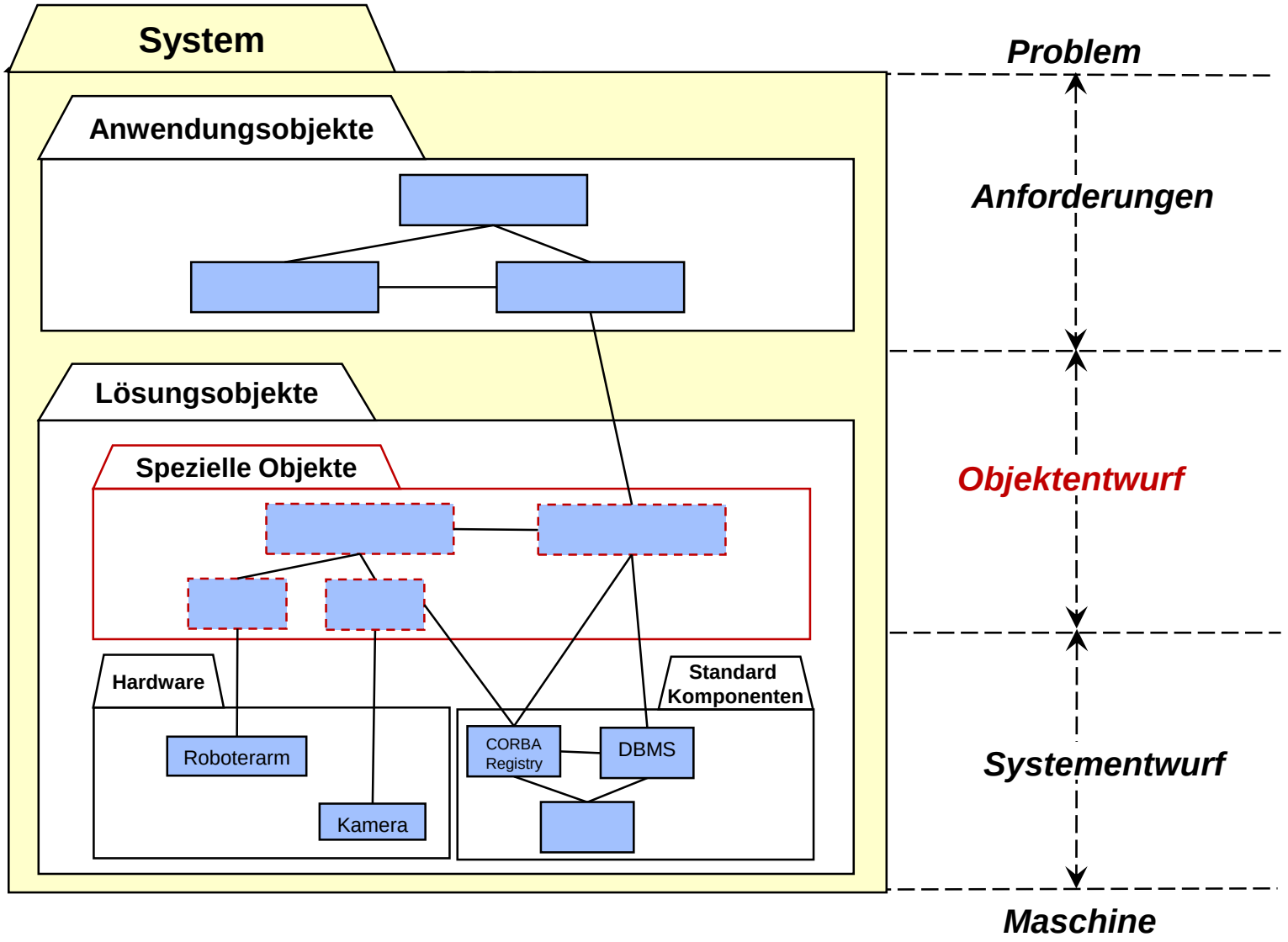
universitätbonn

Kapitel 8. „Objektentwurf“

Stand: 20.12.2017

Teilweise nach „Brügge & Dutoit“, Kap. 8

Objektentwurf: Die Lücke schließen



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-unterstütze 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.

Objektentwurf im Detail

- ✓ Bisher
  - ✓ Domain-Object Model
  - ✓ Analyse-Objektmodell → Boundary-Controller-Entity
  - ✓ Komponentenspezifikation → gebotene und benötigte Interfaces
  - ✓ Architektur → Subsystem und ihre Interaktion
    → Erste Entwurfsmuster

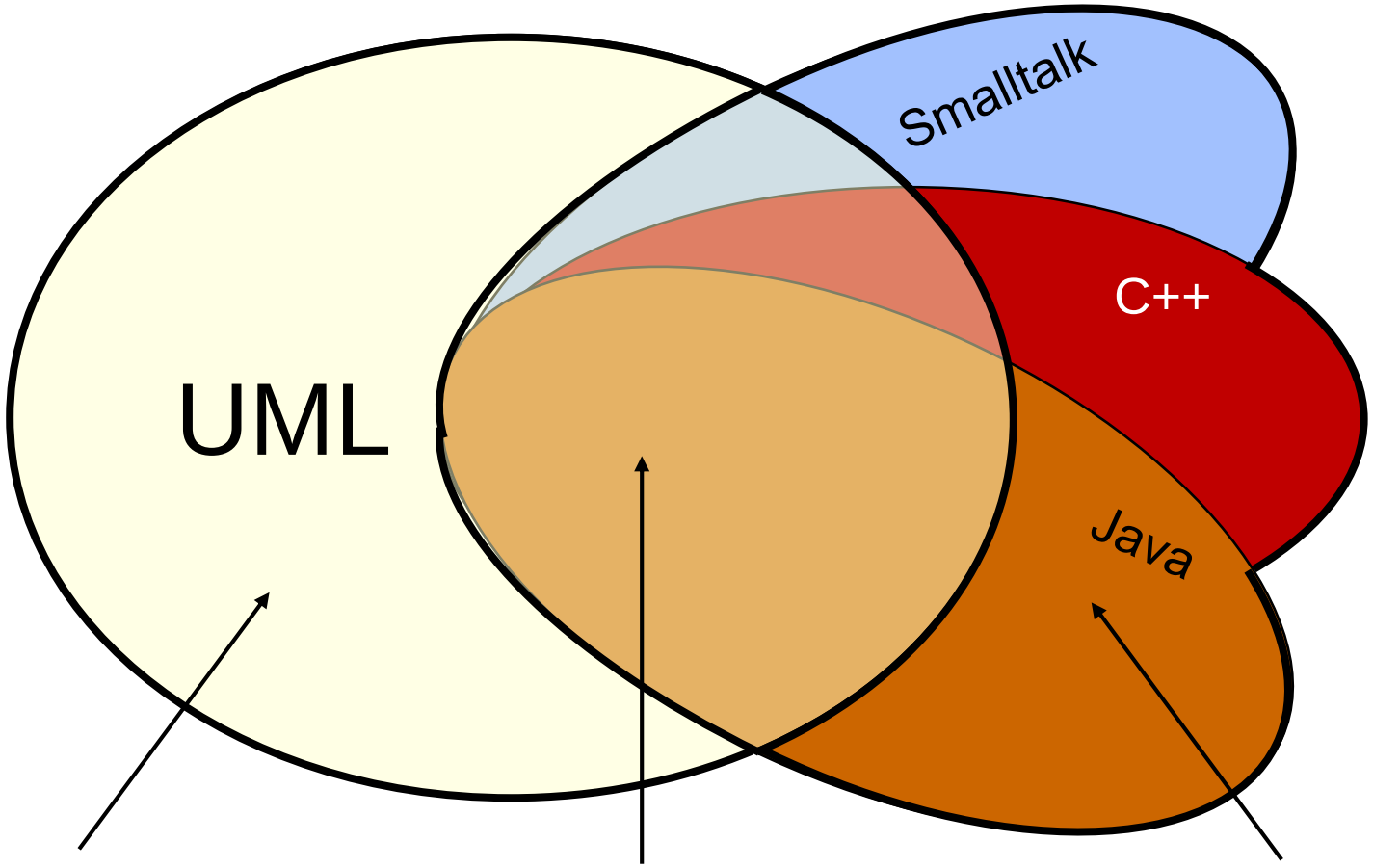
- ➢ Nun: Umformen des Modells zur
  - ➢ Realisierung von UML-Konzepten, die keine Entsprechung in der Programmiersprache haben → „Split-Object“ Entwrufsmuster
  - ➢ Verbesserung von Verständlichkeit, Erweiterbarkeit und anderer „interner Qualitäten“ → Weitere Entwurfsmuster
  - ➢ Optimierung von Reaktionszeit und / oder Speicherbedarf

universitätbonn

UML-Konzepte ohne Entsprechung in der Programmiersprache

Abgeleitete Attribute
Assoziationen als Klassen
Assoziationsklassen
Qualifizierte Assoziationen

Bezug von UML zu gängigen OO Sprachen



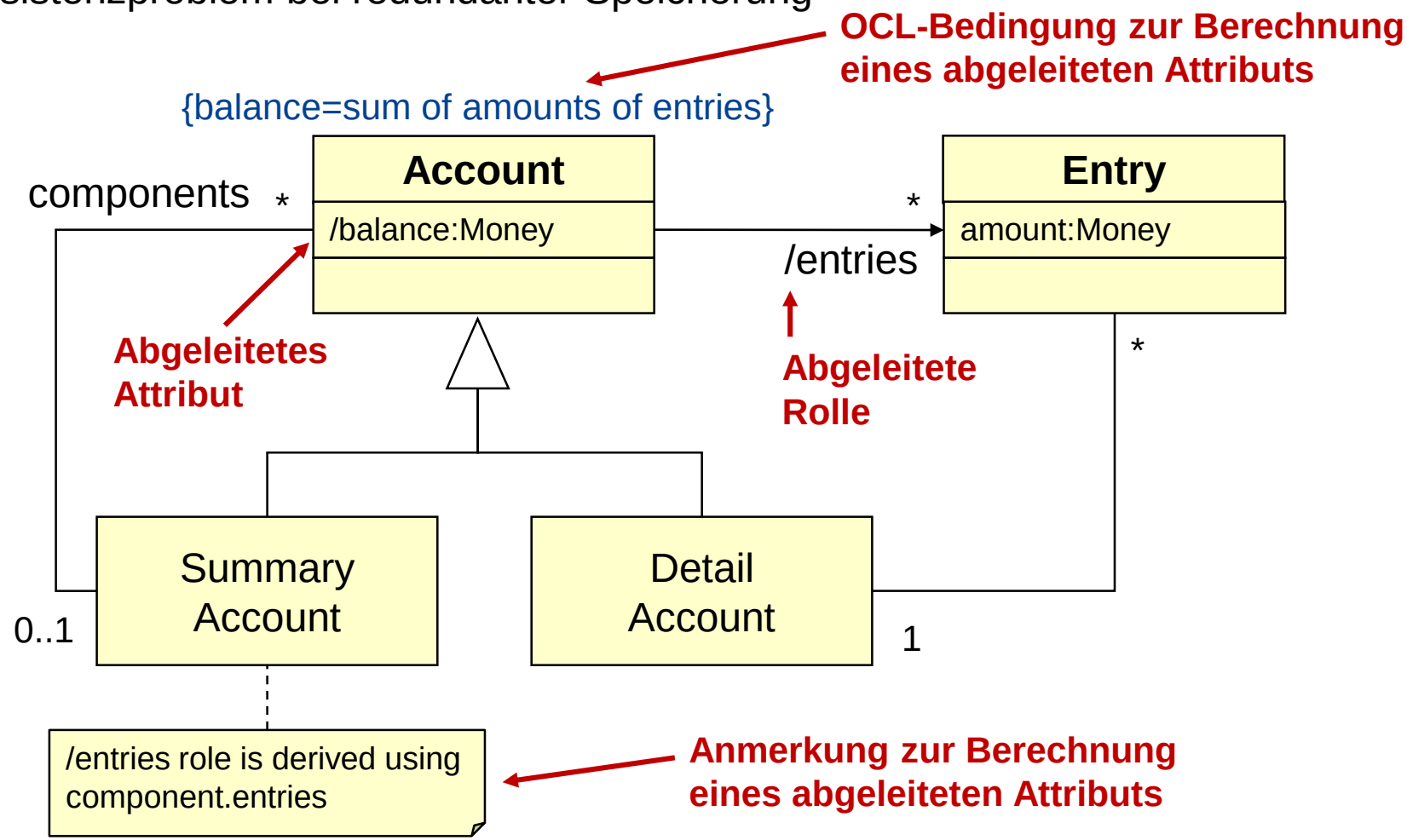
Konzepte ohne direkter Entsprechung in gängigen Sprachen

direkt ineinander abbildbarer gemeinsamer „Kern“

sprachspezifische Details

UML: Abgeleitete Attribute

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



Umsetzung abgeleiteter Attribute

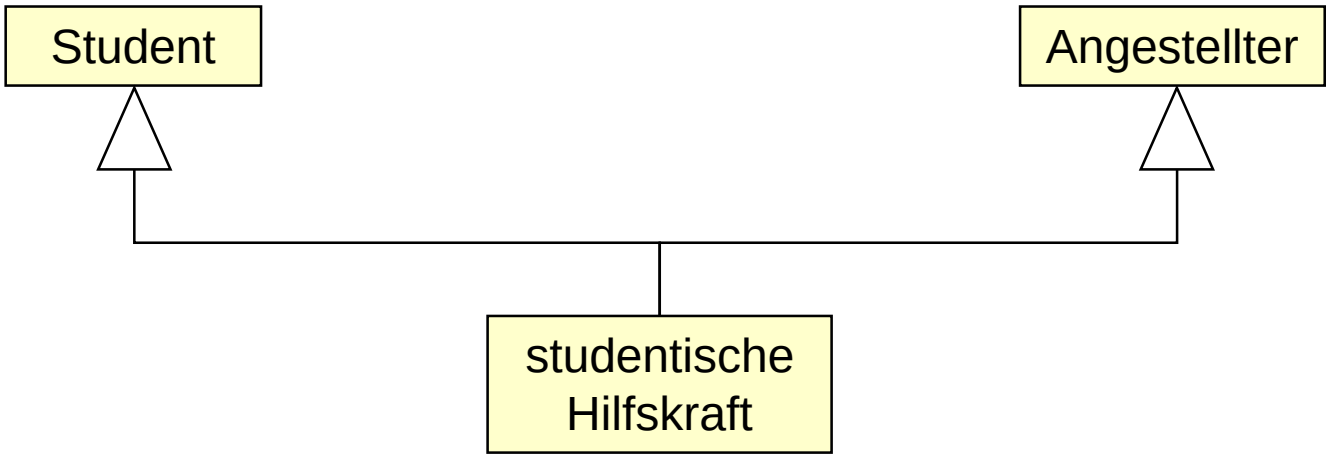
- Abgeleitetes Attribut = Getter-Methode

- Implementierungsoptionen
  - Dynamische Berechnung
    - ⇨ Die erforderlichen Daten werden beim Aufruf berechnet
    - ⇨ Evtl. hoher Berechnungsaufwand
    - ⇨ Werte sind garantiert aktuell
  - Redundante Speicherung
    - ⇨ Das Berechnungsergebnis wird lokal gespeichert
    - ⇨ Weniger Berechnungsaufwand
    - ⇨ Aktuell halten erfordert Implementierung eines Observer-Mechanismus
      - – Das Objekt das das abgeleitete Attribut enthält ist Observer der Objekte, die die Quelldaten enthalten

- Veränderung von abgeleiteten Attributen
  - Oft nicht sinnvoll (→ Was heißt es den Kontostand zu setzen ohne eine neue Buchung durchzuführen???)

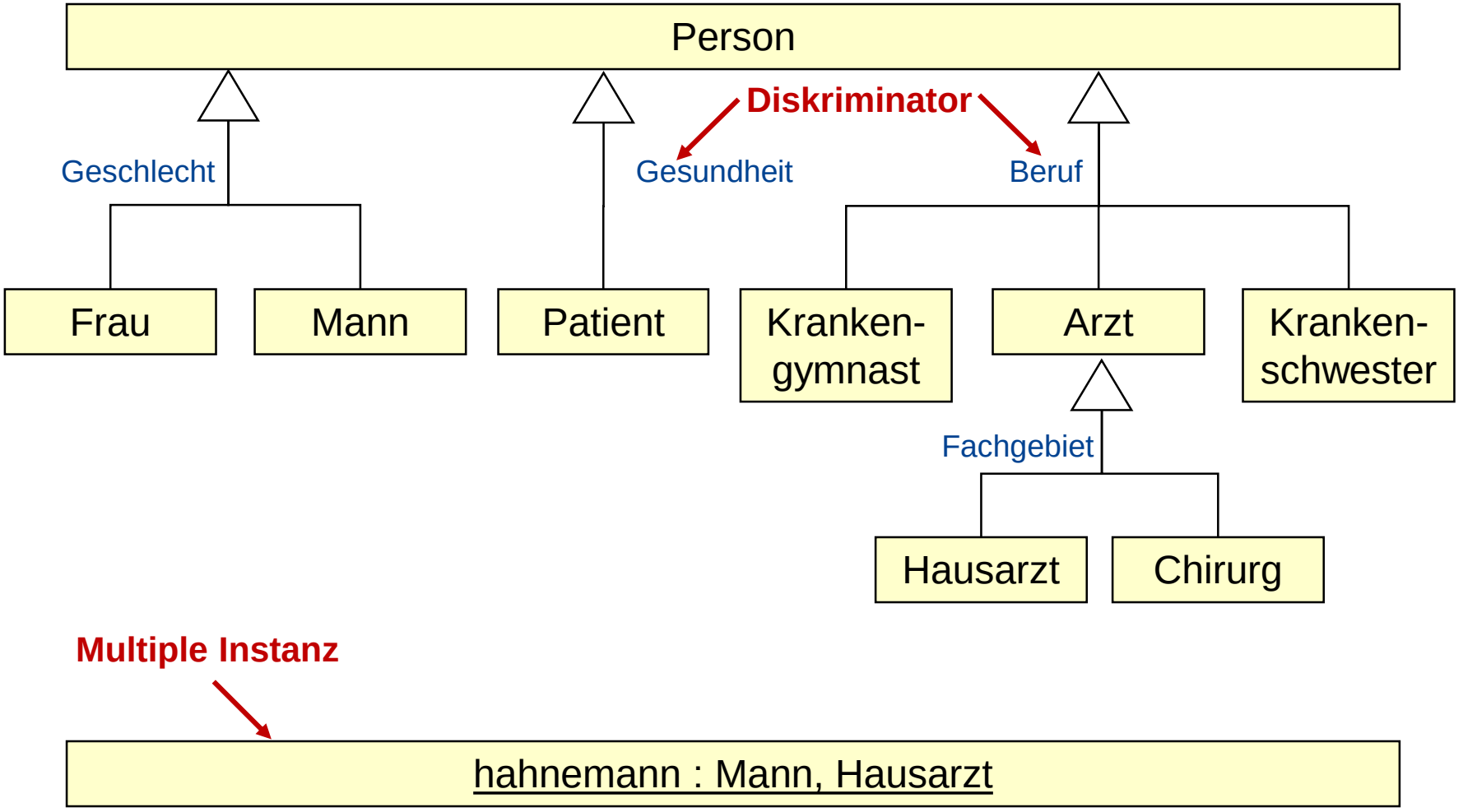
UML, statisches Modell: Multiple Vererbung

- Eine Klasse mit mehreren Oberklassen



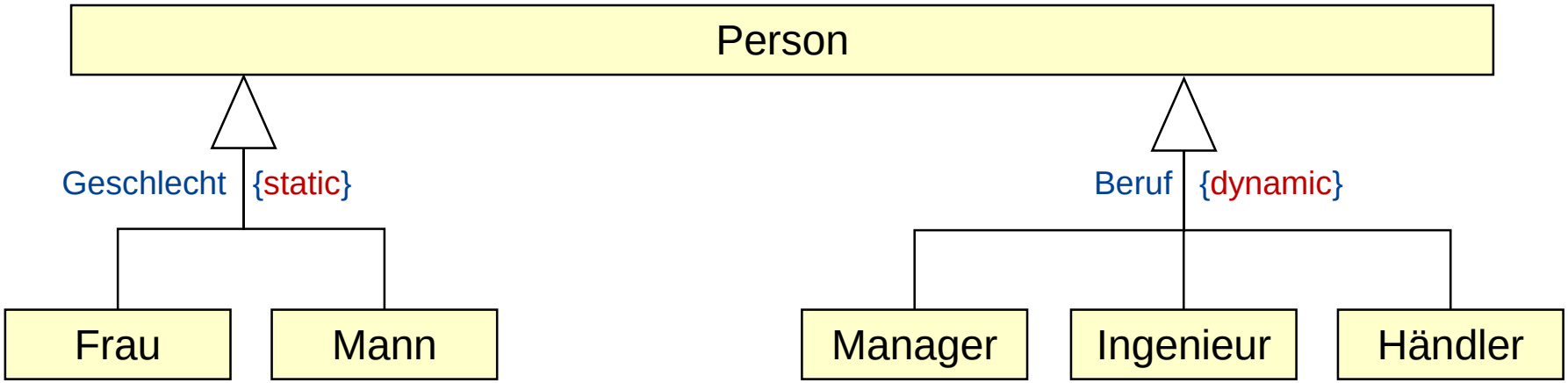
UML, statisches Modell: Multiple Klassifikation

- Ein Objekt kann Instanz mehrerer Klassen sein



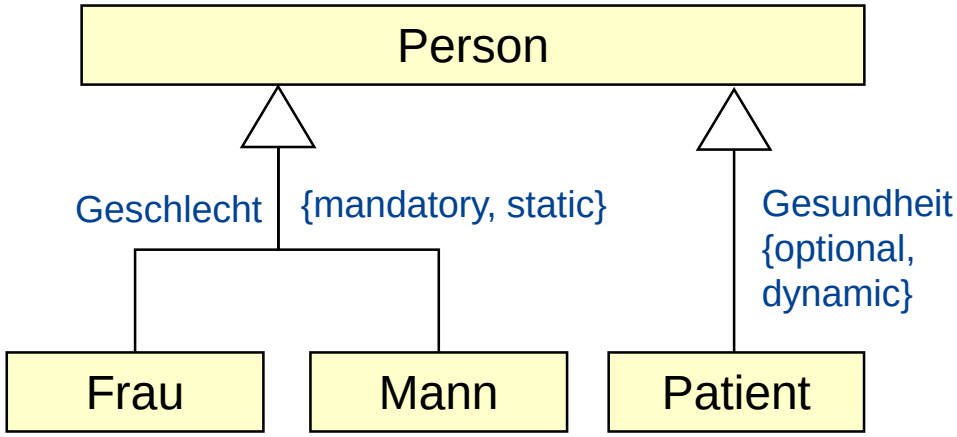
UML, statisches Modell: Dynamische Klassifikation

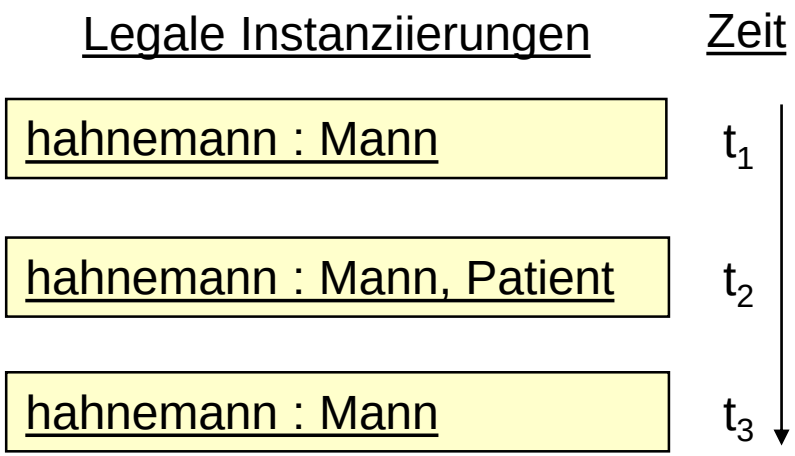
- Eine Objekt kann Instanz mehrerer Klassen sein
- ... und seine Klassenzugehörigkeit ändern



UML: Partitionierung von Unterklassen

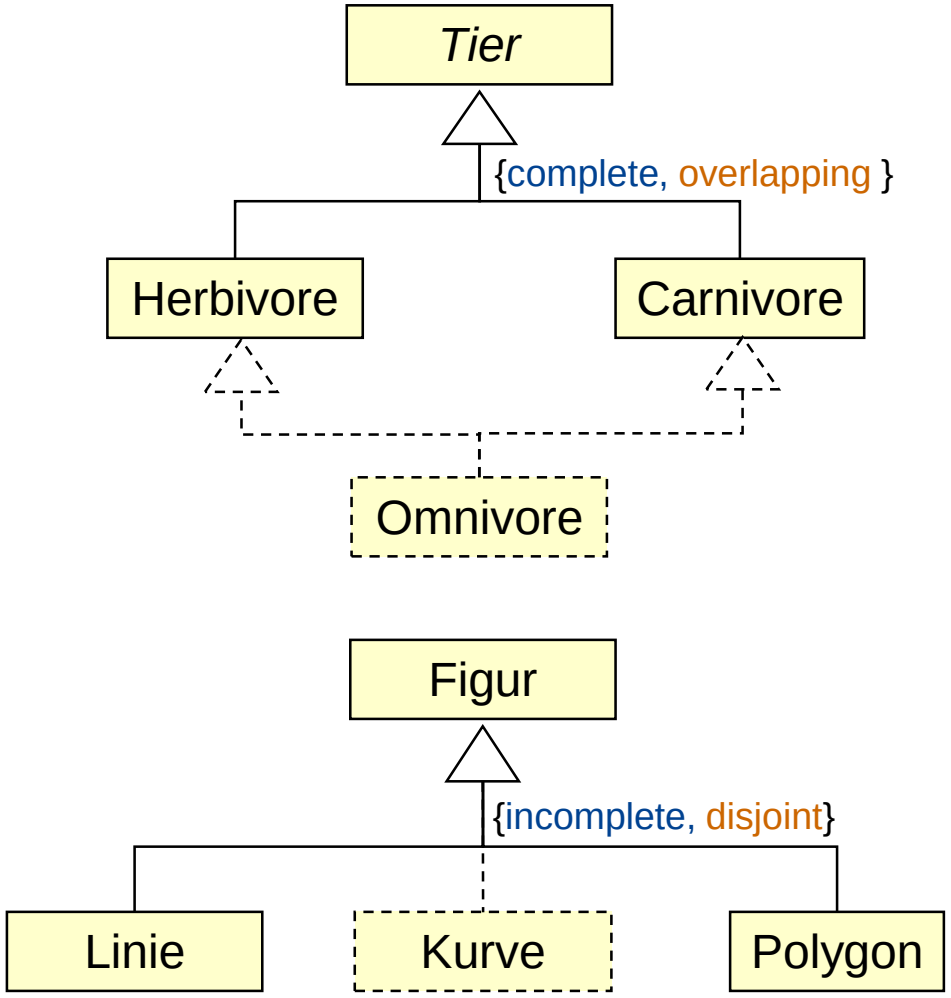
- obligatorisch (mandatory)
  - Objekte müssen einer Klasse aus dieser Partition angehören
- optional (optional) (default)
  - Objekte müssen nicht ...
- statisch (static) (default)
  - Objekte bleiben lebenslang in einer Klasse
- dynamisch (dynamic)
  - Objekte können Klasse wechseln
  - impliziert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



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

- Felder und evtl. zusätzliche Klasse

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

universitätbonn

Umsetzung fortgeschrittener UML-Konzepte in „Kern-UML“

- „Split Object“-Entwurfsmuster

Strategy Pattern als motivierendes Beispiel

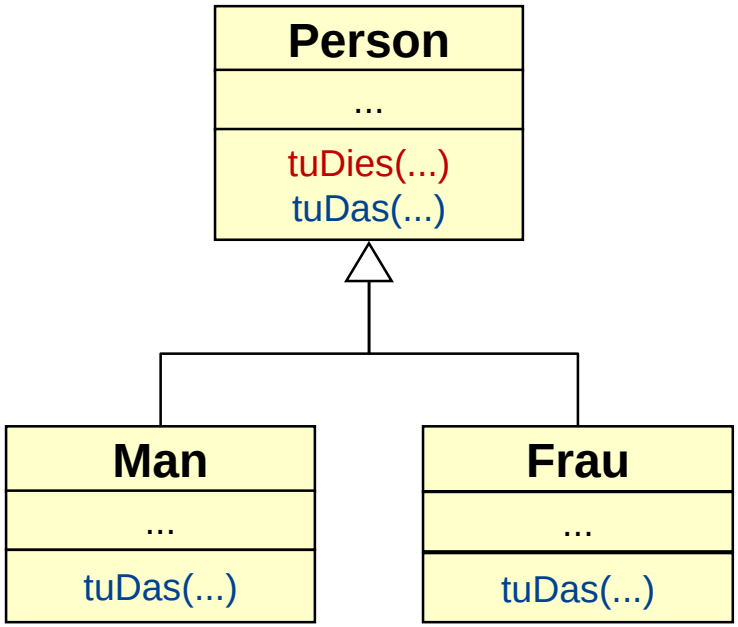
Essenz von Split Objects

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

universitätbonn

Das Strategy Pattern

Motivation: Ein Beispiel



Wie modelliert man

- objektspezifisches
- zustandsspezifisches
- dynamisch veränderbares

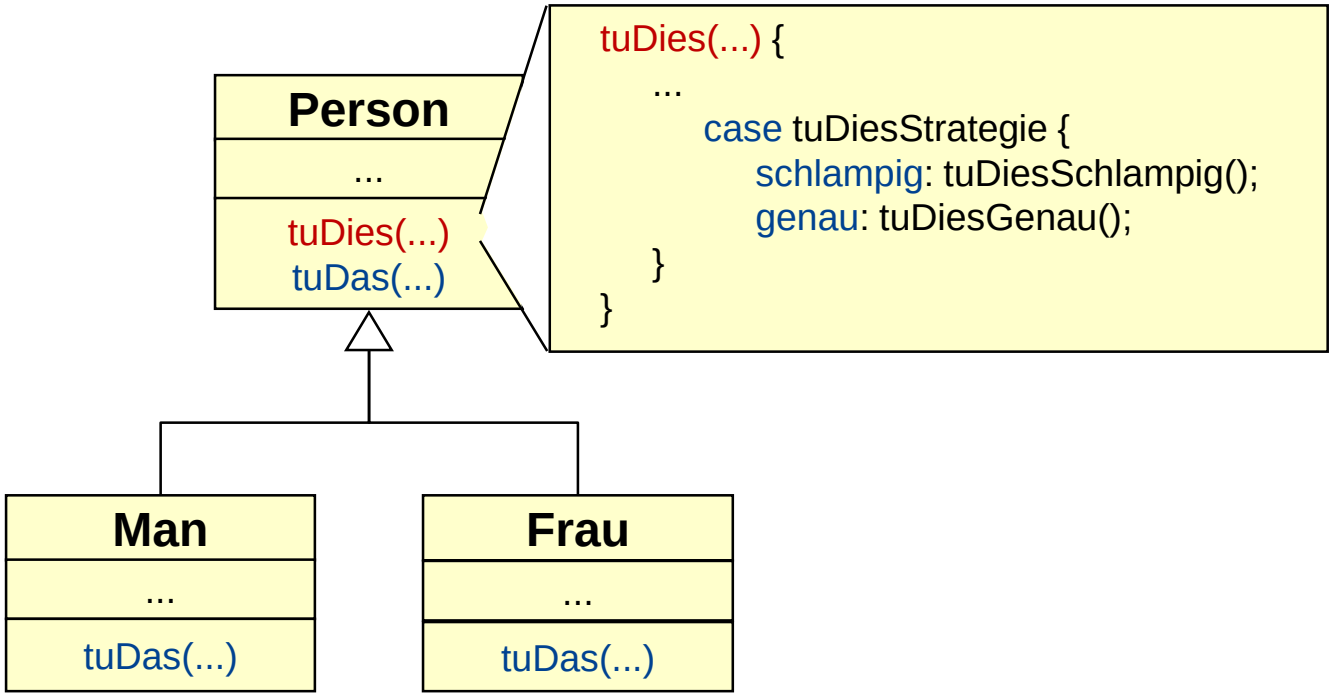
Verhalten?

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

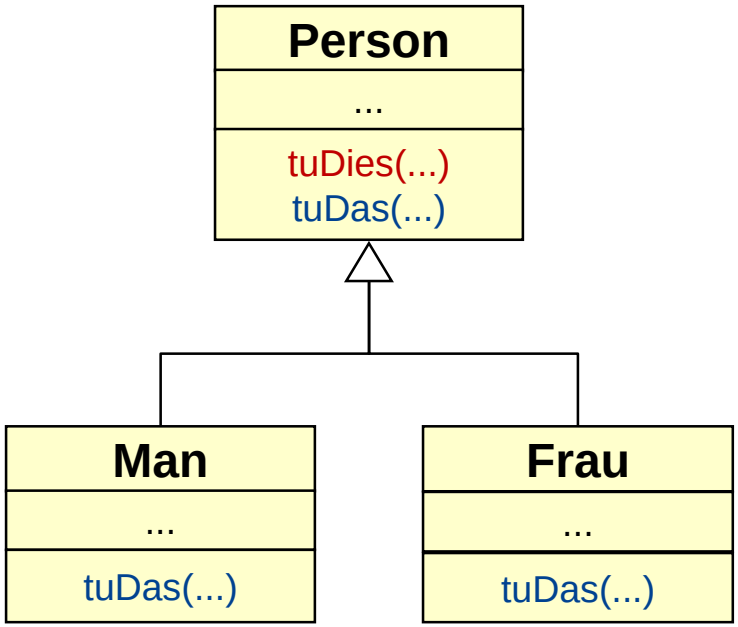
⇨ **tuDies()** ist personenspezifisch verschieden ?

- schlampig
- genau
- …

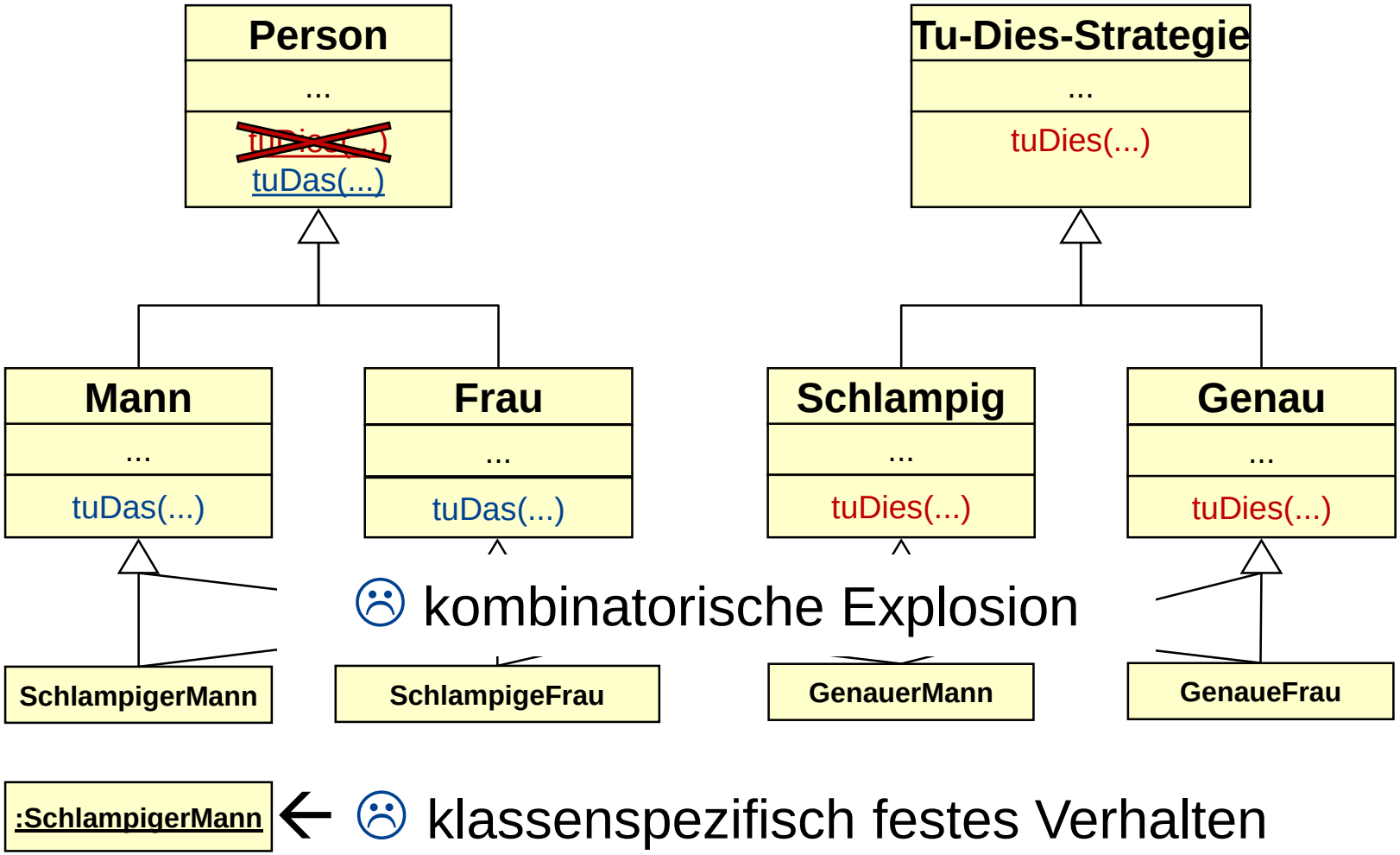
Objektspezifisches Verhalten von tuDies() → 1. Versuch: "Fest Kodieren"



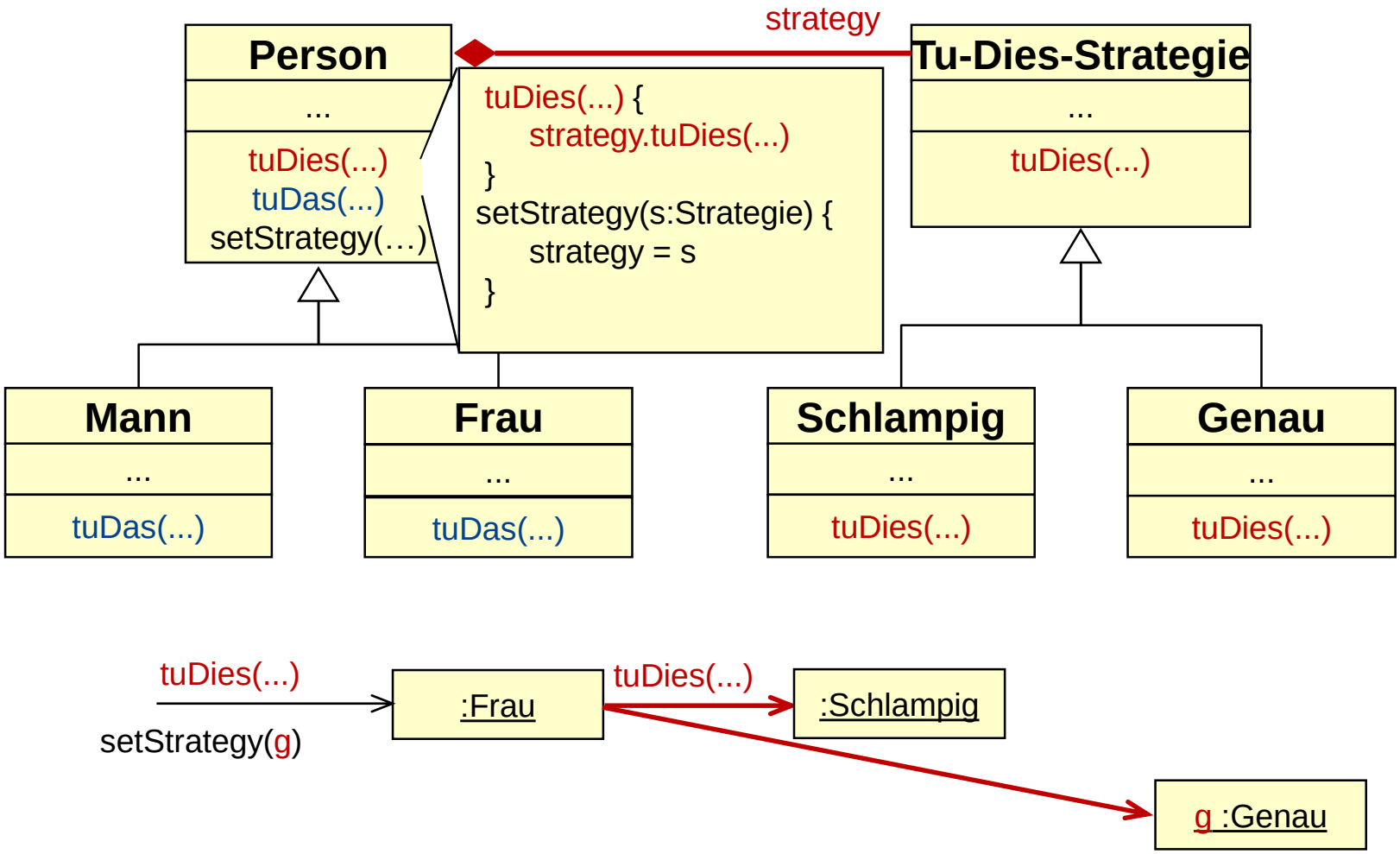
- Schlecht, denn jede neue Alternative („don'tWorryBeHappy", „pingelig", …) erfordert Änderung einer jeden „case"-Anweisung die Varianten von TuDies() beschreibt
- tuDiesGenau() kann evtl. ganz andere Datenstrukturen brauchen als tuDiesSchlampig() – alle zusammen in der Klasse Person unterzubringen ist keine gute Modularisierung (geringe Kohärenz der Klasse)


Person
...
tuDies(...)
tuDas(...)
Man
...
tuDas(...)
Frau
...
tuDas(...)

2. Versuch: “Multiple Vererbung”

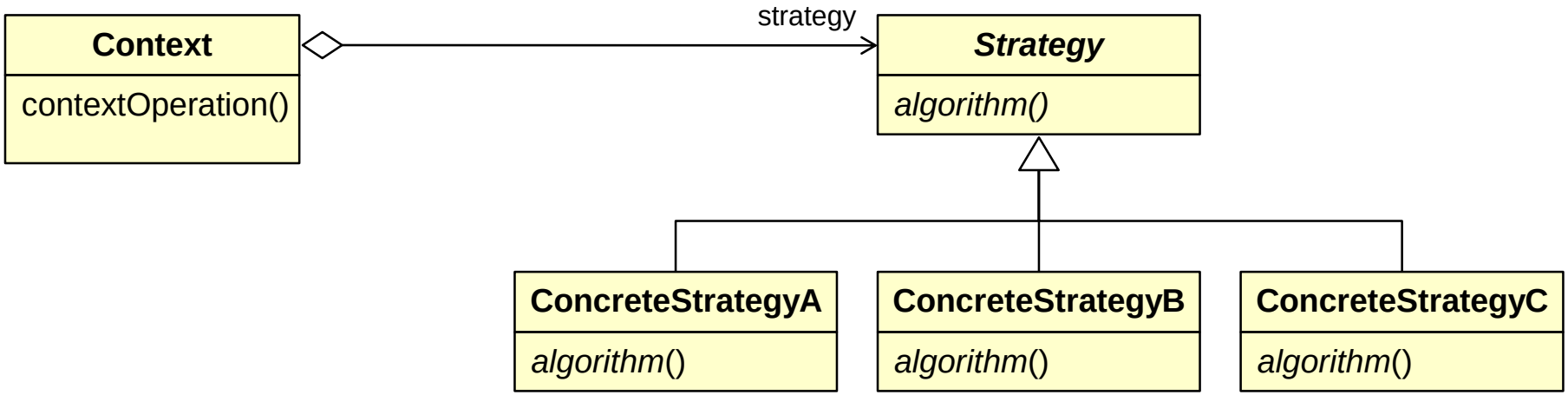


3. Versuch: “Strategy Pattern”



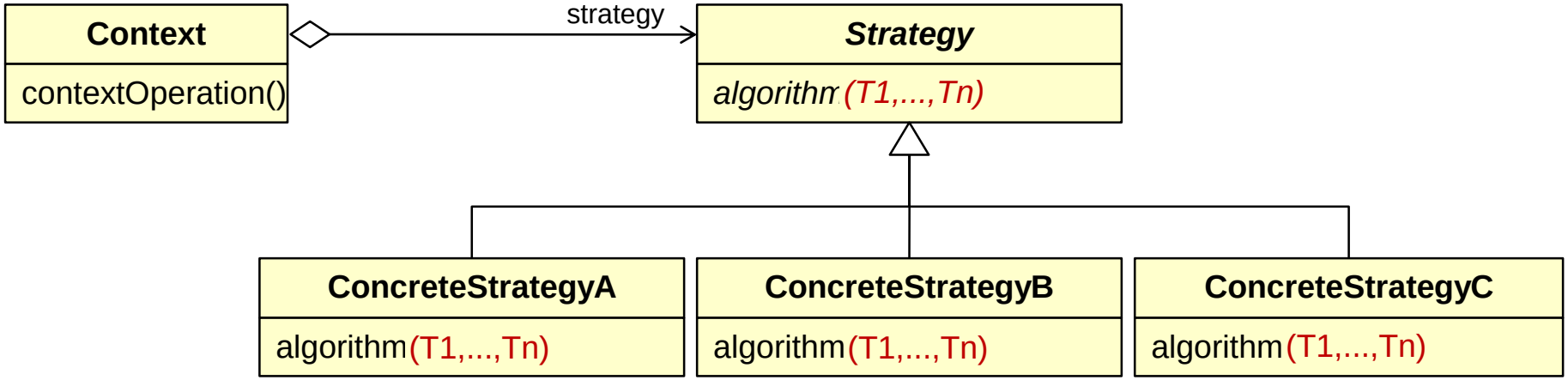
Das Strategy Pattern: Anwendbarkeit und allgemeine Struktur

- 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
▶ Schnittstelle Kontext ↔ Strategie

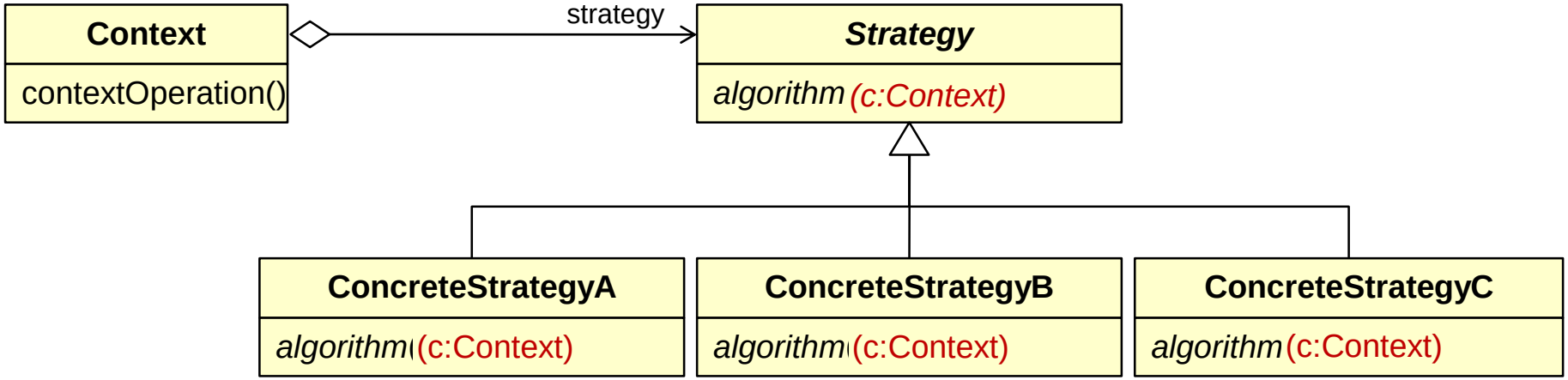
1. Kontext übergibt alle relevanten Daten an die Strategie-Methode
   - einfach
   - überflüssige Parameter
     - ⇨ Jeder von einer Strategie benötigte Parameter taucht in der Schnittstelle auf
   - änderungsanfällig
     - ⇨ StrategyX braucht neuen Parameter → Änderung der Schnittstelle
     - ⇨ … in der gesamten Strategy-Hierarchie!



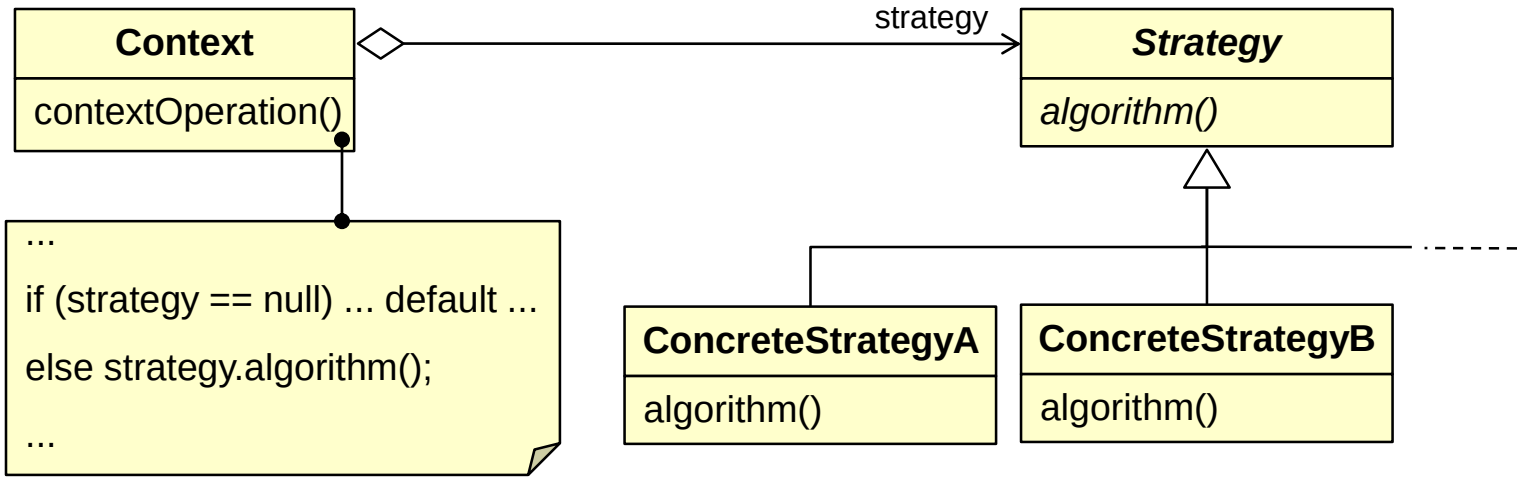
Das Strategy Pattern: Implementierung
▶ Schnittstelle Kontext ↔ Strategie

2. Kontext übergibt nur this an Strategie-Methode

- flexibelste Lösung
  - ⇨ jede Strategie fragt den Kontext nach dem was sie braucht
  - ⇨ Mehrere Kontexte können gleiche Strategie nutzen
    - wenn Strategie keinen eigenen Zustand hat
    - wenn Kontexte alle die gleiche Schnittstelle haben

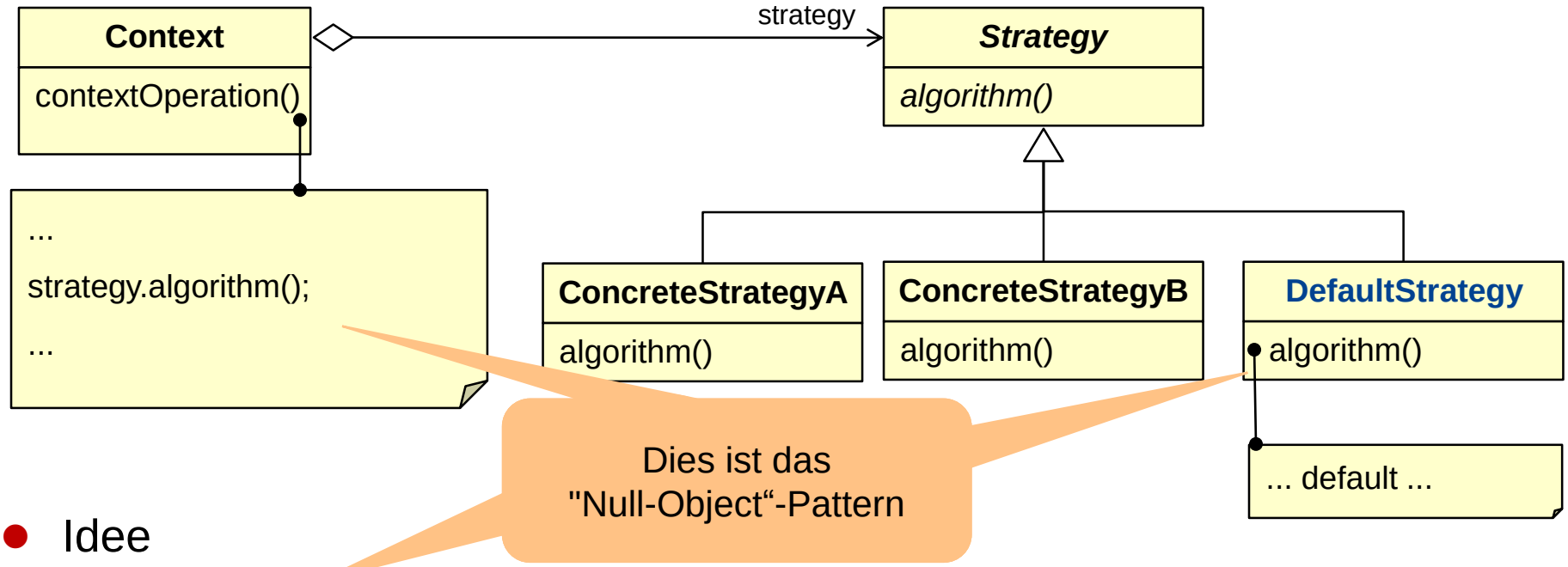


Implementierung: Fallunterscheidung in Kontext



- Vorteile
  - Eine Strategy-Subklasse weniger
- Nachteile
  - Uneinheitliche Lösung: Kontext muss Default-Strategie kennen
  - Besser: „Default-Strategie“-Klasse (nächste Folie)

Implementierung: „Default-Strategie"-Klasse, laut „Null-Object"-Pattern



- 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

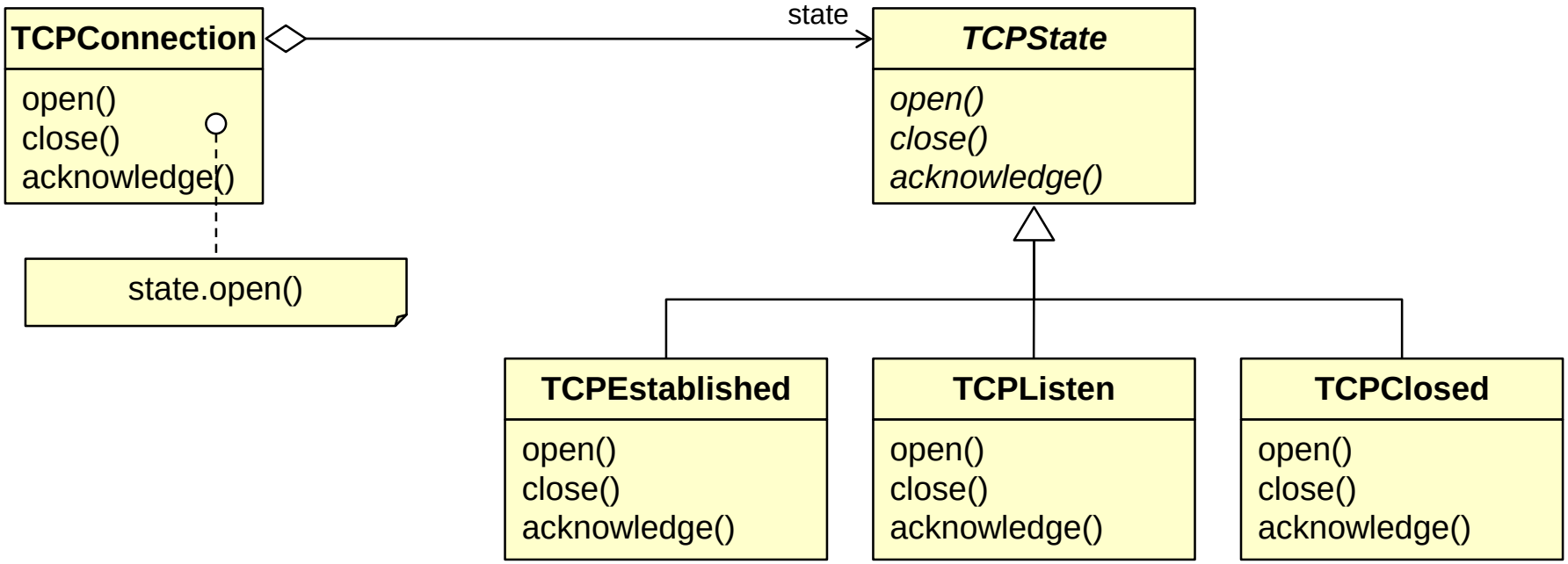
- Prinzip
  - Dekomposition: 2 Objekte
  - Weiterleitung von Anfragen
- Vorteile
  - Dynamik
  - Multiplizität
  - Erweiterbarkeit

universitätbonn

Das State Pattern

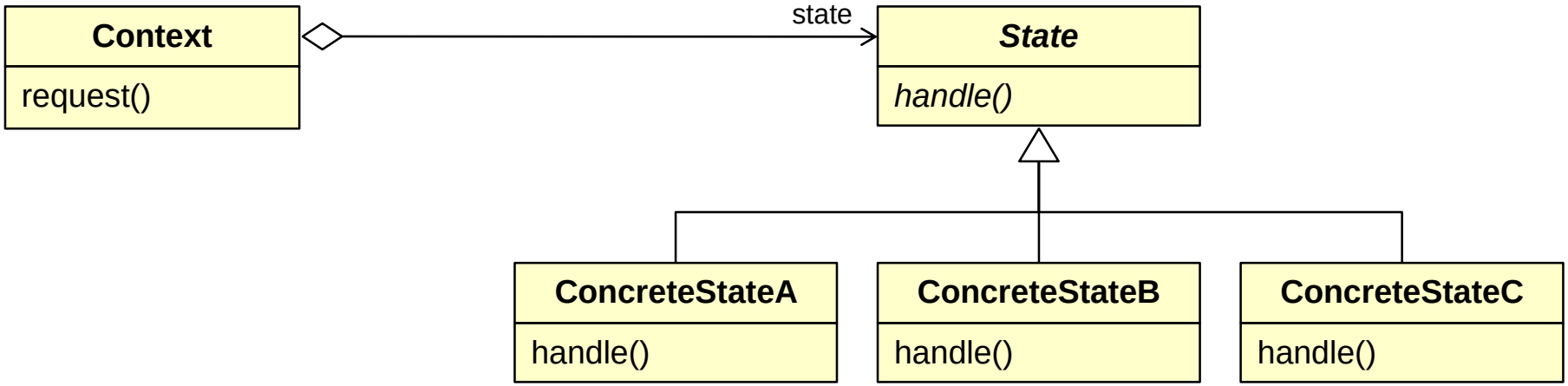
Das State Pattern

- Absicht
  - Objekt soll sein Verhalten ändern können, wenn sein Zustand sich ändert.
  - Viele Methoden gleichzeitig betroffen
- 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

Abgrenzung Strategy / State ▶ Was ist änderbar?

- Bei Strategy-Pattern
  - meist einzelnes Verhalten / einzelne Methode
    - ⇨ Z.B. Video-Rendering-Algorithmus

- Bei State-Pattern
  - meist Menge von Methoden, die gleichzeitig ihr Verhalten ändern, wenn ein bestimmtes Ereignis auftritt
    - ⇨ z.B. Bei einer Person im „Stress“-Zustand ist der Puls erhöht, die Wahrnehmung fokussiert, die Verdauungstätigkeit reduziert, … Es sind also viele Dinge betroffen, die sich alle anders verhalten als z.B. im „Gut gelaunt“-Zustand.
  - State-Pattern modelliert typischerweise einen Zustandsautomat

Abgrenzung Strategy / State ▶ Wer triggert die Änderung des Elternobjektes?

- Bei Strategy-Pattern
  - meist extern veranlasst, d.h. Client ruft setStrategy() auf
  - 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, d.h. eine State-Instanz ruft setState() auf
  - … typischerweise am Ende einer Aktion!
  - State-Pattern modelliert oft einen Zustandsautomat. Daher ist klar, dass hier die Logik der Zustandsübergänge nicht extern bestimmt ist, sondern 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

Split Objects: Prinzipien

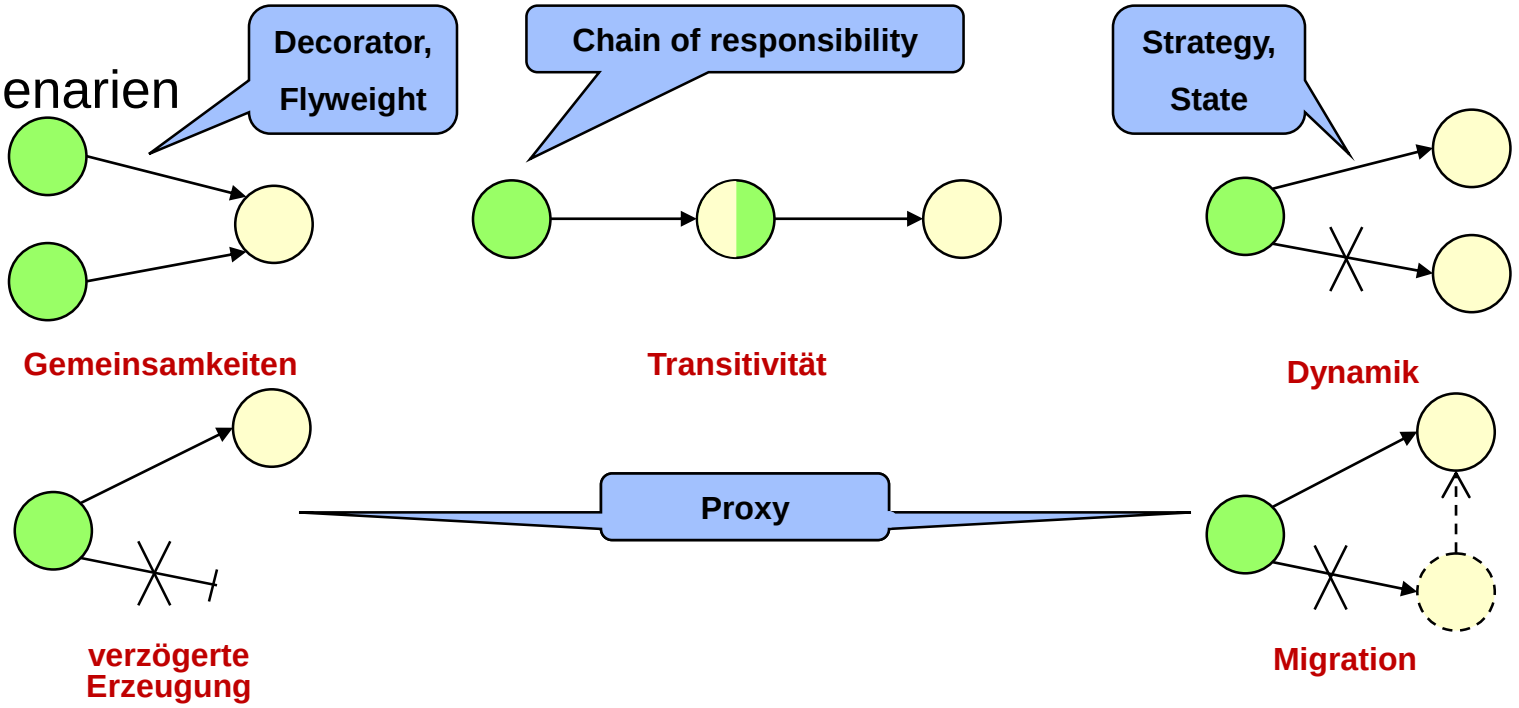
Was sind also "Split Objects"?

- 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

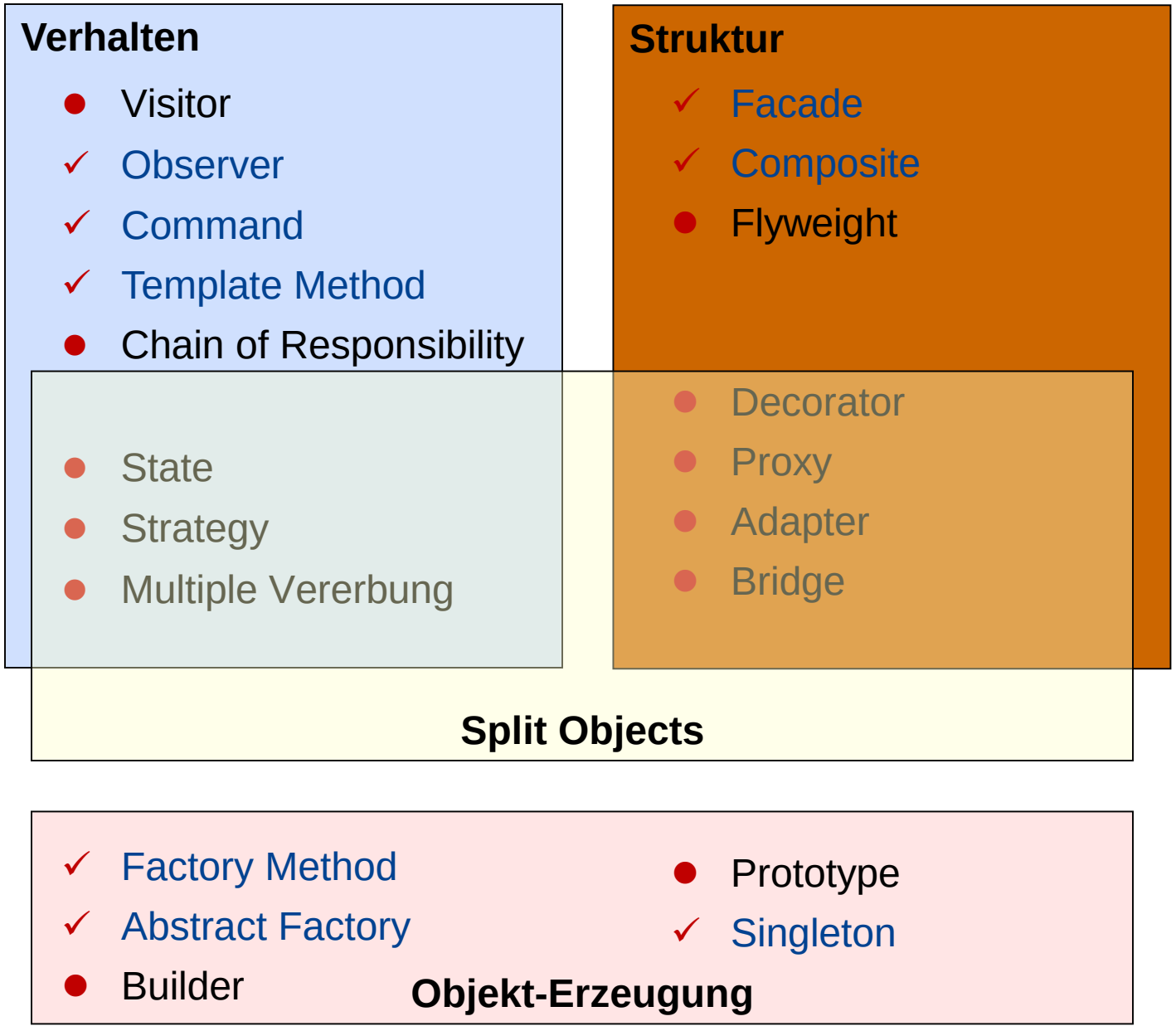
- Technik
  - Mehrere physikalische Objekte
  - Nur eines davon ist nach außen hin sichtbar 
  - Es stellt das Interface des konzeptuellen Gesamtobjektes zur Verfügung
  - ... indem es die Fähigkeiten der anderen mit benutzt
- Szenarien



Split Objects sind die Grundlage vieler Design Patterns

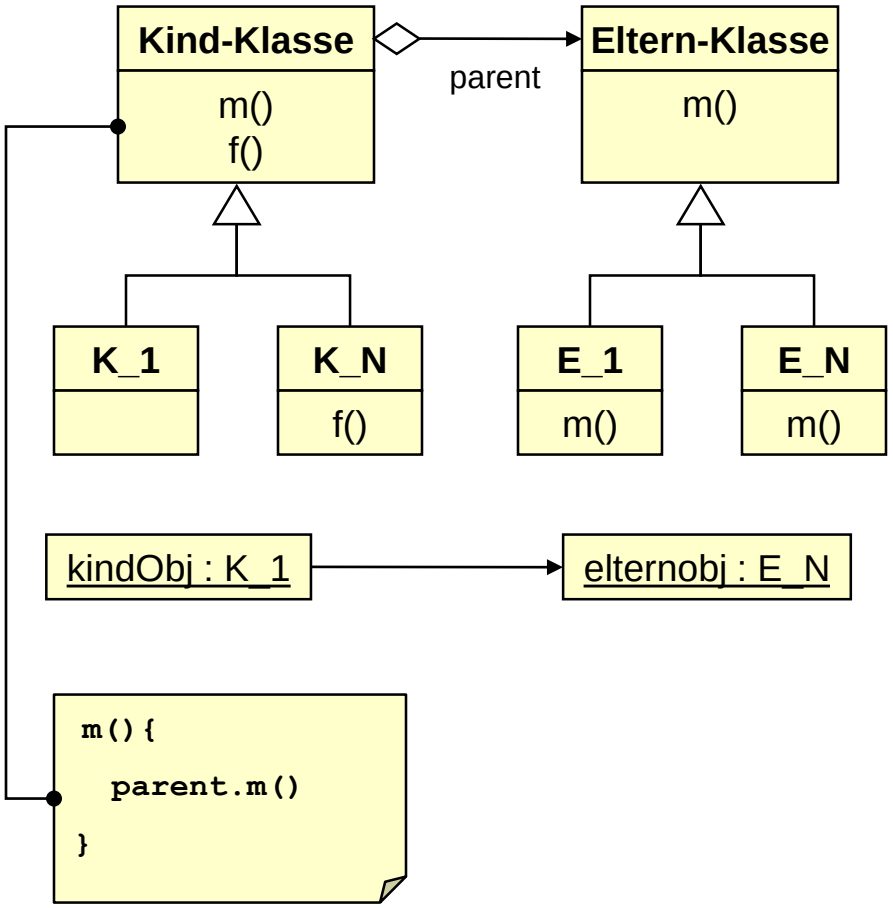
- Vorausschauende Dekomposition
  - Proxy
  - Strategy
  - State
  - Flyweight
- Unvorhergesehene Komposition
  - Adapter
  - Decorator
  - Chain of Responsibility

"Gang of Four"-Patterns: Überblick und Klassifikation



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

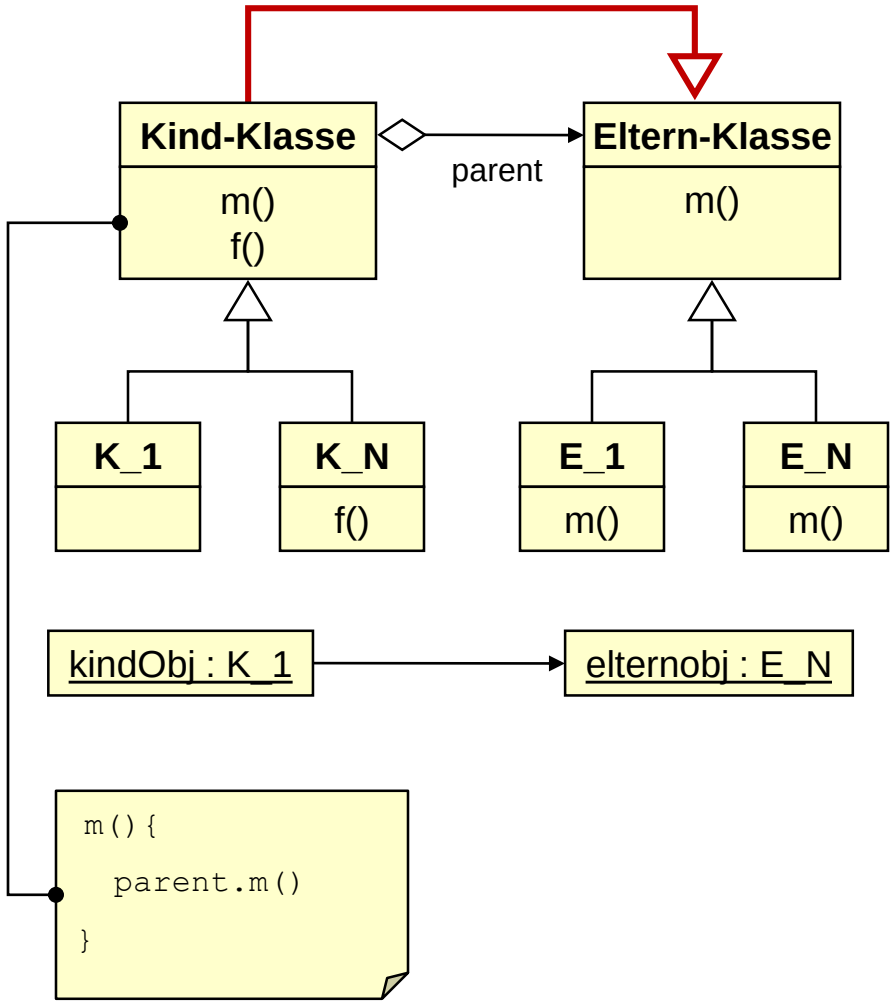
- Forwarding
  - Kindobjekt leitet empfangene Nachricht an Elternobjekt weiter
- Subtyping
  - Kindklasse bietet volles Interface der Elternklasse
- Overriding
  - im Kontext weitergeleiteter Nachrichten werden Methoden des Kindobjekts benutzt
  - ... anstelle entsprechender Methoden des Elternobjekts

Zusammengesetzte Beziehungen

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

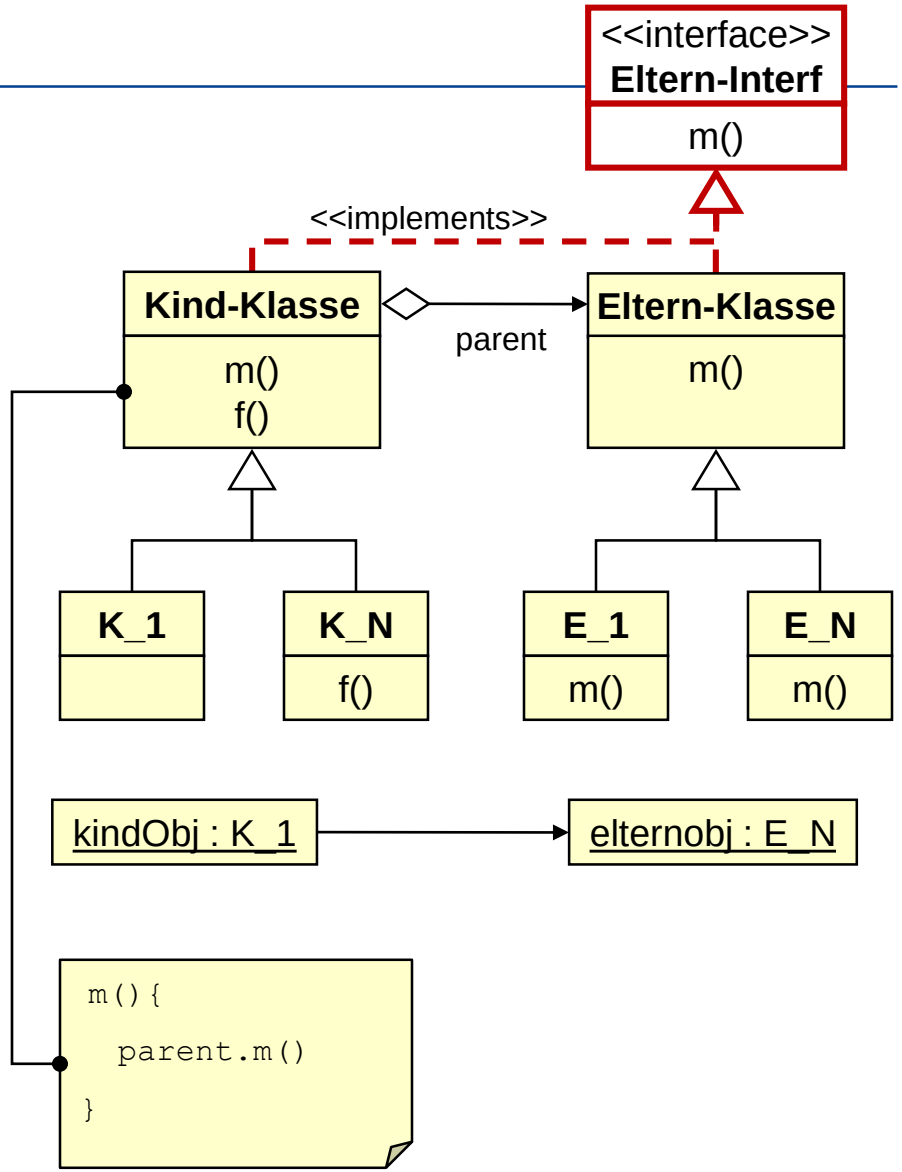
Implementierung der Subtypbeziehung: Variante 1

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



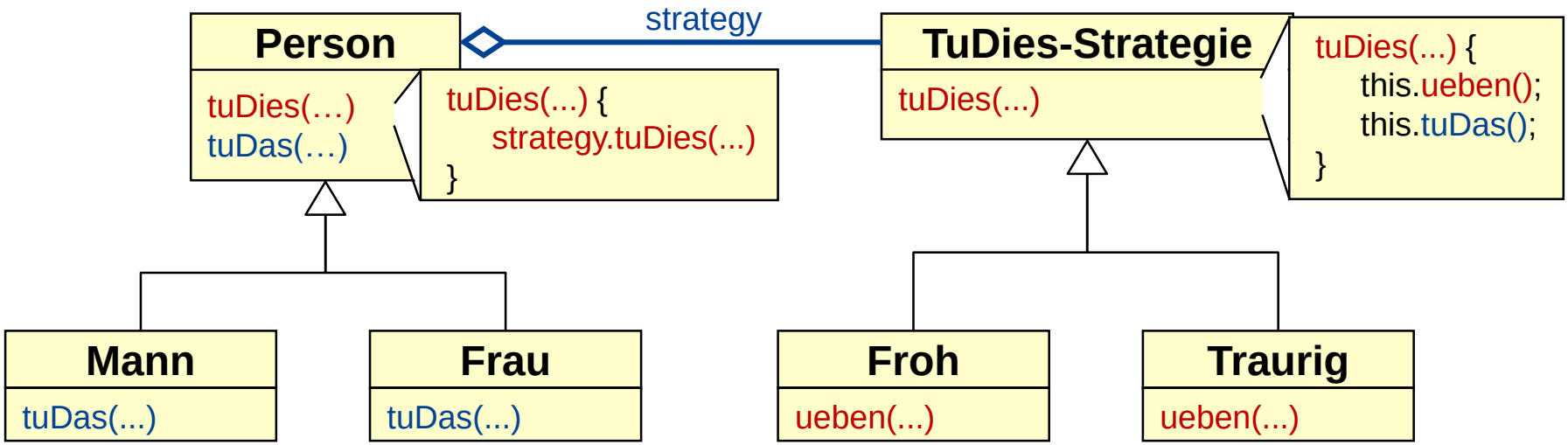
Implementierung der Subtypbeziehung: Variante 2

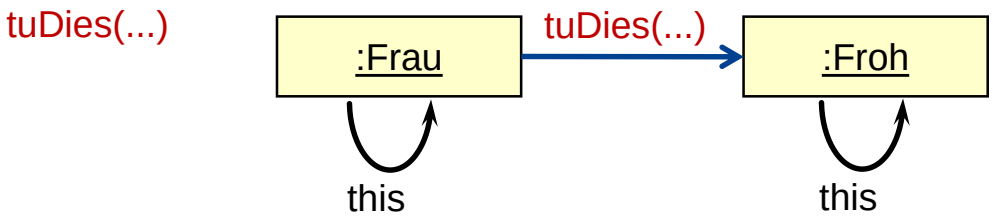
- Idee
  - Kind implementiert gleiches Interface wie die Elternklasse
  - Eltern-Interface anstatt Eltern-Klasse in Typdeklarationen verwenden
- Vorteil
  - keine Datenduplizierung
  - saubere Trennung
    - ⇨ Subtyping
    - ⇨ Code Wiederverwendung



Wie modelliert man Overriding?

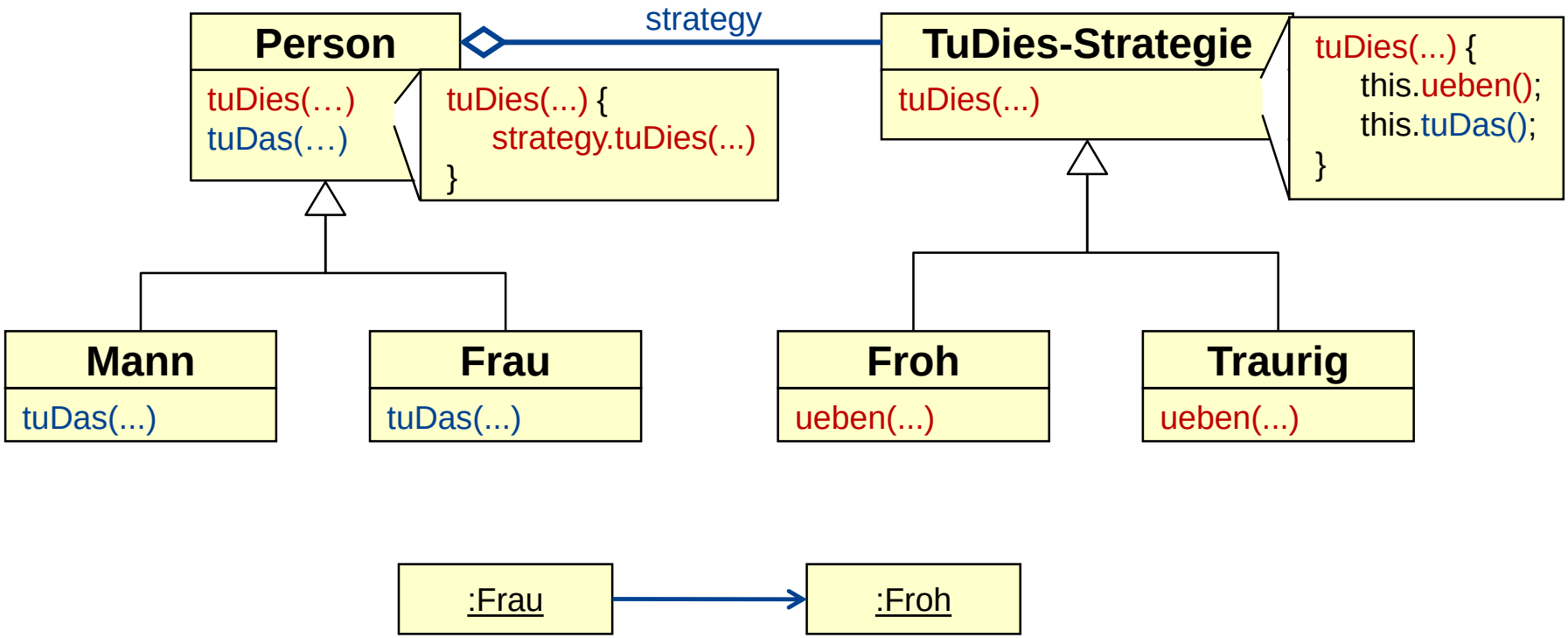
Strategy Pattern als Beispiel des Overriding-Problems





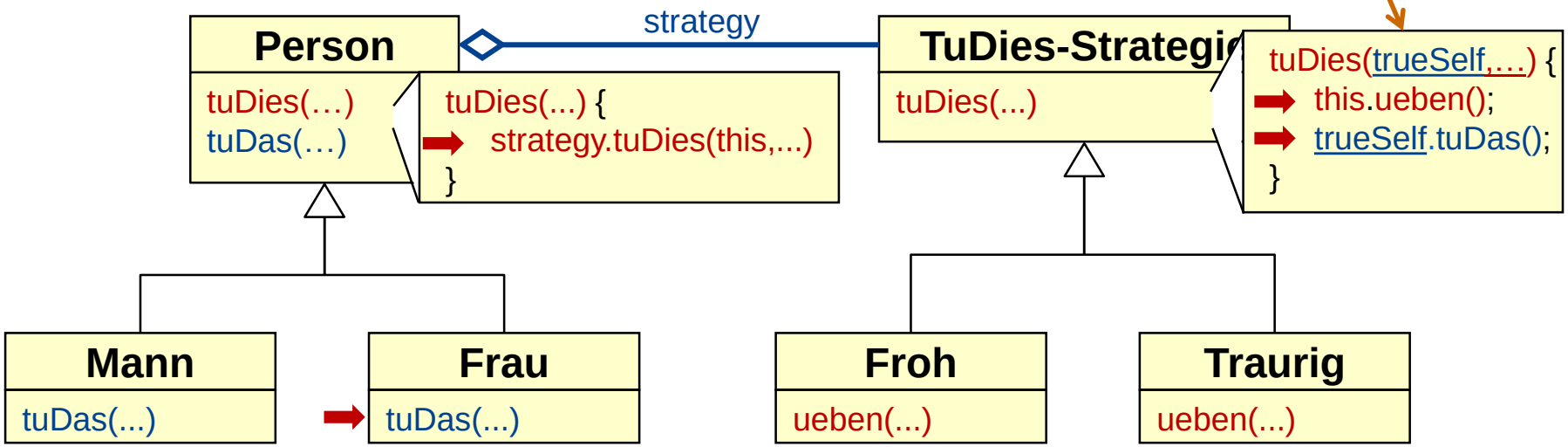
- **“Schizophrene Objekte”: verschiedenes “this” in ursprünglicher Nachricht und weitergeleiteter Nachricht, obwohl beide das gleiche konzeptionelle ein Objekt „fröhliche Frau“ ansprechen.**

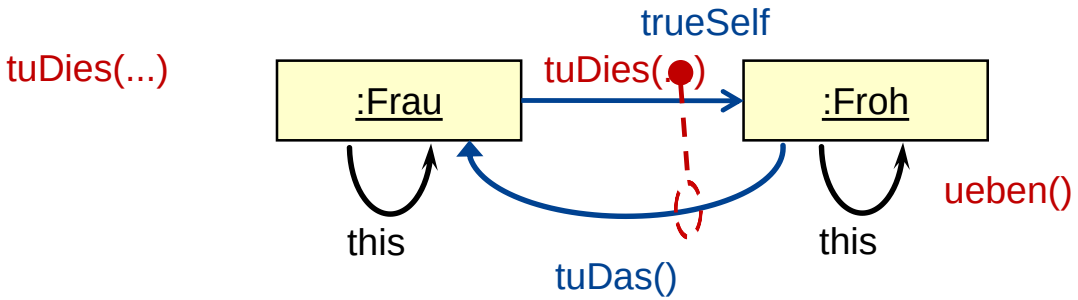
Overriding-Simulation: "this" explizit machen



Overriding-Simulation: "this" explizit machen

Typ von trueSelf muss Person sein, damit tuDas() aufrufbar ist ☹





Heureka!

Heureka?

Overriding-Simulation: Schwachpunkte

- Festlegung des Typs von trueSelf auf Person
  - “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

universitätbonn

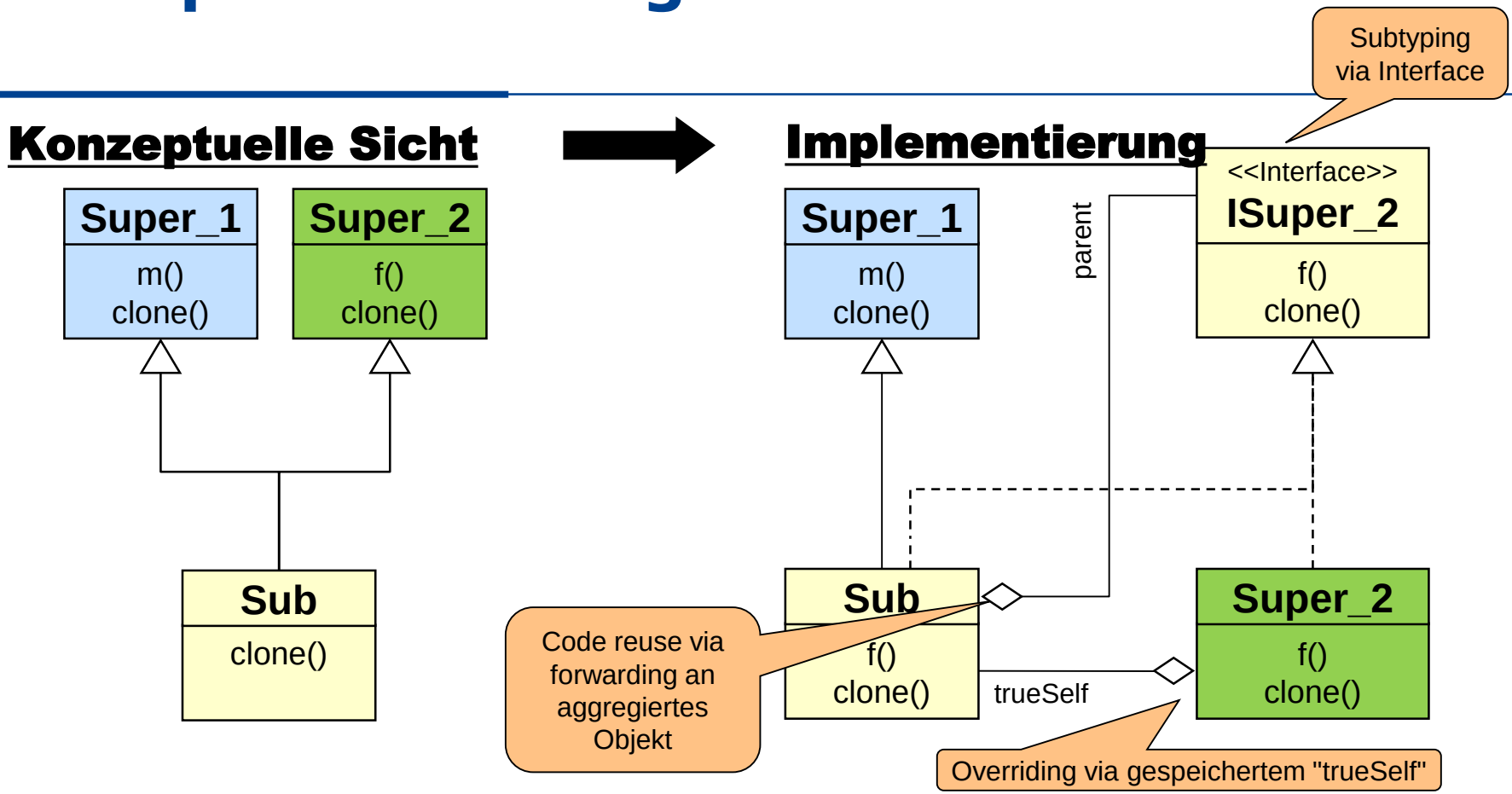
Auf "Split Objects"-Idee basierende Patterns

3. Simulation von Multipler Vererbung in Java

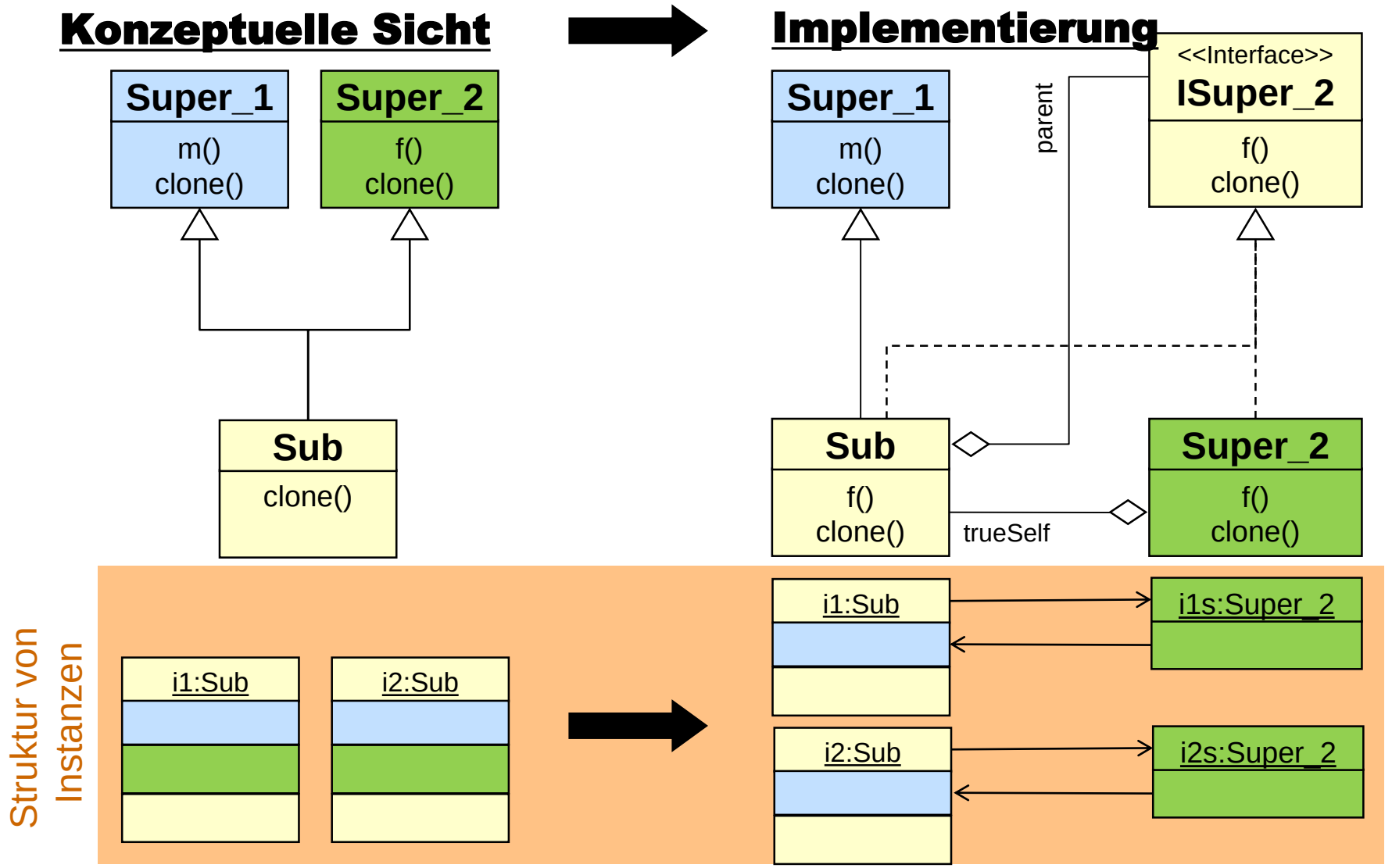
Multiple Vererbung in Java

- Absicht
  - Interface und Code mehrerer "Oberklassen" wiederverwenden
  - ... obwohl Java nur Einfachvererbung erlaubt
- Motivation
  - komplexe Klassen nicht reinmplementieren
- Anwendbarkeit
  - keine semantischen Konflikte zwischen "Oberklassen"-Methoden (z.B. zwei clone()-Methoden mit unterschiedlicher Bedeutung – shallow / deep)
- Vorgehen
  - Code Wiederverwendung durch Aggregation und Forwarding
  - Subtyping durch gemeinsames Oberinterface
  - Overriding durch gespeichertes „trueSelf“

Multiple Vererbung in Java



Multiple Vererbung in Java



Implementierung

- 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

Implementierung: Probleme

- Code-Reuse
  - Schutz von protected-Variablen und -Methoden aufgehoben
  - manuelle Propagierung von Änderungen des Oberklassen-Interfaces → aufwendig, fehleranfällig
- 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:

    ⇨ expr.var;

    ↓

    ⇨ expr.getVar(); + getVar() { return parent.getVar(); }

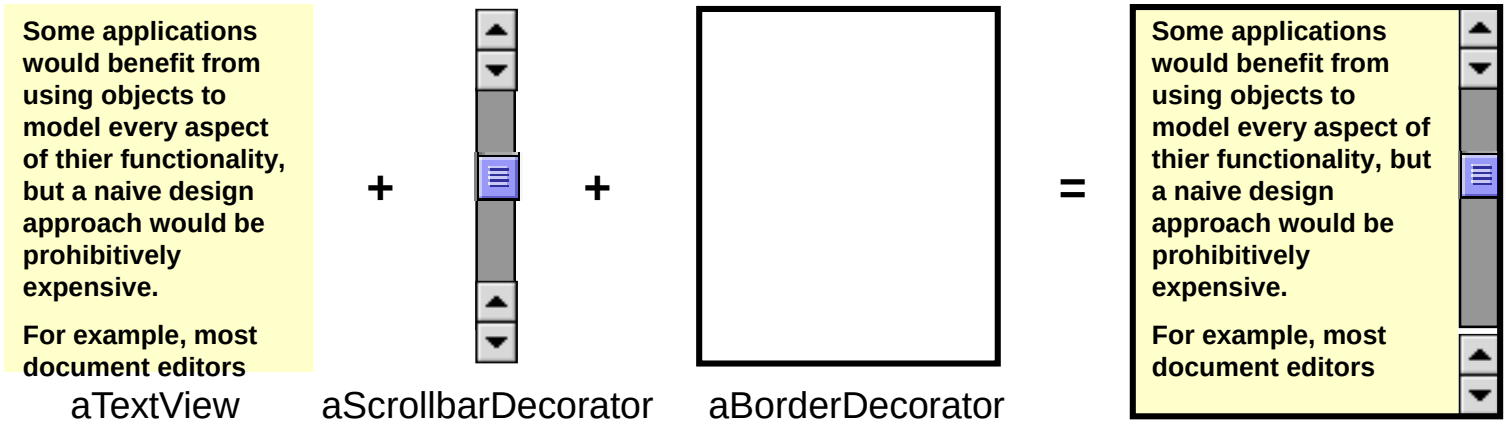
universitätbonn

Auf "Split Objects"-Idee basierende Patterns

4. Das Decorator Pattern

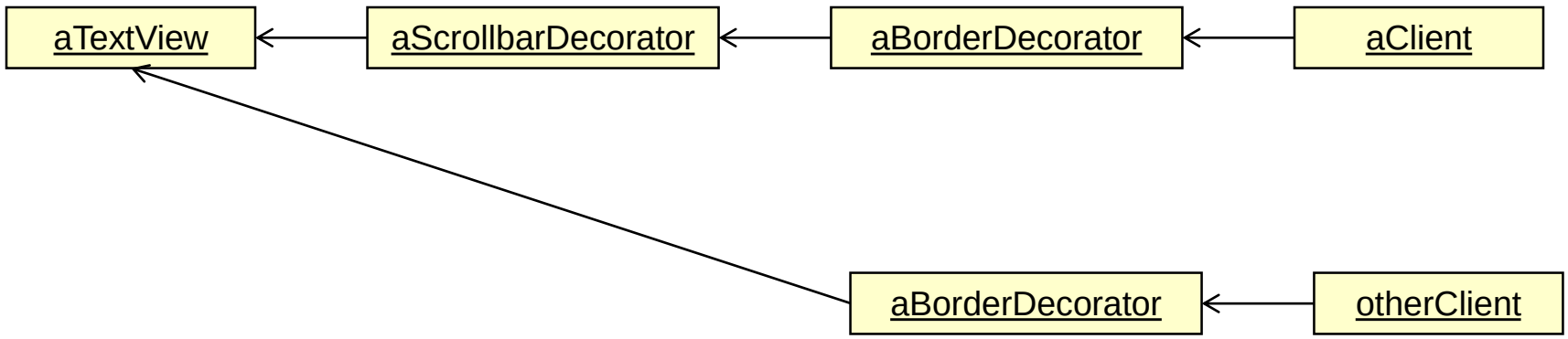
Das Decorator Pattern: Motivation

- Absicht
  - vorhandenen Objekten zusätzliche Fähigkeiten geben
  - 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

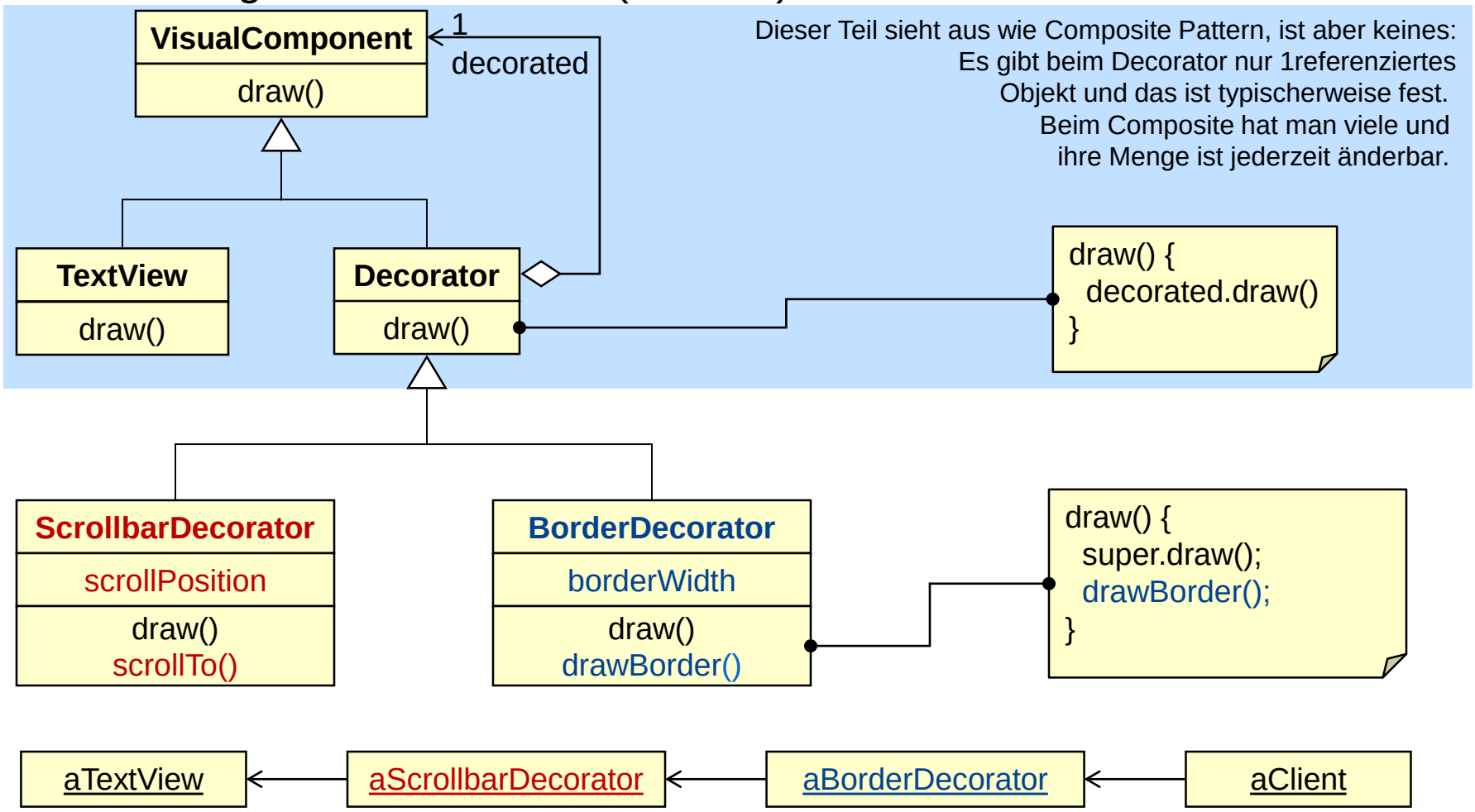
- Veränderte oder zusätzliche Fähigkeiten
  - ... sind zusätzliche Objekte
  - ... mit erweiterter Schnittstelle
  - ... zwischen ursprünglichem Objekt und Client



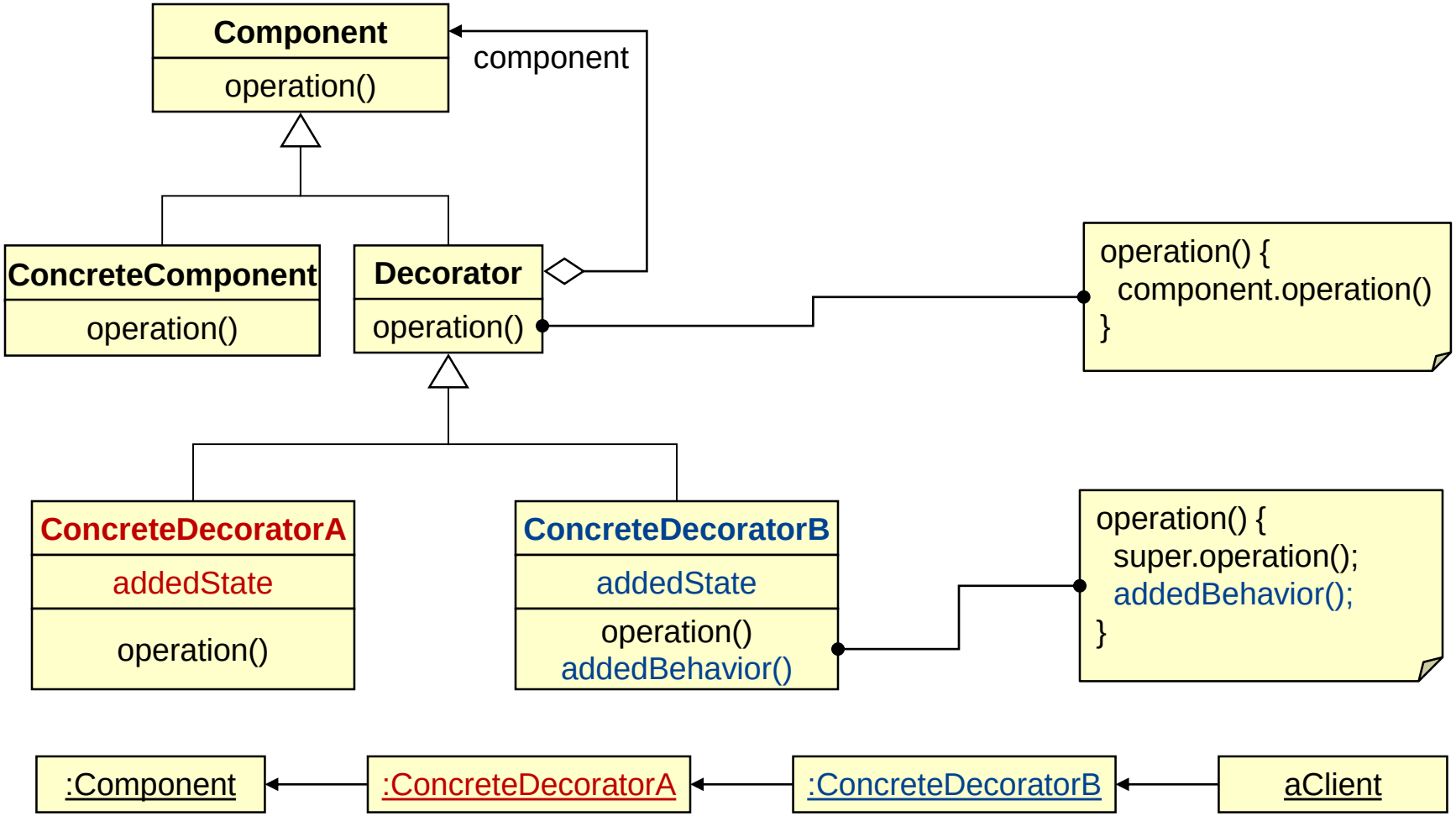
- Context-spezifische Sicht
  - ... entsteht durch Zugriff über verschiedene Decorators des gleichen Objekts

Das Decorator Pattern: Klassenhierarchie

- Vorschlag aus Gamma & al (für C++)



Decorator: Schema / Rollen



Das Decorator Pattern: Anwendbarkeit

- objekt-spezifische Eigenschaften
- kontext-spezifische Eigenschaften

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

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

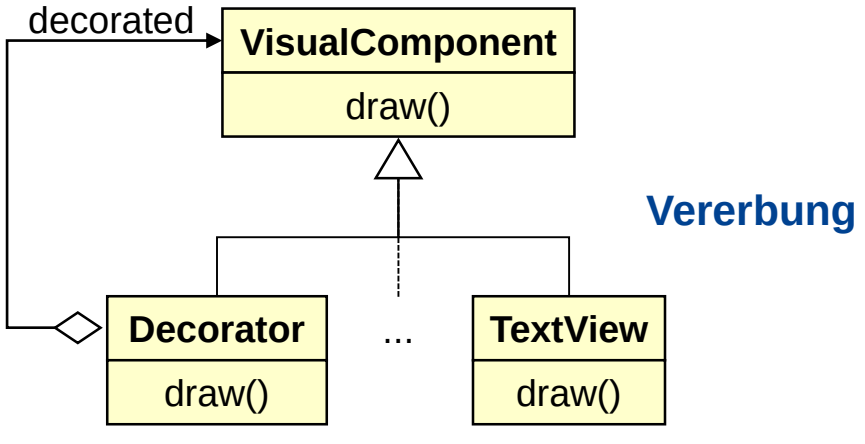
- wenn Vererbung nicht anwendbar ist

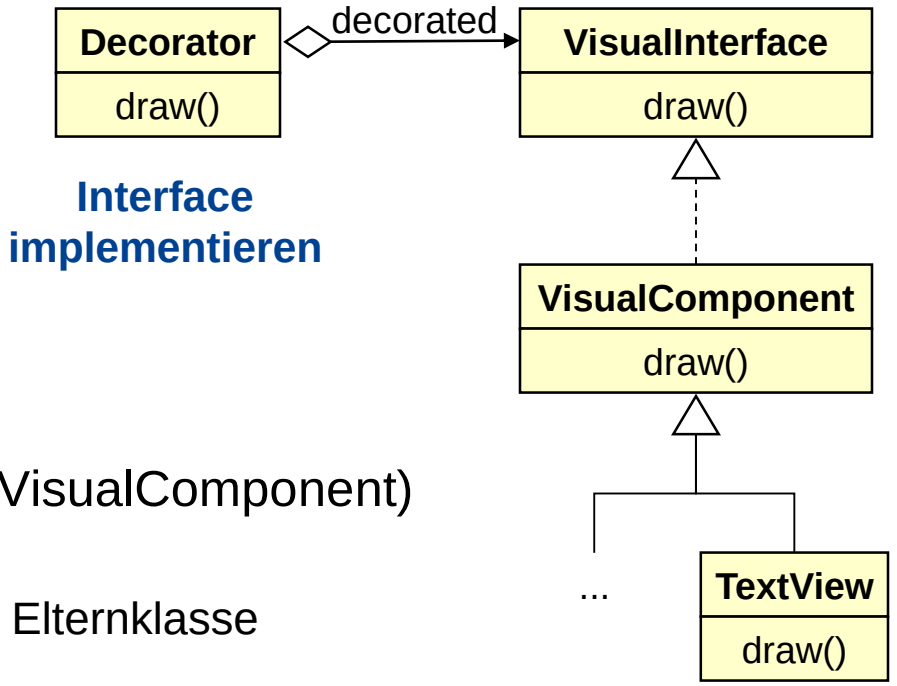
- wenn die Identität des "Hauptobjekts" nicht wichtig ist
  - siehe nächste Folie

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



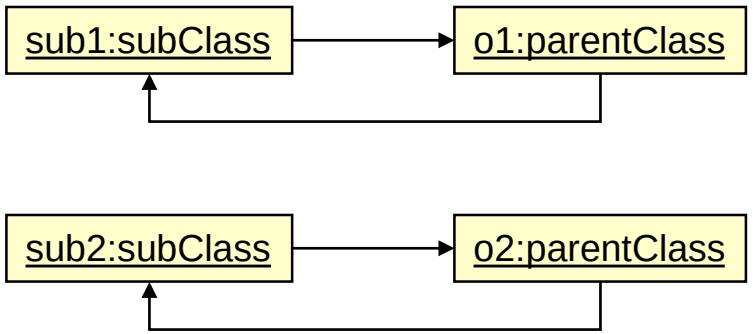


- Erweiterung der Eltern-Schnittstelle (VisualComponent)
  - von Elternklasse erben oder ...
  - gleiches Interface implementieren wie Elternklasse
- Eltern-Klasse (VisualComponent)
  - "schlankes" Interface
    - ⇨ nur was wirklich für alle Unterklassen gilt
  - keine (wenig) Variablen
    - ⇨ sonst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

Simulation multipler Vererbung versus Decorator

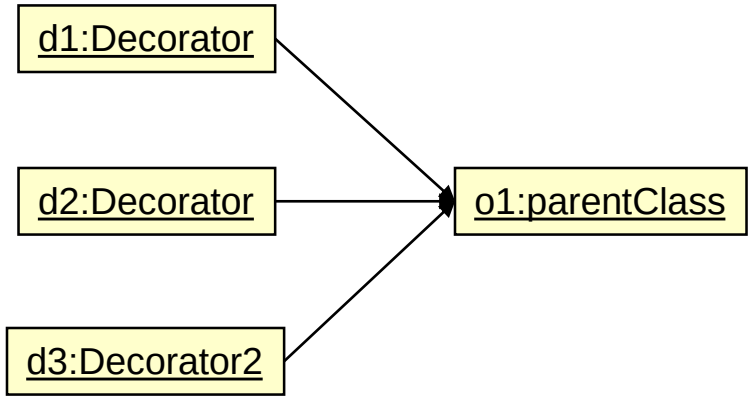
Simulation Multipler Vererbung

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



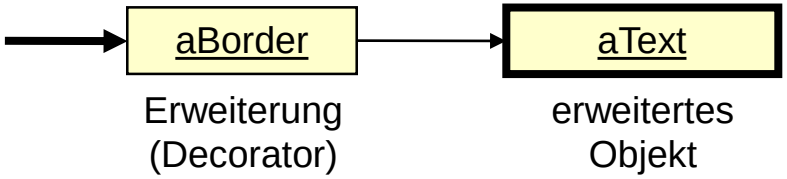
Decorator Pattern

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



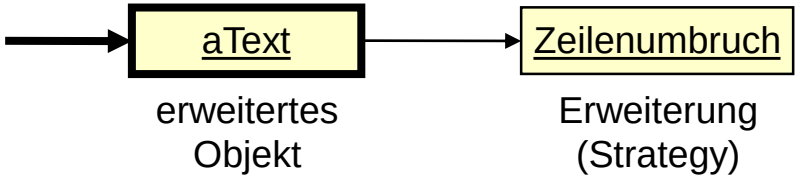
Decorator versus Strategy

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

universitätbonn

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

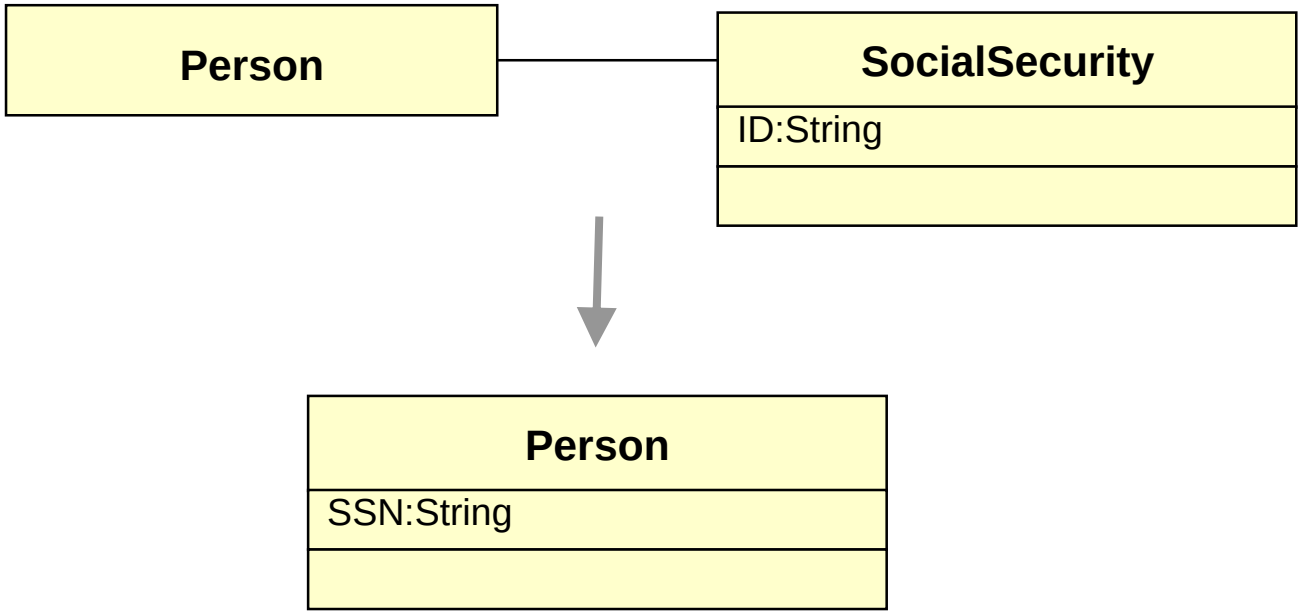
1. Umordnen der Ausführungsreihenfolge
   - Eliminiere „tote Pfade“ so früh wie möglich. (Verwende Wissen über Verteilung, Frequenz der Pfadtraversierung)
   - Grenze Suche so früh wie möglich ein
   - Überprüfe ob die Ausführungsreihenfolge von Schleifen umgekehrt werden sollte
2. Hinzufügen redundanter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

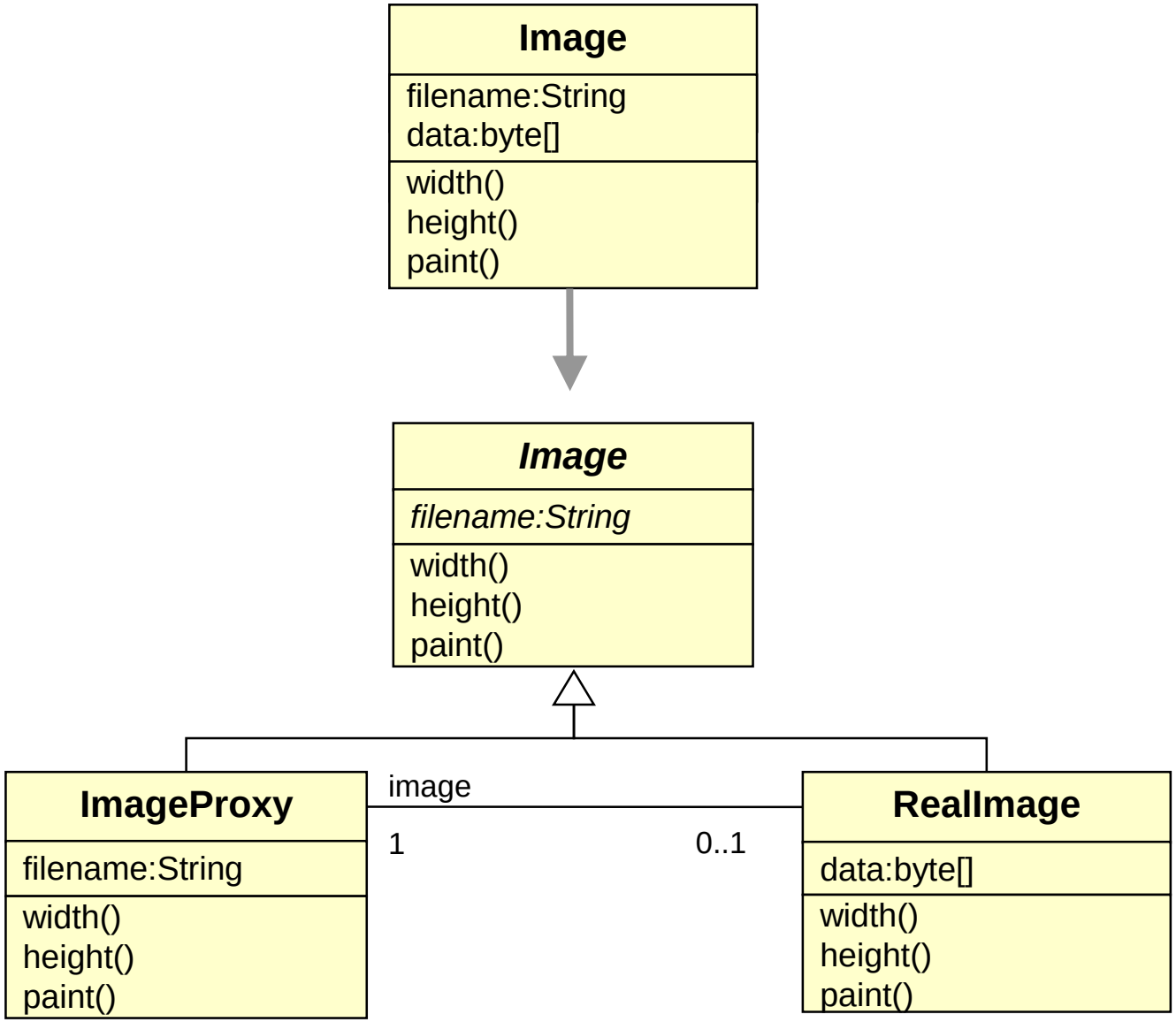
- Verwandle eine Klasse in ein Attribut, wenn get() und set() die einzigen Methoden sind, die von ihr definiert werden.



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