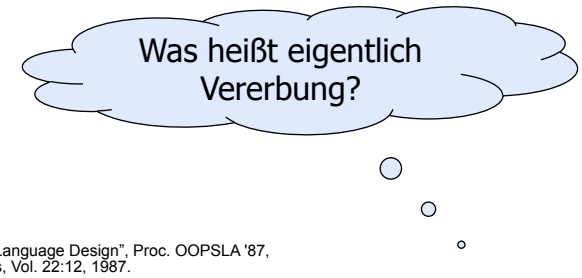
Motivation: Vererbung

- 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.



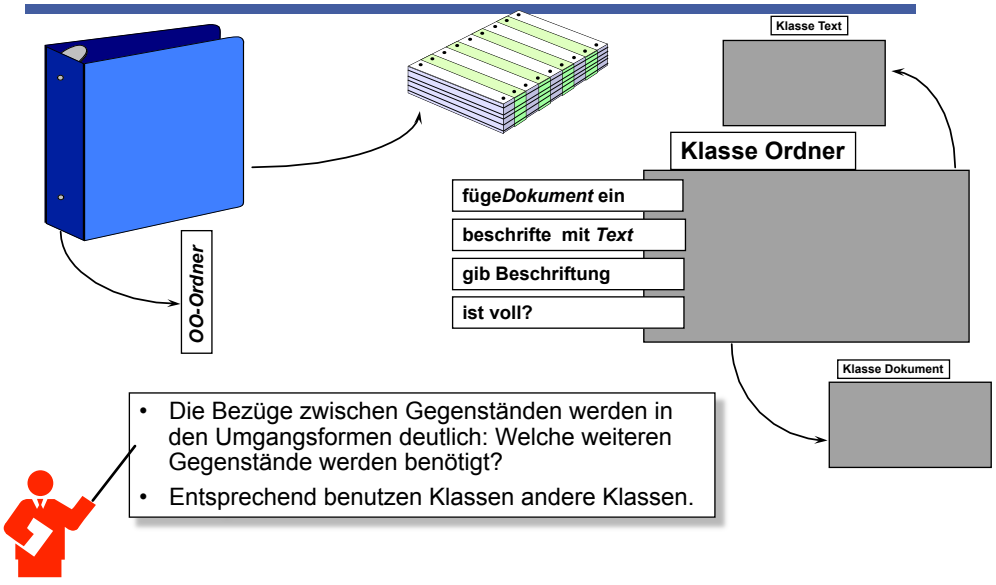
Wegner, P.: “Dimensions of Object-Based Language Design”, Proc. OOPSLA '87, Orlando, Florida; in ACM SIGPLAN Notices, Vol. 22:12, 1987.

„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).

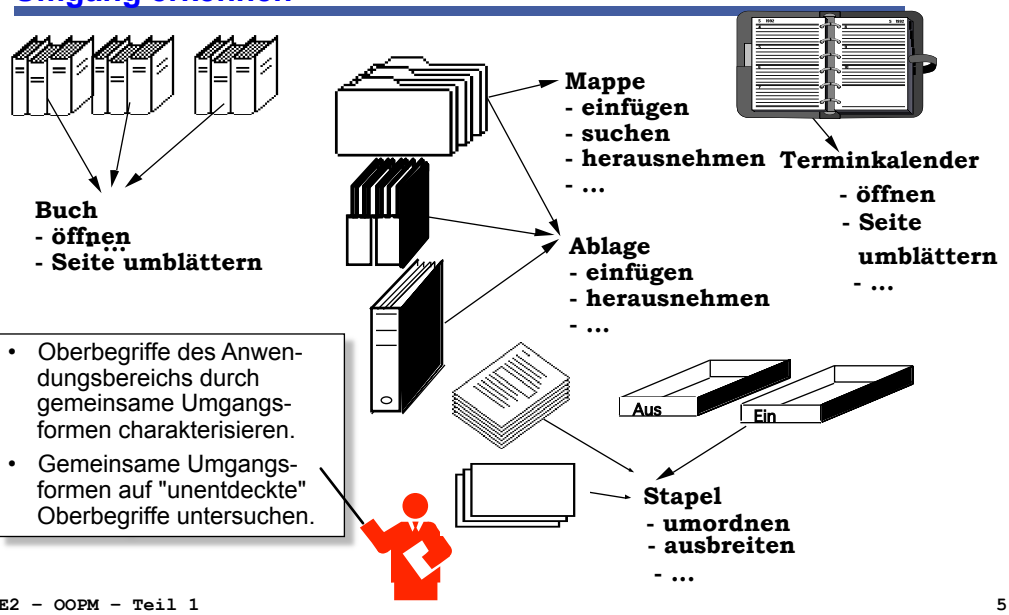
These:
Vererbung ist das GoTo der Objektorientierung!

Umgangsformen verweisen auf andere Gegenstände



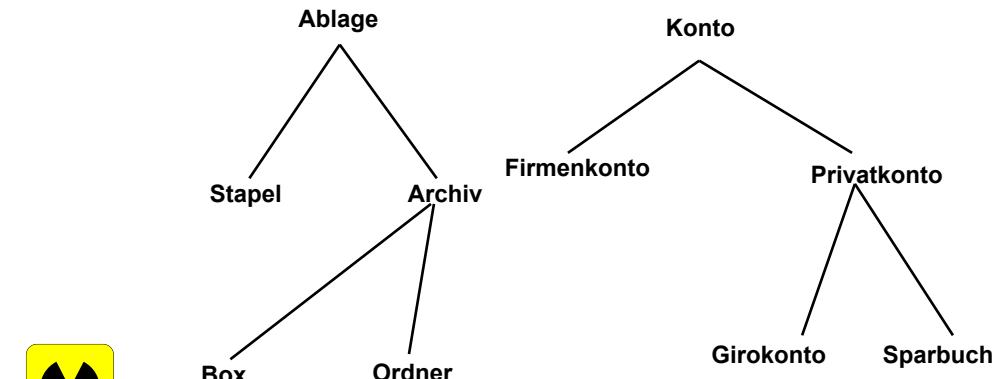
- Die Bezüge zwischen Gegenständen werden in den Umgangsformen deutlich: Welche weiteren Gegenstände werden benötigt?
- Entsprechend benutzen Klassen andere Klassen.

Ähnlichkeiten der Gegenstände im Umgang erkennen



- Oberbegriffe des Anwendungsbereichs durch gemeinsame Umgangsformen charakterisieren.
- Gemeinsame Umgangsformen auf "unentdeckte" Oberbegriffe untersuchen.

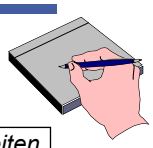
Oberbegriffe identifizieren



Diskussionen über "reine" Begriffe sind oft wenig zielführend, da unklar ist, was abstrahiert wird.

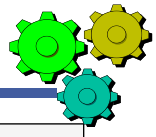
- Oft werden im Anwendungsbereich Oberbegriffe verwendet.
- Die Anordnung von Begriffen in Hierarchien verschafft eine konzeptionelle Übersicht über das Anwendungsgebiet.

Über Begriffshierarchien



- Ausgehend von gemeinsamen Umgangsformen bestimmen wir *Ähnlichkeiten* von unterschiedlichen Gegenständen. Diese Gemeinsamkeit drücken wir generalisierend in einem Oberbegriff aus. Durch solche *Generalisierung* oder *Klassifikation* entstehen *Begriffshierarchien*. Sie bilden als Teil der jeweiligen Fachsprache die Grundlage zur Zusammenarbeit und fördern das Verständnis eines Anwendungsgebiets.
- Begriffe können auch *spezialisiert* oder *konkretisiert* werden. Im Sinne einer verständlichen Begriffsbildung ist es günstig, wenn sich die Unterbegriffe eines Oberbegriffs durch ein *Charakteristikum* unterscheiden.

Entwurf von Typhierarchien: Erkennen gleichartiger Umgangsformen ...



Modellierung der Umgangsformen mit Materialien in einem Bürokontext:

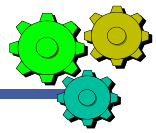
Stapel
- Stapel drauflegen
- *Dokument einfügen*
- *Dokument auswählen*
- *Dokument entnehmen*
- auflösen
- ausbreiten

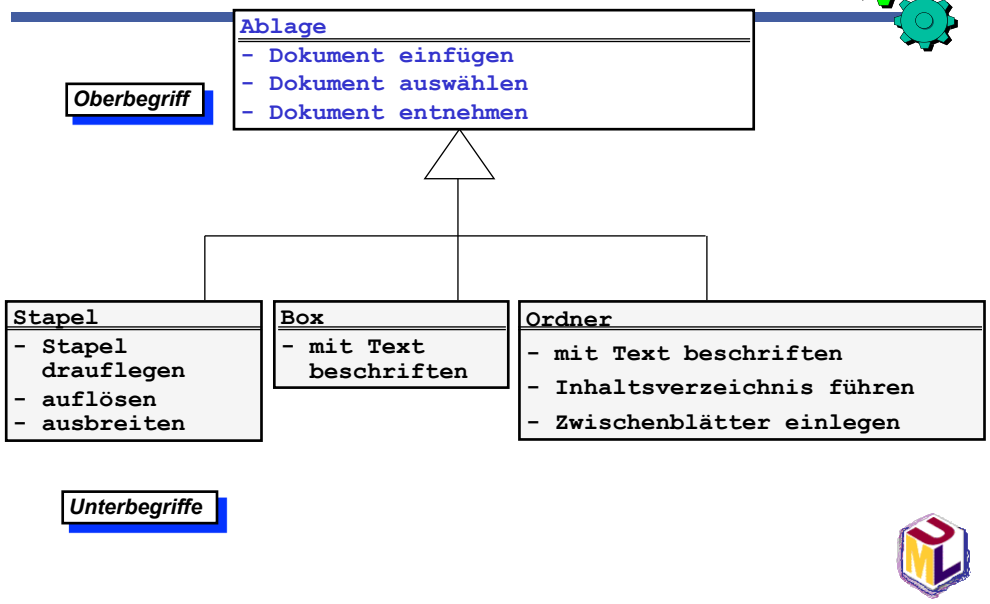
Ordner
- mit Text beschriften
- Inhaltsverzeichnis führen
- *Dokument einfügen*
- *Dokument auswählen*
- *Dokument entnehmen*
- Zwischenblätter einlegen

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

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

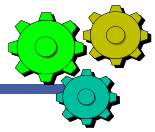
... als Grundlage einer Typhierarchie

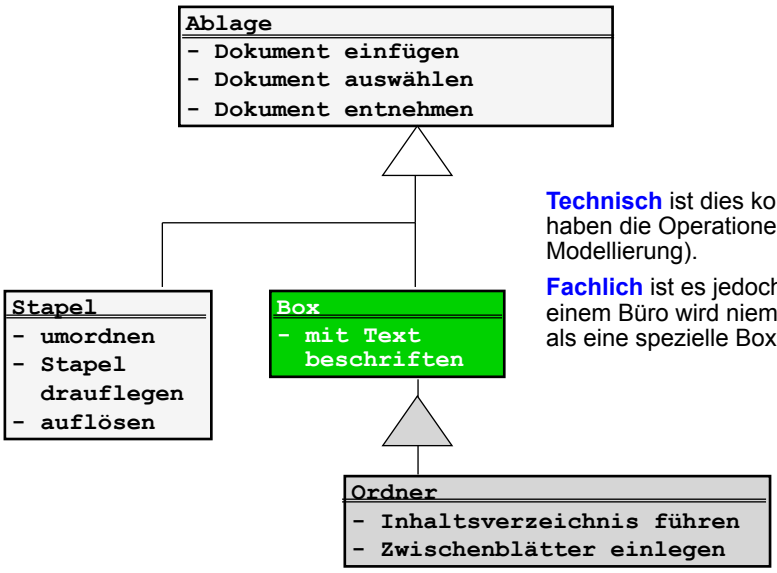




SE2 – OOPM – Teil 1 9

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



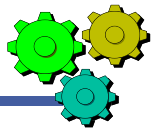


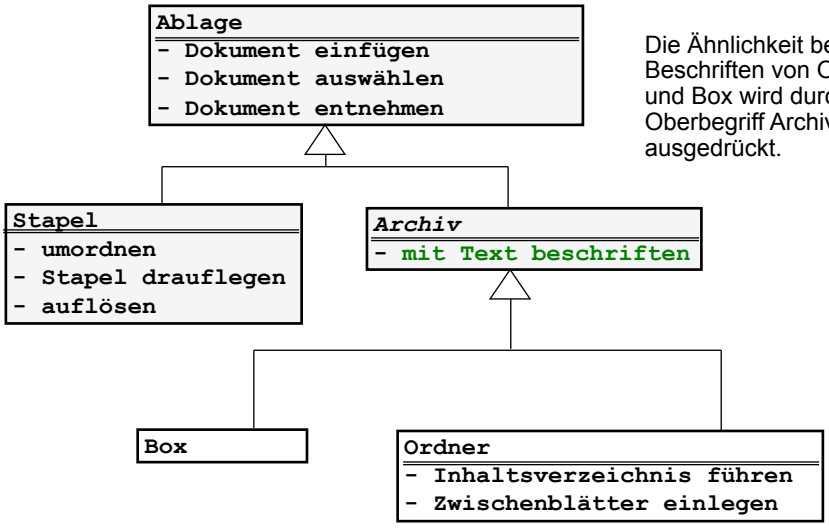
Technisch ist dies korrekt (alle Typen haben die Operationen aus der Modellierung).

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

SE2 – OOPM – Teil 1 10

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

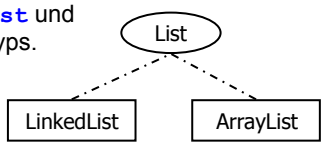




Die Ähnlichkeit beim Beschriften von Ordner und Box wird durch den Oberbegriff Archiv ausgedrückt.

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?

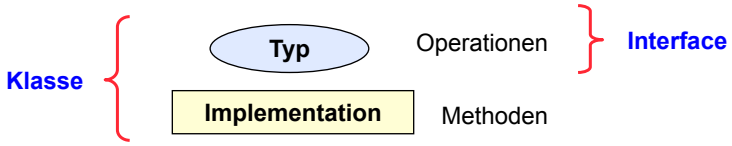
Außensicht, öffentliche Eigenschaften, Dienstleistungen, Typ

- 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

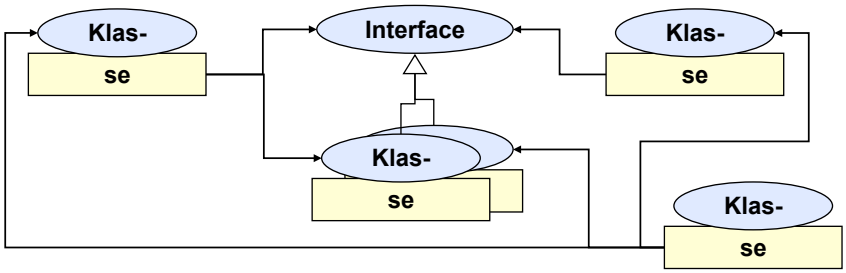
Die objektorientierte Klasse: Typ und Implementation

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

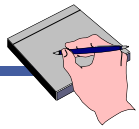


Statik von OO Systemen: Geflechte von Typen

- 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`)



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.

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.

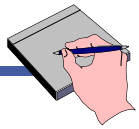
```
liste1 = new LinkedList<String>(); // 1. dynamischer Typ von liste1
```

- 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.

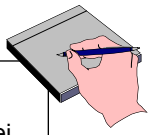
Dynamisches Binden



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!

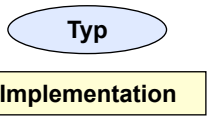
Ein wesentliches Merkmal objektorientierter Systeme: Polymorphie

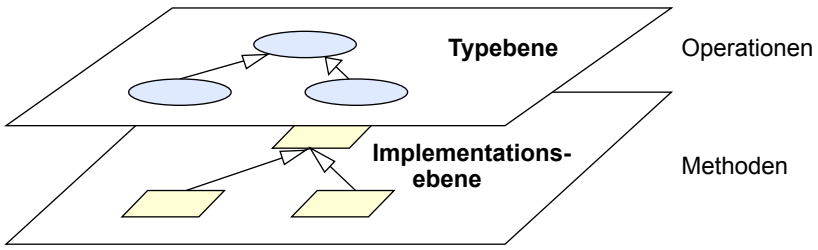


- Polymorphie heißt Vielgestaltigkeit.
- Fachlich: Wir generalisieren Gegenstände in Begriffshierarchien. Dabei abstrahieren wir auch oft ihre Umgangsformen. Dann beschreiben wir bei Oberbegriffen generalisierte Konzepte des Umgangs. In den Unterbegriffen können diese Umgangsformen unter Verwendung des gleichen Begriffs unterschiedlich konkretisiert werden.
- Technisch: In objektorientierten Programmen ist Polymorphie die Fähigkeit eines Bezeichners, zur Ausführungszeit nicht nur auf Objekte des statisch im Text deklarierten Typs, sondern auf Objekte eines Subtyps verweisen zu können.

Vererbung: Auf Typ- und Implementationsebene

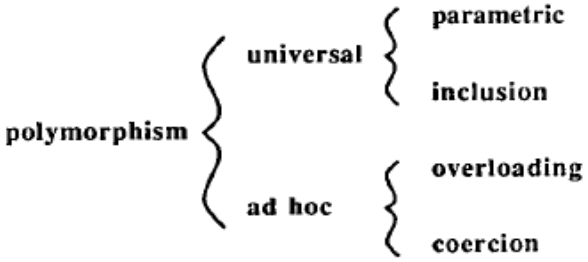
- Mit Vererbung werden **hierarchische Beziehungen** modelliert.
- Die Unterscheidung von Schnittstelle bzw. Typ und Implementation ermöglicht eine differenzierte Betrachtung von Hierarchien auf Typ- und Implementationsebene.





Polymorphie nach Cardelli und Wegner (1985)

polymorph:
griechisch für „vielgestaltig“.

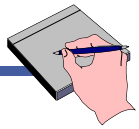


Entsprechung in Java:

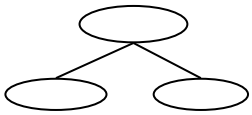
- **Generizität (seit Java 5)**
- **Vererbung et al.**
- **Operatoren, insbes. +; Überladen von Methoden- und Konstruktornamen**
- **Typumwandlungen für primitive Typen**

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

Subtyp-Polymorphie

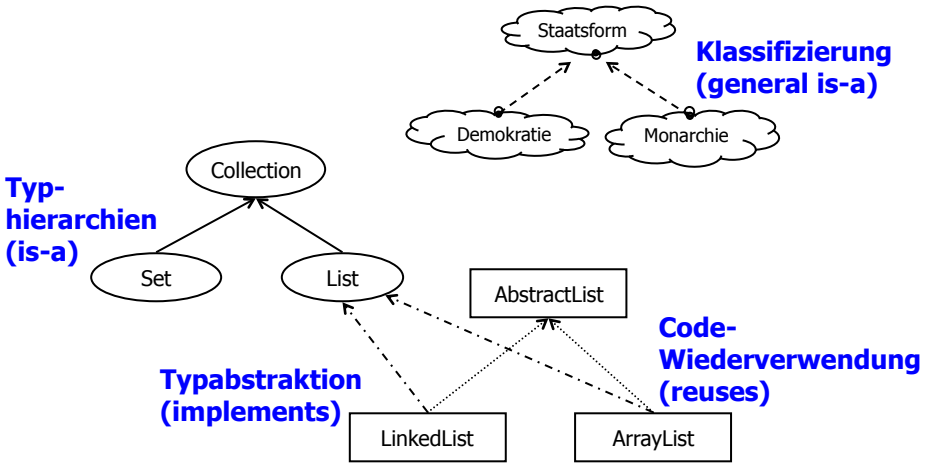


- **Inclusion Polymorphism** nach Cardelli u. Wegner
- Im Kontext objektorientierter Sprachen meist kurz **Polymorphie**
- Eng mit **Ersetzbarkeit** (engl. substitutability) verknüpft: Variable eines Supertyps kann auf Exemplare von Subtypen verweisen.
- Somit auch eng verknüpft mit der Unterscheidung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



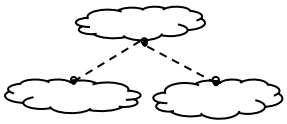
- Erfordert **dynamisches Binden**!
- Zentrales **technisches Konzept** objektorientierter Sprachen
- Voraussetzung für Typhierarchien und Typabstraktion

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.



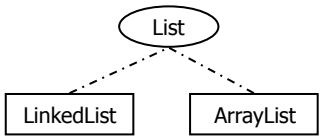
- Häufig im Zusammenhang mit Vererbung genannt; wird in ihrer allgemeinen Form jedoch **nicht** durch die Mechanismen **von Programmiersprachen unterstützt**!

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

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

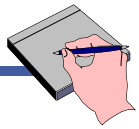
Typabstraktion (bekannt aus SE1)

- 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.

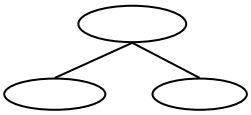


- Unterschiede zeigen sich möglicherweise in der Effizienz (falls nicht Teil der Spezifikation). Je nach Benutzungsprofil wirken sich die Implementationen unterschiedlich aus.
- Setzt **dynamisches Binden** nur dann voraus, wenn mehrere Implementationen im gleichen Programm aktiv sein sollen!
- In Java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

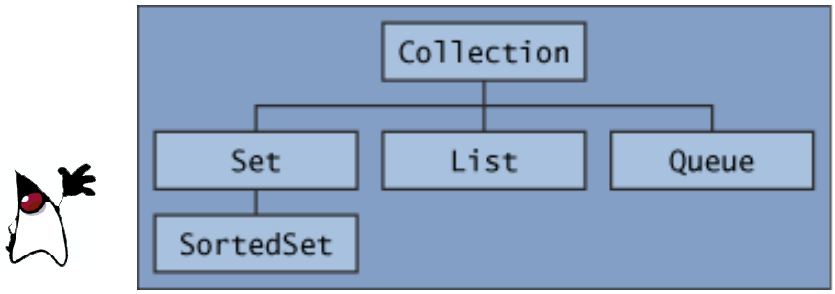


- Das Bilden von Typhierarchien wird auch **Subtyping** genannt.
- Basiert auf **Subtyp-Polymorphie**.

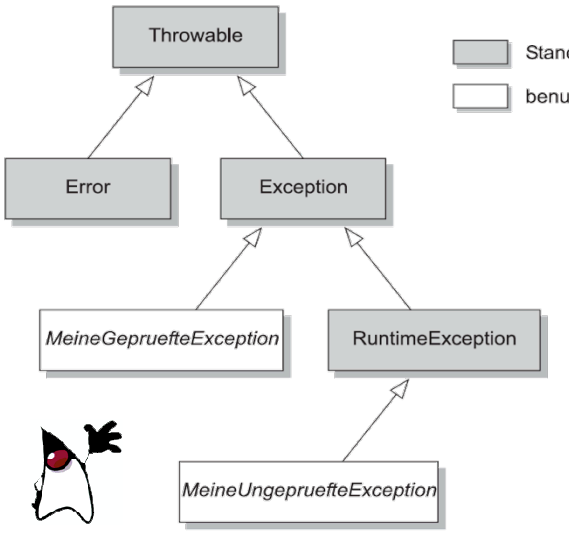
- Wir gehen noch ausführlich auf Subtyping ein.

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.



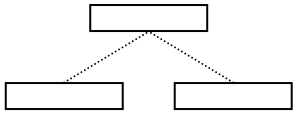
Beispiel: Javas Exception-Hierarchie



- `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.

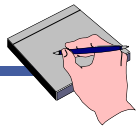
Code-Wiederverwendung

- 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.

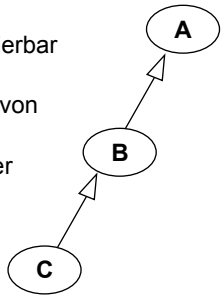


- Theoretisch (und auch praktisch, wie etwa in der Sprache **Sather**) durch einfaches Kopieren von Quelltexten realisierbar (**statisch**).
- Meist jedoch ebenfalls über dynamisches Binden realisiert.
- Wir gehen in einer späteren Vorlesung noch ausführlich auf Code-Wiederverwendung ein.

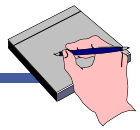
Einige weitere Begriffe



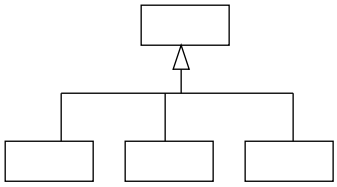
- Wenn wir uns auf **Hierarchien von Typen** beziehen, werden wir im Folgenden die Begriffe **Super- und Subtyp** verwenden.
- Wenn wir uns auf **Hierarchien von Implementationen** (Klassen) beziehen, werden wir im Folgenden die Begriffe **Ober- und Unterklasse** verwenden.
- Subtyp- und Unterklassenbeziehungen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



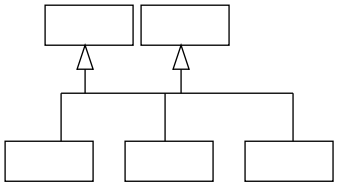
Einfach- und Mehrfachvererbung



Einfachvererbung



Mehrfachvererbung

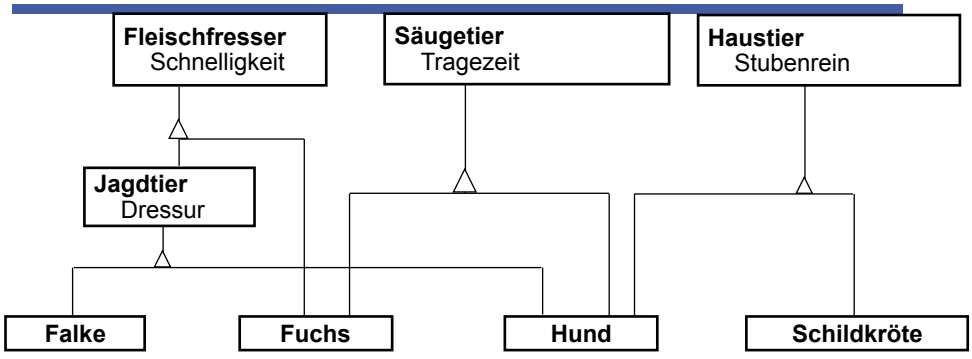


- Sind Vererbungshierarchien baumförmig, d.h. hat eine Klasse nur **eine unmittelbare Oberklasse** und beliebig viele Unterklassen, dann sprechen wir von **Einfachvererbung**.
- Hat eine Klasse mehr als eine unmittelbare Oberklasse, sprechen wir von **Mehrfachvererbung**.

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

SE2 – OOPM – Teil 1 31

Ein (naives) Beispiel für Mehrfachklassifikation



Mehrfachvererbung muß gut begründet sein, denn:

- Entwürfe mit Mehrfachvererbung sind aufwendig zu erschließen und weiterzuentwickeln.
- Die Möglichkeit der Mehrfachklassifikation verleitet zu schlechten, an Strukturen orientierten Entwürfen.

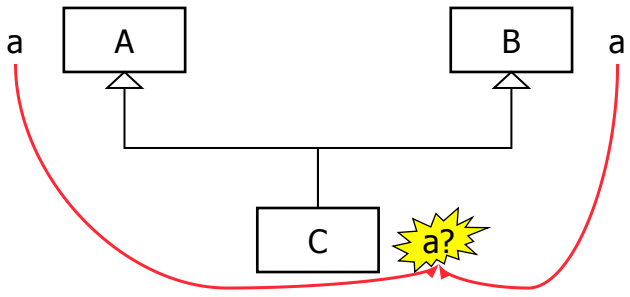
SE2 – OOPM – Teil 1 32

Mehrfachvererbung – Programmiersprachliche Diskussion

- 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)**

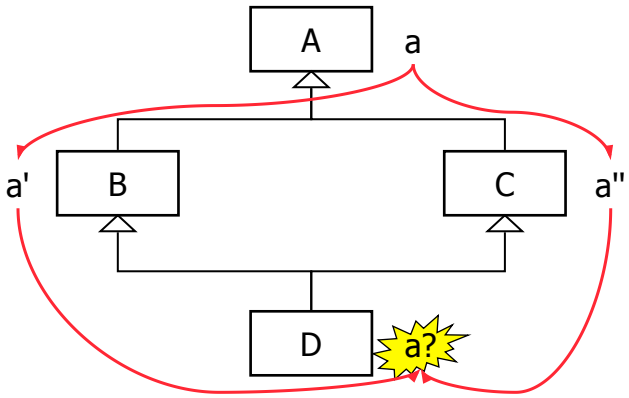
Vererbung verschiedener, aber gleichnamiger Merkmale

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

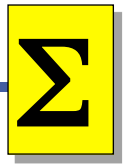


Diamant-Vererbung

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



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.

