

Kapitel 5 „Objektorientierte Modellierung“

Stand: 05.12.2016



Hoffentlich klarer
(S. 53 bis 56)

Identifikation und Spezifikation von Schnittstellen
Modellierungsprinzipien

Schnittstellen

| | |
|--------------------------------------|--------------------|
| <u>Schnittstellenidentifikation:</u> | CRC-Karten |
| <u>Schnittstellenfestlegung:</u> | Signaturen |
| <u>Schnittstellenverfeinerung:</u> | Design by Contract |

Schnittstellen-Identifikation: CRC-Cards

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

Class (Klassen-Name)

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

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

Responsibility
(Aufgaben)

Collaboration
(Zusammenarbeit)

Einsatz von CRC-Cards

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

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

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

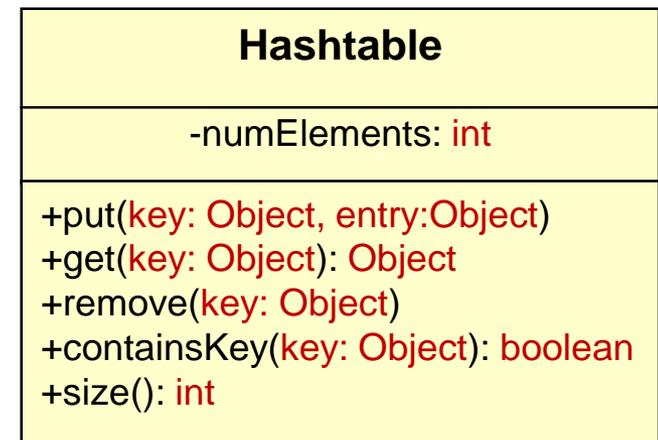
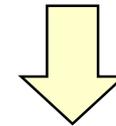
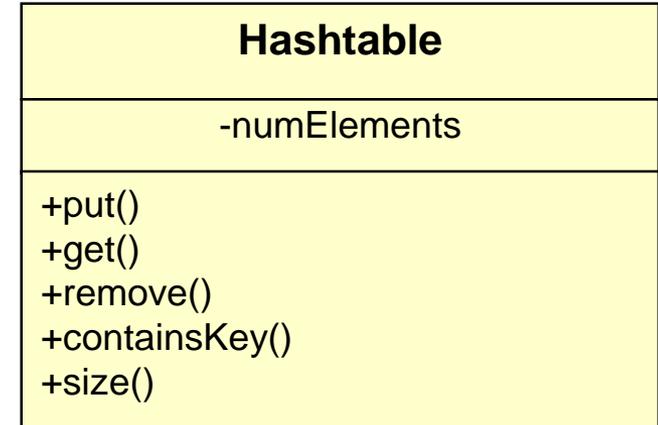
Was kommt nach CRC-Cards?

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

Schnittstellen-Spezifikation: Signaturen = Schnittstellen-Syntax

Spezifikation der Schnittstellen

- In Anforderungsanalyse: Identifikation von
 - ◆ Attributen - ohne ihren Typ anzugeben
 - ◆ Operationen - ohne ihre Parameter(typen) anzugeben
- Später: Hinzufügen von Typangaben
- Typ
 - ◆ Klasse
 - ◆ Interface
 - ◆ primitiver Typ
- Signatur
 - ◆ Methodename
 - ◆ Parametertypen
 - ◆ Ergebnistyp(en)



Warum reichen Signaturen nicht aus?

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

- Sie sagen nur etwas darüber, wie man eine Operation aufruft.
- Sie unterscheiden nicht verschiedene Aufrufende
 - **Zugriffsrechte** (≠/≠ Sichtbarkeiten!)
- Sie sagen nicht, was der spezifizierte Typ selber von seiner Umgebung braucht, um die angebotenen Operationen realisieren zu können.
 - **Benutzte Schnittstellen** (*Required Interfaces*)
- Sie sagen nicht, in welcher Reihenfolge verschiedene Operationen des gleichen Objektes aufgerufen werden müssen.
 - **Interaktionsspezifikation** (*Behaviour Protocols*)
- Sie sagen nicht, was die aufgerufene Operation macht.
 - **Verhaltenszusicherungen** (*Contracts*)

Schnittstellen-Spezifikation: Verhaltensspezifikation durch „Design by Contract“

Design by Contract (DBC)

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

Design by Contract: Vorbedingungen („Preconditions“)

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

Design by Contract: Nachbedingung (Postcondition)

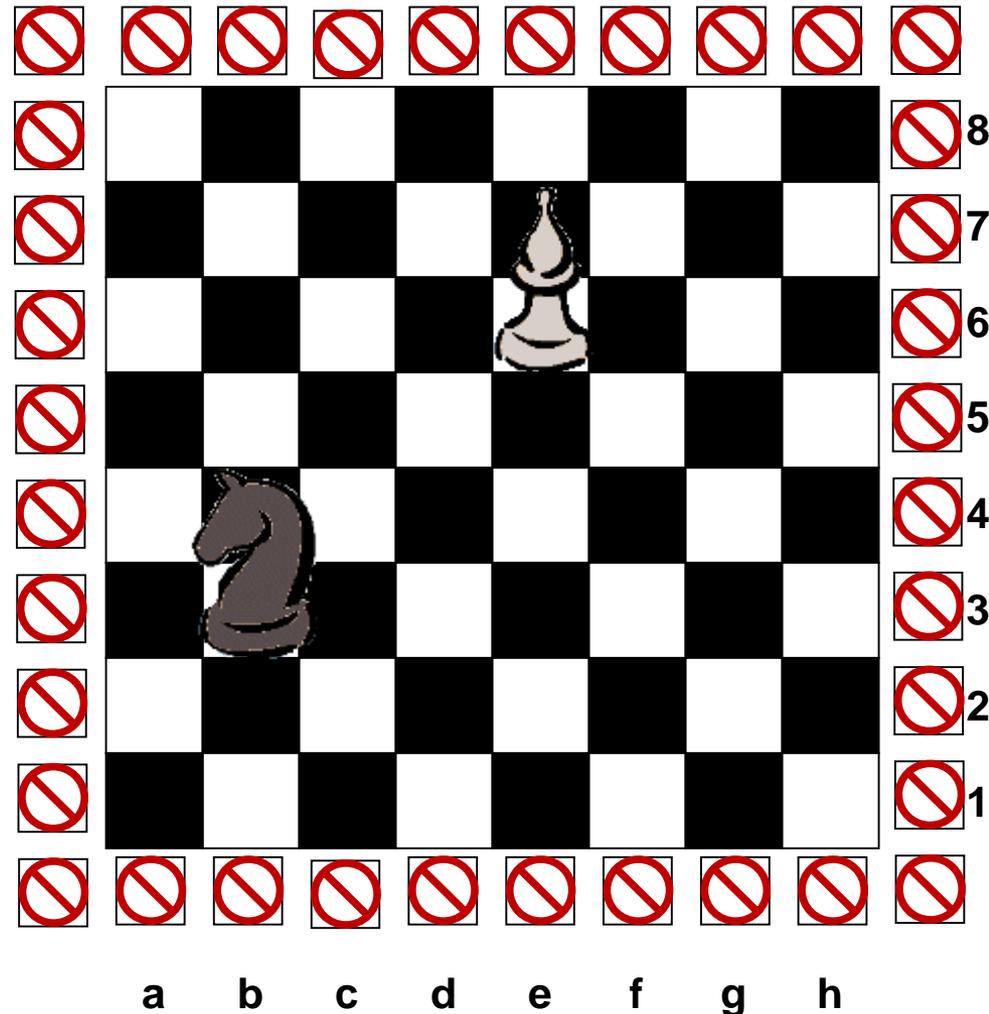
- Definition
 - ◆ Postcondition beschreibt **deklarativ** das Ergebnis eines korrekten Aufrufs
 - ◆ Sagt aus, **was** getan wird, nicht **wie** es getan wird
- Technische Realisierung
 - ◆ Assertion die wahr sein muss, *nachdem* eine Operation ausgeführt wird.
- Beispiel: Operation “Ziehe die Wurzel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

Design by Contract: Klasseninvariante (Class Invariant)

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

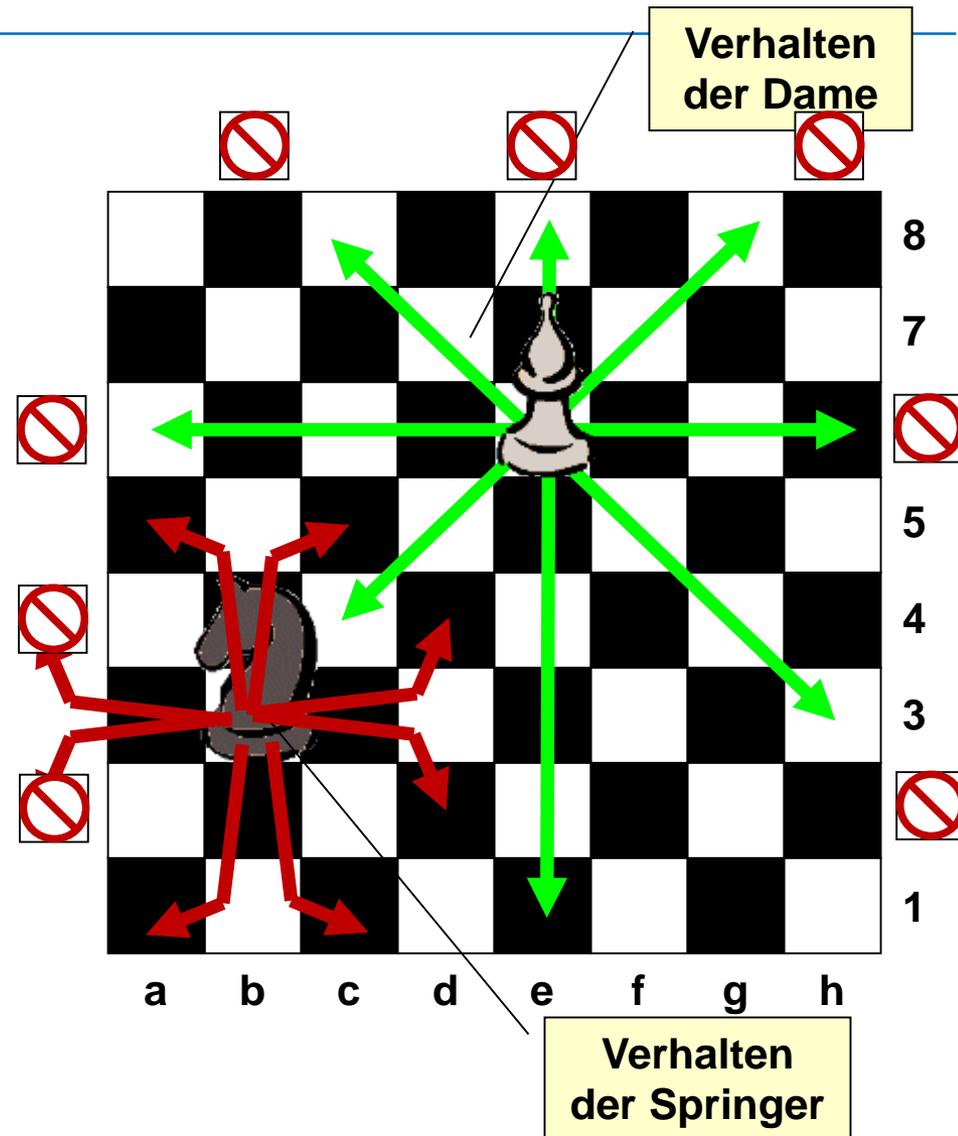
DBC spezifiziert legale Zustände

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



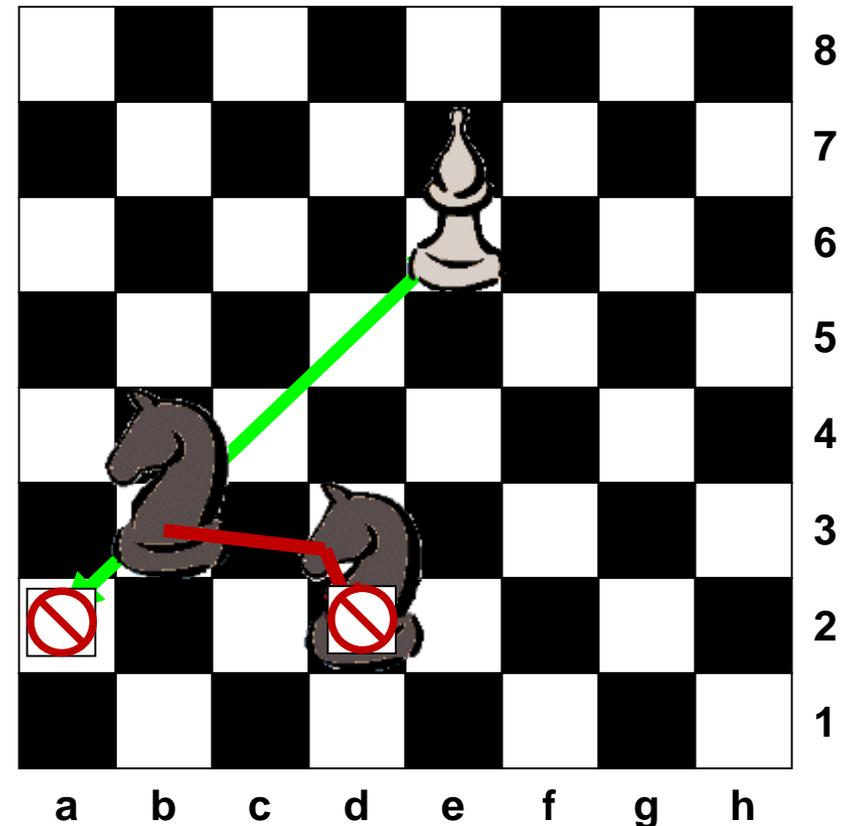
DBC spezifiziert legales Verhalten

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



DBC spezifiziert legales Verhalten

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



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

- Spezifikation von Bedingungen für Werte von Modellelementen
 - ◆ „Constraint“ in OCL = „Assertion“ in DBC
 - ◆ Noch keine Bedingungen an den Kontrollfluss möglich.

- Beispiel: OCL Constraints für Hashtabellen

- ◆ Invariante:

⇒ `context Hashtable inv: numElements >= 0`

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

Dargestellte Art von Assertion

OCL Ausdruck:

Namen beziehen sich auf Elemente des Modells

- ◆ Vorbedingung:

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

- ◆ Nachbedingung:

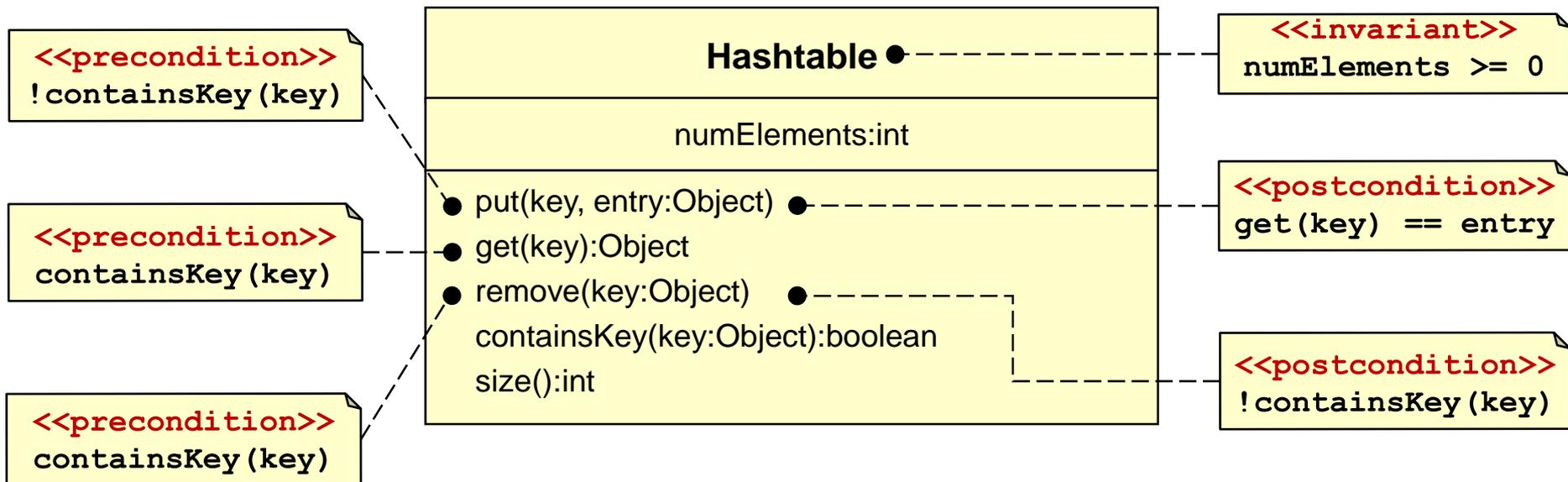
⇒ `context Hashtable::put(key, entry) post: containsKey(key) and get(key) = entry`

Notation für „Methode ‚put‘ aus Typ ‚Hashtable‘ ”

Gültigkeitsbereich

Formulierung von Kontrakten in UML

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



Besondere Schlüsselworte in Nachbedingungen („postconditions“)

- Problem
 - ◆ In Nachbedingungen muss manchmal der **Zustand vor und nach** der Ausführung der Operation unterscheiden werden.
- Syntax
 - ◆ **expr@pre** = Der Wert des Ausdrucks **expr** **vor** Ausführung der **Operation** auf die sich die Nachbedingung bezieht.
 - ◆ **expr@post** = Der Wert des Ausdrucks **expr** **nach** Ausführung der **Operation** auf die sich die Nachbedingung bezieht.
- Beispiele
 - ◆ **context** **Person::birthdayHappens()** **post:** $\text{age} = \text{age@pre} + 1$
 - ◆ **age@pre** bezieht sich hier auf den Wert des Feldes **age** vor Beginn der Ausführung der Operation **birthdayHappens()** → „Alter Wert“
 - ◆ **a.b@pre.c** – Alter Wert des Feldes **b** von **a**. Darin der neue Wert des Feldes **c**.
 - ◆ **a.b@pre.c@pre** – Alter Wert des Feldes **b** von **a**. Darin der alte Wert des Feldes **c**.

Weitere OCL-Schlüsselworte

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

DBC-Unterstützung in Java

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

DBC-Unterstützung in Java

❓ Anwendung

- ❓ Man kann Assertions an beliebigen Stellen des Quellcodes verwenden um logische Ausdrücke zu überprüfen.
 - ❓ Liefert ein solcher Ausdruck `false` zurück, wird ein `AssertionError` ausgelöst und die Ausführung des Programms stoppt.
 - ❓ Die ausgelösten Fehler sind vom Typ `Error` und nicht vom Typ `Exception`
 - ❓ Sie müssen daher nicht abgefangen werden!

❓ Syntax

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

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

oder:

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

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

Warum DBC-Unterstützung in Java?

- ❑ Der Effekt des `assert` Statements könnte auch mit einer `if`-Anweisung und explizitem Werfen von ungeprüften Ausnahmen realisiert werden.

- ❑ Die Vorteile von sprachunterstützten Assertions sind
 - ❑ Lesbarkeit
 - ❑ Der Quellcode wird klarer und kürzer
 - ❑ Auf den ersten Blick ist zu erkennen, dass es sich um eine Korrektheitsüberprüfung handelt und nicht um eine Verzweigung im Programmablauf
 - ❑ Effizienz
 - ❑ Sie lassen sich für die Laufzeit wahlweise an- oder ausschalten
 - ❑ Somit verursachen sie praktisch keine Verschlechterung des Laufzeitverhaltens.

Design by Contract: Sprachunterstützung

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

Subtyping / Ersetzbarkeit und Kontrakte

B ist ein Subtyp von **A**



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



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



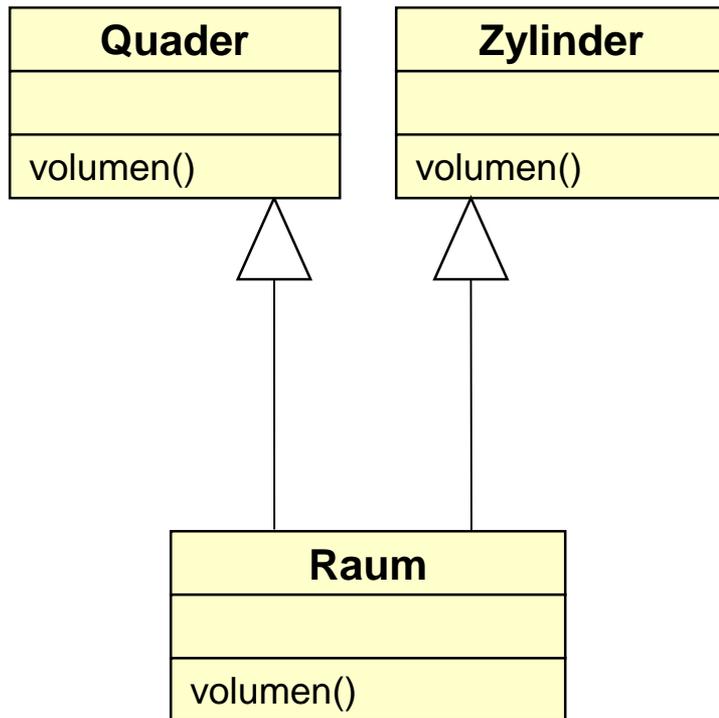
Instanzen von **B** haben mindestens alle Methoden von **A**,
und zwar mit (gleichen oder) stärkeren Nachbedingungen
und (gleichen oder) schwächeren Vorbedingungen

OO Modellierungs-Prinzipien

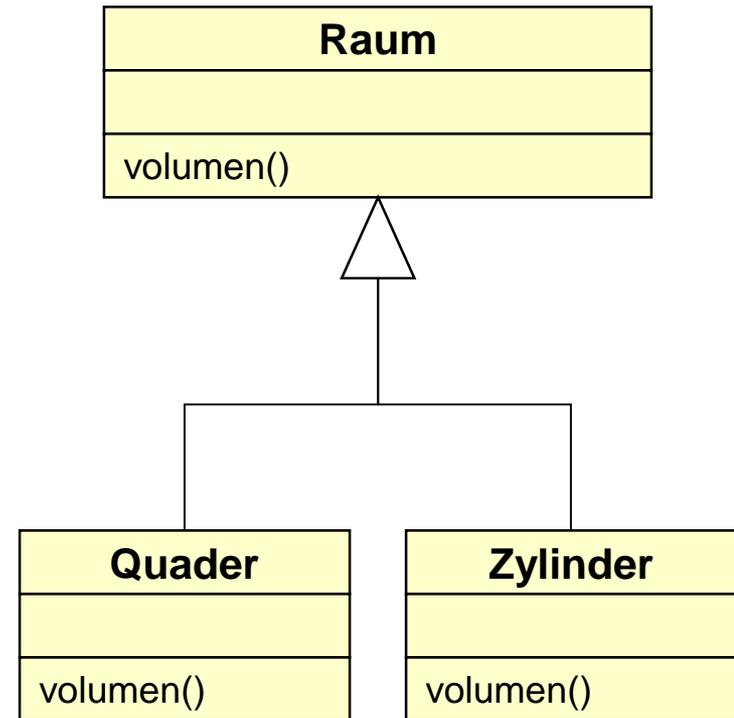
Ein Quiz

OOM-Quiz: „Büroräume“

Was halten Sie hiervon?

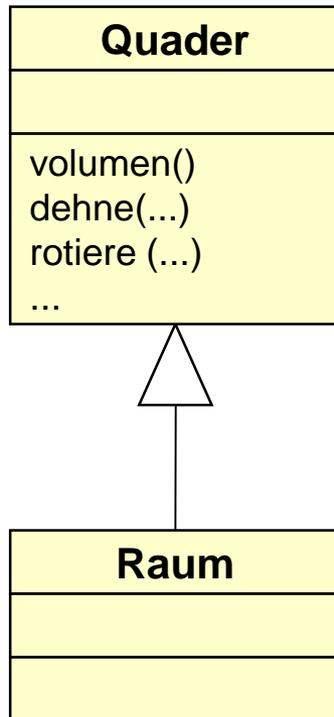


Und hiervon?

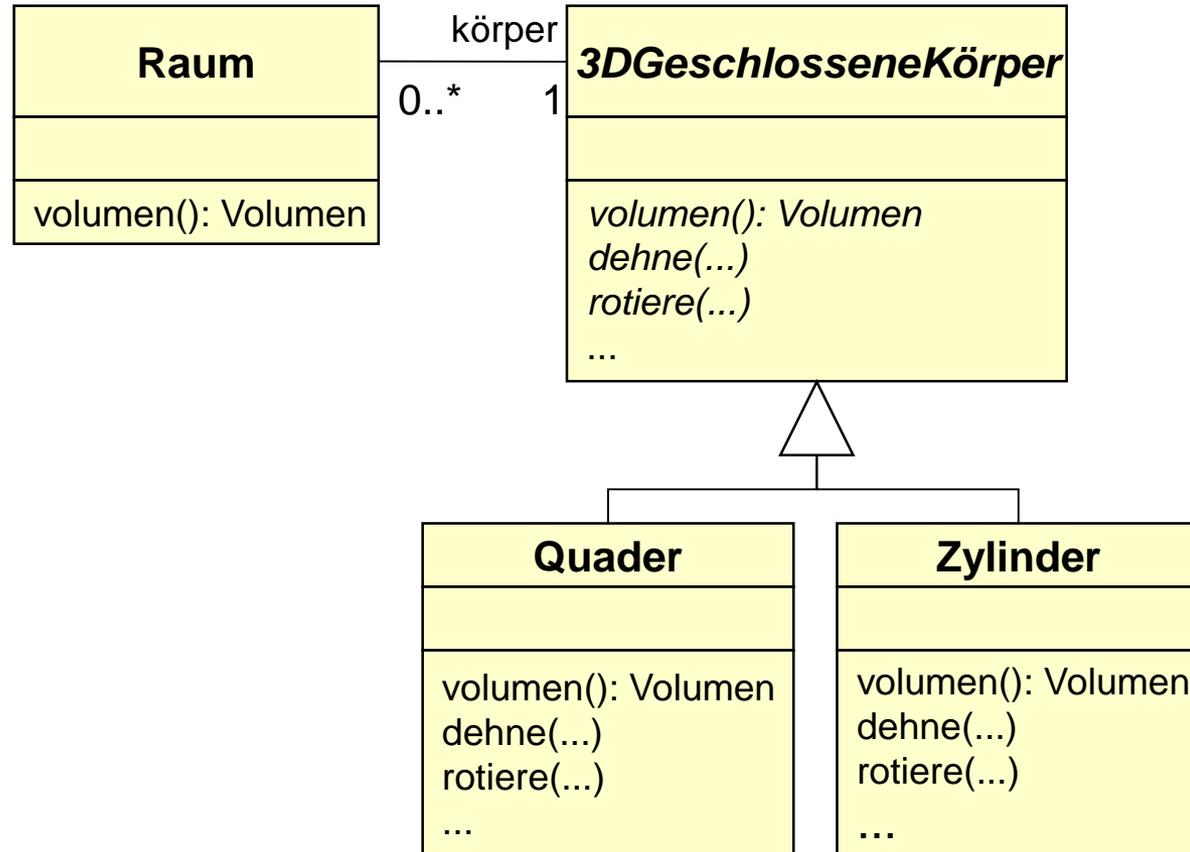


OOM-Quiz: „Bürräume“

Was halten Sie hiervon?



Und hiervon?



Problem: Fehlende Ersetzbarkeit

- Unterklasse hat eine stärkere Invariante als die Oberklasse
 - ◆ Dreidimensionale Körper dürfen rotiert werden
 - ◆ Büroräume 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 Schnittstellen achten
 - ◆ Sie sind das Kriterium um über Ersetzbarkeit zu entscheiden
- Frage: Wie finden wir Schnittstellen?
 - ◆ CRC Cards → Signaturen
 - ◆ DBC als Verfeinerung → Verhaltensbeschreibung durch Assertions

Subtyping / Ersetzbarkeit und Kontrakte

B ist ein Subtyp von **A**



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



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



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

OOM-Quiz

Was halten Sie hiervon?



Panda

~~MiouMiou:GefährdeteSpezies~~

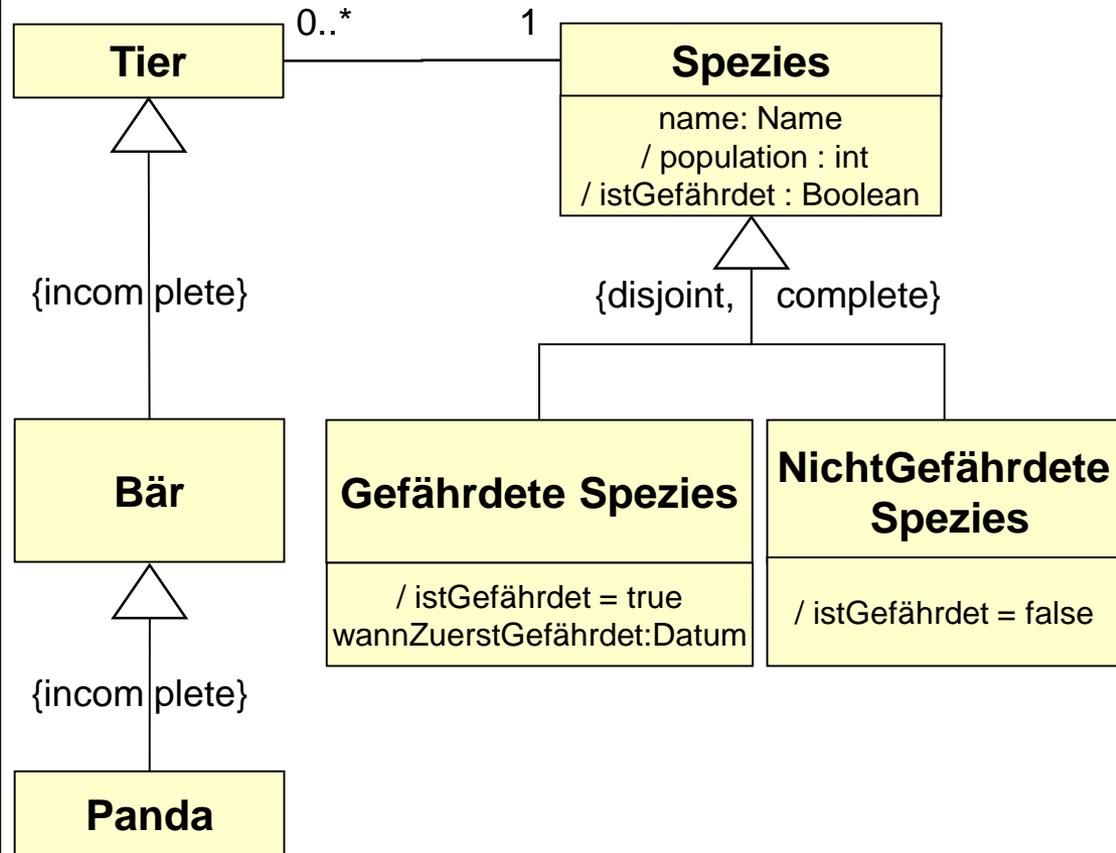
MiouMiou:Panda

Hallo, ich bin MiouMiou

Ich bin ein Panda! 😊

Wer hat behauptet ich sei eine Spezies? 😞

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. Mobiltelefone allgemein, „Nokia 6678“, ...)
 - ◆ "Produkt" → einzelnes Produkt (z.B. mein Mobiltelefon)
- Frage: „Wer/was 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?

| |
|-------------------|
| Real |
| ... |
| arcTan() : Winkel |

Und hiervon?

| |
|-------------------------------------|
| <<utility>> Trigonometrie |
| ... |
| arcTan(Real):Winkel |

| |
|-------------|
| Real |
| ... |
| |

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

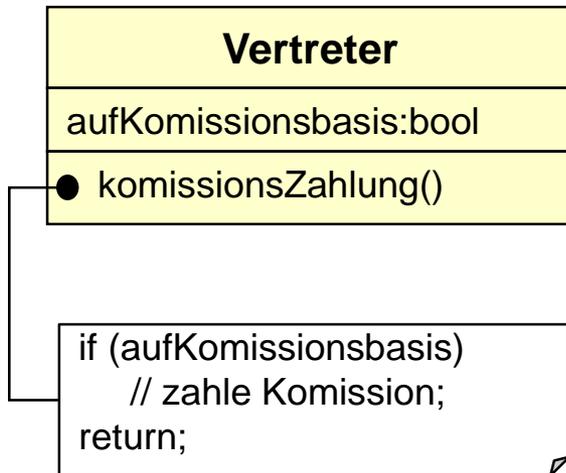
- Behandlung

- ◆ Bezug auf nicht-inhärente Klasse entfernen
- ◆ Eventuell Teile der Klasse in andere Klassen auslagern

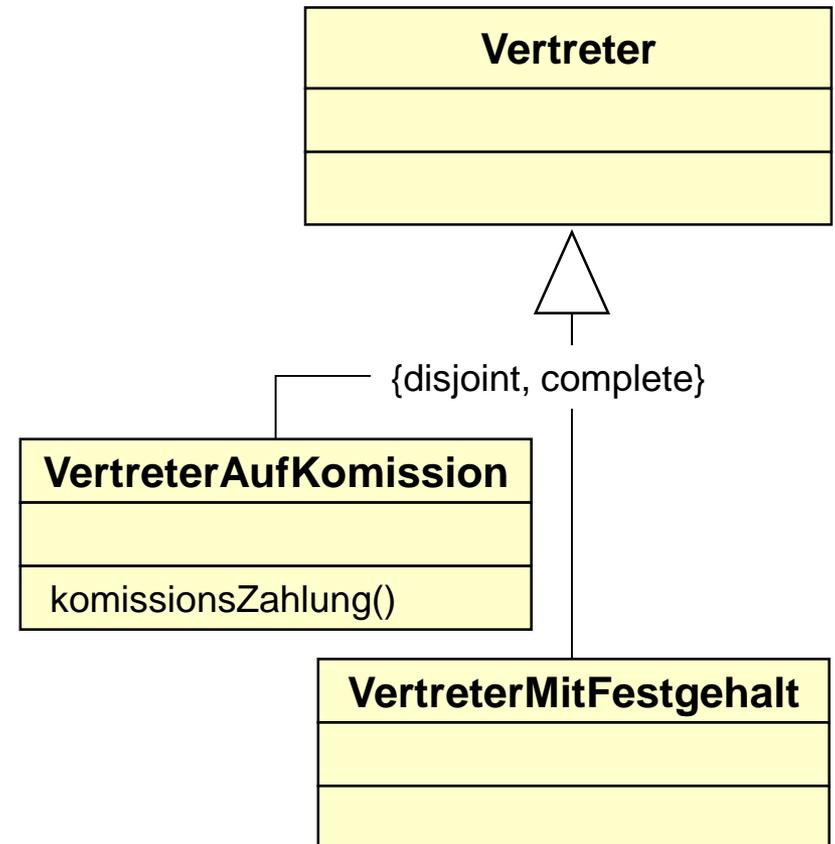
⇒ siehe Refactorings (z.B. „Move Method“, „Move Field“, „Split Class“)

OOM-Quiz

Was halten Sie hiervon?



Und hiervon?

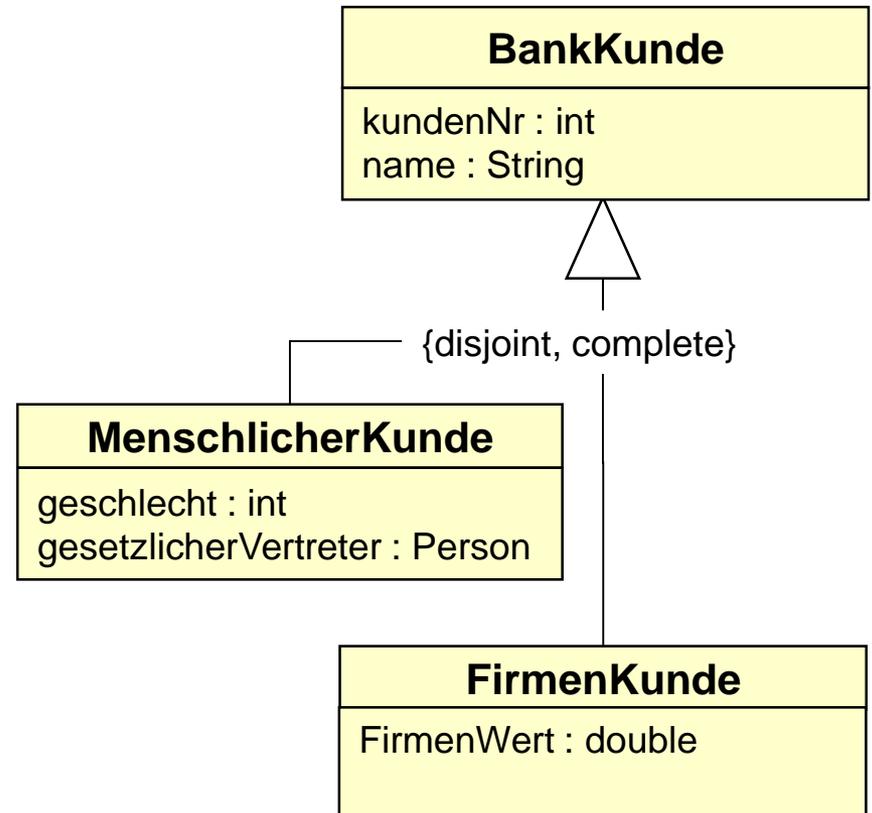


OOM-Quiz

Was halten Sie hiervon?



Und hiervon?

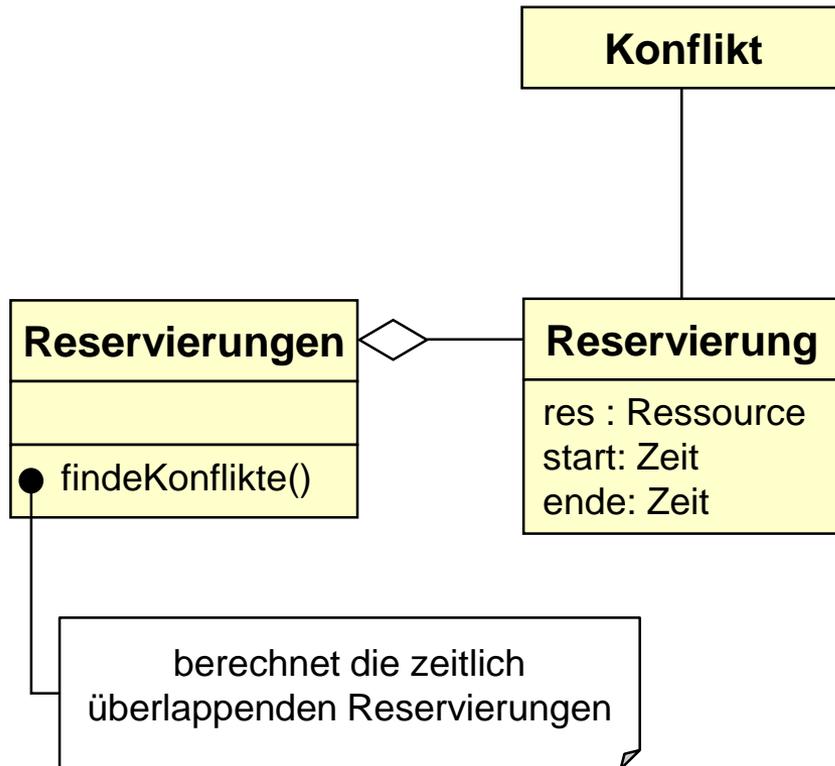


Prinzip: Nicht einheitliche Eigenschaften vermeiden!

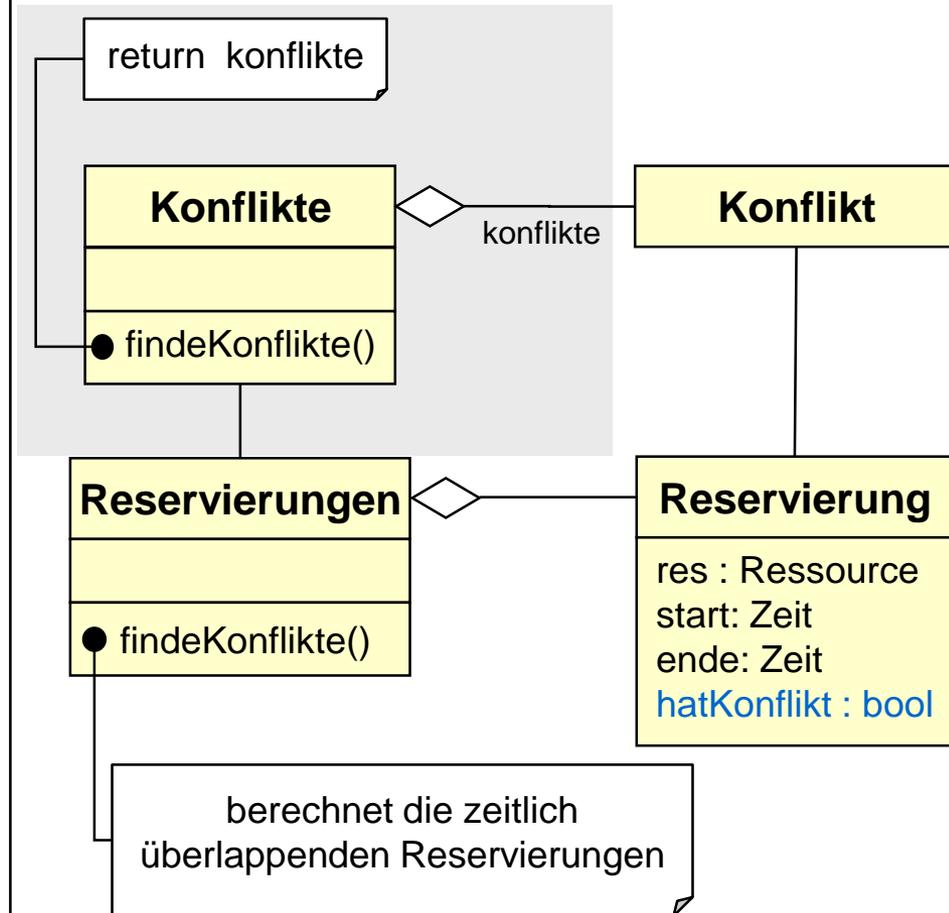
- Symptom
 - ◆ bestimmte Eigenschaften (Methoden oder Variablen) einer Klasse sind nur für manche Instanzen gültig
- Konsequenzen
 - ◆ Abhängigkeit von bestimmten Fallunterscheidungen
 - ◆ Unklare Funktionalität
 - ◆ Wartung erschwert
- Behandlung
 - ◆ Klasse aufsplitten
 - ◆ Evtl. Klasse einführen die „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?

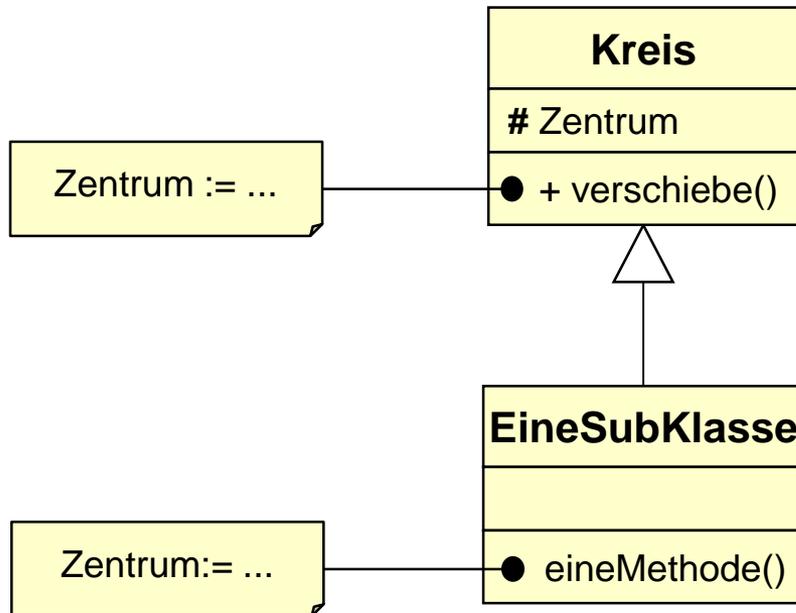


Prinzip: Keine Redundanzen im Modell!

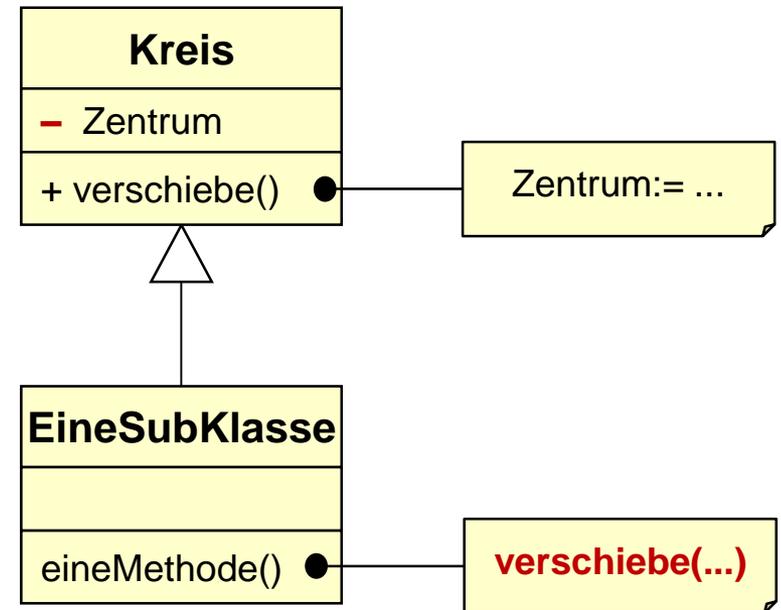
- Redundantes Modell
 - ◆ Mehrfache Wege um die gleiche Information zu erhalten
 - ◆ Speicherung abgeleiteter Informationen
- Problem
 - ◆ Bei Änderungen zur Laufzeit, müssen die Abgeleiteten Informationen / Objekte konsistent gehalten werden
 - ◆ Zusatzaufwand bei Implementierung und zur Laufzeit (Benachrichtigung über Änderung und Aktualisierung abgeleiteter Infos → „Observer“)
 - ◆ 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?



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

Es gibt Abhängigkeit von ...

- **Namen (Existenz)**

- ◆ `int i;`
- ◆ `i := 7` // abhängig von Deklaration von i

- **Namen (Nicht-Existenz)**

- ◆ `int i;`
- ◆ `int j;` // abhängig davon, dass i nicht in j umbenannt wird – j darf nicht zwei mal deklariert werden!

- **Typ**

- ◆ `int i;`
- ◆ `i := j;` // Typ von j muss mit Typ von i zuweisungskompatibel sein

- **Relative Position**

- ◆ op1 muss unbedingt vor op2 stattfinden

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)

- Verberge Interna eines Subsystems
 - ◆ Definiere Schnittstellen zum Subsystem (→ Facade)
- Verberge Interna einer Klasse
 - ◆ Nur Methoden der selben Klasse dürfen auf deren Attribute zugreifen
- Vermeide transitive Abhängigkeiten
 - ◆ Führe eine Operation nicht auf dem Ergebnis einer anderen aus.
 - ⇒ Schreibe eine neue Operation, die die Navigation zur Zielinformation kapselt.

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

- Klasse sollte nur von „Freunden“ (= 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 transitiven Zugriff kapselt.

◆ Zielkonflikt

- ⇒ Vermeidung transitiver Abhängigkeiten vs. „schlanke“ Schnittstellen.

Gibt es „gute“ Abhängigkeiten?

Stabilität

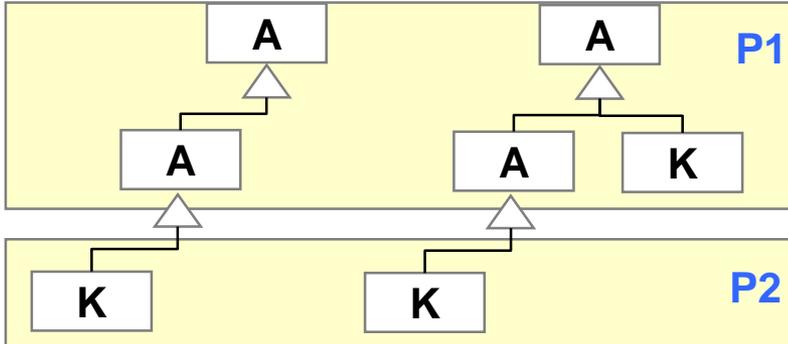
Abstraktheit

Robert C. Martins Abhängigkeits-Prinzipien

Abstraktheit eines Paketes

Abstrakte und konkrete Typen

- A = abstrakter Typ
 - ◆ enthält abstrakte Methode(n)
- K = Konkreter Typ
 - ◆ enthält keine abstrakte Methode



Abstraktheit eines Pakets p

Definition

- a_p = Anzahl abstrakter Typen in p
- k_p = Anzahl konkreter Typen in p
- $abs_p = \frac{a_p}{a_p + k_p}$

Anwendung auf P1 und P2:

- $a_{P1} = 4 / 4+1 = 4/5 = 80\%$
- $a_{P2} = 0 / 0+2 = 0/2 = 0\%$

Idee der Abstraktheit-Metrik: Je abstrakter ein Paket ist um so

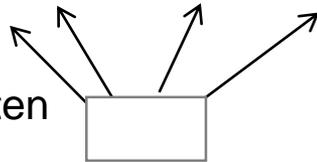
- leichter ist es wiederverwendbar
- häufiger wird es wiederverwendet

(In)Stabilität eines Typs oder Pakets

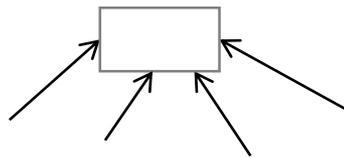
Abhängigkeiten

Es werden alle Assoziationen und Vererbungs-Beziehungen gezählt:

- efferente (e)
= Eigene Abhängigkeiten von Anderen



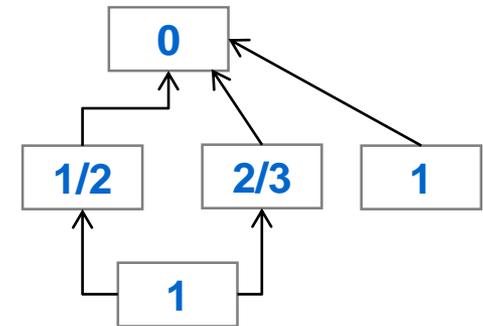
- afferente (a)
= Abhängigkeiten anderer von mir



Instabilität

$$I = \frac{e}{e+a}$$

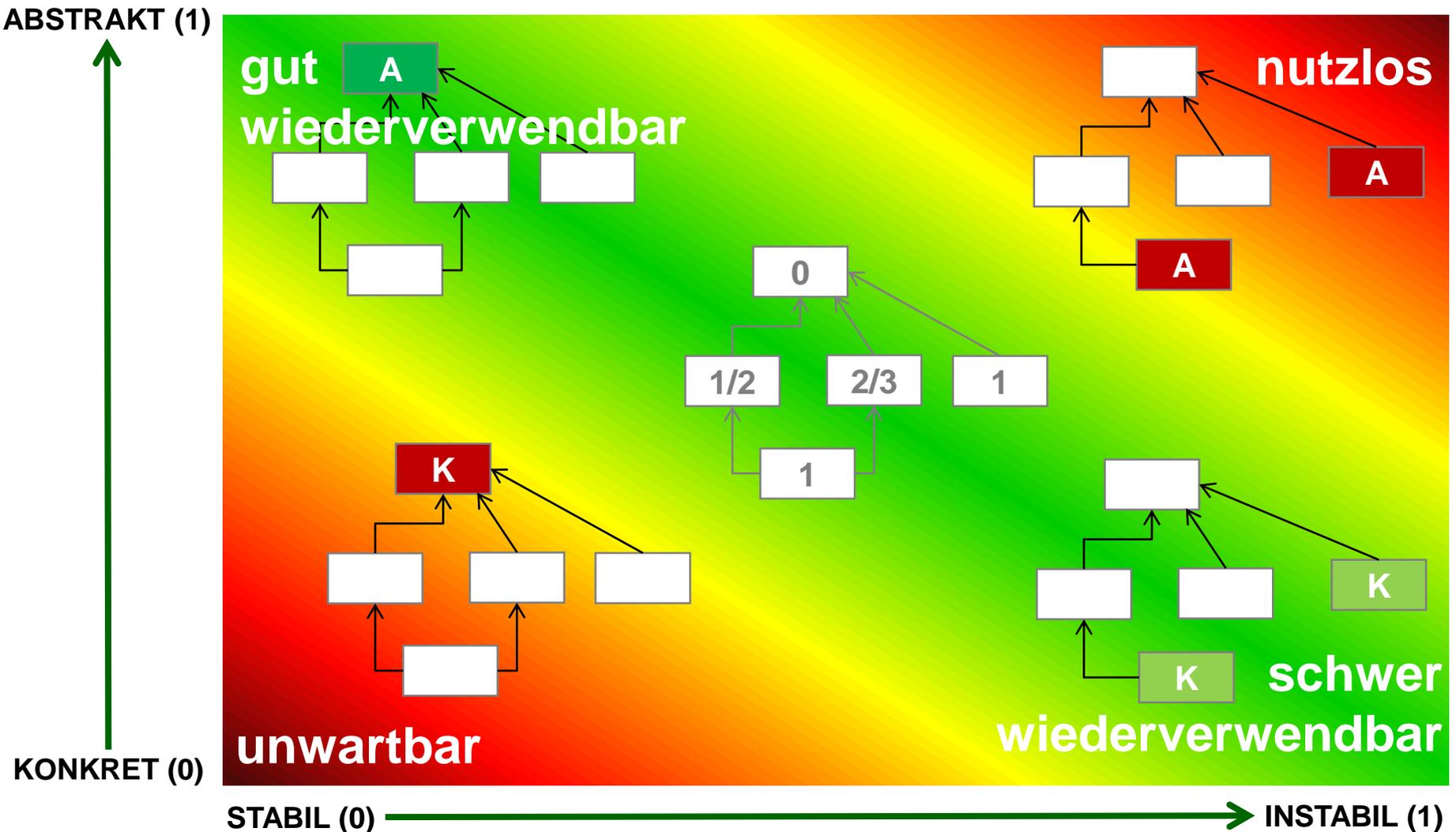
0 = maximal stabil



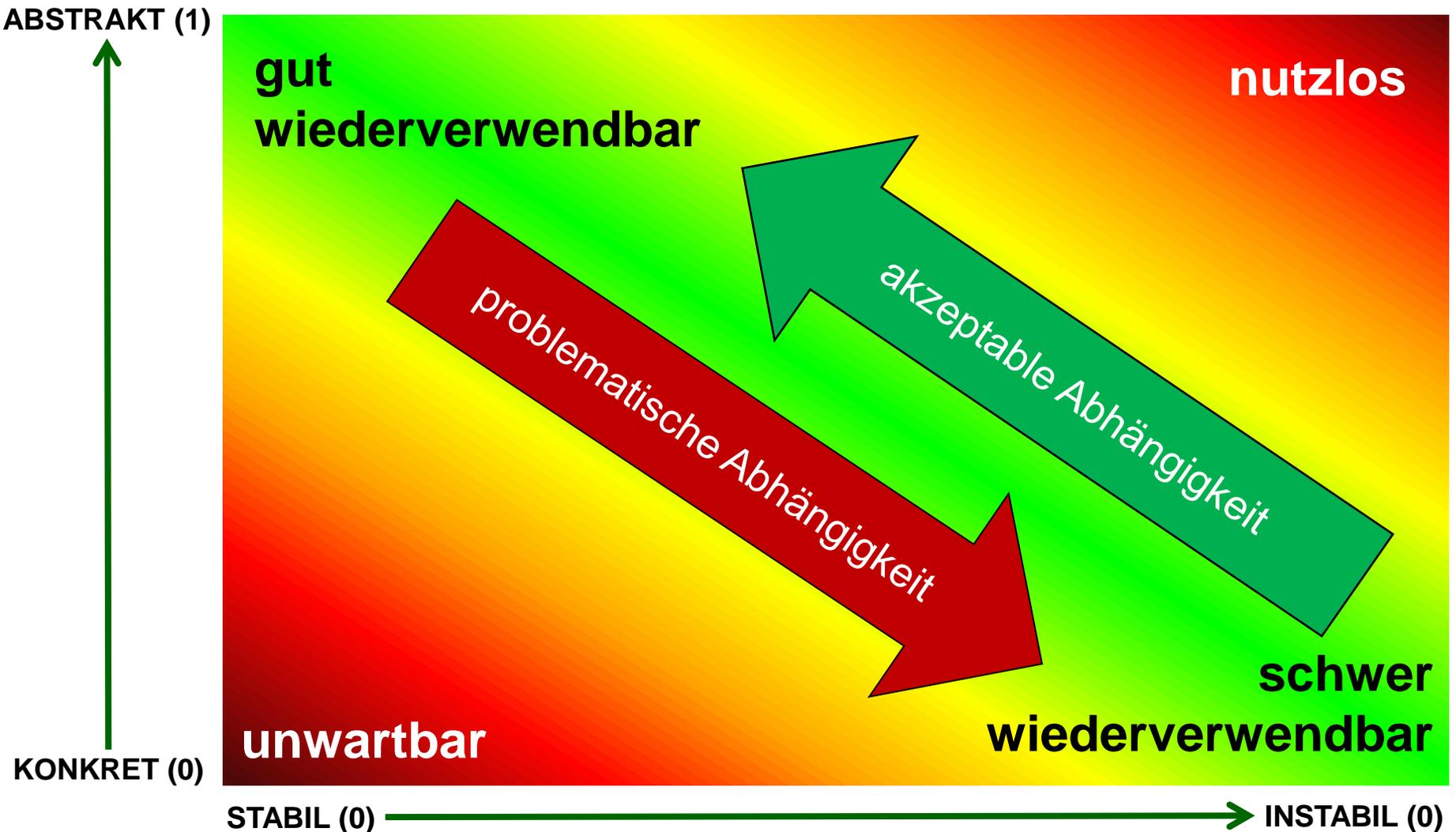
1 = maximal instabil

Idee der Instabilitäts-Metrik: Instabilität ist ein Maß dafür, wie leicht ein Typ zu ändern ist. Ein instabiler Typ ist leicht zu ändern, weil die Konsequenzen von Änderungen (für Andere) null sind.

Klassifikation nach Stabilität und Abstraktheit



Gute uns schlechte Abhängigkeiten



Entwurfs-Prinzipien

(„Design Principles“, Robert C. Martin, 1996)

- Dependency Inversion Principle (DIP)

- ◆ Abhängigkeiten nur zu Abstrakterem!

- Stable Dependencies Principle (SDP)

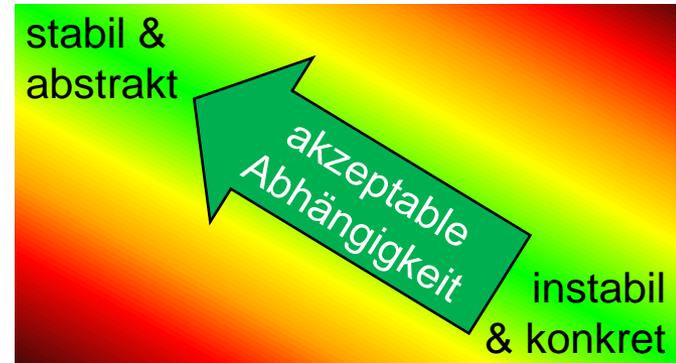
- ◆ Abhängigkeiten nur zu Stabilerem!

- Stable Abstractions Principle (SAP)

- ◆ Abstraktion und Stabilität sollten zueinander proportional sein.
- ◆ Maximal stabile Pakete sollten maximal abstrakt sein.
- ◆ Instabile Pakete sollten konkret sein.

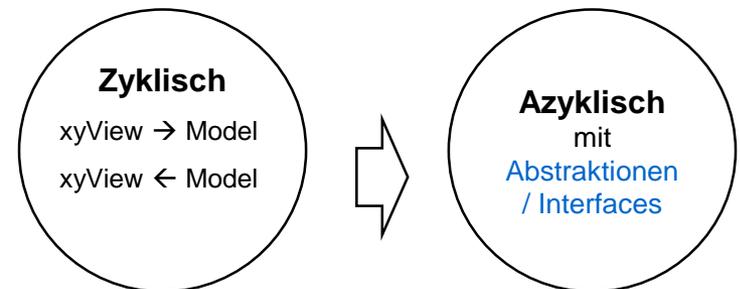
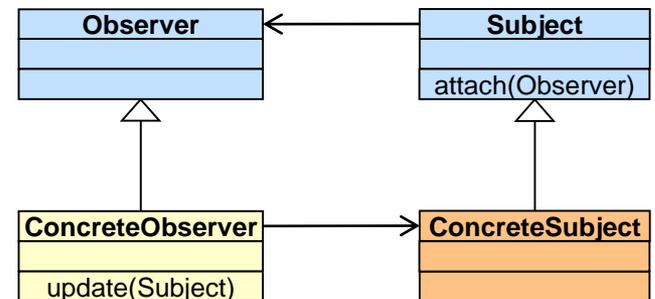
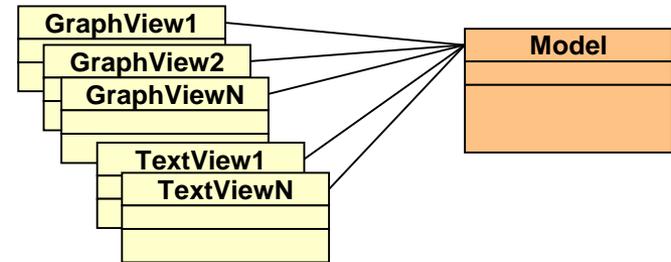
- Acyclic Dependencies Principle (ADP)

- ◆ Der Abhängigkeitsgraph veröffentlichter Komponenten muss azyklisch sein!



Aufbrechen zyklischer Abhängigkeiten am Beispiel „Observer“

- Design ohne Observer → zyklisch
 - ◆ Gegenseitige Abhängigkeiten der konkreten Typen.
- Design mit Observer → azyklisch
 - ◆ ConcreteObserver ist von Concrete Subject abhängig.
 - ◆ Concrete Subject ist nur von abstraktem Subject abhängig.
- Modellierungsprinzip
 - ◆ Aufbrechen zyklischer Abhängigkeiten durch Einführung von **Abstraktionen** von denen beide Partner azyklisch abhängig sind.



Zusammenfassung & Ausblick

OO Modellierung: Rückblick

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

OO Modellierung: Literatur

- 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: Literatur

- Design by Contract (ff)
 - ◆ Bertrand Meyer: “Applying Design by Contract”
Erschienen in “Computer”, Vol. 25 Issue 10, October 1992, page 40-51,
IEEE Computer Society Press Los Alamitos, CA, USA,
<http://dx.doi.org/0.1109/2.161279>
- Abhängigkeiten (Stable dependency principle, ...)
 - ◆ Robert C. Martin: Design Principles and Design Patterns
http://www.objectmentor.com/resources/articles/Principles_and_Patterns.PDF
- Aufbrechen von ungünstigen Abhängigkeiten mit 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>