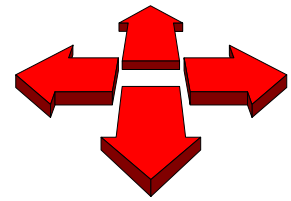
Worte vorweg

- Wir bieten wieder ein **Tutorium** an, in dem Inhalte aus Vorlesung und Übungen wiederholt und vertieft werden können.
- Inzwischen steht der endgütige Termin fest:
  - **Termin: dienstags 18 bis 20 Uhr in D-010**

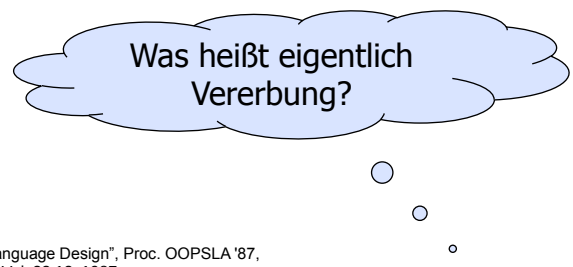
Abstraktion: Polymorphie und Vererbung

- Polymorphie-Begriff
- Übersicht über Vererbungskonzepte
- Typhierarchien: Subtyping



Motivation: Vererbung

- Vererbung ist nach Wegner die **definierende Eigenschaft** objektorientierter Programmiersprachen.
- Ohne ein Verständnis von Vererbung kein vollständiges Verständnis für OOP!
- Aber: der Begriff ist stark überladen; Vererbung wurde auch schon als das **„Goto der Neunziger Jahre“** bezeichnet.



Wegner, P.: “Dimensions of Object-Based Language Design”, Proc. OOPSLA '87, Orlando, Florida; in ACM SIGPLAN Notices, Vol. 22:12, 1987.

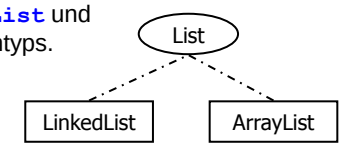
„Inheritance Considered Harmful“

- In Anlehnung an Dijkstras „**GoTo Statement Considered Harmful**“.
- Sinnvoll: Unterscheidung der **Konzepte**, die durch Vererbung unterstützt werden sollen, und der **Mechanismen**, die verschiedene Sprachen anbieten.
- „**Harmful**“ werden die **Mechanismen**, die von Programmiersprachen angeboten werden, wenn sie mit **zu vielen Konzepten** überladen werden (wie beim GoTo).
- Vererbung ist insbesondere dann „**harmful**“, wenn die Klassen großer Systeme unsystematisch mit Vererbung von einander abhängig werden (ähnlich wie beim GoTo).

These:
Vererbung ist das GoTo der Objektorientierung!

Bisher: Abstraktionsmittel Interface

- Wir haben bisher **Interfaces** primär als Abstraktion von verschiedenen Implementationen eines Datentyps kennen gelernt.
  - Beispiel: Datentyp `List` als Interface, `LinkedList` und `ArrayList` als Implementationen dieses Datentyps.



- Wir wenden uns nun den Konzepten zu, die die Grundlage für Polymorphie und Vererbung in Programmiersprachen bilden.

Aus SE1: Die Doppelrolle einer Klasse

- Aus Sicht der Klienten einer Klasse ist interessant:
  - Welche **Operationen** können an Exemplaren der Klasse aufgerufen werden?
  - Welchen Typ haben die Parameter einer **Operation** und welches Ergebnis liefert sie?
  - Was sagt die Dokumentation (Kommentare, javadoc) über die Benutzung?

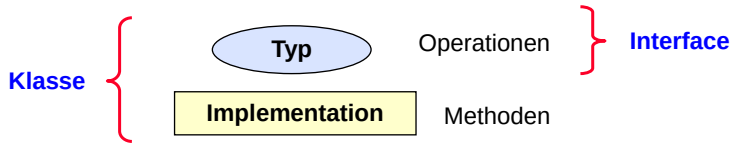
Außensicht, öffentliche Eigenschaften, Dienstleistungen, Typ

- Für die Implementation der **Methoden** einer Klasse ist relevant:
  - Wie sind die Operationen in den **Methodenrümpfen** umgesetzt?
  - Welche **Exemplarvariablen/Felder** definiert die Klasse?
  - Welche (privaten) **Hilfsmethoden** stehen in der Klasse zur Verfügung?

Innensicht, private Eigenschaften, Implementation

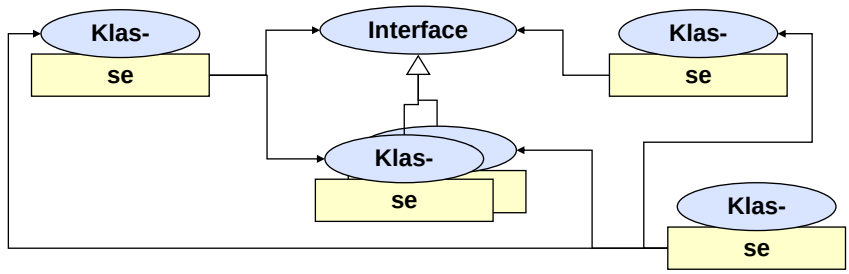
Die objektorientierte Klasse: Typ und Implementation

- Die Unterscheidung von Schnittstelle bzw. Typ und Implementation kennen wir bereits aus SE1. Eine **Klasse definiert beides**.
- **Interfaces** hingegen sind reine Typinformationen, ohne Implementation.



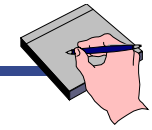
Statik von OO Systemen: Geflechte von Typen

- Ein objektorientiertes (Java-)System besteht in seiner statischen Sicht aus einer **Menge von Typen** (Klassen und Interfaces) und **Implementationen**.
- Diese **benutzen** sich gegenseitig ausschließlich über ihre **Schnittstellen**, indem sie Operationen aufrufen.
- Zu einem Interface kann es verschiedene Implementationen geben, die auch nebeneinander in einem System zum Einsatz kommen können.
  (Bsp.: Interface `List` mit Implementationen `LinkedList` und `ArrayList`)



SE2 – OOPM – Teil 1 9

Wiederholung: Statischer und dynamischer Typ



- In objektorientierten Sprachen muss der statische vom dynamischen Typ einer **Referenzvariablen** unterschieden werden.
- Der **statische Typ einer Variablen** wird durch ihren **deklarierten Typ** definiert. Er heißt statisch, weil er zur Übersetzungszeit feststeht.

```
List<String> liste1;  // List<String> ist hier der statische Typ von liste1
```

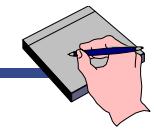
- Der statische Typ legt die **Operationen** fest, die über die Variable aufrufbar sind.

```
liste1.add("Simpson");  // add ist hier eine Operation
```

- Ein Compiler kann bei der Übersetzung prüfen, ob die genannte Operation tatsächlich im statischen Typ definiert ist.

SE2 – OOPM – Teil 1 10

Wiederholung: Statischer und dynamischer Typ (II)



- Der **dynamische Typ einer Referenzvariablen** hängt von der Klasse des Objektes ab, auf das die Variable zur Laufzeit verweist.

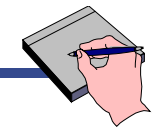
```
liste1 = new LinkedList<String>(); // 1. dynamischer Typ von liste1
```

- Er bestimmt die Implementation und ist dynamisch in zweierlei Hinsicht:
  - Er kann erst zur Laufzeit ermittelt werden.
  - Er kann sich während der Laufzeit ändern.

```
liste1 = new ArrayList<String>(); // neuer dynamischer Typ von liste1
```

- Ein **Objekt** hingegen **ändert seinen Typ nicht**; es bleibt sein Leben lang ein Exemplar seiner Klasse.
- Der dynamische Typ einer Variablen (bzw. der Typ des referenzierten Objektes) entscheidet darüber, welche **konkrete Methode** bei einem **Operationsaufruf** ausgeführt wird. Da diese Entscheidung erst zur Laufzeit getroffen werden kann, wird dieser Prozess **dynamisches Binden** (einer Methode) genannt.

Dynamisches Binden



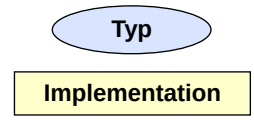
Dynamisches Binden:

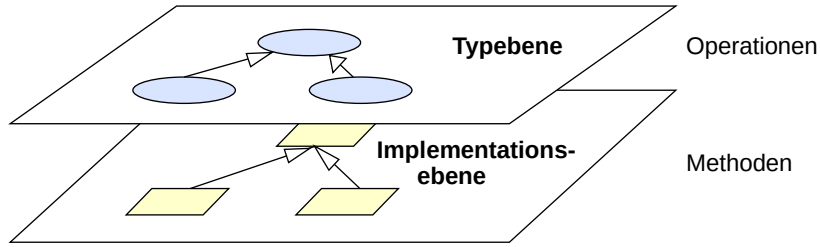
Da erst zur Laufzeit ein konkretes Objekt den **dynamischen Typ** einer Variablen bestimmt, kann der Compiler beim Aufruf einer Operation durch einen Klienten zur Übersetzungszeit nicht festlegen, **welche Methode** tatsächlich **aufzurufen** ist; diese Entscheidung muss deshalb zur Laufzeit (**dynamisch**) getroffen werden.

In Java werden lediglich die Aufrufe privater Exemplarmethoden statisch gebunden. Alle anderen Aufrufe an Exemplare werden dynamisch gebunden!

Vererbung: Auf Typ- und Implementationsebene

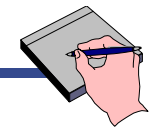
- Mit Vererbung werden **hierarchische Beziehungen** modelliert.
- Die Unterscheidung von Schnittstelle bzw. Typ und Implementation ermöglicht eine differenzierte Betrachtung von Hierarchien auf Typ- und Implementationsebene.



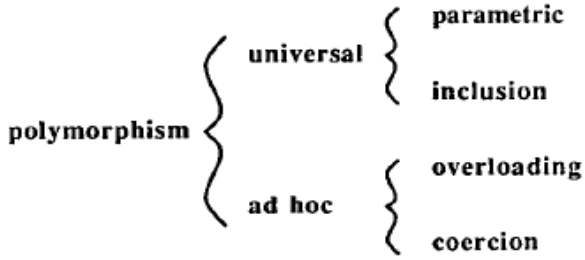


SE2 – OOPM – Teil 1 13

Polymorphie nach Cardelli und Wegner (1985)



polymorph:
griechisch für "vielgestaltig".



Entsprechung in Java:

Generizität (seit Java 5)

Vererbung et al.

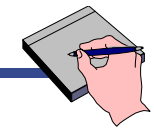
Operatoren, insbes. +; Überladen von Methoden- und Konstruktornamen

Typumwandlungen für primitive Typen

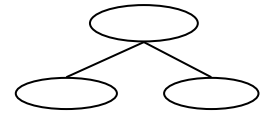
© Cardelli, L., Wegner, P.: "On Understanding Types, Data Abstraction and Polymorphism", *Computing Surveys,* Vol. 17:4, S. 471-522, 1985.

SE2 – OOPM – Teil 1 14

Subtyp-Polymorphie

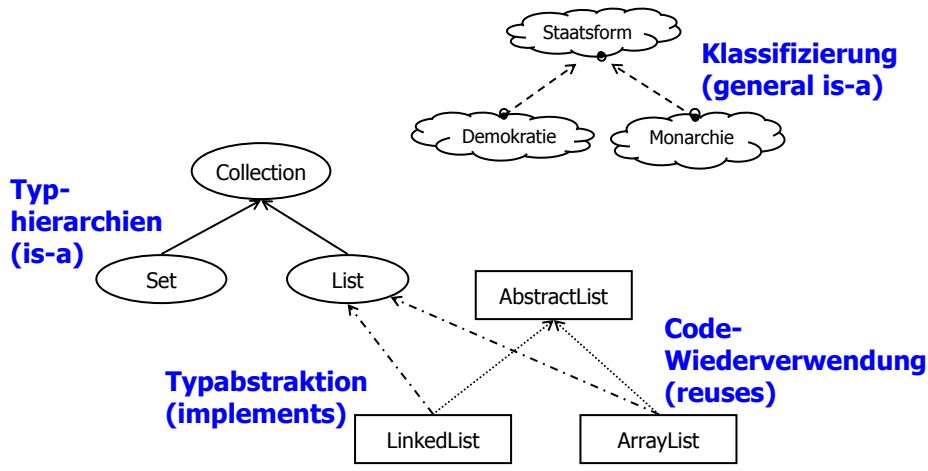


- **Inclusion Polymorphism** nach Cardelli u. Wegner
- Im Kontext objektorientierter Sprachen meist kurz **Polymorphie** genannt.
- Eng mit **Ersetzbarkeit** (engl. substitutability) verknüpft: Variable eines Supertyps kann auf Exemplare von Subtypen verweisen.
- Somit auch eng verknüpft mit der Unterscheidung von **statischem** und **dynamischem Typ** einer Referenz-Variablen.



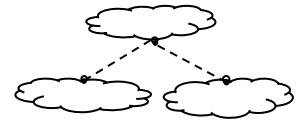
- Erfordert **dynamisches Binden**!
- Zentrales **technisches Konzept** objektorientierter Sprachen
- Voraussetzung für Typhierarchien und Typabstraktion

Übersicht: Zentrale „Vererbungs“-konzepte



Klassifizierung

- Allgemeines Verständnis von **Ist-ein-Beziehungen**
- Vgl. **Taxonomien** in der Biologie
- **Ontologien**, semantische Netze
- Beispiele:
  - ein Quadrat *ist ein* Rechteck.
  - ein Emu *ist ein* Vogel.

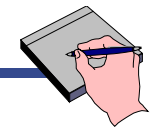


- Häufig im Zusammenhang mit Vererbung genannt; wird in ihrer allgemeinen Form jedoch **nicht** durch die Mechanismen **von Programmiersprachen unterstützt**!

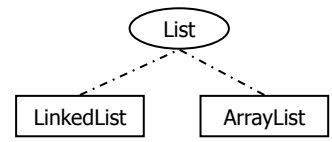
Subclassing ≠ Subtyping ≠ Is-a
LaLonde u. Pugh, Journal of Object-oriented Programming, January 1991

- Wir gehen in dieser Veranstaltung nicht weiter auf Klassifizierung ein.

Typabstraktion (bekannt aus SE1)

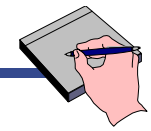


- Die Implementierung eines abstrakt, aber vollständig beschriebenen Typs (vgl. **abstrakter Datentyp**) kann auf unterschiedliche Weise erfolgen.
- Beispiel: Der Typ `List` kann mit einer `LinkedList` und mit einer `ArrayList` implementiert werden.
- Idealerweise wird für einen Klienten nur der Typ sichtbar, die **Implementation** ist **austauschbar**.
- Typischerweise definieren die Implementationen keine zusätzlichen Operationen.

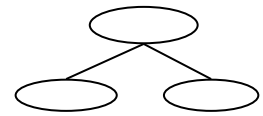


- Unterschiede zeigen sich möglicherweise in der Effizienz (falls nicht Teil der Spezifikation). Je nach Benutzungsprofil wirken sich die Implementationen unterschiedlich aus.
- Setzt **dynamisches Binden** nur dann voraus, wenn mehrere Implementationen im gleichen Programm aktiv sein sollen!
- In Java über **Subtyp-Polymorphie** realisierbar.

Typhierarchien



- Durch das Anordnen von **Typen** in einer hierarchischen Beziehung entstehen **Super-** und **Subtypen**.
- Ein **Subtyp** definiert mindestens alle Operationen seines Supertyps; typischerweise bietet ein Subtyp **weitere Operationen** an.
- Auch hier gilt **Ersetzbarkeit**: Ein Supertyp ist durch jeden seiner Subtypen ersetzbar.
- Das Bilden von Typhierarchien wird auch **Subtyping** genannt.
- Basiert auf **Subtyp-Polymorphie**.

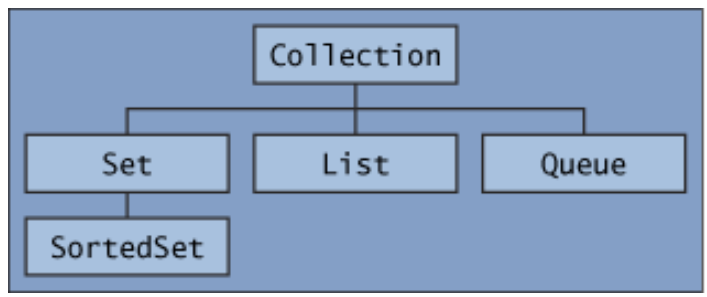


- Wir gehen noch ausführlich auf Subtyping ein.

SE2 – OOPM – Teil 1 19

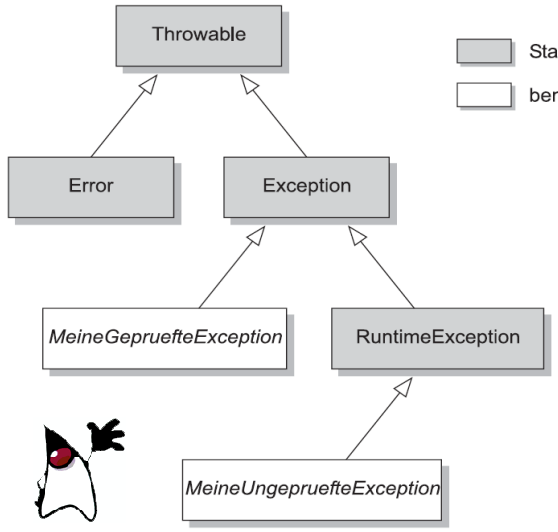
Beispiel: Typ-Hierarchie im Java Collections Framework

- Der Typ `Collection` ist der Supertyp der Typen `Set`, `List` und `Queue` (und transitiv auch der Supertyp von `SortedSet`).
- An allen Stellen, an denen eine `Collection` erwartet wird, kann ein Exemplar eines der Subtypen eingesetzt werden.
- An allen Stellen, an denen ein `Set` erwartet wird, kann auch ein Exemplar von `SortedSet` eingesetzt werden.



SE2 – OOPM – Teil 1 20

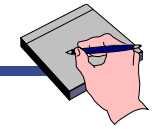
Beispiel: Javas Exception-Hierarchie



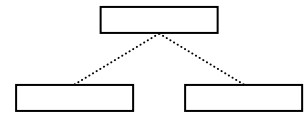
- `Throwable` ist der Supertyp aller Exception-Klassen bzw. –Typen.
- `RuntimeException` ist der Supertyp aller ungeprüften Exception-Typen (abgesehen von `Error`).
- In einem Exception-Handler (kommt noch) kann ein Supertyp stellvertretend für alle seine Subtypen aufgeführt werden.

SE2 – OOPM – Teil 1 21

Code-Wiederverwendung



- Auch **Implementationsvererbung** (engl.: inheritance) oder **Subclassing** genannt.
- Eine **Subklasse** erbt die **Methoden** und **Felder** ihrer **Superklasse**.
- Der geerbte Code wird für spezielle Anforderungen angepasst, indem Methoden **definiert**, **überschrieben** oder **erweitert** werden.

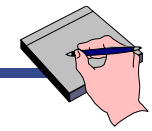


- Theoretisch (und auch praktisch, wie etwa in der Sprache **Sather**) durch einfaches Kopieren von Quelltexten realisierbar (**statisch**).
- Meist jedoch ebenfalls über dynamisches Binden realisiert.

- Wir gehen in einer späteren Vorlesung noch ausführlich auf Code-Wiederverwendung ein.

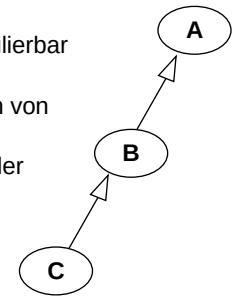
SE2 – OOPM – Teil 1 22

Einige weitere Begriffe



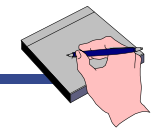
- Wenn wir uns auf **Hierarchien von Typen** beziehen, werden wir im Folgenden die Begriffe **Super- und Subtyp** verwenden.
- Wenn wir uns auf **Hierarchien von Implementationen** (Klassen) beziehen, werden wir im Folgenden die Begriffe **Ober- und Unterklasse** verwenden.

- Subtyp- und Unterklassenbeziehungen sind unter anderem **transitiv**; eine Typ kann deshalb **mehrere Supertypen** haben und eine Klasse **mehrere Oberklassen**.
- Ist eine solche Beziehung zwischen zwei Typen/Klassen formulierbar ohne die Beteiligung weiterer, nennen wir sie **unmittelbar**.
  - Im Beispiel rechts ist C ein Subtyp von B und transitiv auch von A.
  - A ist jedoch nur der **unmittelbare Supertyp** von B und B der **unmittelbare Supertyp** von C.

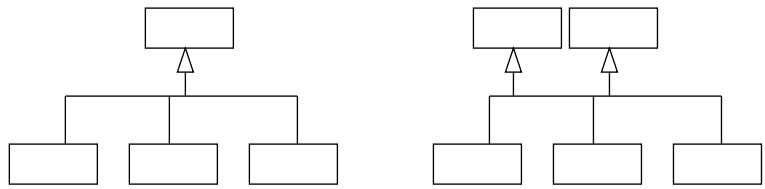


SE2 – OOPM – Teil 1 23

Einfach- und Mehrfachvererbung



Einfachvererbung **Mehrfachvererbung**



- Sind Vererbungshierarchien baumförmig, d.h. hat eine Klasse nur **eine unmittelbare Oberklasse** und beliebig viele Unterklassen, dann sprechen wir von **Einfachvererbung**.
- Hat eine Klasse mehr als eine unmittelbare Oberklasse, sprechen wir von **Mehrfachvererbung**.

Java erlaubt nur Einfachvererbung zwischen Klassen, bietet aber Mehrfachvererbung zwischen Interfaces.

SE2 – OOPM – Teil 1 24

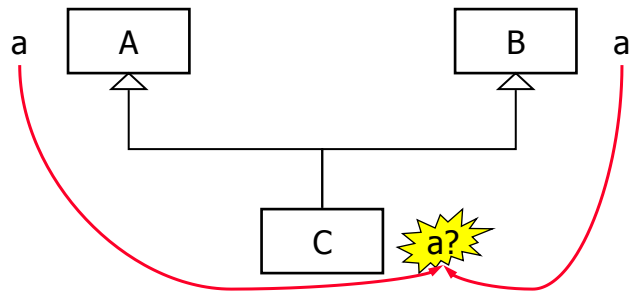
Mehrfachvererbung

- Die Notwendigkeit von Mehrfachvererbung in Programmiersprachen wurde Anfang der 90er Jahre heiß in der Forschergemeinde diskutiert.
- Vorläufiges Ergebnis für neuere Sprachen (wie Java, C#):
  - **Mehrfach-Subtyping ist nützlich und gewünscht.**
  - **Mehrfach-Implementationsvererbung ist (eher) kompliziert.**
- Das größte Problem bei Mehrfachvererbung ist der Umgang mit **Namenskollisionen**.
- Allgemein unterscheidet man dabei:
  - **Vererbung verschiedener, aber gleichnamiger Merkmale**
  - **Vererbung eines Merkmals über verschiedene Wege (Diamant-Vererbung)**

SE2 – OOPM – Teil 1 25

Vererbung verschiedener, aber gleichnamiger Merkmale

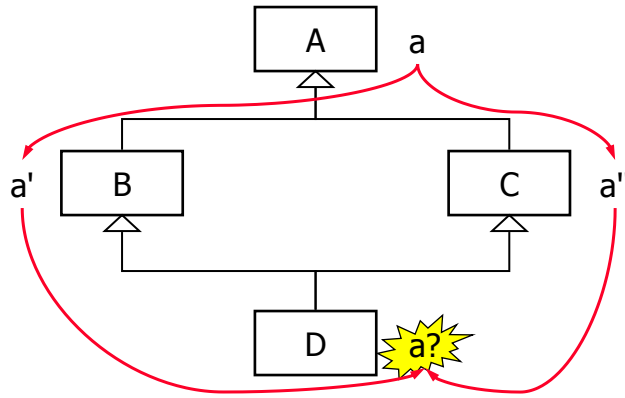
- Horizontale Namenskollision: verschiedene geerbte Eigenschaften wurden voneinander unabhängig mit gleichen Namen in Superklassen definiert.



SE2 – OOPM – Teil 1 26

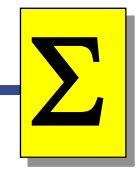
Diamant-Vererbung

- Ein Merkmal wird über verschiedene Vererbungspfade geerbt. Auf jedem Vererbungspfad können Eigenschaften verändert worden sein.



SE2 – OOPM – Teil 1 27

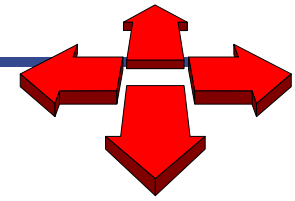
Vorläufige Zusammenfassung



- **Vererbung** ist eine zentrale Eigenschaft objektorientierter Programmiersprachen; **aber**: Vererbung ist auch einer der am stärksten missbrauchten und missverstandenen **Sprachmechanismen**.
- Vererbung als Begriff ist stark **überladen**; viele verschiedene **Konzepte** werden darunter zusammengefasst. Die wichtigsten sind:
  - **Subtyp-Polymorphie** auf Typebene für das Formulieren von **Typhierarchien** (Subtyping) und für **Typabstraktion**.
  - **Implementationsvererbung** für das hochflexible Kombinieren von ausführbaren Quelltext-Elementen (vor allem Methoden).
- SoftwaretechnikerInnen sollten diese sehr verschiedenen Konzepte klar voneinander trennen können; wir werden sie uns deshalb getrennt voneinander im Folgenden näher ansehen.
- In der Praxis treten diese beiden Konzepte meist gemeinsam auf; umso wichtiger ist ein klares Verständnis der Unterschiede.

SE2 – OOPM – Teil 1 28

Übersicht Subtyping



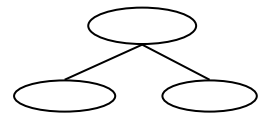
- Fachliche Typhierarchien
- Technische Eigenschaften des Subtyping
- Ko- und Kontravarianz
- Subtyping und Zusicherungen

Allgemeines

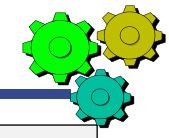
- Beim Subtyping werden **Typen** hierarchisch miteinander in Beziehung gesetzt.
- Eine Subtyp-Beziehung sollte eine **ist-ein-Beziehung** ausdrücken (aber nicht jede ist-ein-Beziehung ist eine Subtyp-Beziehung).
- Nur eine genaue Kenntnis der technischen Grundlagen von Subtyping versetzt uns in die Lage, **fachliche Hierarchien** in einem Anwendungsbereich geeignet in **Typ-Hierarchien** abzubilden.

Fokus beim Subtyping:

Ersetzbarkeit!



Entwurf von Typhierarchien: Erkennen gleichartiger Umgangsformen ...



Modellierung der Umgangsformen mit Materialien in einem Bürokontext:

Stapel
- Stapel drauflegen
- *Dokument einfügen*
- *Dokument auswählen*
- *Dokument entnehmen*
- auflösen
- ausbreiten

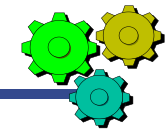
Ablage
- *Dokument einfügen*
- *Dokument auswählen*
- *Dokument entnehmen*

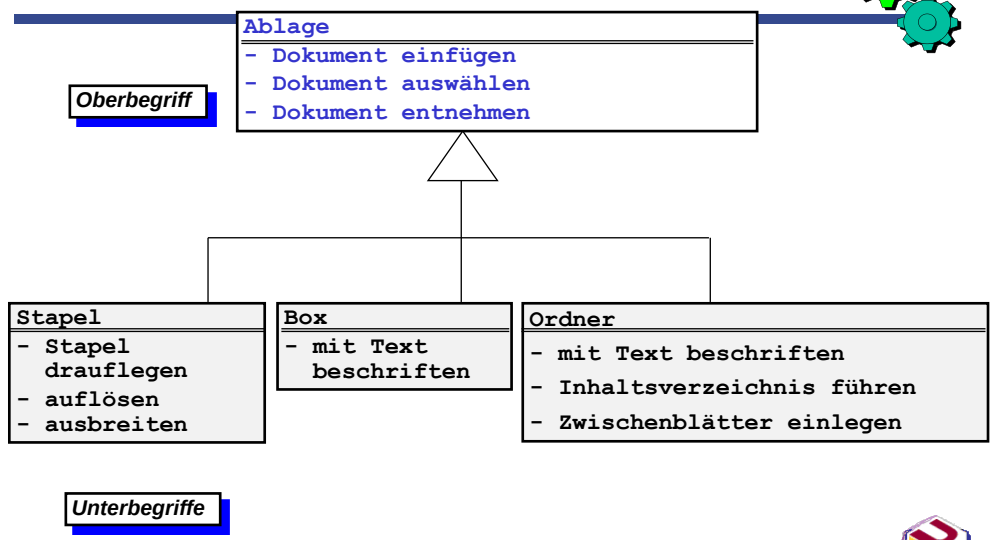
Ordner
- mit Text beschriften
- Inhaltsverzeichnis führen
- *Dokument einfügen*
- *Dokument auswählen*
- *Dokument entnehmen*
- Zwischenblätter einlegen

Box
- *Dokument einfügen*
- *Dokument auswählen*
- *Dokument entnehmen*
- mit Text beschriften

SE2 – OOPM – Teil 1 31

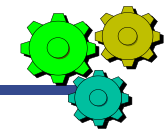
... als Grundlage einer Typhierarchie

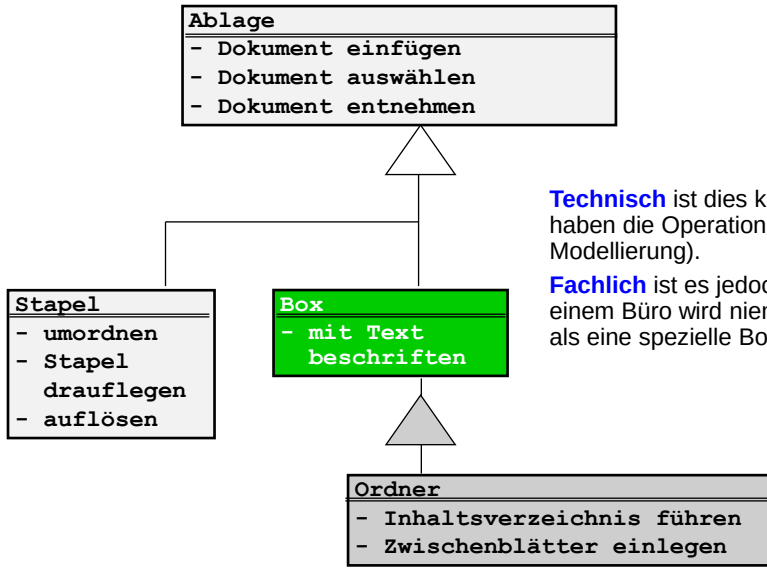




SE2 – OOPM – Teil 1 32

Eine schlechte Verwendung von Subtyping: "Box" als Supertyp von "Ordner"



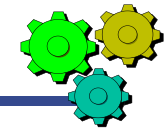


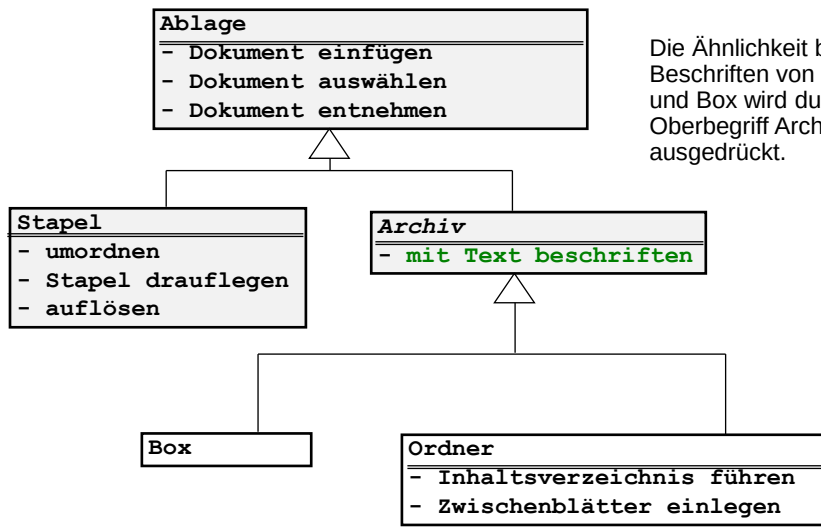
Technisch ist dies korrekt (alle Typen haben die Operationen aus der Modellierung).

Fachlich ist es jedoch **unsauber**: In einem Büro wird niemand einen Ordner als eine spezielle Box ansehen.

SE2 – OOPM – Teil 1 33

Subtyping soll spezielle Klassifikation ausdrücken (*"verstehen als"*, *"is-a"*)

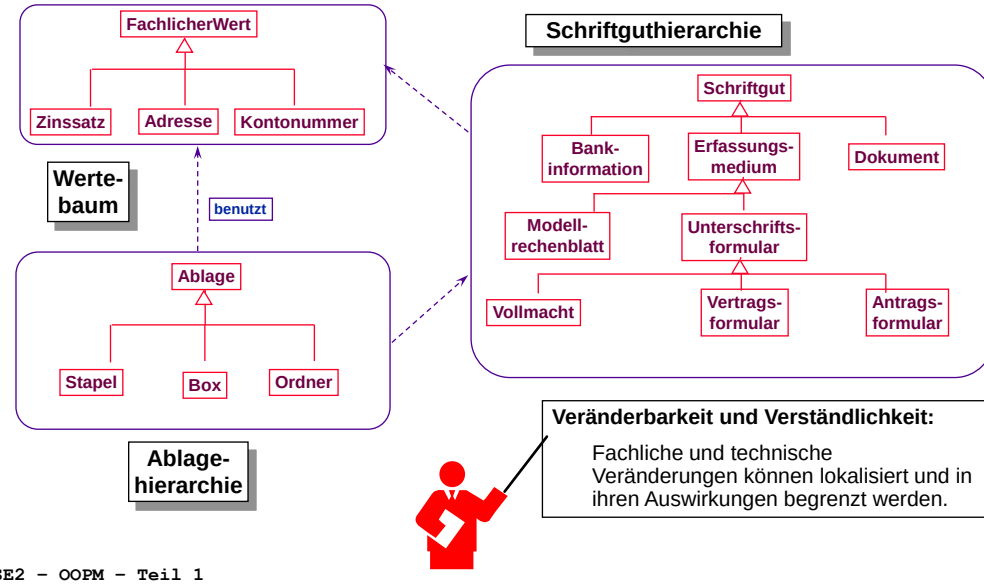




Die Ähnlichkeit beim Beschriften von Ordner und Box wird durch den Oberbegriff Archiv ausgedrückt.

SE2 – OOPM – Teil 1 34

Die Softwarearchitektur spiegelt die fachlichen Begriffe



Veränderbarkeit und Verständlichkeit:

Fachliche und technische Veränderungen können lokalisiert und in ihren Auswirkungen begrenzt werden.

SE2 – OOPM – Teil 1 35

Noch einmal: Erkennen gleichartiger Umgangsformen ...

Sparbuch

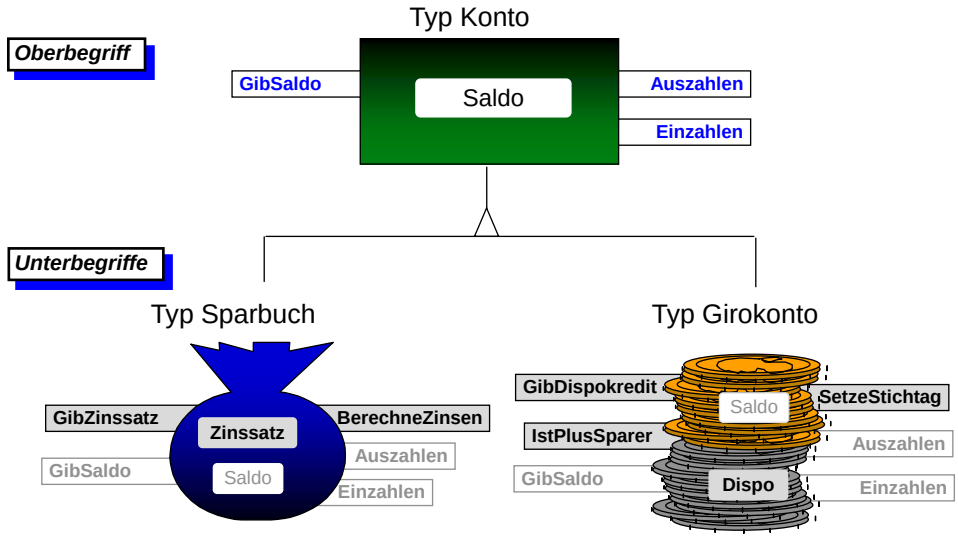
```
GibSaldo
Auszahlen
Einzahlen
GibZinssatz
BerechneZinsen
```

Girokonto

```
GibSaldo
Auszahlen
Einzahlen
GibDispokredit
GibStichtag
IstPlussparer
```

SE2 – OOPM – Teil 1 36

... als Grundlage einer Typhierarchie



Subtyping in Java

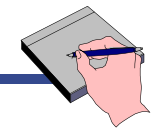
- In Java besteht eine **Subtyp-Beziehung**…
  - **zwischen Interfaces**, die über die **extends**-Beziehung verknüpft sind;
  - **zwischen einer Klasse und einem Interface**, die über eine **implements**-Beziehung verknüpft sind (denn eine Klasse definiert in Java immer auch einen Typ);
  - **zwischen Klassen**, die miteinander über eine **extends**-Beziehung verknüpft sind (denn Klassen definieren in Java immer Typen).

Das Schlüsselwort `extends` ist in Java überladen: Es wird sowohl für Subtyping zwischen Interfaces verwendet als auch für Subtyping und Subclassing zwischen Klassen.

Interfaces und Subtyping

- Ein **Interface** in Java ist eine **reine Typbeschreibung** und ist deshalb immer Teil einer Subtyp-Beziehung.
- Die häufigsten Verwendungen von Interfaces sind…
  - für **Typabstraktion**: Ein Interface beschreibt einen abstrakten Datentyp, der von mehreren Klassen implementiert werden kann. Der Name des Interfaces ist meist ein **Substantiv** (`Set`, `List`, `Map`, etc.).
  - für **adjektivische Abstraktionen**: Ein Interface beschreibt nur eine Teilfunktionalität/Rolle, deren Operationen von implementierenden Klassen neben anderen Operationen angeboten werden. Der Name des Interfaces ist meist ein **Adjektiv** (`Comparable`, `Iterable`, etc.).
  - für **reine Typhierarchien**: Mehrere Interfaces beschreiben Abstraktionen, die in einer hierarchischen Beziehung zueinander stehen; siehe etwa die Interfaces im Java Collection Framework (`Collection` als Supertyp von `List` und `Set` etc.).
- Eine weniger übliche Verwendung stellen die so genannten **Marker-Interfaces** dar: Ein Marker-Interface definiert keine Operationen, sondern dient nur zur Typprüfung. Beispiele sind `java.lang.Clonable` und `java.io.Serializable`.

Redefinieren vs. Redeklarieren



- **Redefinieren** ist das Ändern der Implementation einer Operation in einer Subklasse (später mehr dazu beim Thema Implementationsvererbung).
  - **eine andere Methode für dieselbe Operation**
  - Prominentes Beispiel aus SE1: `equals` aus `Object`:
    - `equals` ist eine **Operation** des **Typs** `Object`
    - `equals` hat eine Implementation in der **Klasse** `Object` (Prüfung auf Identität)
    - wir können `equals` für eine eigene Klasse redefinieren, indem wir in der eigenen `equals-Methode` definieren, wie Gleichheit für Exemplare unserer Klasse definiert sein soll.
    - es bleibt **dieselbe Operation**!
- **Redeklarieren** ist das Ändern der **Signatur** einer Operation in einem Subtyp bzw. einer Subklasse.
  - **Änderung der Schnittstelle**

Grenzen beim Redeklarieren

- Aus Gründen der Typsicherheit sollte die **Ersetzbarkeit** erhalten bleiben.
- In statisch typsicheren Sprachen ist ein Redeklarieren deshalb nur in sehr begrenztem Maße möglich.
- Der Name* einer Operation und die Anzahl (und Reihenfolge) der Parameter müssen gleich bleiben.
- Einzig möglich: **Typänderungen** für Parameter und Ergebnisse

*Eiffel erlaubt auch das Umbenennen von Methoden beim Redeklarieren. Dies trägt nicht zum besseren Verständnis des Quelltextes bei.

Formale Definition von Subtyping

Der **Typ einer Operation** kann geschrieben werden als *Op(S):T* für eine Operation, die einen Parameter vom Typ *S* bekommt und ein Ergebnis vom Typ *T* liefert. Der Typ einer weiteren Operation, *Op(R):U*, ist ein **Subtyp** der ersten Operation, wenn *S* ein Subtyp von *R* ist und *U* ein Subtyp von *T*:

$(Op(R): U) <: (Op(S): T)$, wenn $S <: R$ und $U <: T$

Diese Definition gilt analog für Operationen mit *n* Parametern ($n \geq 0$) und mit *m* Ergebnistypen ($m \geq 0$).

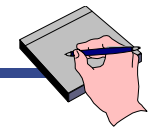
Ein **Objekttyp** definiert eine Menge von *n* Operationen m_i mit Typ T_i für $1 \leq i \leq n$, verkürzt geschrieben als $ObjektTyp\{ m_i : T_i \}_{1 \leq i \leq n}$.

Ein Objekttyp ist ein **Subtyp** eines anderen Objekttyps, wenn er mindestens die Operationen des anderen Typs definiert und jede dieser Operationen ein Subtyp des Operationstyps des anderen Objekttyps ist:

$ObjektTyp\{ m_j : S_j \}_{1 \leq j \leq m} <: ObjektTyp\{ m_i : T_i \}_{1 \leq i \leq n}$,

wenn $n \leq m$ und für alle $i \leq n$ ist $S_i <: T_i$

Ko- und Kontravarianz



- Die formale Definition von Subtyping ist ein Ergebnis der **objektorientierten Typtheorie**.
- Alle Theorie ist grau; aus der Definition lassen sich jedoch wichtige und praktisch relevante Aussagen für die statische Typsicherheit ableiten:
  - **Ergebnistypen** können **kovariant** (d.h. der Spezialisierungsrichtung folgend) angepasst werden.
  - **Typen von Parametern** können **kontravariant** (d.h. entgegen der Spezialisierungsrichtung) angepasst werden.
- Insbesondere heißt dies auch, dass **kovariante** Anpassungen von **Parametertypen** nicht statisch typsicher sind; entsprechende Gegenbeispiele lassen sich leicht konstruieren.
  - An diesem Umstand scheitert beispielsweise die statische Typsicherheit in **Eiffel**!

Zulässig: Kovarianz für Ergebnistypen

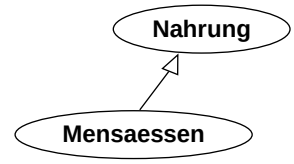
```
interface Person {
  public Person clone();
}

interface Student extends Person {
  public Student clone();
}
```

Redeklaration, da Änderung der Signatur.

Es bleibt **dieselbe** Operation.

In Java bis 1.4 nicht zugelassen, obwohl die VM es schon immer unterstützt hat.
Seit Java 1.5 zugelassen!

Zulässig: Kontravarianz für Parametertypen

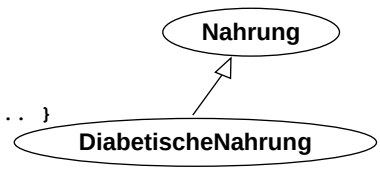



```
class Nahrung { ... }
class Mensaessen extends Nahrung { ... }

interface Person {
    void verzehre(Mensaessen param);
}

interface Student extends Person {
    void verzehre(Nahrung param);
}
```

Redeklaration, da Änderung der Signatur (kein Overloading wie in Java).

theoretisch interessant, praktisch irrelevant.

Nicht typsicher: Kovarianz für Parametertypen




```
class Nahrung { ... }
class DiabetischeNahrung extends Nahrung { ... }

interface Person {
    void verzehre(Nahrung param);
}

interface Diabetiker extends Person {
    void verzehre(DiabetischeNahrung param);
}
```

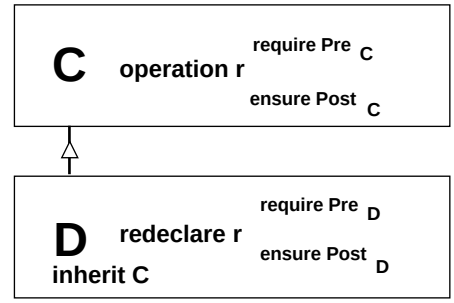
Kovariant, da Redeklaration **mit** der Vererbungsrichtung.

```
Benutzung:
Person p = new DiabetikerImpl();
Nahrung nahrung = new Nahrung();
p.verzehre(nahrung);
```


Vertragsmodell und Subtyping: Das Prinzip

Metapher:

- **Verträge** können an **Zulieferer** weitergegeben werden.
- Ein Zulieferer darf **höchstens** die Voraussetzungen fordern, die der Anbieter gefordert hat.
- Ein Zulieferer muss **mindestens** die Leistung erbringen, die der Anbieter versprochen hat.



§

Bei der Redeklaration einer Operation muss die Unterklasse den Vertrag von der Oberklasse übernehmen. Es muss gelten:

Pre_D **schwächer als** Pre_C : $Pre_C \Rightarrow Pre_D$

$Post_D$ **stärker als** $Post_C$: $Post_D \Rightarrow Post_C$

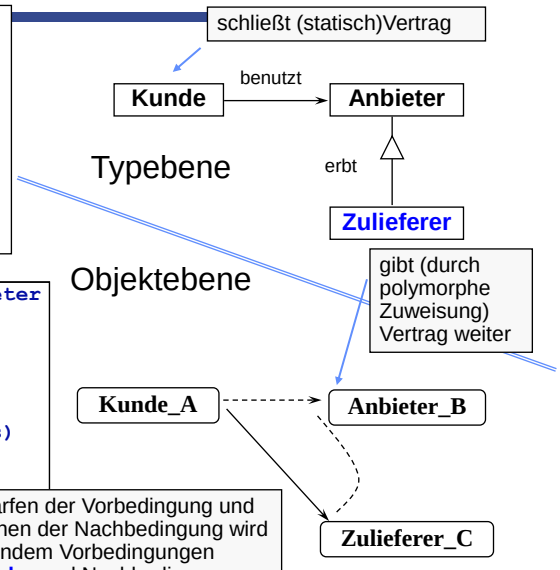
Die Prüfung der Vertragsübernahme ist ein i.a. unentscheidbares Problem.

SE2 – OOPM – Teil 1 47

Zusicherungen und Subtyping: Ein Beispiel

```
class Anbieter
{
   /**
    * @require preis > 500
    * @ensure Toleranz < 20
    */
   liefereBohrstueck(int preis)
   {
      ...
   }
}
```

```
class Zulieferer extends Anbieter
{
   /**
    * @require preis > 400
    * @ensure Toleranz < 10
    */
   liefereBohrstueck(int preis)
   {
      ...
   }
}
```



Ein Verschärfen der Vorbedingung und ein Aufweichen der Nachbedingung wird verhindert, indem Vorbedingungen immer mit **oder** und Nachbedingungen immer mit **und** verknüpft werden.

SE2 – OOPM – Teil 1 48

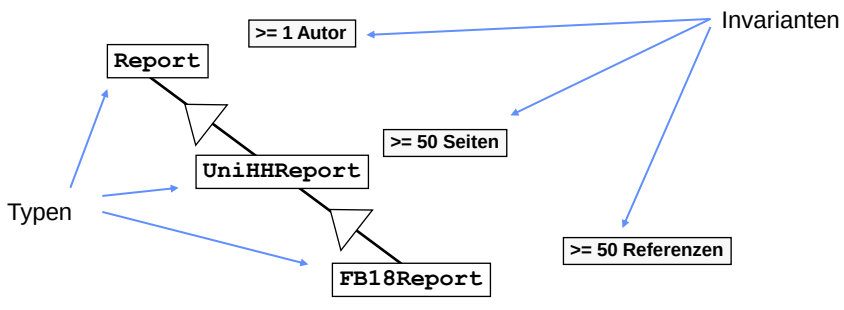
Zusicherungen und Subtyping in Programmiersprachen

- Da ein Compiler nicht überprüfen kann, ob die Regeln für Subtyping und Zusicherungen eingehalten werden, wurden in der Sprache **Eiffel** defensive Regeln für das Vertragsmodell festgelegt:
- **Vor- und Nachbedingungen**:
  - Eine Vorbedingung kann in Subtypen nur durch eine **oder-Verknüpfung** erweitert werden - die Bedingung wird höchstens abgeschwächt.
  - Eine Nachbedingung kann in Subtypen nur durch eine **und-Verknüpfung** erweitert werden - die Bedingung wird höchstens verschärft.
- **Invarianten**:
  - Eine Invariante kann in Subtypen nur durch eine **und-Verknüpfung** ergänzt werden - die Bedingung wird höchstens verschärft.

Analoge Regeln gelten in allen aktuellen Programmiersprachen, die das Vertragsmodell anbieten: **D**, **Fortress**, **Ada 2012**, …

SE2 – OOPM – Teil 1 49

Invarianten und Subtyping



- Die (Gesamt-) Invariante des Typs `FB18Report` ist **>= 1 Autor und >= 50 Seiten und >= 50 Referenzen**

- **Invarianten** werden mitvererbt („ge-undet“), d.h. entlang der Vererbungsrelation verschärft.
- Die Randbedingungen für **Verträge** werden somit anspruchsvoller.

SE2 – OOPM – Teil 1 50

Ko- und Kontravarianz und Zusicherungen

- Den Regeln für Ko- und Kontravarianz und denen für Subtyping und Zusicherungen liegt **dasselbe Prinzip** zugrunde:
  - Ein Klient sollte sich auf die statisch vereinbarten Verträge (Signatur und Zusicherungen einer Operation) verlassen können.
  - Wenn aufgrund von Subtyp-Polymorphie ein Exemplar eines Subtyps „hinter“ einer Variablen steckt, dann sollte sich dieses Exemplar stets so verhalten, wie der statische Typ es verspricht.
- Eine **kovariante Parametertypanpassung** widerspricht diesem Prinzip ebenso wie die **Verschärfung einer Vorbedingung**:
  - Das gerufene Exemplare akzeptiert in beiden Fällen nur einen kleineren Wertebereich;
  - der Klient liefert potenziell etwas aus dem größeren Wertebereich;
  - es kann deshalb, trotz pflichtbewusstem Klienten, zur Laufzeit zu Fehlern kommen!

Weiteres Beispiel: Generizität und Subtyping

Ist eine **Liste von Strings** ein Subtyp einer **Liste von Objekten**?

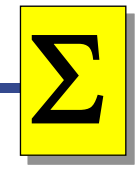
```
List<Object> objektListe;
```

Kovariante Typ-Parameter!

```
List<String> stringListe = new ArrayList<String>();

objektListe = stringListe; // zulässig?   // Nein, Fehler zur
                                           // Übersetzungszeit!
...
objektListe.add(new Object());
String s = stringListe.get(1);
```

Die kovariante Anpassung eines Typparameters bedeutet meist (und hier tatsächlich bei der Operation `add`) auch eine kovariante Parametertypanpassung in Operationen – und ist somit nicht typsicher!

Zusammenfassung Polymorphie und Vererbung



- **Vererbung** ist eine zentrale Eigenschaft objektorientierter Programmiersprachen; **aber**: Vererbung ist auch einer der am stärksten missbrauchten und missverstandenen **Sprachmechanismen**.
- Ein zentrales Vererbungskonzept auf Typebene ist (**Subtyp-**) **Polymorphie** für das Formulieren von **Typhierarchien** (Subtyping) und für **Typabstraktion**.
  - Grundidee dabei immer: **hinter einer Schnittstelle** können sich **verschiedenartige** (polymorphe) Typen und Implementationen verbergen, von deren **Unterschieden** auf Ebene der Benutzung **durch den Klienten** bewusst **abstrahiert** werden soll.
- **Implementationsvererbung** für das Kombinieren von ausführbaren Quelltext-Elementen (vor allem Methoden) werden wir in einer eigenen Vorlesung ausführlich betrachten.