universitätbonn | ROOTS

Kapitel 8
„Objektorientierte Modellierung“

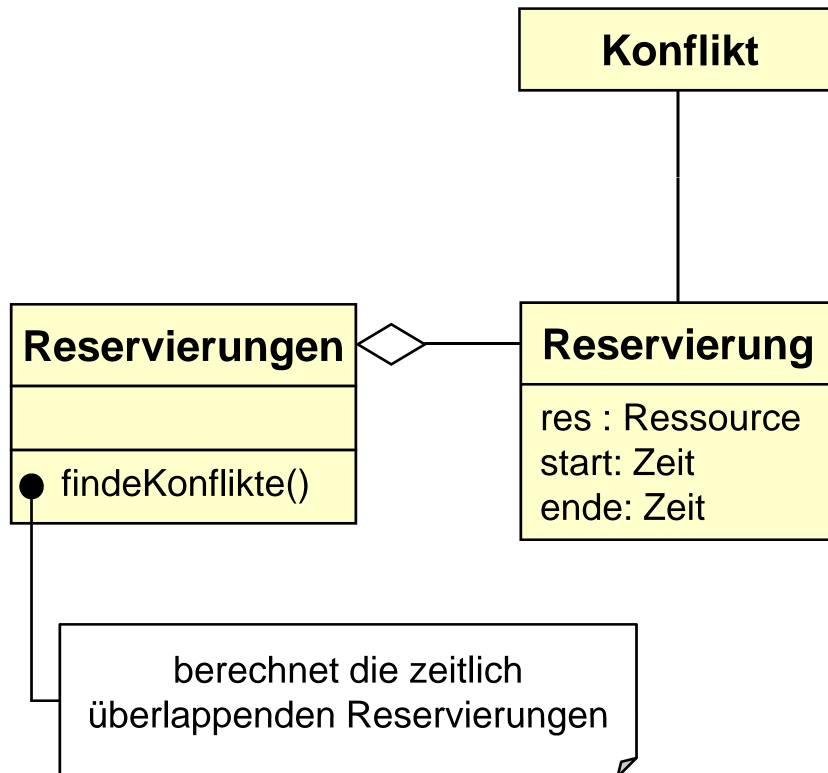
(→ Nicht in Brügge & Dutoit)

Stand: 6.01.2009

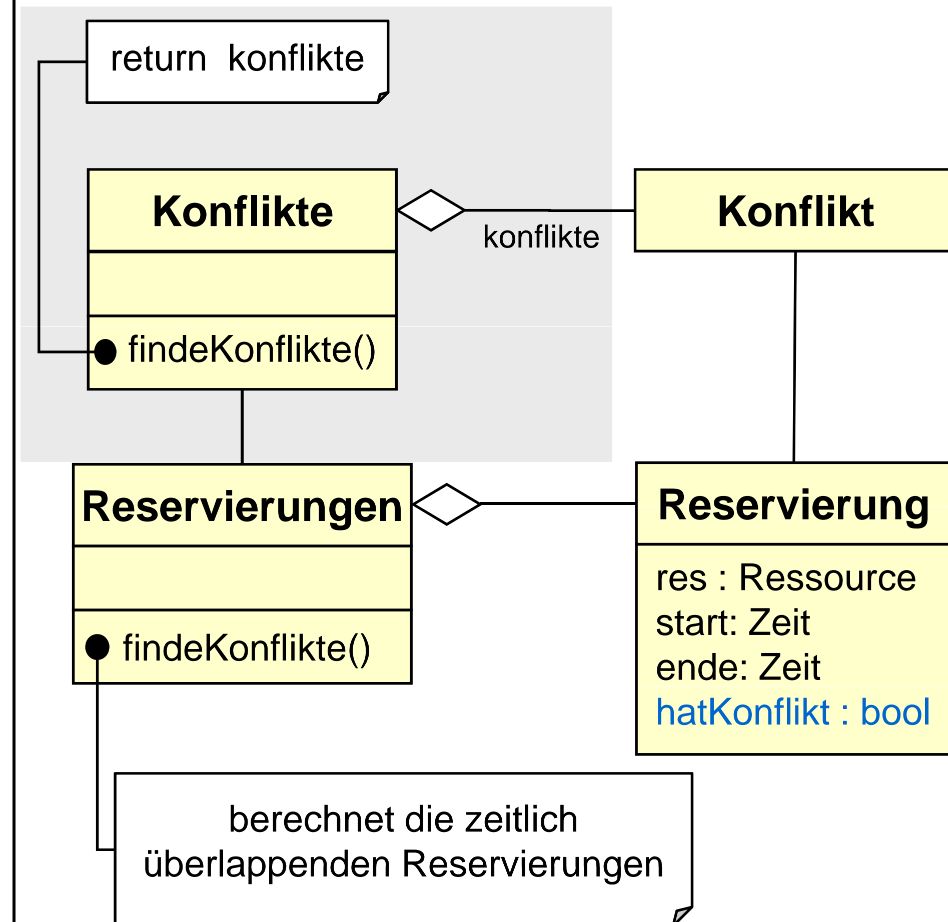
Vermeide Redundanzen
Vermeide nicht-einheitliches Verhalten
Vermeide Verwirrung von Klasse und Instanz
Vermeide fehlende Ersetzbarkeit in Typhierarchien
Vermeide Bezug auf nicht-Inhärente Typen
Reduziere Abhängigkeiten
Was sind akzeptable Abhängigkeiten?
Wie kann ich zyklische Abhängigkeiten aufbrechen?

OOM-Quiz

Was halten Sie hiervon?



Und hiervon?



Prinzip: Keine Redundanzen im Modell!

- Redundantes Modell
  - Mehrfache Wege um die gleiche Information zu erhalten
  - Speicherung abgeleiteter Informationen

- Problem
  - Bei Änderungen zur Laufzeit, müssen die Abgeleiteten Informationen / Objekte konsistent gehalten werden
  - Zusatzaufwand bei Implementierung (Benachrichtigungsmechanismus) und zur Laufzeit (Benachrichtigung über Änderung und Aktualisierung abgeleiteter Infos)
  - Änderungen im Design müssen evtl. an vielen Stellen nachgezogen werden

- Behandlung
  - Redundanz entfernen
  - ... falls sie nicht als Laufzeitoptimierung unverzichtbar ist, da die Berechnung der abgeleiteten Informationen zu lange dauert

OOM-Quiz

Was halten Sie hiervon?

Real

...

arcTan() : Winkel

Und hiervon?

<<utility>>
Trigonometrie

...

arcTan(Real):Winkel

Real

...

universitätbonn ROOTS

Prinzip: Kein Bezug auf nicht-inhärente Klassen!

- Inhärente Klasse
  - Eine Klasse A ist für eine Klasse B inhärent, wenn sie Charakteristika von B definiert

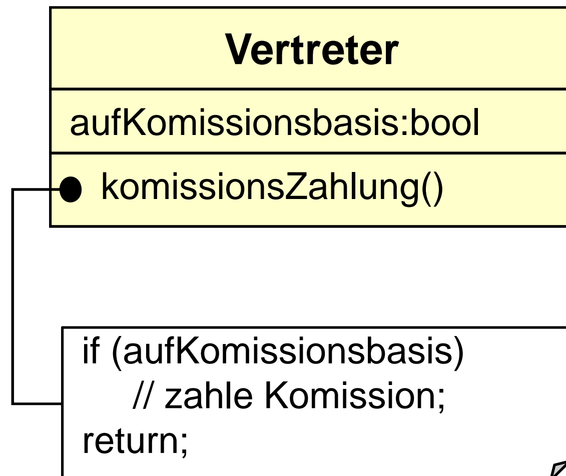
  Anders ausgedrückt:
  - Eine Klasse A ist für eine Klasse B inhärent, wenn B nicht ohne A definiert werden kann

- Problem
  - Bezug auf **nicht**-inhärente Klassen führt unnötige Abhängigkeiten ein

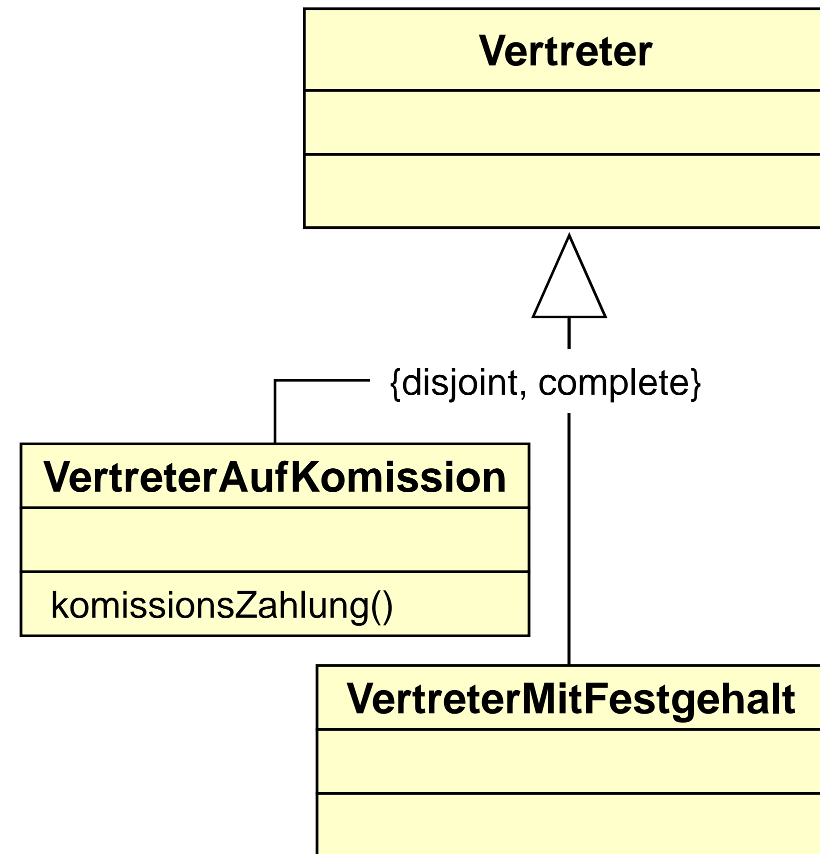
- Behandlung
  - Bezug auf nicht-inhärente Klasse entfernen
  - Eventuell Teile der Klasse in andere Klassen auslagern
    - ⇨ siehe Refactorings (z.B. „Move Method“, „Move Field“, „Split Class“)

OOM-Quiz

Was halten Sie hiervon?

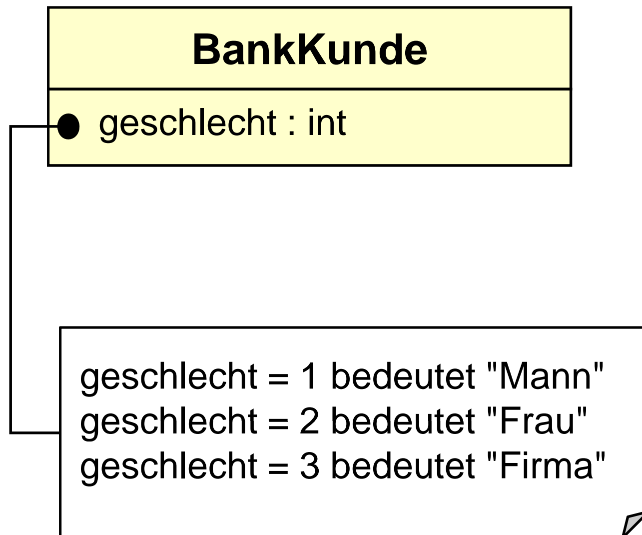


Und hiervon?

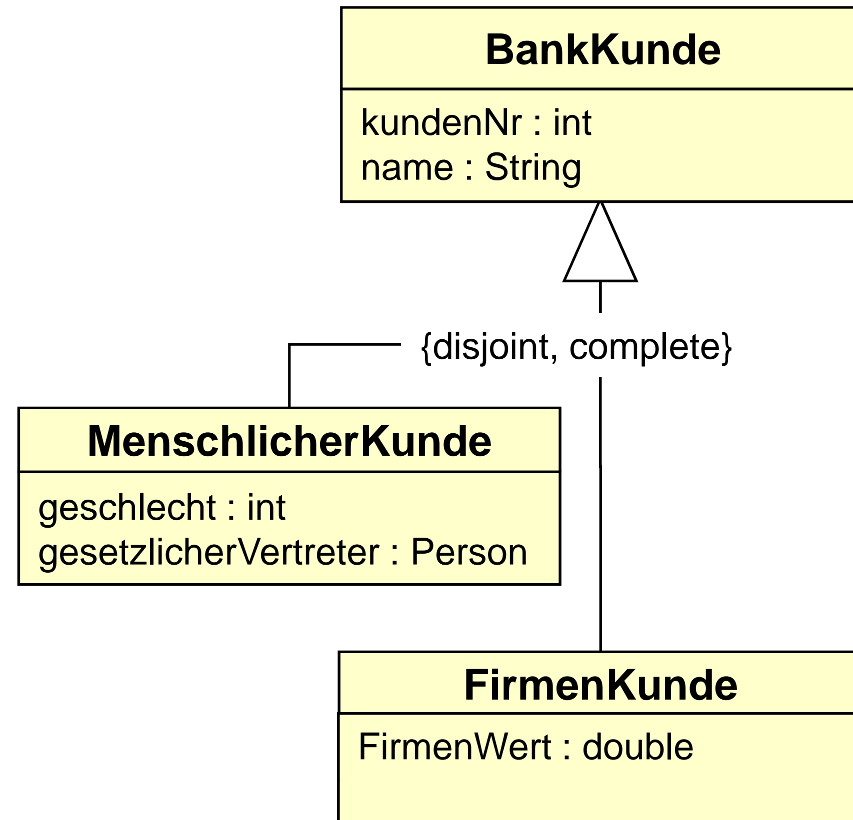


OOM-Quiz

Was halten Sie hiervon?



Und hiervon?

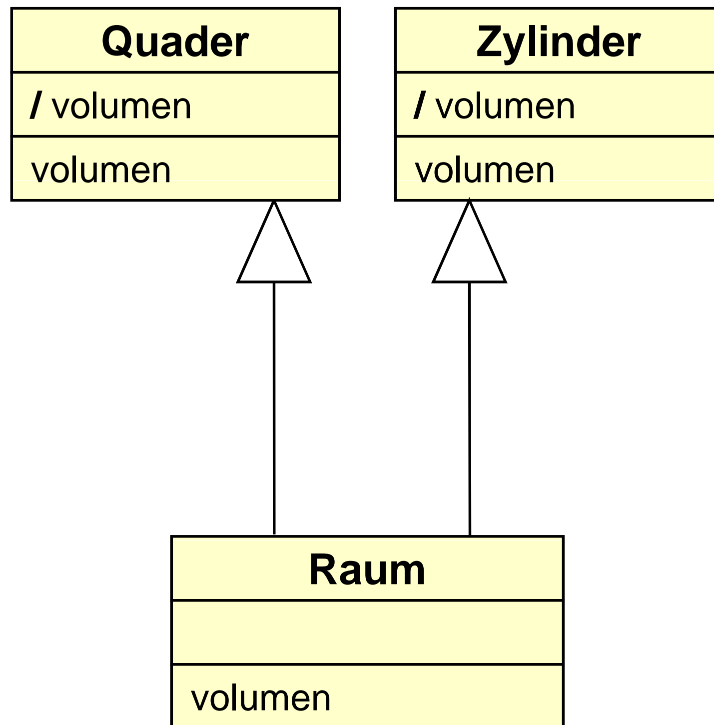


Prinzip: Nicht einheitliche Eigenschaften vermeiden!

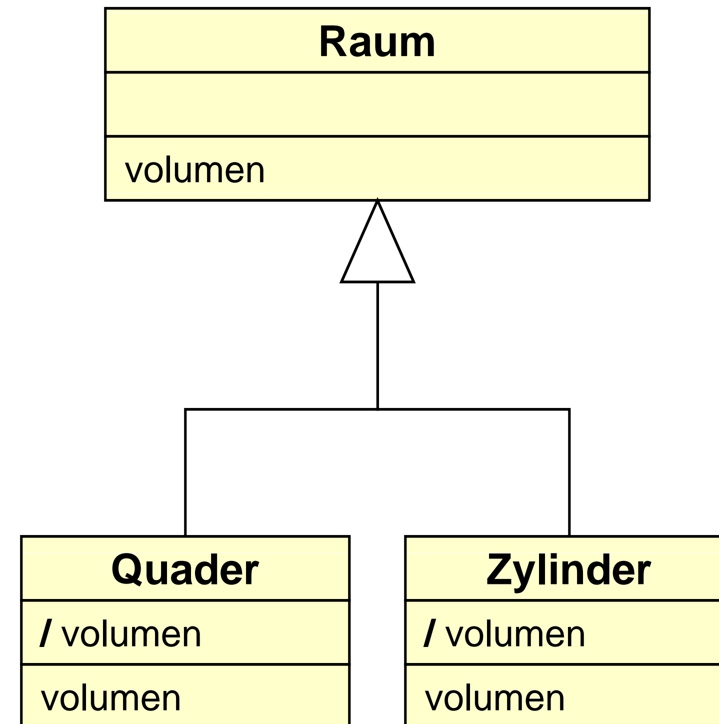
- Symptom
  - bestimmte Eigenschaften (Methoden oder Variablen) einer Klasse sind nur für manche Instanzen gültig
- Konsequenzen
  - Abhängigkeit von bestimmten Fallunterscheidungen
  - Unklare Funktionalität
  - Wartung erschwert
- Behandlung
  - Klasse aufsplitten
  - Evtl. Klasse einführen die „alle anderen Fälle“ darstellt
    - ⇨ Beispiel: „VertreterMitFestgehalt“
    - ⇨ Dadurch komplette Partition möglich
    - ⇨ Klarere Bedeutung der Klassen
      - – Walter Hürsch: „Should superclasses be abstract?“, p. 12-31, ECOOP 1994 Proceedings, LNCS 821, Springer Verlag.

OOM-Quiz

Was halten Sie hiervon?

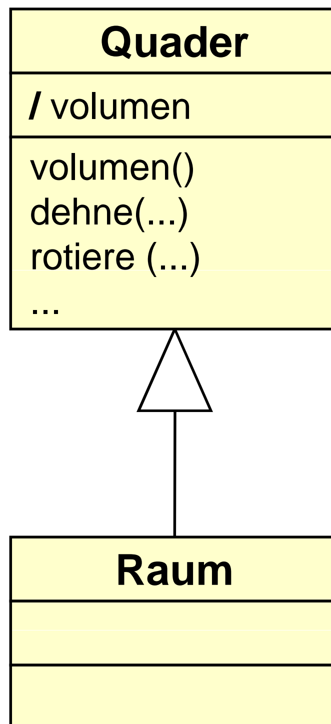


Und hiervon?

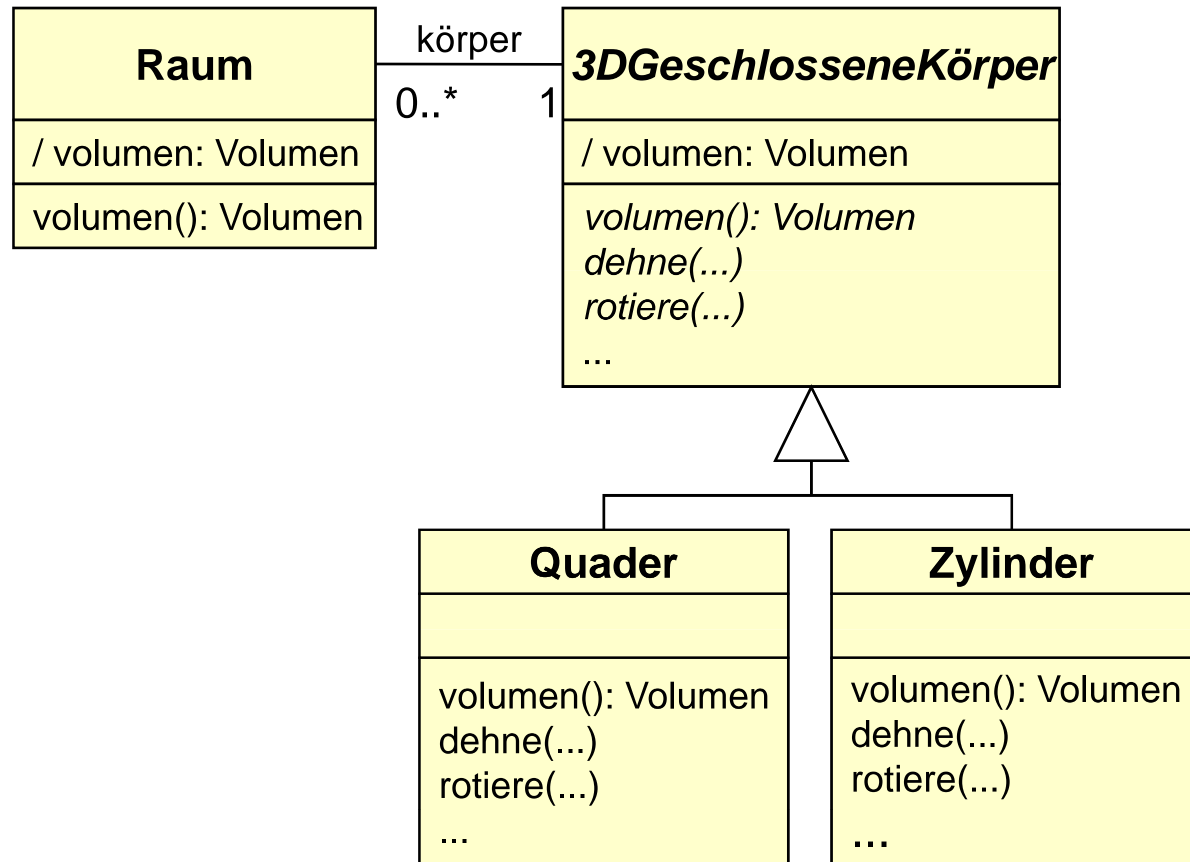


OOM-Quiz

Was halten Sie hiervon?



Und hiervon?

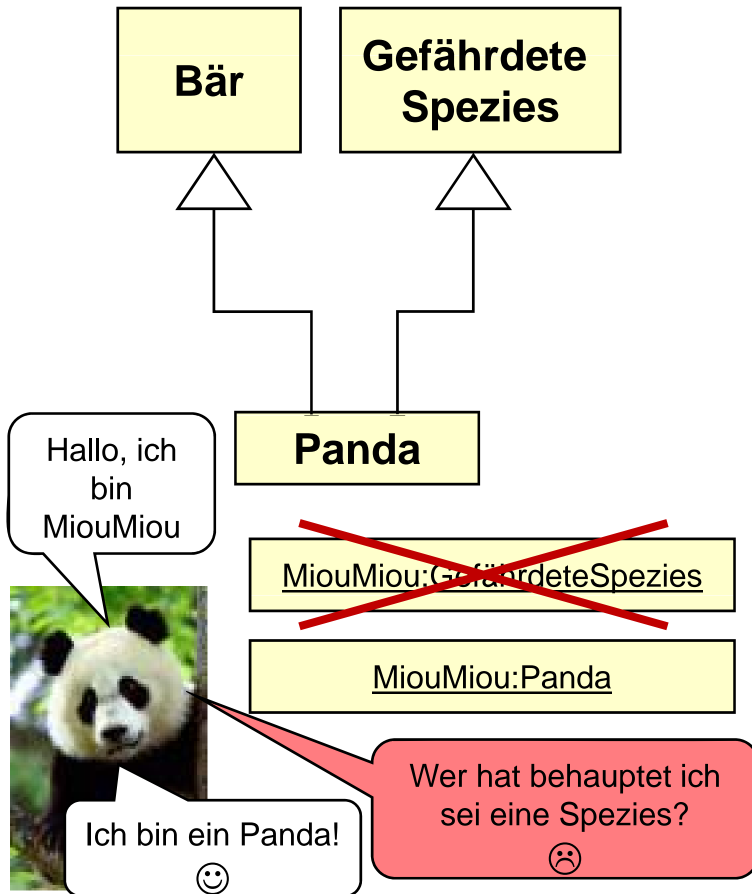


Problem: Fehlende Ersetzbarkeit

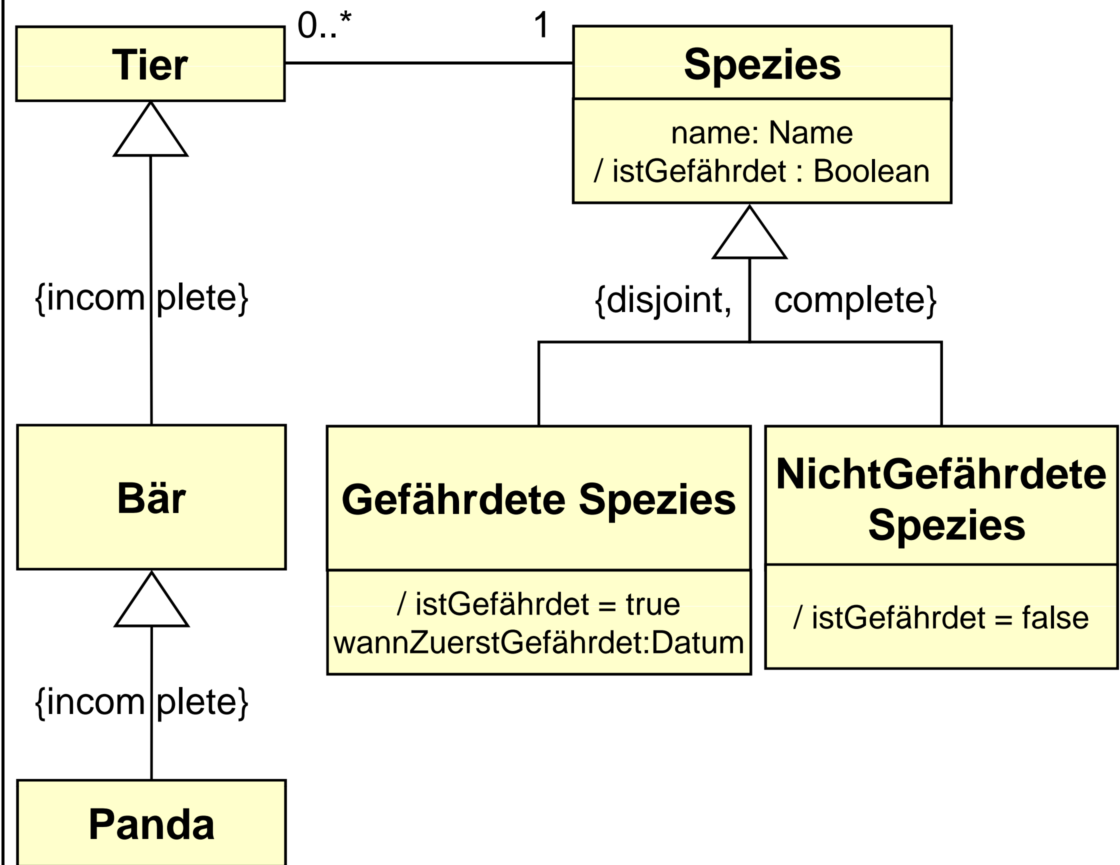
- Unterklasse hat eine stärkere Invariante als die Oberklasse
  - Dreidimensionale Körper dürfen rotiert werden
  - Räume dürfen nicht rotiert werden!
- Daraus resultiert fehlende Ersetzbarkeit
  - In einem Kontext wo man Rotierbarkeit erwartet darf kein nicht-rotierbares Objekt übergeben werden
- Wichtig: Frühzeitig auf Interfaces achten
  - Sie sind das Kriterium um über Ersetzbarkeit zu entscheiden
- Frage: Wie finden wir Interfaces?
  - → CRC Cards!

OOM-Quiz

Was halten Sie hiervon?



Und hiervon?

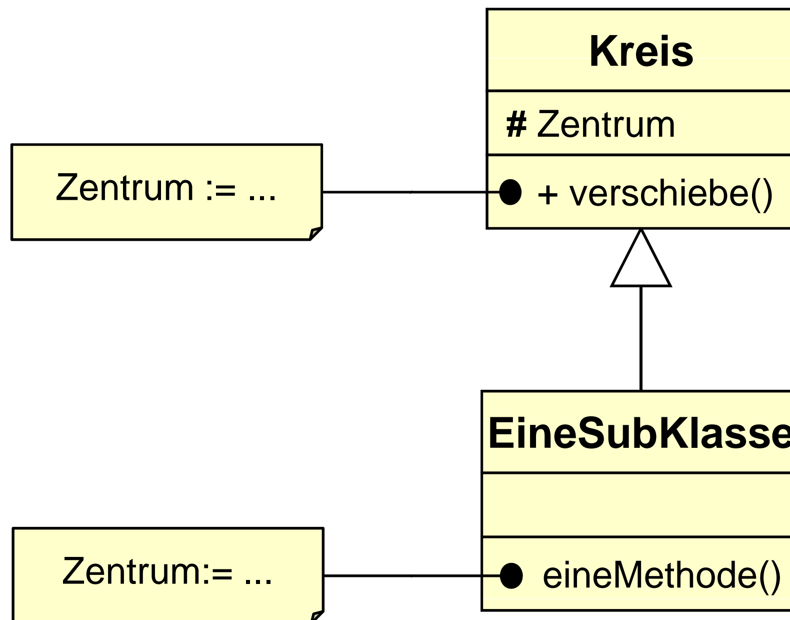


Prinzip: Keine Verwirrung von Klassen und Instanzen!

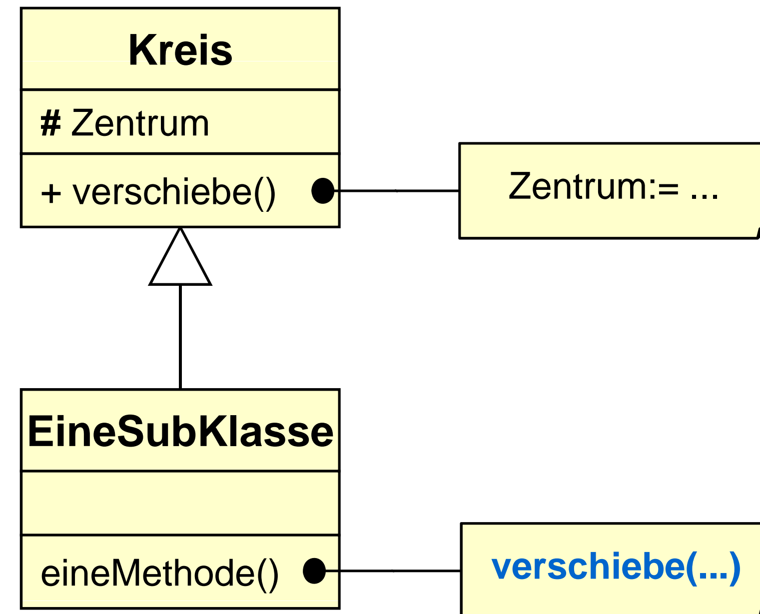
- Falle: Natürliche Sprache unterscheidet oft nicht zwischen
  - Klasse: "Panda" als Name einer Spezies
  - Instanz: "Panda" als Bezeichnung für ein einzelnes Tier
- Ähnlich im technischen Bereich
  - "Produkt" → Produktline (z.B. Telefone)
  - "Produkt" → einzelnes Produkt (z.B. mein Handy)
- Frage: „Wer ist Instanz wovon?“
  - "Miou-Mio ist ein Bär": OK
  - "Miou-Mio ist ein Panda": OK
  - "Miou-Mio ist eine Spezies": FALSCH!
- Alternatives Kriterium
  - Ersetzbarkeit
  - Bsp.: Kann ich "Miou-Miou" überall da einsetzen, wo ich eine Spezies erwarte?

OOM-Quiz

Was halten Sie hiervon?



Und hiervon?



Prinzip: Abhängigkeiten vermeiden!

- Eine Abhängigkeit zwischen A und B besteht wenn
  - Änderungen von A Änderungen von B erfordern (oder zumindestens erneute Verifikation von B)
  - ... um Korrektheit zu garantieren

- Eine Kapselungseinheit ist
  - eine **Methode**: kapselt Algorithmus
  - eine **Klasse**: kapselt alles was zu einem Objekt gehört

- Prinzip
  - Abhängigkeiten zwischen Kapselungseinheiten reduzieren
  - Abhängigkeiten innerhalb der Kapselungseinheiten maximieren

- Nutzen
  - Wartungsfreundlichkeit

Beispiele: Abhängigkeiten von ...

- **Konvention**
  - if order.accountNumber > 0 // was bedeutet das?
- **Wert**
  - Problem: Konsistent-Haltung von redundant gespeicherten Daten
- **Algorithmus**
  - a) implizite Speicherung in der Reihenfolge des Einfügens wird beim Auslesen vorausgesetzt
  - b) Einfügen in Hashtable und Suche in Hashtable müssen gleichen hash-Algorithmus benutzen
- **Impliziten Annahmen**
  - Werte die Hash-Schlüssel müssen unveränderlich sein
    - ⇨ Achtung bei Aliasing

Reduktion von Abhängigkeiten durch „Verbergen von Informationen“ („information hiding”)

- “Need to know” Prinzip → „Schlanke Schnittstelle“
  - Zugriff auf eine bestimmte Information nur dann allgemein zulassen, wenn dieser wirklich gebraucht wird.
  - Zugriff nur über wohldefinierte Kanäle zulassen, so dass er immer bemerkt wird.
- Je weniger eine Operation weiß...
  - ... desto seltener muss sie angepasst werden
  - ... um so einfacher kann die Klasse geändert werden.
- Zielkonflikt
  - Verbergen von Informationen vs. Effizienz

Information Hiding (2)

- Verbergen von internen Objekten an den Grenzen des Subsystems
  - Definiere abstraktes Schnittstellen die zwischen dem System und der externen Welt sowie zwischen Subsystemen vermitteln (→ Facade)
- Verberge Datenstrukturen einer Klasse
  - Nur Methoden der selben Klasse dürfen auf deren Attribute zugreifen
- Führe eine Operation nicht auf dem Ergebnis einer anderen aus.
  - Schreibe eine neue Operation, die die Navigation zu der Zielinformation kapselt. (→ „Law of Demeter": keine „transitiven" Abhängigkeiten)

Law of Demeter („Talk only to your friends!")

- **Klasse sollte nur von „Freunden", d.h. den Typen der eigenen Felder, Methoden- und Ergebnisparameter abhängig sein.**
- **Insbesondere sollte sie nicht Zugriffsketten nutzen, die Abhängigkeiten von den Ergebnistypen von Methoden der Freunde erzeugen. Beispiel:**
  - **Nicht: param.m().mx().my()….;**
  - **Sondern: param.mxy(); wobei die methode mxy() den transtiven Zugriff kapselt.**

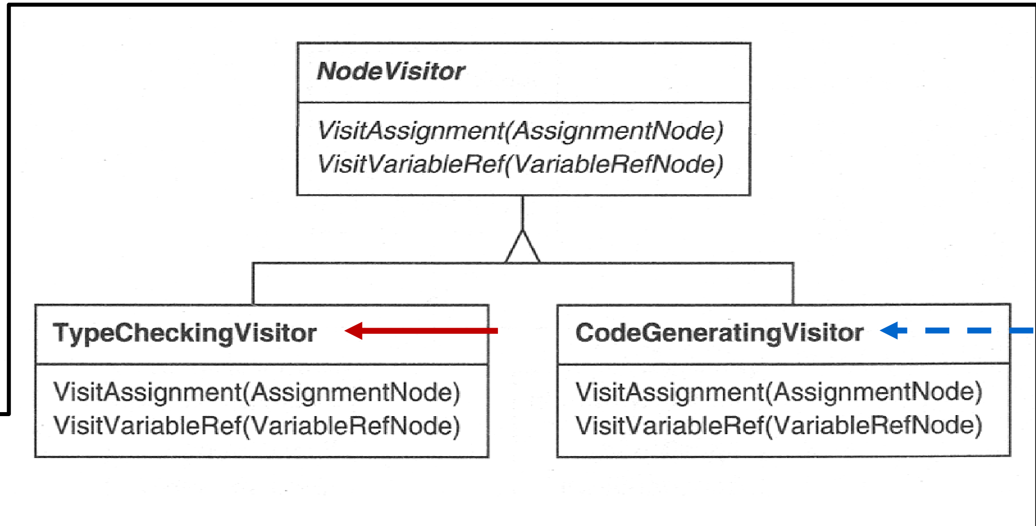
- Zielkonflikt
  - Verbergen von Informationen vs. „schlanke" Schnittstellen.

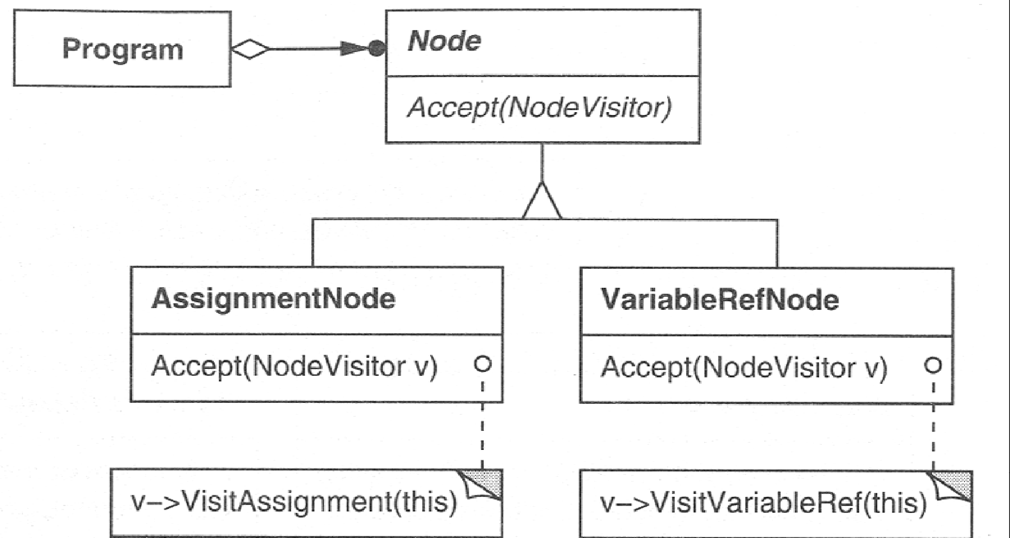
Gibt es „gute“ Abhängigkeiten?

Kriterien für „problematische“ und „akzeptable“ Abhängigkeiten am Beispiel des Visitor Patterns

Erinnerung an das Visitor-Pattern

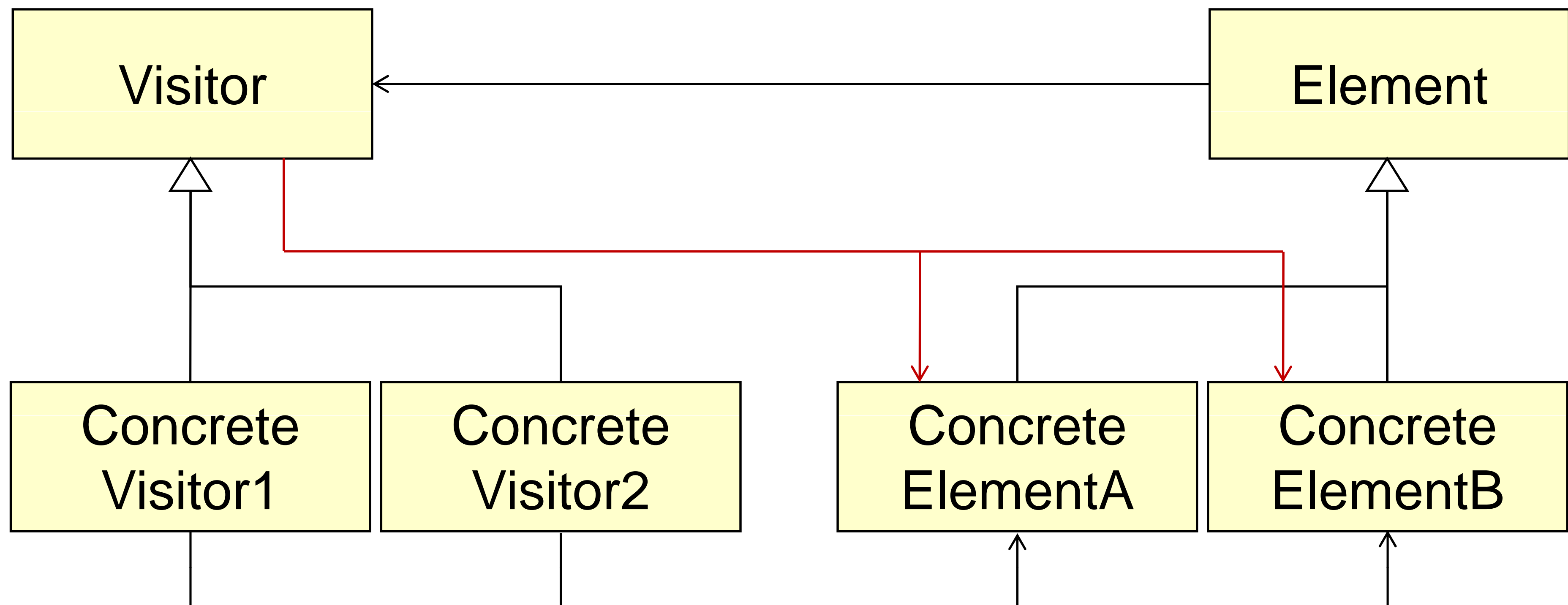
- Ziel
  - Neue Operationen auf Objekten sollen definiert werden, ...
  - ... ohne dass die Klassen dieser Objekte geändert werden müssen!
- Idee
  - Zusammenfassung dieses Codes in Visitor-Objekte...
  - ... und Bereitstellung von "akzeptierenden" Methoden in der betroffenen Klassenhierarchie.





Abhängigkeiten im Visitor-Pattern

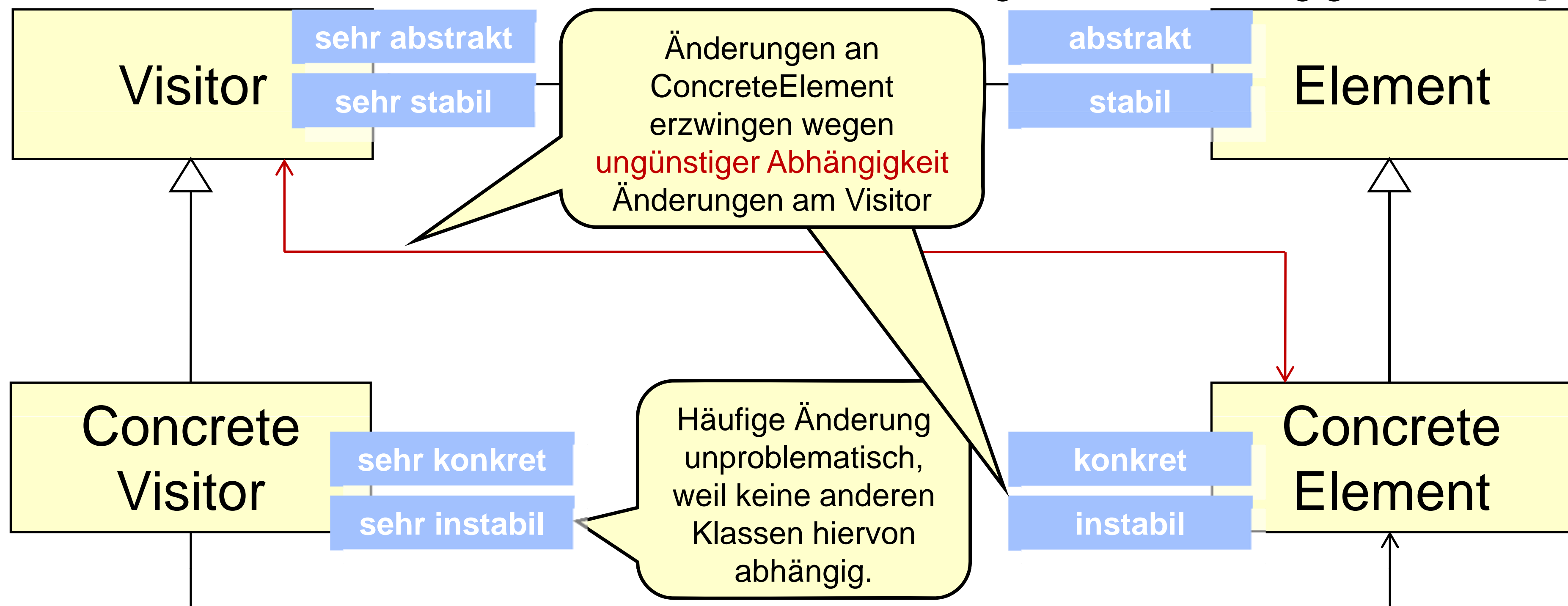
- ConcreteVisitor1 und ConcreteVisitor2 haben gleiche Abhängigkeiten.
- ConcreteElementA und ConcreteElementB ebenso.
- Ab jetzt in der Darstellung nur ein Repräsentant
- Bei Vererbung sind damit immer mehrere abgeleitete Klassen gemeint.



© 2000-2008 Dr. G. Kniesel

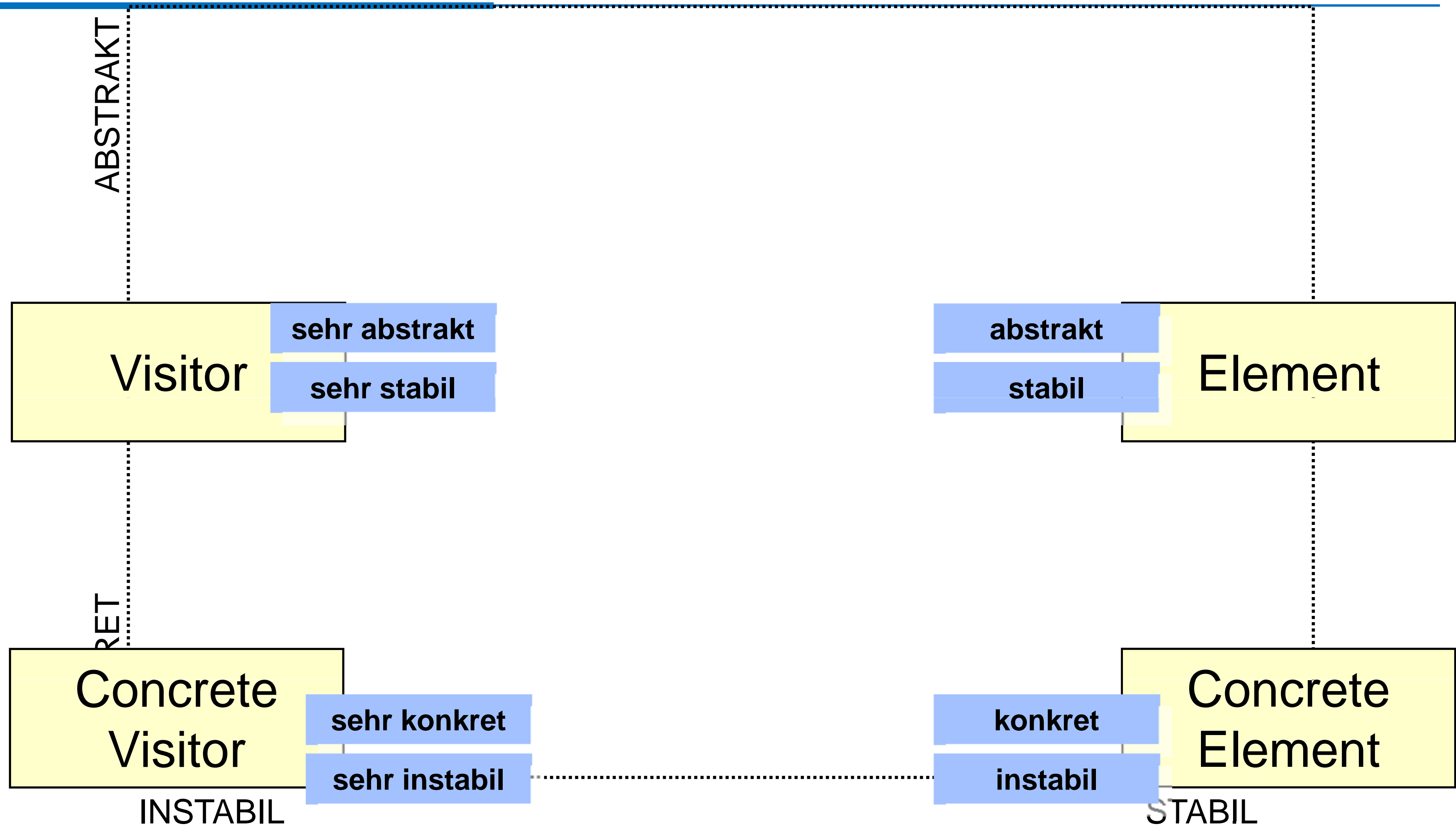
Stabilität und Abstraktheit im Visitor-Pattern

- Abstraktheit:
  - Grad der Unabhängigkeit von einer bestimmten Anwendung
  - Abstrakte Klassen, Interfaces
- Stabilität:
  - Geringe Wahrscheinlichkeit sich ändernder Kundenanforderungen
  - [Hoher Anpassungsaufwand wegen vieler abhängiger Klassen]



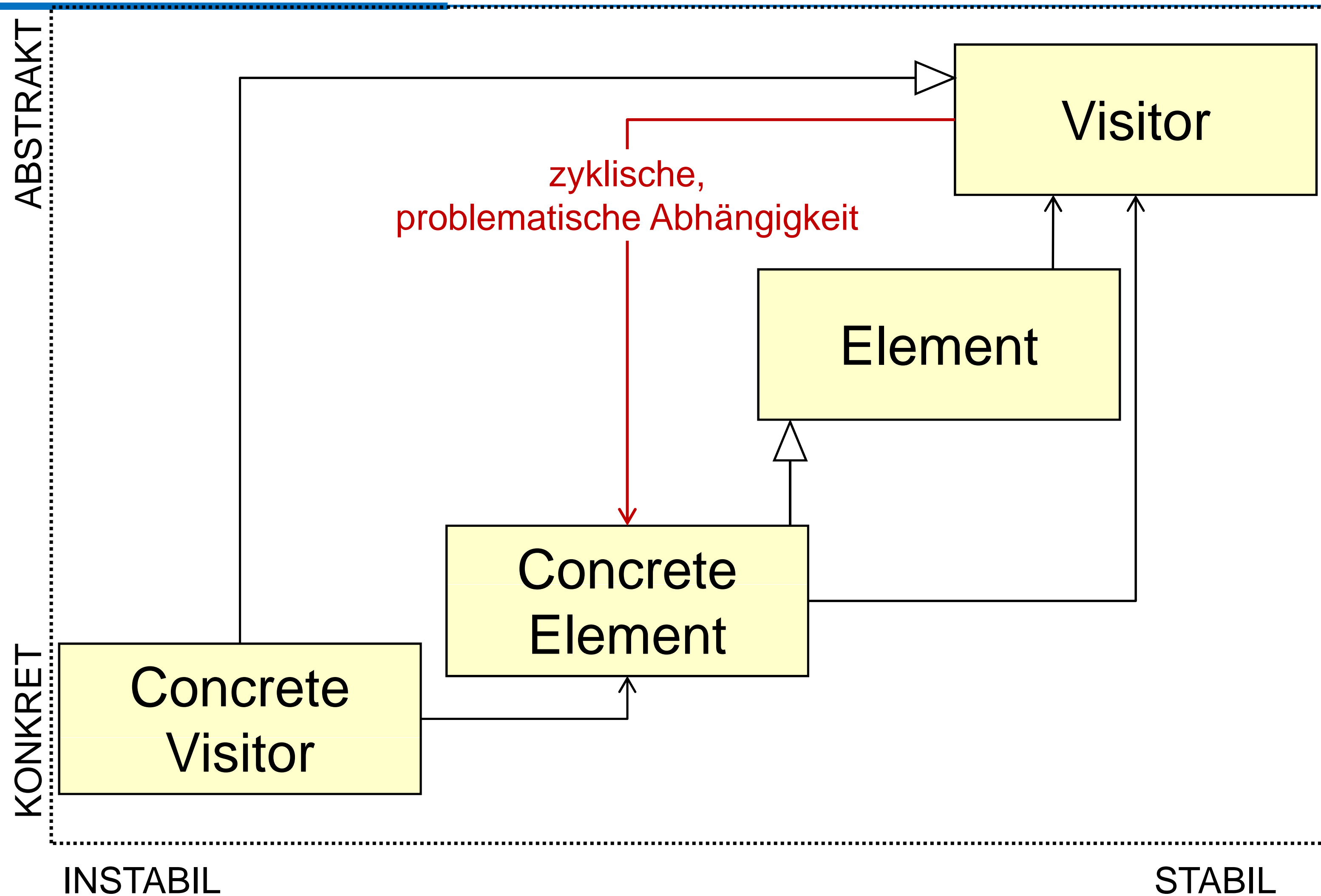
universitätbonn ROOTS

Bewertung der Abhängigkeiten im Visitor-Pattern

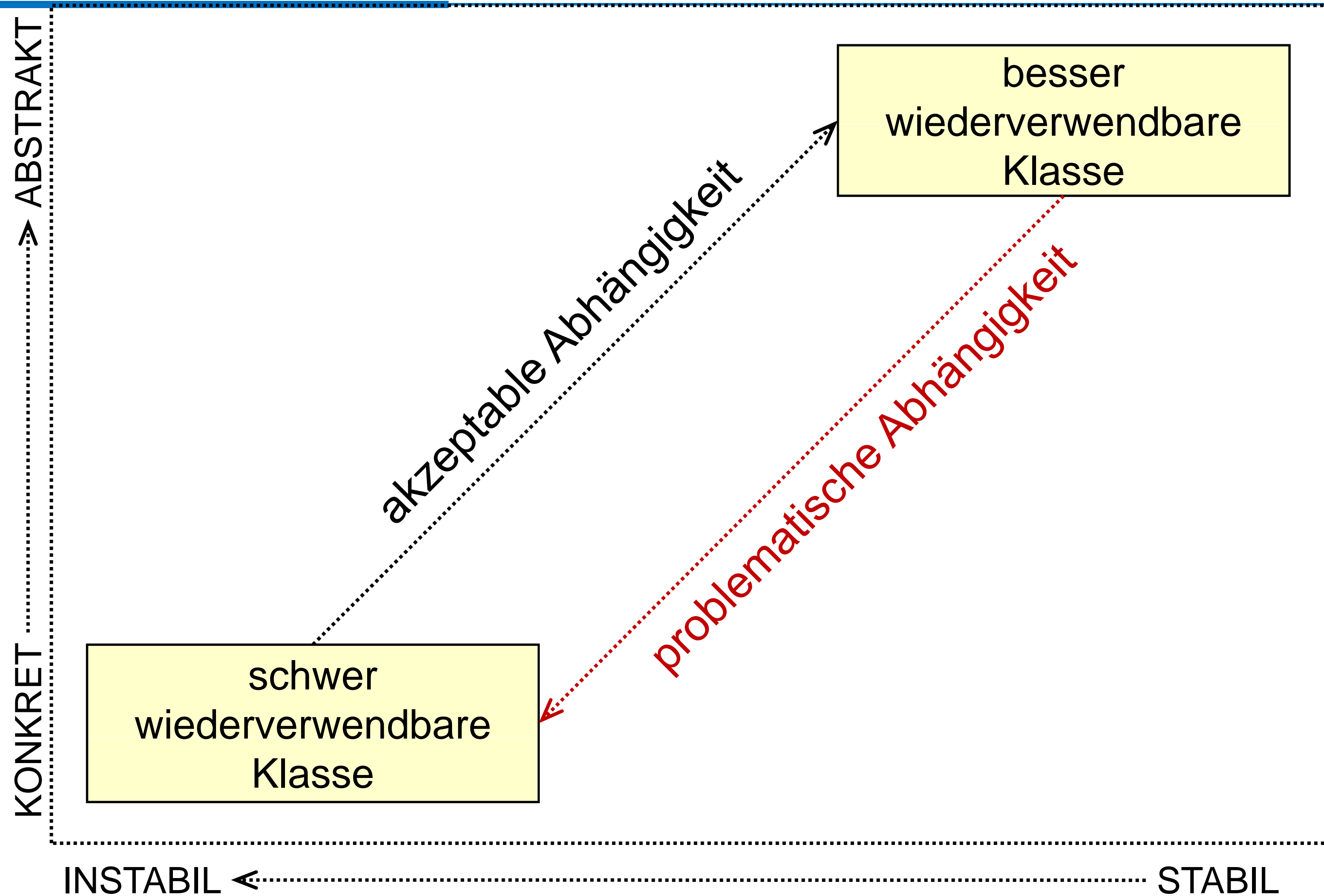


universitätbonn ROOTS

Bewertung der Abhängigkeiten im Visitor-Pattern



Ungünstige Abhängigkeiten mit einem Blick erkannt



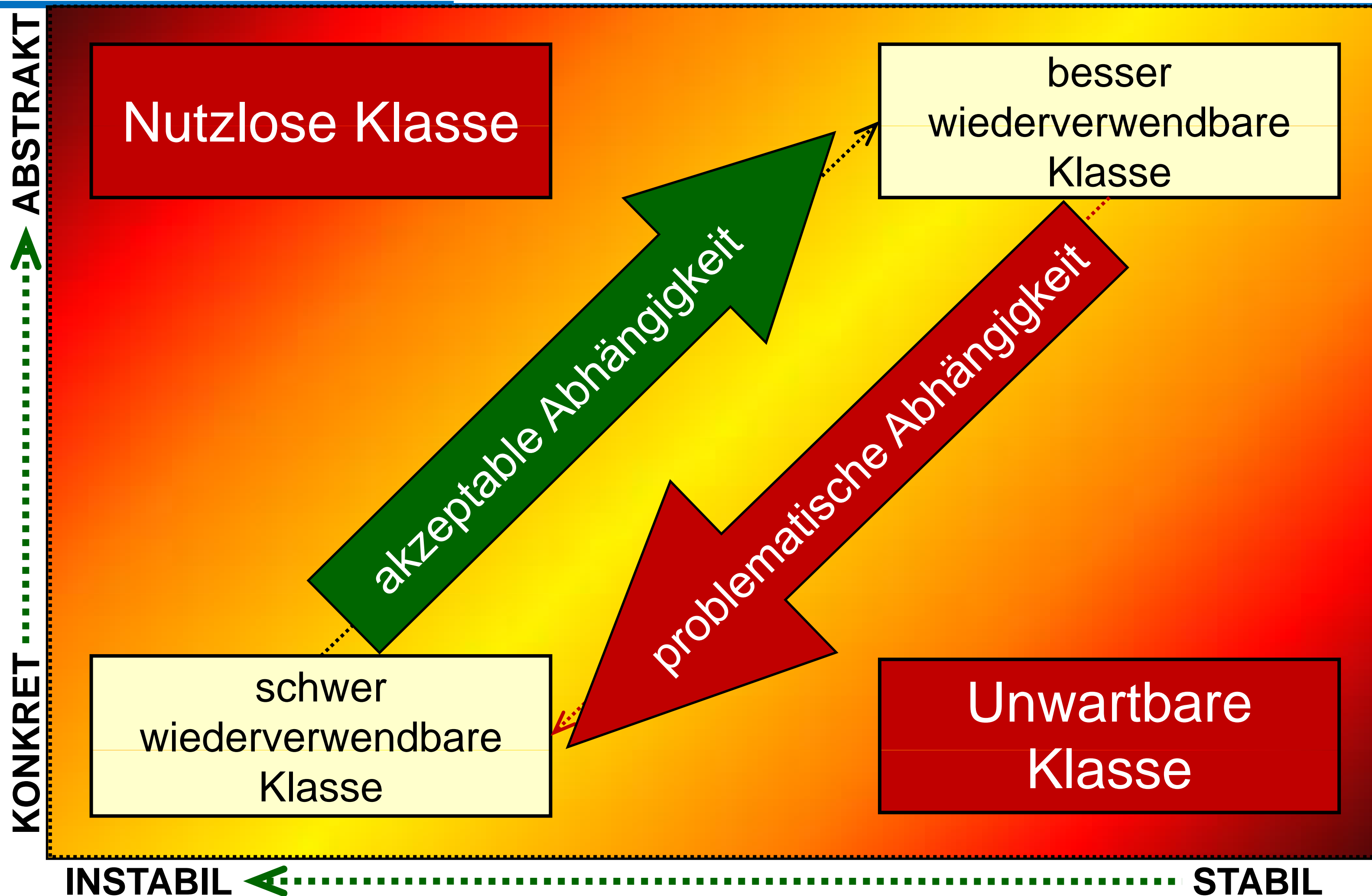
Vorzeichen von schlechtem Design
(Bad Signs of Rotting Design, Robert C. Martin, 1996)

- Starrheit
  - Code ist schwierig zu ändern
  - Die Zurückhaltung etwas zu ändern wird zur Regel
- Zerbrechlichkeit
  - Selbst kleine Änderungen können zu verketteten Effekten führen
  - Der Code wird an unerwartet Stellen inkonsistent
- Unbeweglichkeit
  - Der Code ist so verworren, dass es unmöglich ist etwas wiederzuverwenden
  - Ein Modul könnte in einem anderen System wiederverwendet werden, jedoch ist der Aufwand und das Risiko zu groß das Modul aus seinem Umfeld zu lösen
- Zähigkeit
  - Einfacher zu “hacken”als das Ausgangs Design zu erhalten
- → Abhängigkeits-Management notwendig!!!

Entwicklungs - Prinzipien (Design Principles, Robert C. Martin, 1996)

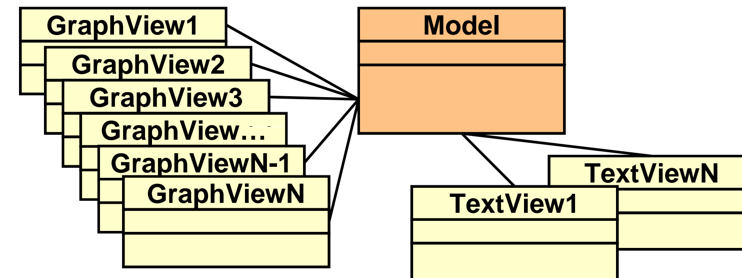
- Dependency Inversion Principle (DIP)
  - Abhängigkeiten nur zu Abstrakterem, nicht zu Konkreterem!
- Acyclic Dependencies Principle (ADP)
  - Der Abhängigkeitsgraph veröffentlichter Komponenten muss azyklisch sein!
- Stable Dependencies Principle (SDP)
  - Abhängigkeiten nur zu Stabilerem, nicht zu Instabilerem!
- Stable Abstractions Principle (SAP)
  - Abstraktion und Stabilität eines Paketes sollten zueinander proportional sein. Maximal stabile Pakete sollten maximal abstrakt sein. Instabile Pakete sollten konkret sein.

Dependency Inversion Principle (DIP) & Stable Dependency Principle (SDP)

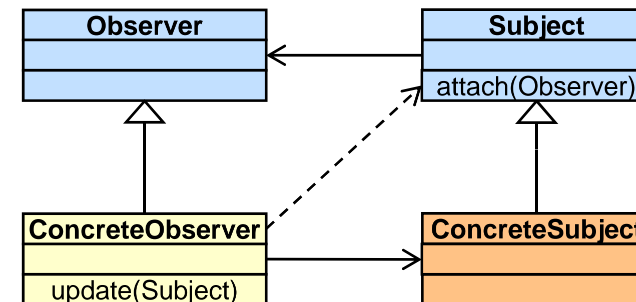


Beispiel: Abhängigkeiten mit und ohne Observer

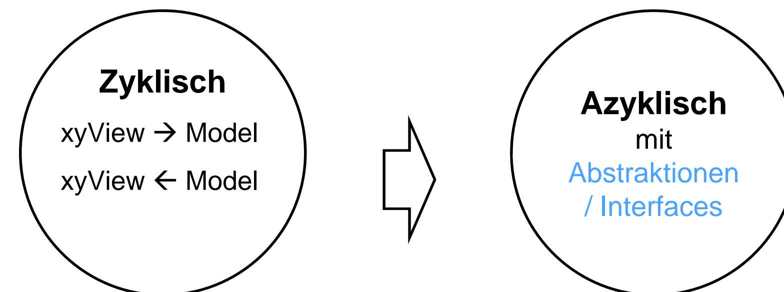
- Ohne Observer
  - Gegenseitige Abhängigkeiten der konkreten Typen.



- Mit Observer
  - ConcreteObserver ist von Concrete Subject abhängig.
  - Concrete Subject ist nur von abstraktem Subject abhängig.



- Modellierungsprinzip
  - Aufbrechen zyklischer Abhängigkeiten durch Einführung einer Abstraktion von der beide Partner azyklisch abhängig sind.



universitätbonn ROOTS

Zusammenfassung & Ausblick

© 2000-2008 Dr. Günter Kniesel

OO Modellierung: Rückblick

- CRC-Cards → Identifikation
  - Klassen
  - Operationenen
  - Kollaborationen
- Design by Contract → Verfeinerung des Verhaltens
  - Vorbedingungen
  - Nachbedingungen
  - Invarianten
  - Ersetzbarkeit
- Prinzipien → Strukturierung von OO-Modellen: Vermeiden von
  - Redundanzen
  - Nicht-einheitlichem Verhalten
  - Verwirrung von Klasse und Instanz
  - Fehlender Ersetzbarkeit
  - Abhängigkeiten von nicht-inhärenten Typen
  - Abhängigkeiten von spezielleren oder instabileren Typen

OO Modellierung: Weiterführende Informationen

- Modellierungs-Prinzipien („Quiz“)
  - Page-Jones, „Fundamentals of Object-Oriented Design in UML“, Addison Wesley, 1999
  - Sehr empfehlenswert!

- CRC-Cards
  - Konzentriert: Fowler & Scott, „UML distilled“ (2te Ausgabe), Addison Wesley
  - Original-Artikel: http://c2.com/doc/oopsla89/paper.html

- Design by Contract
  - Konzentriert: Fowler & Scott, „UML distilled“ (2te Ausgabe), Addison Wesley
  - Original-Buch: Bertrand Meyer, „Object-oriented Software Construction“, Prentice Hall, 1997.
    - ⇨ Ein Klassiker!

OO Modellierung: Weiterführende Informationen

- Abhängigkeiten (Stable dependency principle, …)
  - Robert C. Martin: Design Principles and Design Patterns
    http://www.objectmentor.com/resources/articles/Principles_and_Patterns.PDF
- Brechen von Abhängigkeiten mit Hilfe Aspektorientierter Programmierung
  - Martin E. Nordberg: Aspect-Oriented Dependency Inversion.
    OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems, October 2001
    http://www.cs.ubc.ca/~kdvolder/Workshops/ OOPSLA2001/submissions/12-nordberg.pdf
  - Jan Hannemann and Gregor Kiczales:
    "Design Pattern Implementation in Java and AspectJ",
    http://www.cs.ubc.ca/~jan/papers/oopsla2002/oopsla2002.html
  - Wes Isberg: Tutorial, AOSD 2004 (Folien nicht öffentlich verfügbar)
    http://aosd.net/2004/tutorials/goodaop.php
  - Vorlesung AOSD, Wintersemester 2006, Universität Bonn
    http://roots.iai.uni-bonn.de/teaching/vorlesungen/2006aosd

Halde: Folien evtl. ins Kapitel „Entwurfsmuster“ übertragen