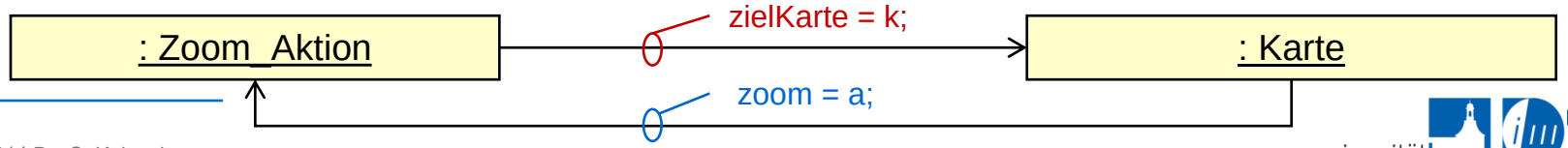
Objektentwurfsmodell vor der Transformation



Objektentwurfsmodell nach der Transformation

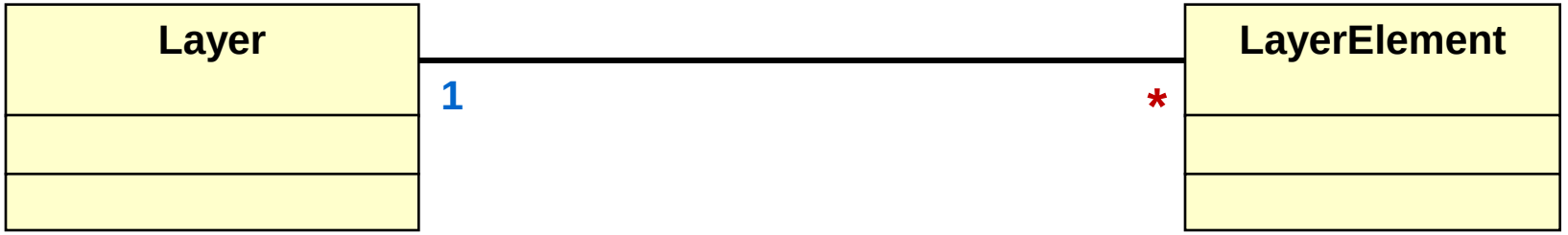


Beispiel-Instanziierung nach der Transformation



Bidirektionale 1:N Assoziation

Objektentwurfsmodell vor der Transformation



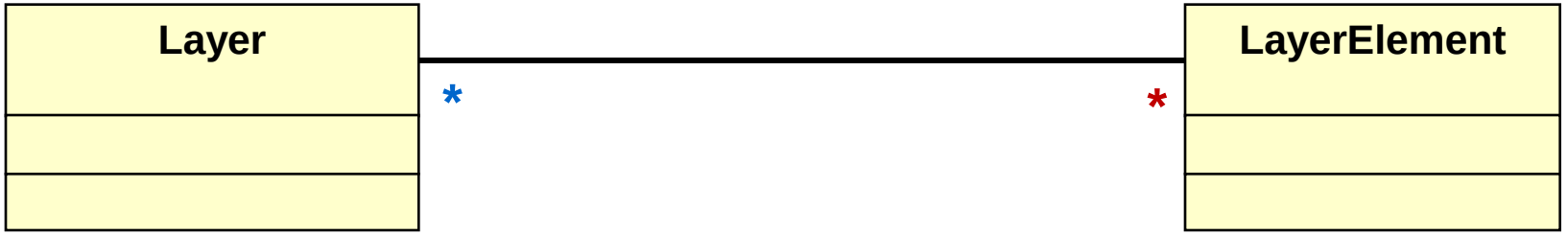
Objektentwurfsmodell nach der Transformation

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



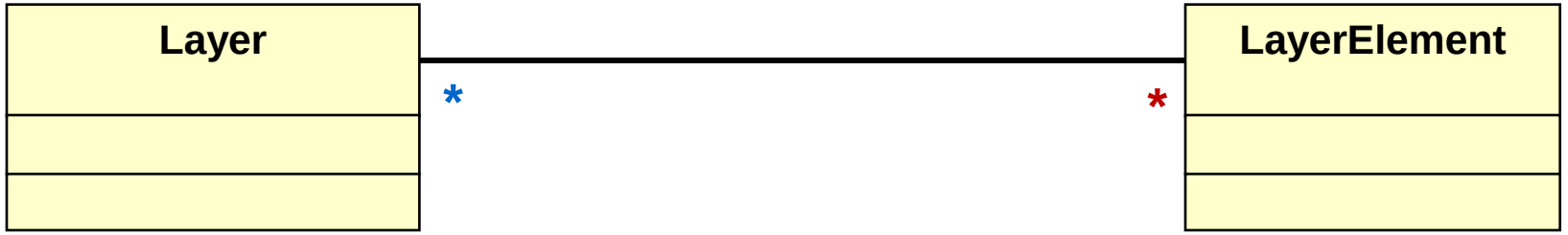
Objektentwurfsmodell nach der Transformation

Beispiel-Instanziierung nach der Transformation

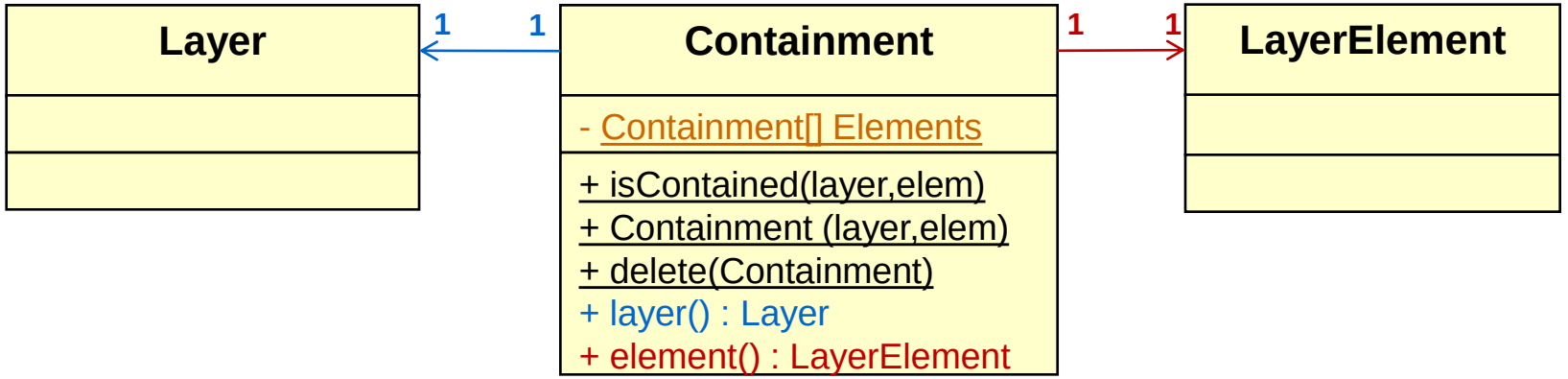
- Problem 1: Deadlockfreies setzen von Rückreferenzen nicht mehr trivial.
- Problem 2 (aller bisherigen Varianten): Die Beziehung wird fest in den Klassen verdrahtet. Das schafft zusätzliche Abhängigkeiten.

Bidirektionale N:M Assoziation (Assoziation als Klasse)

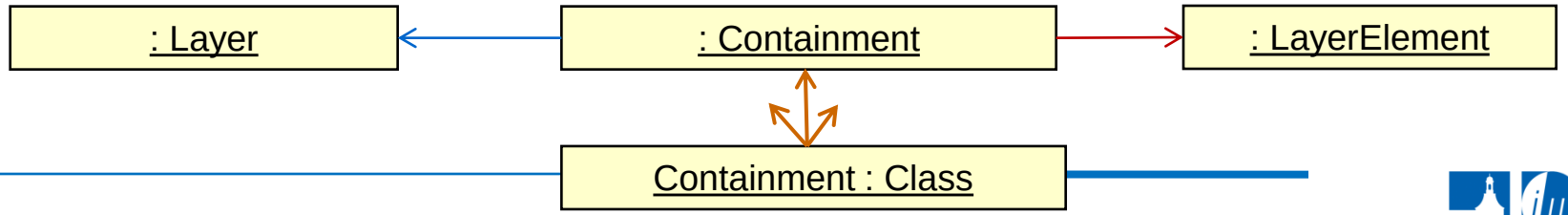
Objektentwurfsmodell vor der Transformation



Objektentwurfsmodell nach der Transformation

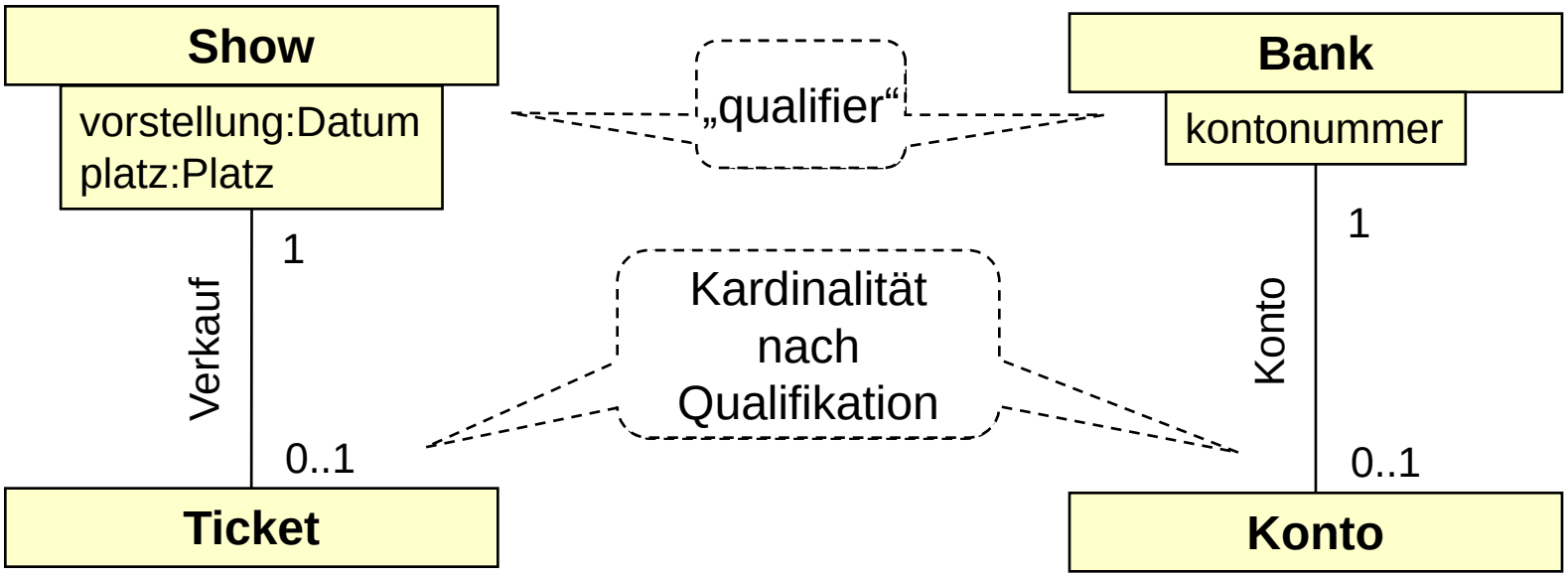


Beispiel-Instanziierung nach der Transformation



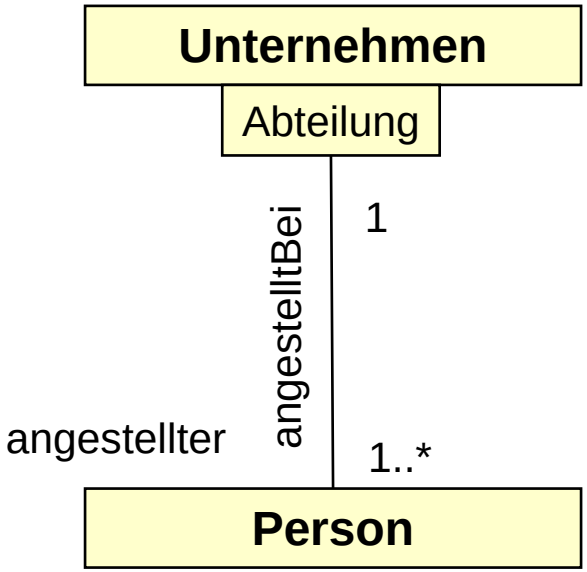
Qualifizierte Assoziation

- 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

- Qualifier können auch partielle Indizes sein
  - Sie bestimmen evtl. nicht ein einziges Element, sondern eine (Teil)Menge
  - Beispiel:
    - ⇨ In einer Abteilung können mehrere Angestellte beschäftigt sein



Qualifizierte Assoziation: Umsetzung

Objektentwurfsmodell vor der Transformation

Objektentwurfsmodell nach der Transformation

Bank
-Konten:Hashtable
+konten() +neuesKonto(kontonr:Kontonummer) +kontoLöschen(kontonr:Kontonummer)

Person
- bank : Bank
+getBank() +setBank(B:Bank)

- Der Qualifier „kontonr“ 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
   - Umsetzung von UML_{High} als Patterns
4. Optimierung des Objektmodells
   - Umformen des Objektmodells unter Berücksichtigung von Performanzkriterien, wie Reaktionszeit oder Speicherbedarf.

Entwurfsoptimierung: Vorsicht!

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

- Erst „hot spots“ identifizieren!
  - Welche Programmstellen verbrauchen die meiste Zeit?
  - Werkzeuge: „Profiler“, zB
    - ⇨ Eclipse TPTP (generische Schnittstelle)
    - ⇨ JProfiler (kommerzielles Werkzeug)

Aktivitäten während der Entwurfsoptimierung

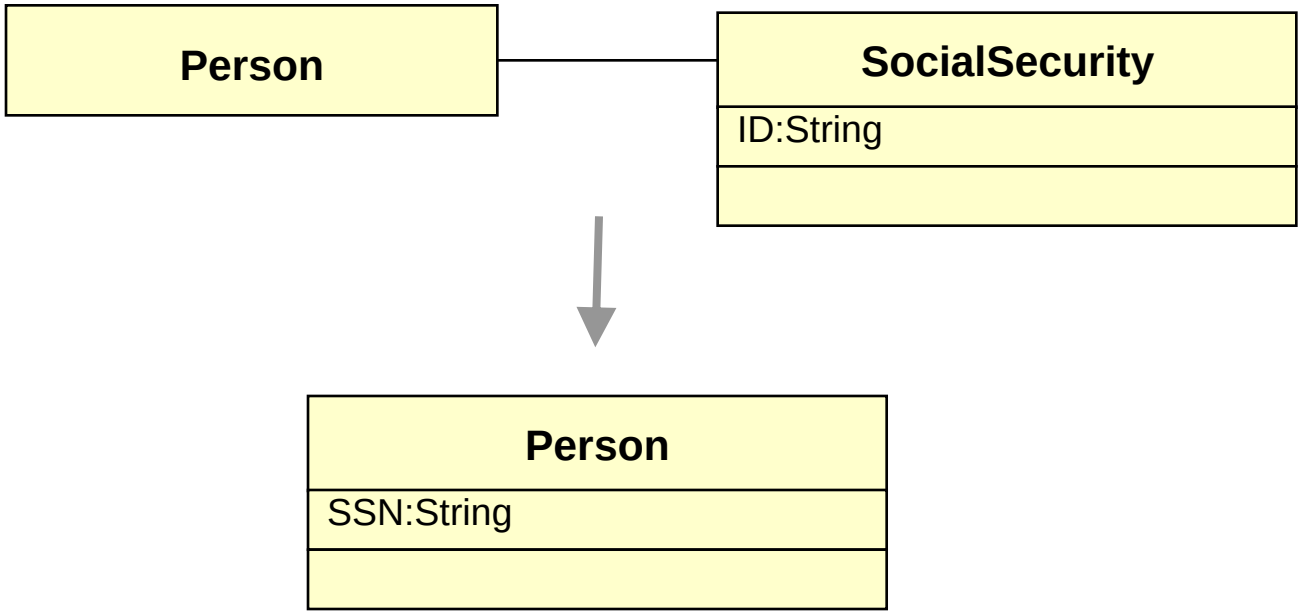
1. Umordnen der Ausführungsreihenfolge
   - Eliminiere „tote Pfade“ so früh wie möglich. (Verwende Wissen über Verteilung, Frequenz der Pfadtraversierung)
   - Grenze Suche so früh wie möglich ein
   - Überprüfe ob die Ausführungsreihenfolge von Schleifen umgekehrt werden sollte
2. Hinzufügen redundanter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

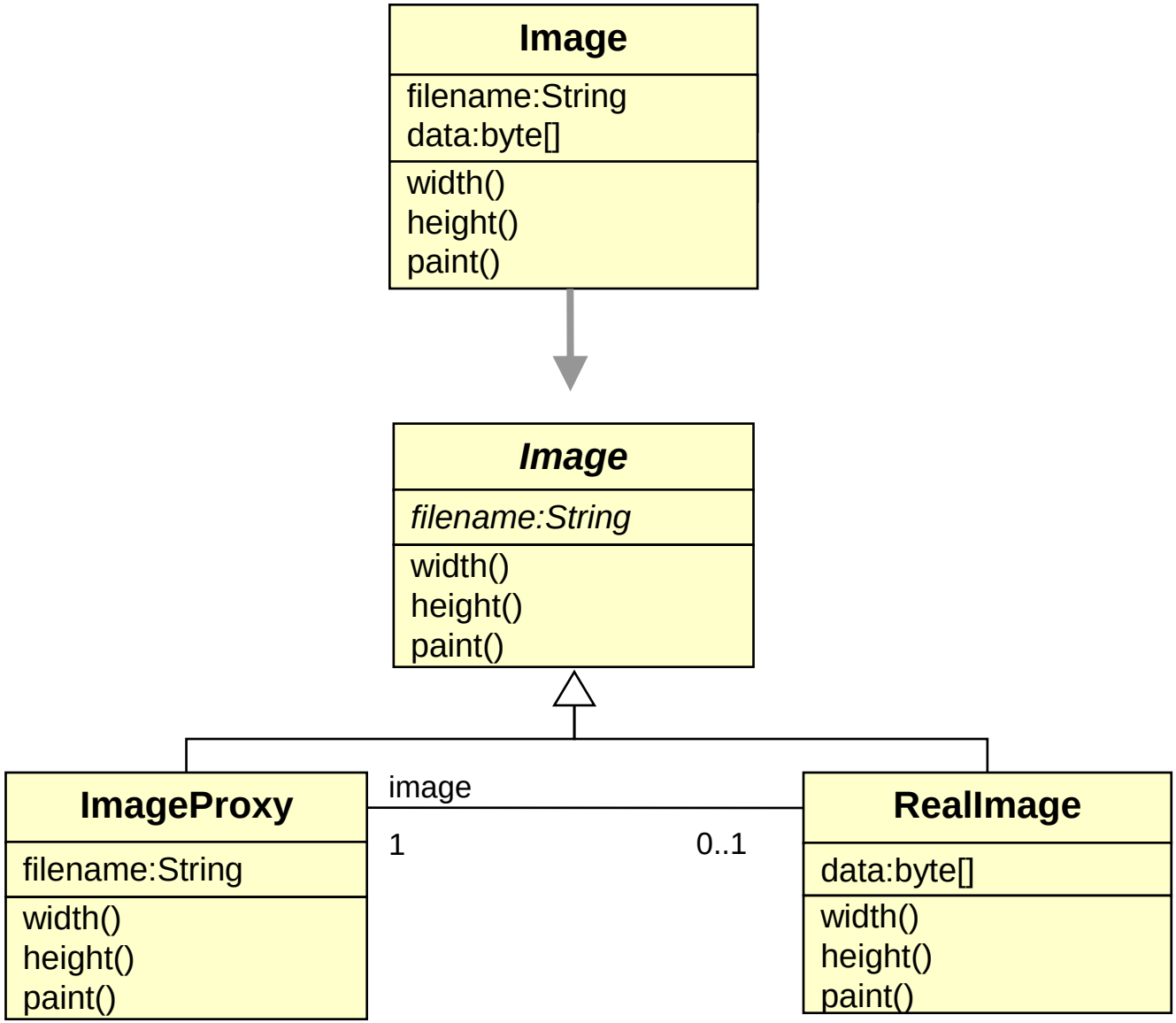
- Verwandle eine Klasse in ein Attribut, wenn get() und set() die einzigen Methoden sind, die von ihr definiert werden.



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 Code der die Änderung durchführt kennt auch alle davon abhängigen abgeleiteten Attribute und setzt sie explizit (*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.
      → *Observer*, *event notification, data trigger:* Das Objekt das ein abgeleitetes Attribut enthält ist Observer der Objekte aus deren Daten der abgeleitete Wert bestimmt wird.

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 Verfeinerung + Restrukturierung des Objektmodells (Design Patterns, …)
  4. Die Optimierung des Objektmodells