Einführung in die Informatik II
Zentrale Konzepte der Objekt-Orientierung

Prof. Bernd Brügge, Ph.D
Institut für Informatik
Technische Universität München

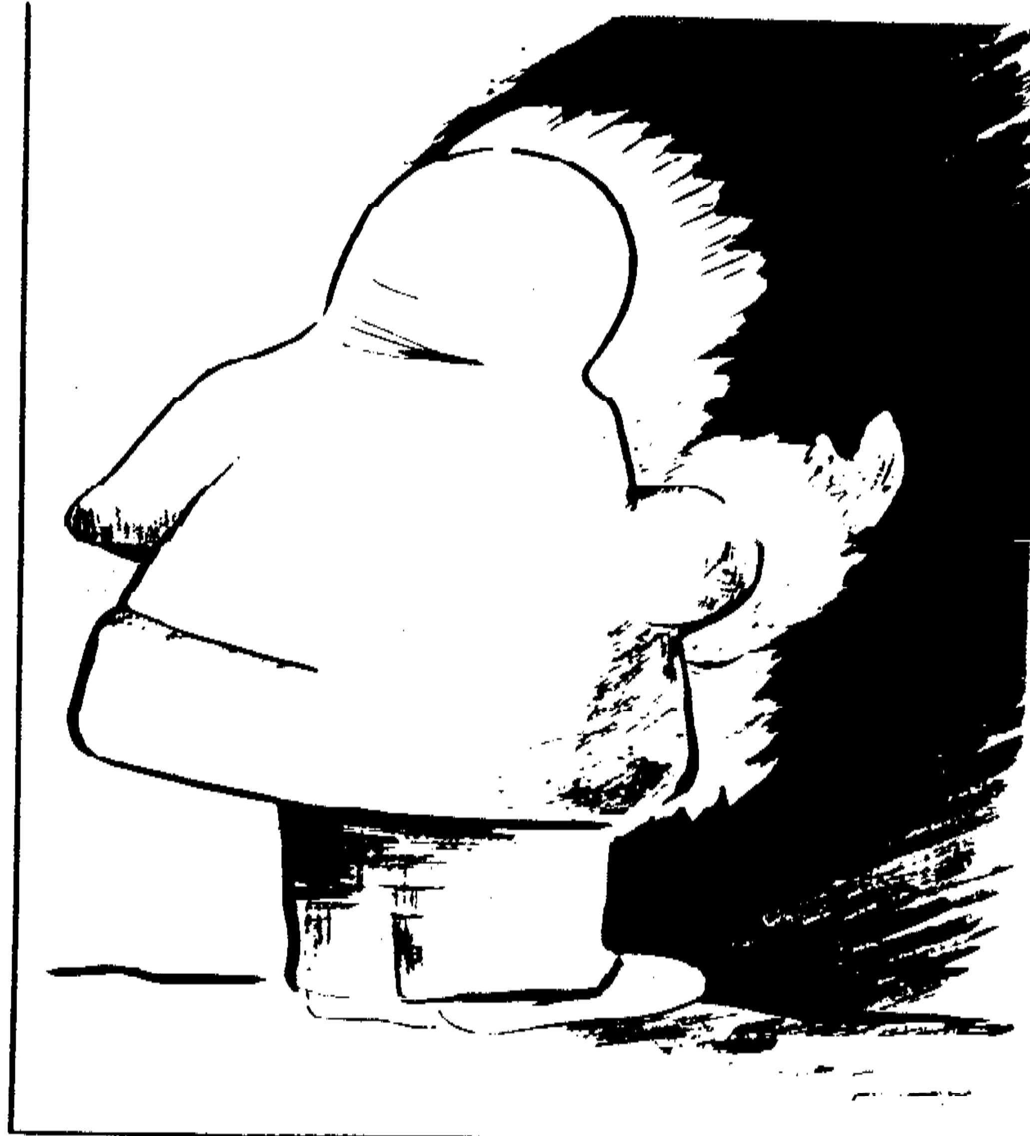
Sommersemester 2001

14.-16. Mai 2001

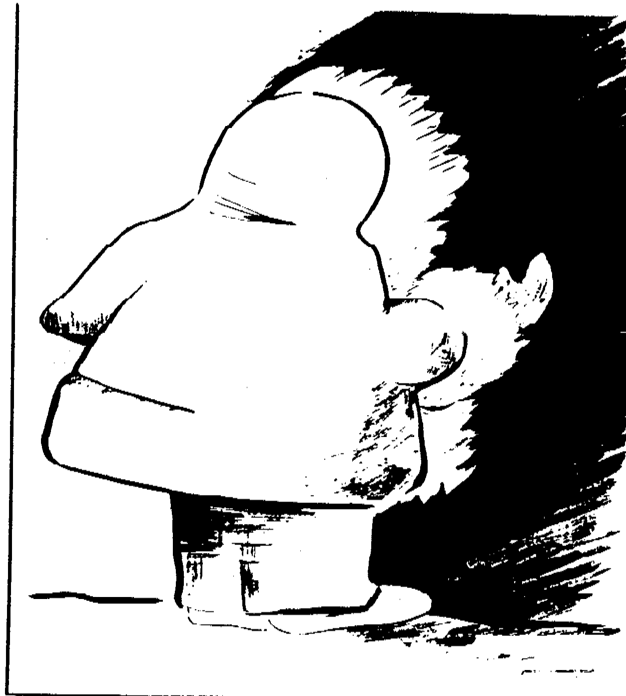
Eine Bemerkung zum Finden von Klassen

- In der letzten Vorlesung hatten wir gezeigt, wie man mit Abbott's Technik Klassen, Attribute, und Operationen finden kann.
  - Die Technik ist rein syntaktisch and daher nur eine Empfehlung
  - Nicht jede Klasse, die man so findet, ist auch sinnvoll.
  - Umgekehrt finden Sie mit Abbott's Analyse ziemlich sicher nicht alle wichtigen Klassen.
- Benutzen Sie die Regeln von Abbott wie andere Heuristiken zum Finden von Klassen: *Mit Vorsicht!*
- Die Technik eignet sich am besten beim Brainstorming während der Analysephase, am besten mit einer Tafel.
  - Zeichnen Sie alle Klassen, die Ihnen oder Ihren Teammitgliedern einfallen, auf die Tafel.
  - Streichen Sie unnütze Klassen dann aus (nicht auswischen!).
  - Verwenden Sie die verbleibenden Klassen in einem ersten vorläufigen Klassendiagramm.

Was ist Das?



Brainstorming-Meeting: Identifikation von Objekten



- Was sehen Sie für Objekte?
- Eskimo
- Gesicht
- Kopf

So könnte die Tafel nach einem Treffen aussehen

Eskimo X

Kopf

Höhle X

Indianer

Nase

Mantel X

Schuh X

Mund

Handschuh X

Kinn

Überblick über diesen Vorlesungsblock

- Was wollen wir mit unseren Systemen erreichen?
  - **Organisation:** Das Wissen der Problemdomäne ist "gebändigt"
  - **Flexibilität**: Das System soll auch an ungeahnte oder späte Änderungen angepasst werden können.
  - **Wiederverwendbarkeit**: Die Lösung soll beim nächsten Problem wieder einsetzbar sein.
  - **Allgemeinheit**: Die Lösung soll bei vielen Problemklassen einsetzbar sein.
- 3 Konzepte aus der Objekt-Orientierung, um dies zu erreichen:
  - **Vererbung**
  - **Dynamische Bindung**
  - **Polymorphismus**
- Der Einsatz dieser Konzepte wird dann am Beispiel von **generischen Klassen** und von **Entwurfsmustern** demonstriert.

Warum Vererbung ?

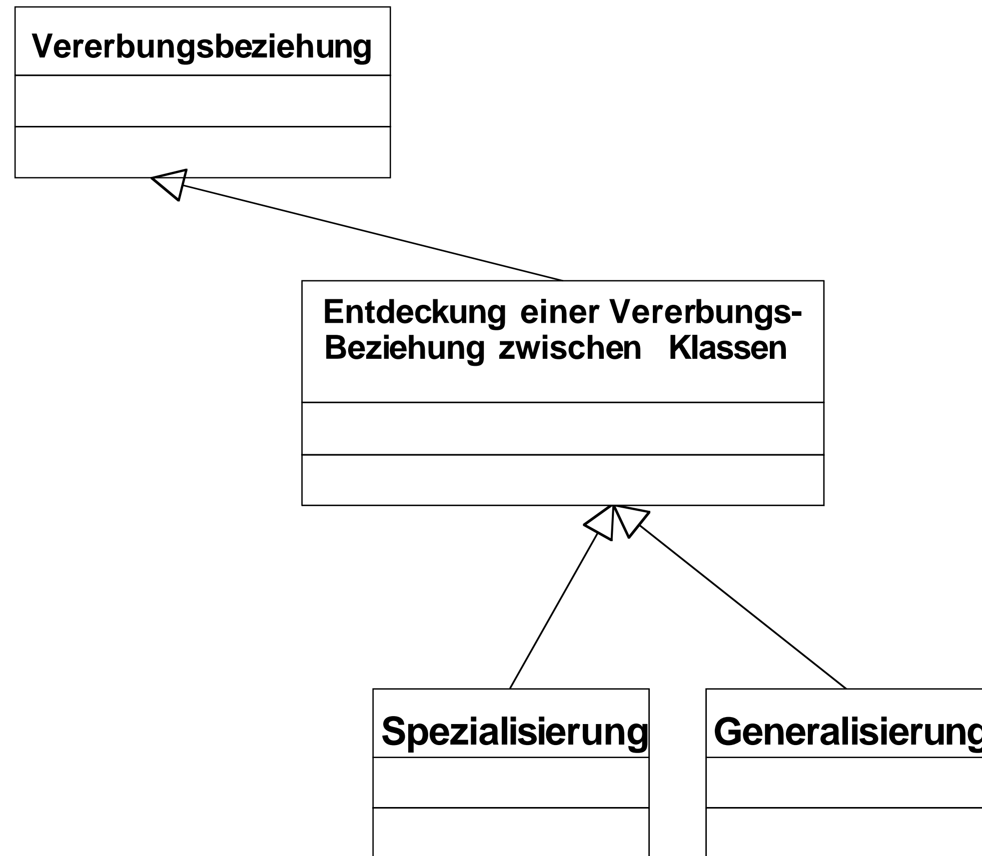
Vererbung ist ein nützliches Konzept, und zwar aus 2 Gründen:

- **1. Organisation:**
  - Vererbung hilft uns bei der Erstellung von *Taxonomien* für Abstraktionen in der Applikationsdomäne
- **2. Wiederverwendbarkeit:**
  - Wir können Modelle und Code durch den Einsatz von Vererbung *besser wiederverwendbar* machen.
- Es gibt viele Arten von Vererbung, wir besprechen die wichtigsten Formen: Spezialisierung, Generalisierung, Spezifikationsvererbung und Implementationsvererbung
  - Auch für die Vererbung kann man eine Taxonomie erstellen
  - Wir benutzen ein Metamodell über die Vererbung. Hoffentlich erhöht dies das Verständnis!
- Fangen wir mit dem Einsatz von Vererbung in der Modellierung an.

Vererbung in der Modellierung

- **Ausgangssituation**:
  - In der Analyse erzeugen wir zunächst Anwendungsfalldiagramme.
  - Daraus finden wir Klassen ("Klassenidentifizierung").
  - Diese Klassen untersuchen wir dahingehend, in welcher Beziehung sie zu einander stehen ("Identifizierung von Beziehungen").
  - Diese Beziehungen sind entweder allgemeine Assoziationen, Aggregationen oder Vererbungen.
- Beim "Entdecken" von Vererbungsbeziehungen unterscheiden wir 2 Vorgehensweisen, die wir Spezialisierung und Generalisierung nennen.
  - Der Unterschied ist rein zeitlich:
    Wer wird zuerst entdeckt, die Oberklasse oder die Unterklasse?
- **Definition Generalisierung:** Das Entdecken einer Vererbungsbeziehung zwischen 2 Klassen, wobei die Unterklasse zuerst entdeckt wird.
- **Definition Spezialisierung:** Das Entdecken einer Vererbungsbeziehung zwischen 2 Klassen, wobei die Oberklasse zuerst entdeckt wird.

Metamodell für Vererbung



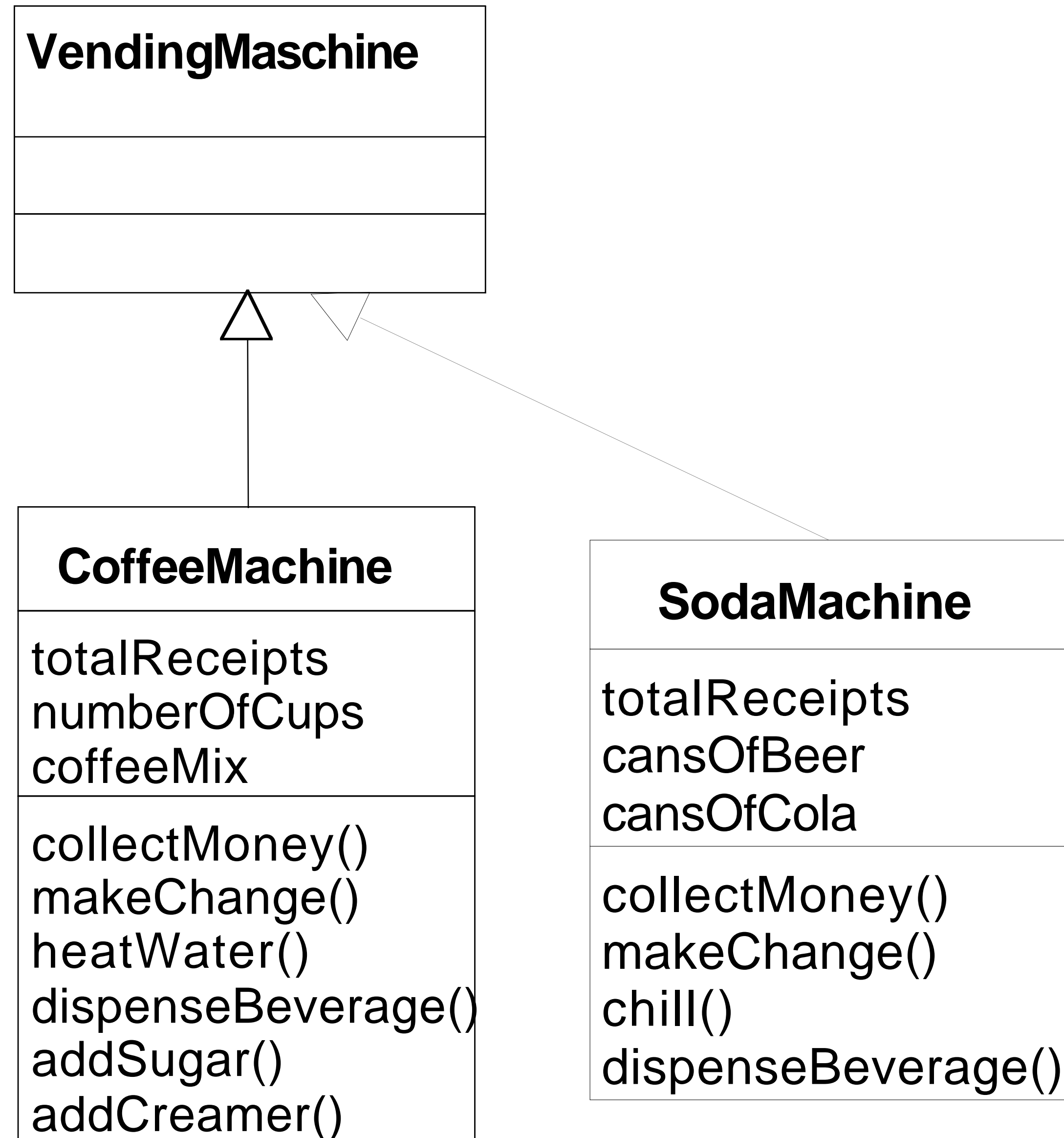
Generalisierung

- Die Unterklasse wird zuerst identifiziert, dann die Oberklasse
- Diese Art der Vererbung kommt oft bei induktiven Vorgehensweisen vor.
  - **Biologie**: Zunächst werden einzelne Tiere (Elefant, Löwe, Wal) entdeckt, dann das Konzept, dass Tiere gemeinsame Merkmale haben (Säugetiere).
- Oberklassen werden auch manchmal unbeabsichtigt gefunden:

Gibt es etwas, was Autos und Flugzeuge gemeinsam haben?

Copyright 2001 Bernd Brügge

Modellierung von Kaffeemaschinen



Generalisierung:

Die Klasse **`CoffeeMachine`** wird zuerst entdeckt, dann die Klasse **`SodaMachine`**, dann die Klasse **`VendingMachine`**

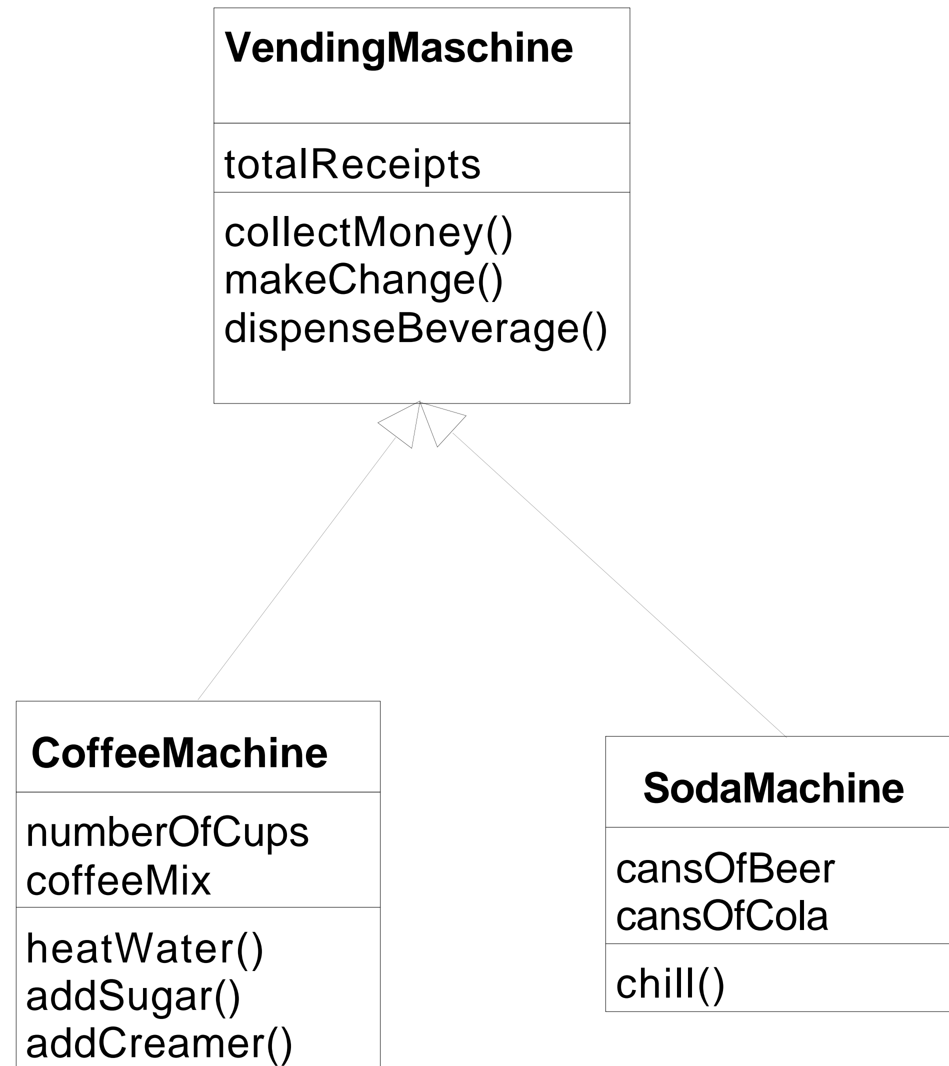
Nach der Generalisierung kommt die (Re-)Strukturierung:

Warum bringen wir

`totalReceipts`
`collectMoney()`
`makeChange()`
`dispenseBeverage()`

nicht in die Oberklasse?

Restrukturierung von Attributen und Operationen ist oft eine Folge der Generalisisierung

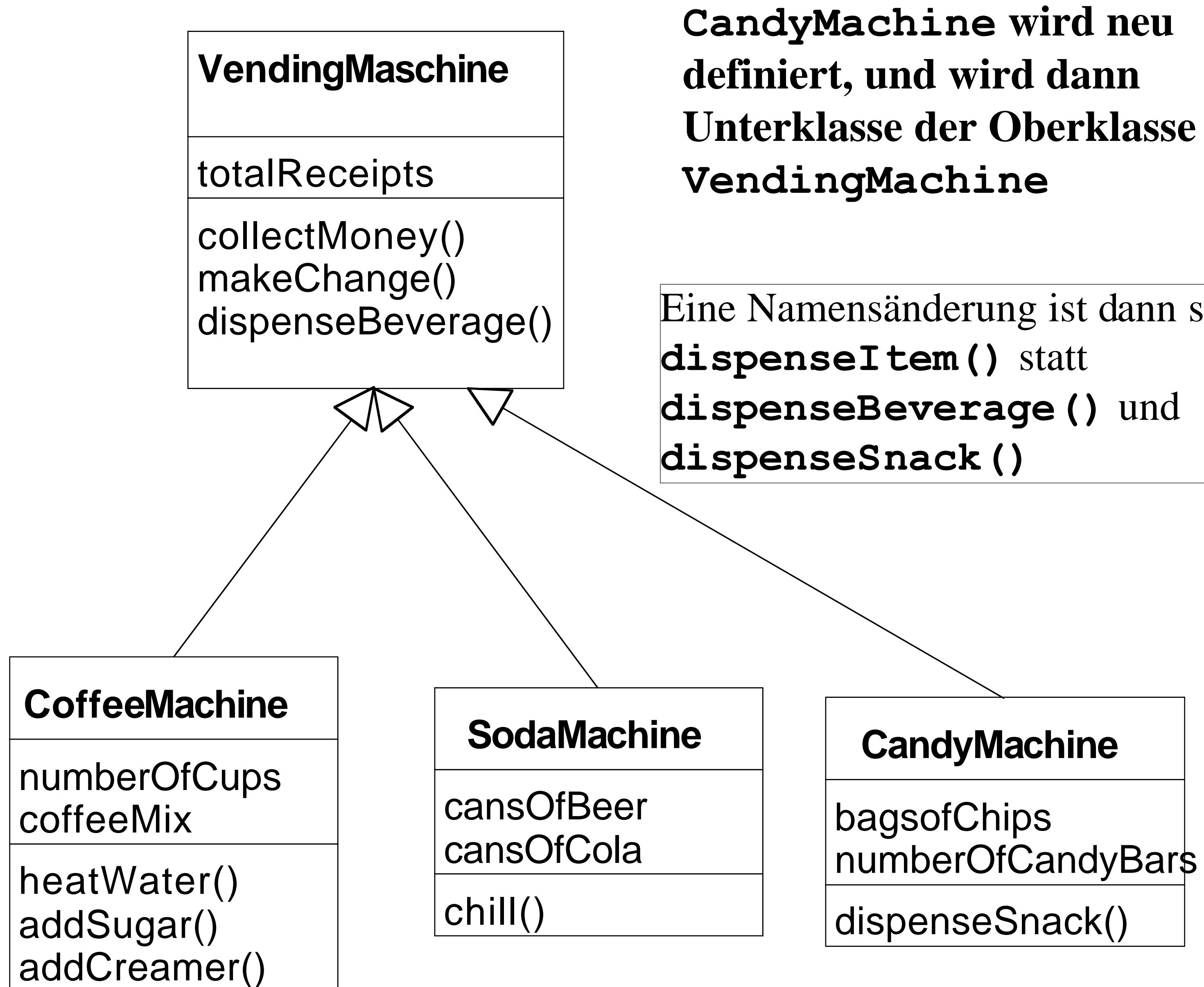


Spezialisierung

- Spezialisierung kommt vor, wenn wir sagen, dass eine Klasse so ähnlich ist wie eine andere Klasse.
- Spezialisierung kommt oft in experimentellen Wissenschaften vor, wenn eine Theorie formuliert wird, die bestimmte Ergebnisse postuliert.
  - **Beispiel Physik**: Die Quarks-Theorie wurde formuliert, und danach musste es bestimmte Teilchen geben. Dann wurden diese Teilchen gesucht (oder werden immer noch gesucht).
- **Fallbeispiel:**
  Wir haben im letzten Jahr ein Projekt abgeschlossen, in dem wir einen Kaffee- und einen Getränke-Automaten entwickelt haben, die beide Verkaufsautomaten waren.
  Im neuen Projekt sollen wir einen Süßigkeiten-Verkaufsautomaten entwickeln:
  "So ein Süßigkeiten-Automat muss möglich sein".

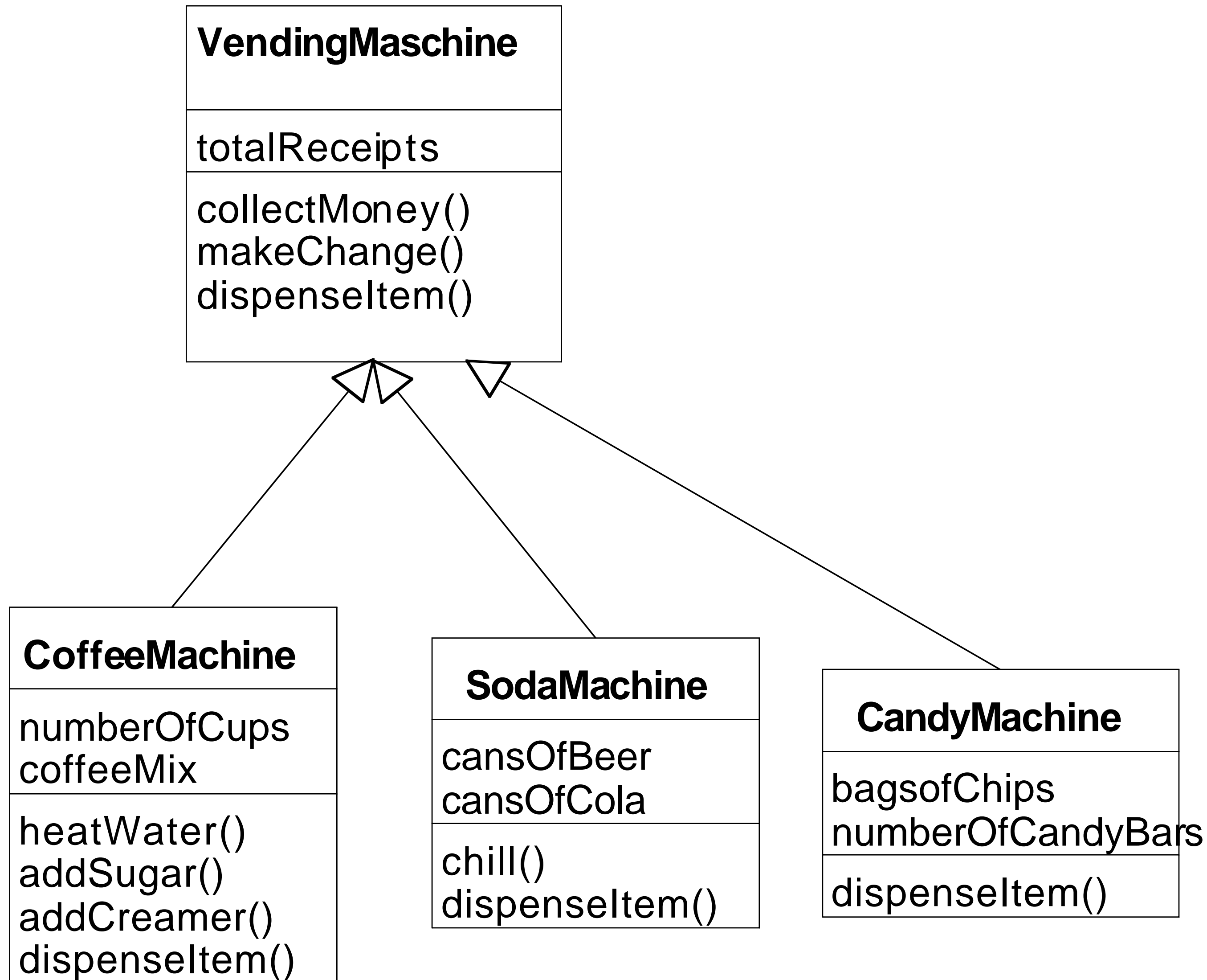
Beispiel einer Spezialisierung

`CandyMachine` wird neu definiert, und wird dann Unterklasse der Oberklasse `VendingMachine`



Eine Namensänderung ist dann sinnvoll:
`dispenseItem()` statt
`dispenseBeverage()` und
`dispenseSnack()`

Beispiel einer Spezialisierung (Fortsetzung)



Copyright 2001 Bernd Brügge

Beispiel für eine schlechte Art, mit Vererbung umzugehen

- Man kann die Operationen der Oberklasse mit völlig neuen Operationen überschreiben.

  Beispiel:

```
Public class SuperClass {
  public int add (int a, int b) { return a+b; }
  public int subtract (int a, int b) { return a-b; }
}

Public class SubClass extends SuperClass {
  public int add (int a, int b) { return a-b; }
  public int subtract (int a, int b) { return a+b; }
}
```

  Wir haben also die Addition als Substraktion redefiniert!
  Java verbietet derartige Konstruktionen nicht. Aber sie verletzen unser Konzept von Vererbung und Spezialisierung.

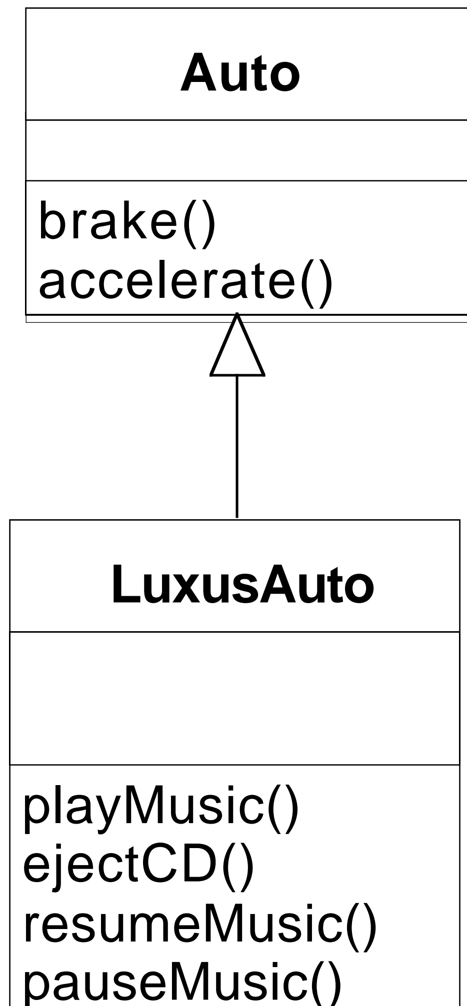
Substituierbarkeit

- **Dogma in Info II**: Eine Unterklasse darf die Bedeutung ihrer Oberklasse nicht verletzen.
  - Andere Formulierung (in der Vorlesung über Verträge): Eine Unterklasse darf den Kontrakt der Oberklasse nicht verletzen.
- **Liskov's Substitutionsprinzip:**
  - Eine Unterklasse muss substituierbar sein, d.h. eine Instanz der Unterklasse kann immer und überall dort eingesetzt werden, wo Instanzen der Oberklasse auftreten.
  - Eine Oberklasse kann prinzipiell immer durch ihre Unterklasse substituiert werden.
- Eine Klasse ist eine Menge von Objekten.
  Eine Unterklasse kann als *Teilmenge* der Oberklasse angesehen werden, wenn das Substitutionsprinzip gilt.

Einfache Vererbung (strict typing)

- **Definition Einfache Vererbung (auch strikte Vererbung genannt):**
  – Die Unterklasse überschreibt keine Operationen der Oberklasse, sondern fügt lediglich zusätzliche Operationen zu den Operationen der Oberklasse hinzu.
- Einfache Vererbung liegt vor, wenn die Unterklasse dem Liskov'schen Substitutionsprinzip gehorcht, und wenn die Methoden der Oberklasse nicht überschrieben werden dürfen.

Beispiel einer einfachen Vererbung



Oberklasse:

```
public class Auto {
   public void drive() {…}
   public void brake() {…}
   public void accelerate() {…}
}
```

Unterklasse:

```
public class LuxusAuto extends Auto {
   public void playMusic() {…}
   public void ejectCD() {…}
   public void resumeMusic() {…}
   public void pauseMusic() {…}
}
```

Vererbungen findet man nicht sofort

- Wenn Sie ein Klassendiagramm erstellen, versuchen Sie nicht sofort, Vererbung zu finden!
  - Das Entdecken von Vererbung bedingt ein sehr gutes Verstehen der Anwendungs- oder Lösungsdomäne.
  - Das Formulieren von Vererbungshierarchien in Modellen ist ein iterativer Vorgang.
  - Stellen Sie sich darauf ein, dass Sie mehrere Runden von Assoziationsentdeckungen durchmachen werden, wobei Sie mal Spezialisierungen und mal Generalisierungen entdecken werden.

Vererbung in Entwurf und Implementierung

- Mit Spezialisierung und Generalisierung kann man Klassen finden, wobei wir uns allerdings noch nicht festlegen, was die Klasse machen soll und wie sie realisiert wird.
- Manchmal wissen wir, was eine Unterklasse machen soll, aber wir haben keine Idee, wie sie es machen soll.
- Dafür führen wir 2 Begriffe ein.
  - **Definition Klassen-Spezifikation:** Beschreibung, *was* eine Klasse machen soll.
  - **Definition Klassen-Implementation:** Beschreibung, *wie* eine Klasse es macht.

Beispiel: Modellierung von Brettspielen

- Nehmen wir an, wir entwickeln ein Brettspiel. Wir können das allgemeine Verhalten eines Brettspieles so spezifizieren:

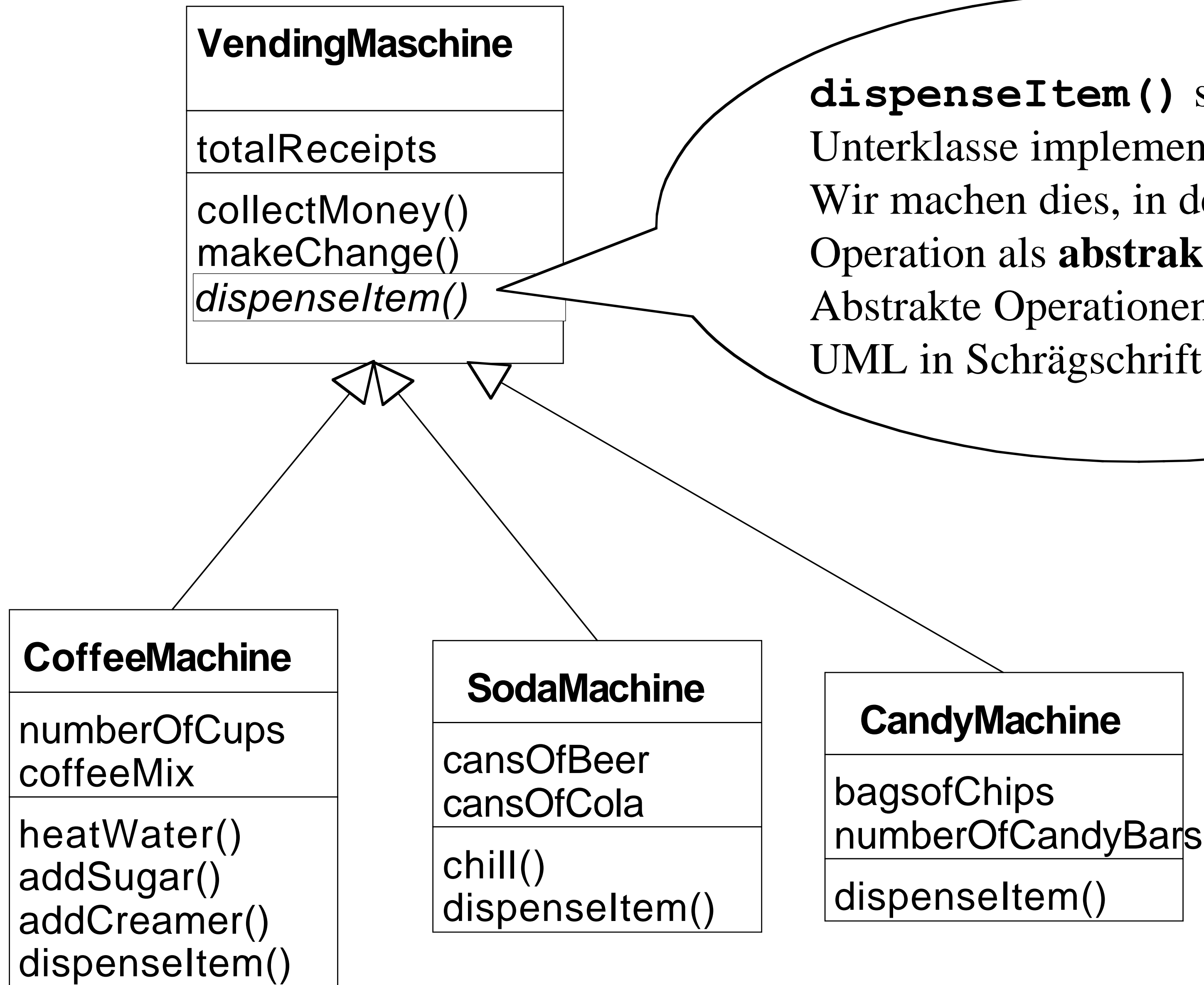
```
Public class BoardGame {
  public void play () {
    while (! finished()) {
      currentPlayer.move();
      currentPlayer = nextPlayer();
    }
  }
  ...
```

- Wir machen hier mehr als nur eine Spezifikation der Schnittstelle: Wir wissen, dass es 2 Spieler gibt. Jeder Spieler führt einen Zug aus, und wartet dann auf den anderen Spieler.
- **`BoardGame`** können wir als Oberklasse für Brettspiele wie Schach, Dame, Othello usw. benutzen. Alle Unterklassen müssen den Code für **`play()`** von **`BoardGame`** erben.

Ein Problem

- Wir wissen, daß jedes Objekt vom Typ **`BoardGame`** in der Lage sein muss, einen Zug auszuführen:
  - Das Spiel **Schach** muss eine öffentliche Operation **`move()`** anbieten.
  - Das Spiel **Mühle** muss eine öffentliche Operation **`move()`** anbieten.
  - …
- Jedes Spiel hat allerdings seine eigenen Regeln, wie dieser Zug auszuführen ist.
  - Im **Schach** wird ein Läufer anders bewegt als ein Stein in **Mühle**.
- Die Deklaration der Operation **`move()`** in der Oberklasse ist also als Bedingung zu sehen, dass alle Unterklassen die Operation **`move()`** erben müssen; aber jede Unterklasse implementiert diese Operation anders.

Ein weiteres Beispiel



`dispenseItem()` soll von jeder Unterklasse implementiert werden. Wir machen dies, in dem wir die Operation als **abstrakt** spezifizieren. Abstrakte Operationen werden in UML in Schrägschrift geschrieben

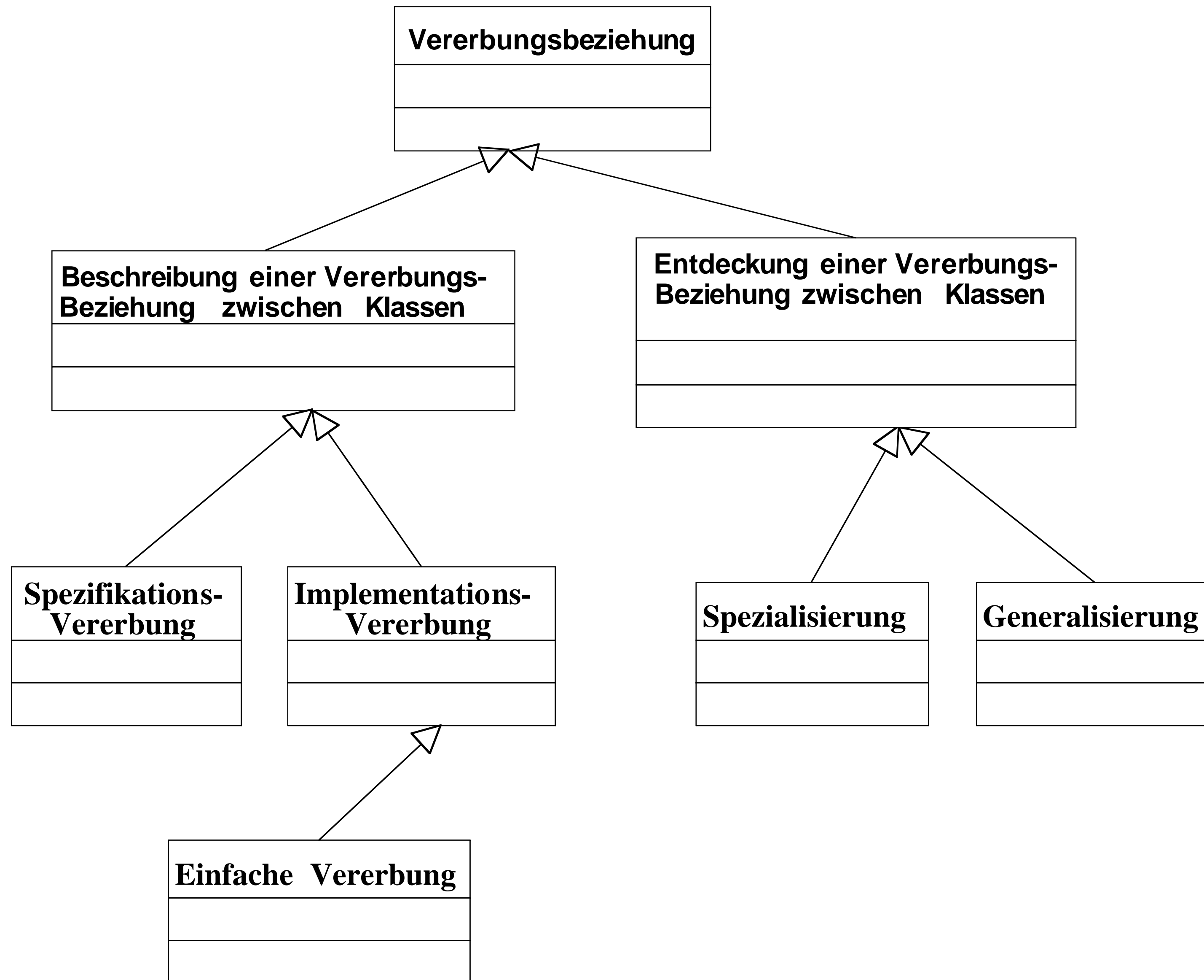
Abstrakte Methoden und Klassen

- **Wiederholung aus Info I:**
  - **Definition Abstrakte Methode:** Eine Methode ohne Implementierung (also ohne Methodenrumpf).
  - **Definition Abstrakte Klasse:** Eine Klasse, die nicht instanziert werden kann.
    Eine Klasse mit abstrakten Methoden ist automatisch abstrakt.
  - **Definition Schnittstelle:** Eine abstrakte Klasse, in der nur abstrakte Methoden deklariert sind (enthält weder Felder (Attribute) noch Methodenrümpfe)
    - Eine Schnittstelle wird primär zur Spezifikation verwendet. Die Implementation wird in den Unterklassen oder durch andere Mechanismen bereitgestellt.
- In UML werden abstrakte Klassen und Operationen in Kursivschrift notiert. Man kann auch die Einschränkung `{abstract}` verwenden (Sinnvoll vor allem beim Brainstorming).

Spezifikations- vs. Implementationsvererbung

- **Definition Spezifikationsvererbung:** Die Kombination von *Vererbung* und *Spezifikation*
  – Die Schnittstelle der Oberklasse wird vollständig vererbt.
  – Implementationen werden von der Unterklasse nicht geerbt.
- **Definition Implementationsvererbung:** Die Kombination von *Vererbung* und *Implementation*
  – Die Schnittstelle der Oberklasse wird vollständig vererbt.
  – Etwaige in der Oberklasse vorhandene Implementationen von Methoden ("Referenzimplementierungen") werden von der Unterklasse geerbt.

Metamodell für Vererbung



Kontraktion

- Kontraktion ist eine spezielle Form der Vererbung, die unbedingt zu vermeiden ist, aber oft während der Implementierungsphase benutzt wird:
- **Definition Kontraktion:** Von der Oberklasse geerbte Implementationen von Methoden werden in der Unterklasse mit leeren Methodenrümpfen überschrieben, um Operationen der Oberklasse "auszublenden".
- Warum ist Kontraktion so beliebt? Beispiel:
  - Wir haben eine Klasse **`Auto`** mit einem sehr guten Stereosystem modelliert, implementiert und geliefert.
  - Die Implementation war sehr zeitaufwendig.
  - Wir werden gebeten, schnell und billig eine Musikanlage **`BoomBox`** zu entwickeln, die über das WWW verkauft werden soll.
  - *Implementationsvorschlag* :-(
    - Wir nehmen den existierenden Code für **`Auto`**.
    - Wir verändern ihn aber nicht, um unnötige Übersetzungen zu vermeiden.
    - Stattdessen erzeugen wir eine Unterklasse **`BoomBox`** von **`Auto`**, und überschreiben alle Methoden von **`Auto`**, die nichts mit dem Spielen von Musik zu tun haben, in **`BoomBox`** durch Methoden mit leeren Rümpfen.

Was wir wollen

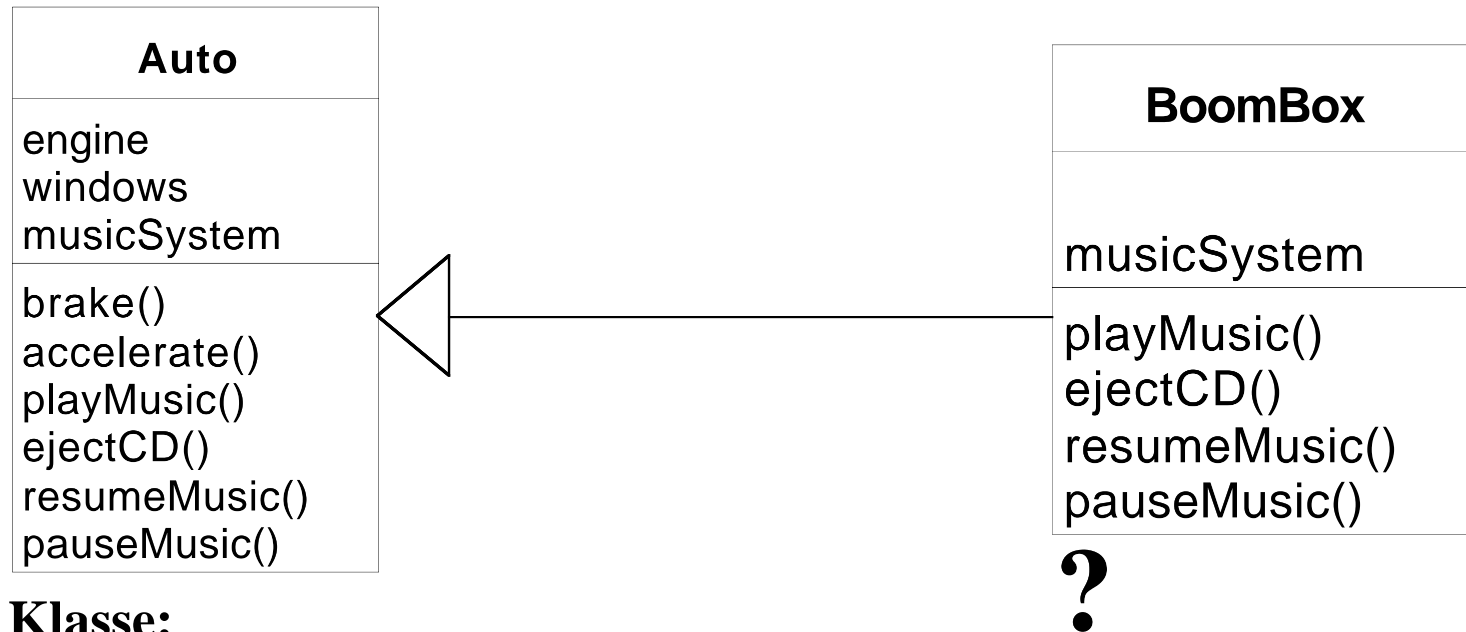
Auto
engine windows musicSystem
brake() accelerate() playMusic() ejectCD() resumeMusic() pauseMusic()

BoomBox
musicSystem
playMusic() ejectCD() resumeMusic() pauseMusic()

Neue Abstraktion!

Copyright 2001 Bernd Brügge

Was wir machen, um Zeit und Geld (?) zu sparen



Existierende Klasse:

```
public class Auto {
  public void drive() {…}
  public void brake() {…}
  public void accelerate() {…}
  public void playMusic() {…}
  public void ejectCD() {…}
  public void resumeMusic() {…}
  public void pauseMusic() {…}
}
```

`Boombox` als Kontraktion von `Auto`:

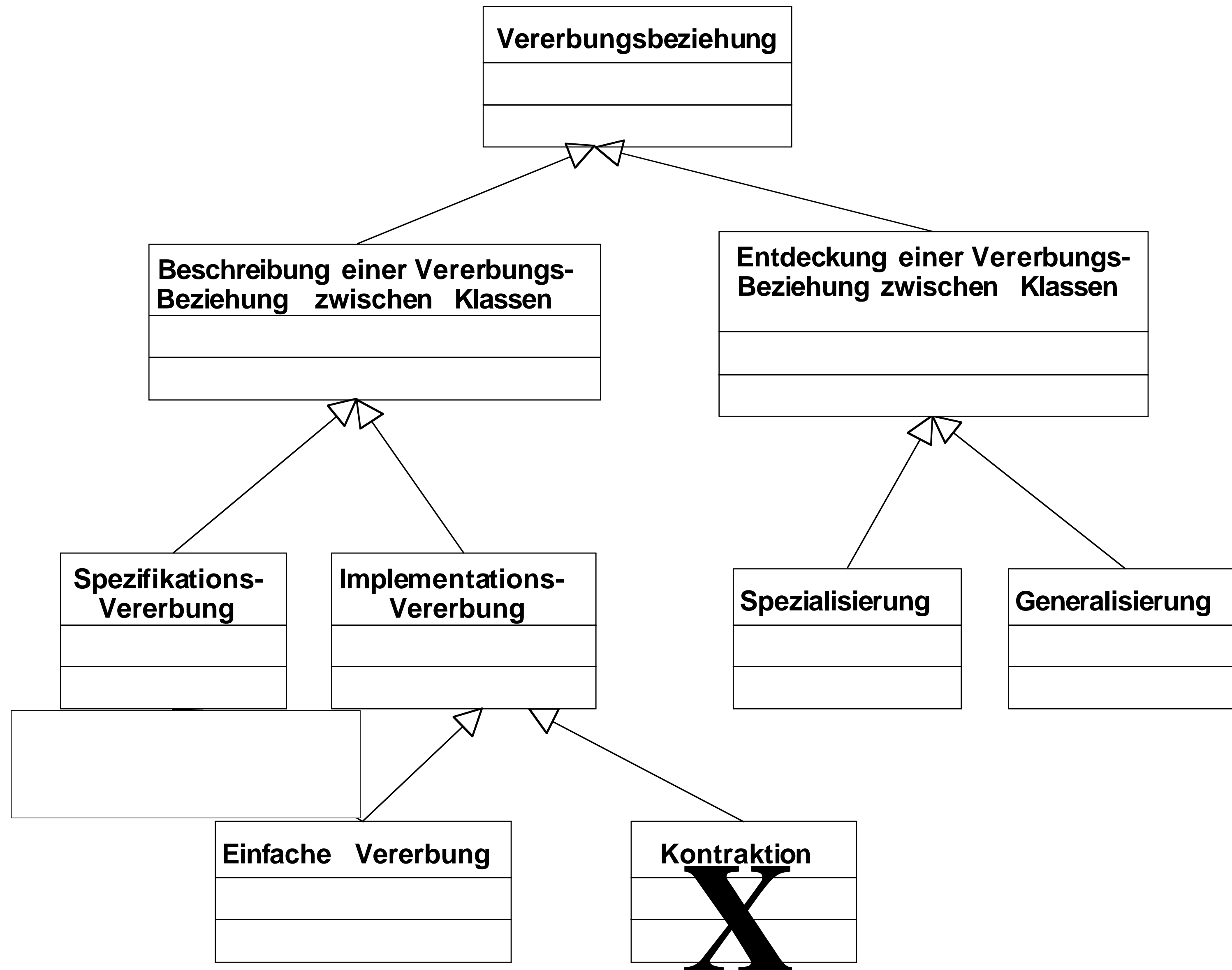
```
public class Boombox extends
  Auto {
  public void drive() {};
  public void brake() {};
  public void accelerate() {};
}
```

Copyright 2001 Bernd Brügge

Warum ist Kontraktion zu vermeiden?

- Eine durch Kontraktion erzeugte Unterklasse entspricht bzgl. Ihrer Funktionalität den Anforderungen, aber:
  - Die Schnittstelle der Unterklasse enthält überflüssige und für diese Klasse unsinnige Operationen.
    - im Beispiel:
      Welche Semantik hat die Operation **`brake()`** für eine **`BoomBox`**?
  - Die Unterklasse passt nicht in die Klassenhierarchie.
    - Im Beispiel:
      Eine **`BoomBox`** ist keine spezielle Form von **`Auto`**.
  - Die Unterklasse verletzt Liskov's Substitutionsprinzip.
    - Im Beispiel:
      Ich kann nicht mein **`Auto`** durch eine **`BoomBox`** ersetzen, um damit zur Arbeit zu fahren.

Vererbung als Anwendungsdomäne: Eine "Vererbungstaxonomie"



Java's Vererbungsmechanismen

- Java bietet mehrere Techniken an, um dem Programmierer bei der Implementierung der verschiedenen Typen von Vererbung zu helfen.
  1. Realisierung von *Spezialisierung* oder *Generalisierung*
    - **Definition von Unterklassen**
    - Schlüsselwort: **`extends`**

  2. Realisierung von *einfacher Vererbung (strikter Vererbung)*
    - **Verbieten des Überschreibens von Methoden**
    - Schlüsselwort: **`final`**

  3. Realisierung von *Implementationsvererbung*
    - **Überschreibbare Methoden**
    - Schlüsselwort: - (Überschreibbare Methoden sind Default in Java)

  4. Realisierung von *Schnittstellenvererbung*
    - **Abstrakte Methoden/Klassen, Spezifikation einer Schnittstelle**
    - Schlüsselworte: **`abstract`**, **`interface`**

Realisierung von Vererbung in Java

- **Ziel**: Wir wollen Vererbung aus dem UML-Modell in Java-Code umsetzen.
- **Frage:** Welche Mechanismen aus der Programmiersprache benutzen wir?
- In Java gibt es folgende Mechanismen:
  - Überschreibbare Methoden (Voreinstellung in Java)
  - Finale Klassen
  - Finale Methoden
  - Abstrakte Methoden
  - Abstrakte Klassen
  - Schnittstellen
- Für jede dieser Mechanismen kann man Heuristiken aufstellen, um sie sinnvoll zu benutzen.

Heuristik: Einsatz von überschreibbaren Methoden

- **Überschreibbare Methoden:** *Eine Methode einer Oberklasse soll so implementiert werden, dass sie das Normalverhalten (default behavior) für die Unterklassen beschreibt.*
- Wir unterscheiden 2 Fälle:

1. Mehr als eine Unterklasse benutzt das Normalverhalten

– Wenn nur eine Unterklasse diese Methode benutzt, dann ist es keine überschreibbare Oberklassenmethode und sollte daher `abstract` sein

2. Objekte der Unterklassen führen die Operation auf verschiedene Arten aus, haben dabei allerdings auch gemeinsames Verhalten

– Mehrere Unterklassen (aber nicht alle) können die Operation unverändert benutzen

⇒ Die Methode muss überschreibbar sein, um Änderungen zu ermöglichen.

– Die Unterklassen verändern die Methode, haben aber auch gemeinsames Verhalten

⇒ Das gemeinsame Verhalten sollte in eine zusätzliche, (mit `protected`) geschützte Methode ("Hilfsmethode" in der Oberklasse) extrahiert werden. Die verbleibende Methode sollte dann `abstract` deklariert werden.

– Wenn man das nicht macht, werden Klassen-Erweiterer schnell vergessen, dass sie die Methode überschreiben müssen.

Heuristiken: Einsatz von finalen Klassen und Methoden

- **Finale Klassen:** *Die Klasse soll nicht erweiterbar sein.*
  – Wir können die Erweiterung verbieten, indem wir die gesamte Klasse als **`final`** markieren.
- **Finale Methoden:** *Wir wollen nur die einfache Vererbung zwischen 2 Klassen modellieren.*
  – Alle Methoden, die mit **`final`** markiert sind, repräsentieren unveränderliche Implementationen, die für alle Unterklassen gültig sein sollen.
  Wenn wir eine Methode also nicht **`final`** markieren, sagen wir gewissermaßen, dass das Verhalten der Methode nicht unveränderlich ist.

Beispiele für finale Methoden und Klassen

- Beispiele für finale Methoden:
  - Klassenmethoden (**`static`**) und private (**`private`**) Methoden sind finale Methoden, können also nicht überschrieben werden.
- Viele Basis-Klassen von Java sind finale Klassen, z.B.:
  - **`System`**: Definiert Klassenvariablen **`in`** und **`out`** (Ein-/Ausgabe).
  - **`Boolean`**, **`Byte`**, **`Character`**, **`Short`**, **`Integer`**, **`Long`**, **`Float`**, **`Double`**: Konvertierung von Grundtyp-Werten in Objekte
  - **`String`**: Zeichenketten
  - **`Math`**: Definiert mathematische Operationen wie **`abs()`**, **`sin()`**, **`cos()`**, **`log()`**, **`max()`**, **`min()`**, **`pow()`**, **`round()`**, ...
- Warum brauchen wir finale Methoden und Klassen?
  - Schutz gegen Missbrauch (durch Überschreiben)
  - Der Compiler kann Optimierungen durchführen, um schnelleren Code zu erzeugen → Vorlesung Compilerbau (Hauptstudium)

Heuristik: Einsatz von abstrakten Methoden

- **Abstrakte Methoden:** *Alle Unterklassen müssen dieselbe Operation implementieren.*
- 2 Gründe:
  1. Wir schreiben Code für Objekte vom Typ der Oberklasse, aber wir benutzen nur Objekte vom Typ der Unterklassen.
     - Beispiel: Eine Oberklasse **`Brettspiel`** wird mit einer Operation **`move()`** geschrieben, aber **`Brettspiel`** hat keine vordefinierte Methode, die dieses Verhalten implementiert. Diese wird in der Unterklasse (**`Schach`**, usw) implementiert.
  2. Wir wollen die Objekte der Unterklassen zwingen, ein einheitliches Verhalten zu zeigen.

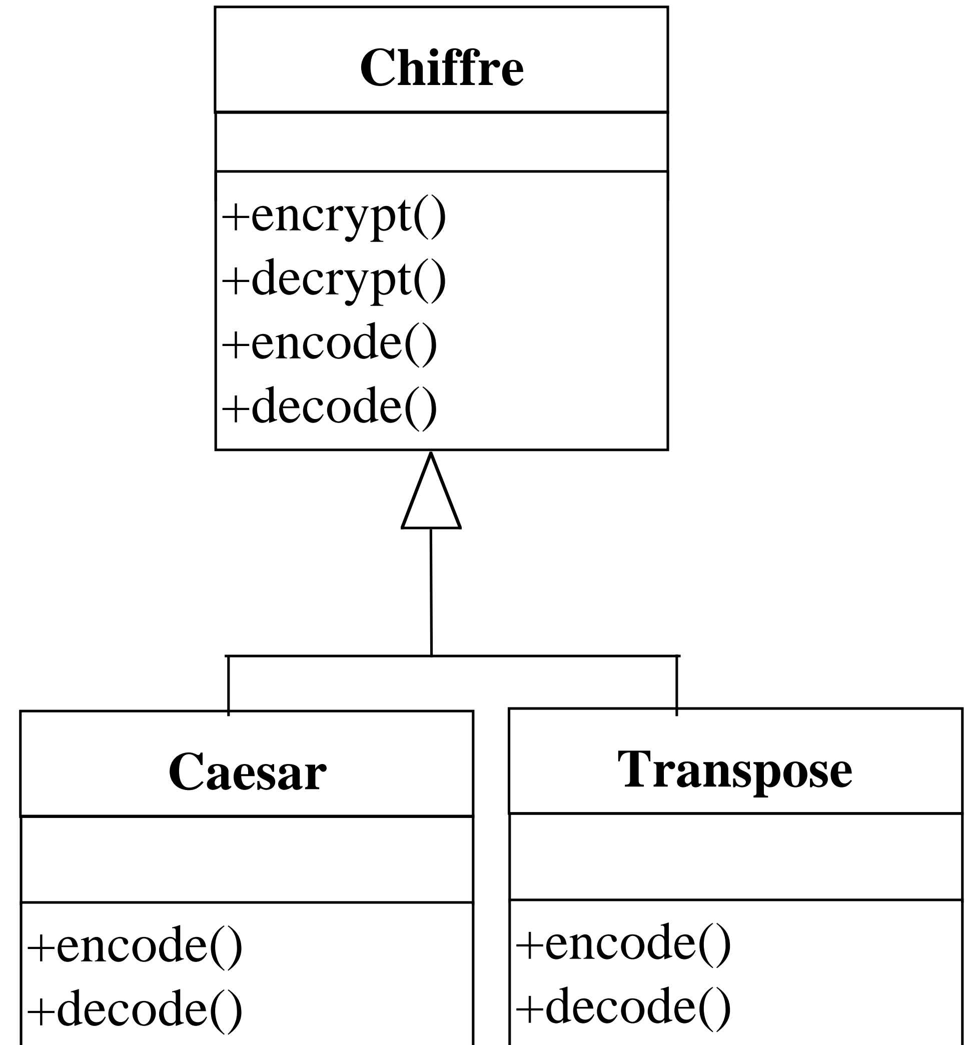
     Beispiel:
     - Initalisierung: "Alle Unterklassen müssen eine Methode **`init()`** haben!"
     - Terminierung: "Alle Objekte müssen nach Aufruf der Methode **`term()`** in einem wohldefinierten Zustand sein!"

Beispiel für die Anwendung von abstrakten Methoden: Kryptographie (siehe Info I - Vorlesung 14)

- Problem: Bereitstellung eines allgemeinen Verschlüsselungsverfahren.
- Anforderungen:
  – Das System stellt mindestens Algorithmen für Caesar- und Transpositionsverschlüsselung bereit.
  – Neue Algorithmen können zur Laufzeit eingebunden werden, ohne dass der Benutzer sein Program erneut compilieren muss.
  – Die Auswahl des besten Algorithmus geschieht zur Laufzeit.

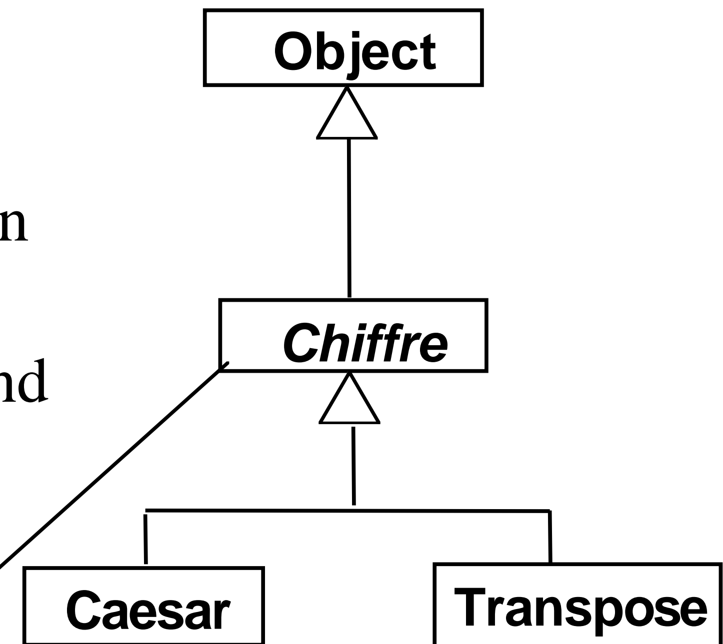
Beispiel: Detaillierter Entwurf

- Wir definieren **`Chiffre`** als Oberklasse, und die einzelnen Chiffrierverfahren als Unterklassen von **`Chiffre`**.
  `Chiffre` bietet 4 Methoden an:
  - **`encrypt()`** verschlüsselt einen Text von Worten.
  - **`decrypt()`** entschlüsselt einen Text von Worten.
  - **`encode()`** wendet einen spezifischen Algorithmus zur Verschlüsselung eines Wortes an.
  - **`decode()`** wendet einen spezifischen Algorithmus zur Entschlüsselung eines Wortes an.



Beispiel: Implementation in Java

- Die Methoden **`encrypt()`** und **`decrypt()`** sind dieselben für jede Unterklasse und können deshalb in der Oberklasse **`Chiffre`** *implementiert* werden.
  - **`Chiffre`** definieren wir als Unterklasse von **`Object`**, weil wir einige Methoden von **`Object`** benötigen.
- Die Methoden **`encode()`** und **`decode()`** sind für jede Unterklasse spezifisch. Sie werden deshalb als *abstrakte Methoden* in der Oberklasse definiert, und dann in den Unterklassen *implementiert.*



Die **`Chiffre`**-Klasse ist abstrakt, weil einige ihrer Methoden noch nicht implementiert sind.

Heuristik: Einsatz von Abstrakten Klassen

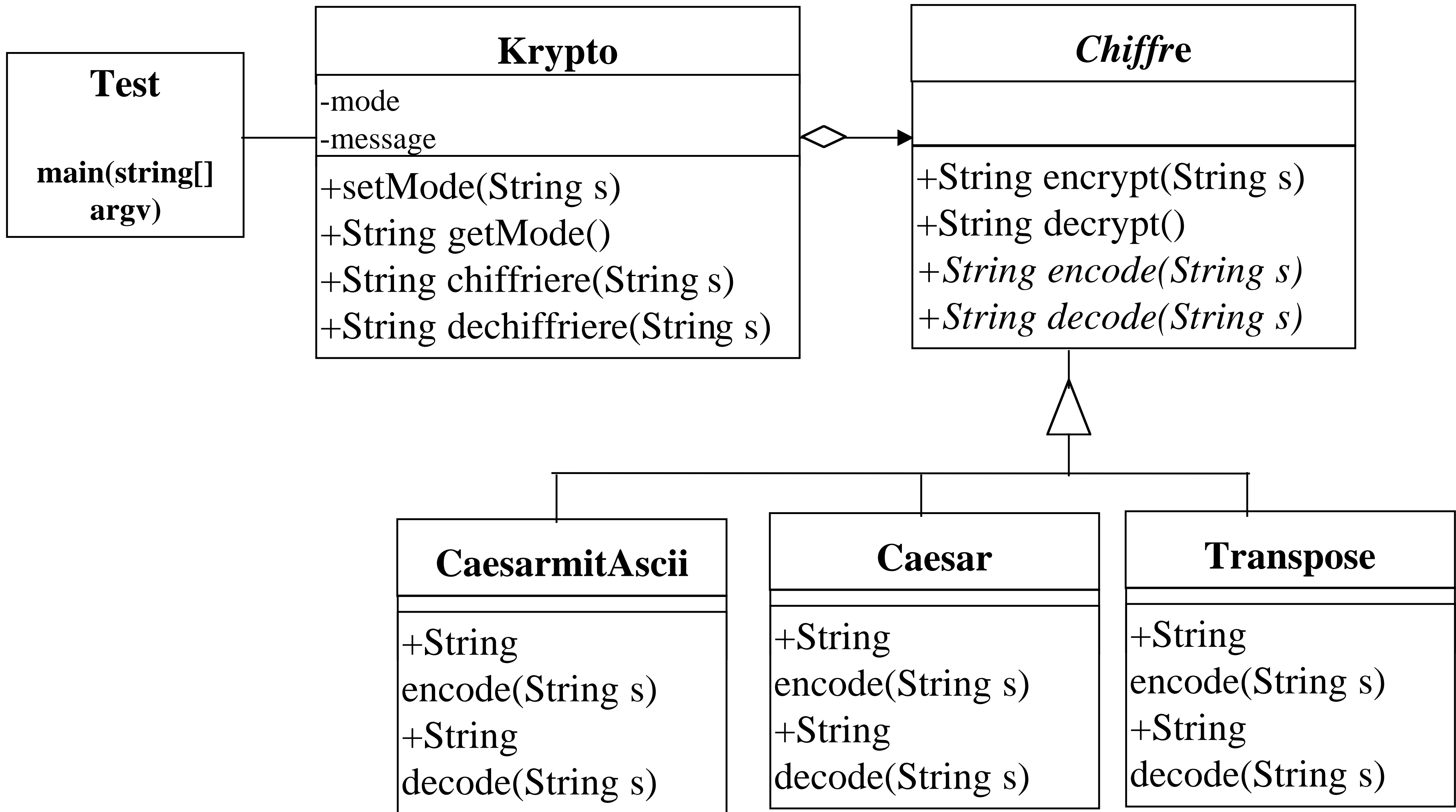
- **Definition Reine Abstrakte Klasse:** Eine Klasse, die nur abstrakte Methoden hat.
- Die Oberklasse hat ein Attribut, das in allen Unterklassen verwendet wird; allerdings kann man noch keine Methoden implementieren, denn die Implementation ist Unterklassen-spezifisch.
  - In diesem Fall benutzen wir eine reine abstrakte Klasse.
  - Wenn die Oberklasse kein gemeinsames Attribut hat, benutzen wir eine Schnittstelle.

Heuristik für den Einsatz von Schnittstellen

Eine Schnittstelle (interface) benutzen wir aus folgenden Gründen :

1. Wir entwerfen eine Anzahl von Klassen, die ein gemeinsames Verhalten haben sollen, aber sonst in keiner Beziehung zueinander stehen.
   – Beispiel: Eine Klasse **`Window`**, die plattformunabhängig (d.h. mit unterschiedlichen Betriebssystemen) verwendbar sein soll.
2. Die Schnittstelle aus dem Entwurfsmodell soll auch im Programm von ihrer Implementation getrennt werden.
   – Beispiel: Ein Studentenverzeichniss soll als Baum, Liste oder Reihung implementierbar sein.
3. Eine Unterklasse soll Merkmale (Verhalten und Attribute) von mehr als einer Oberklasse erben.
   – Java unterstützt keine Mehrfachvererbung. Aber man kann sie sie mit dem **`interface`**-Mechanismus implementieren.

Beispiel für die Trennung zwischen Schnittstelle und Implementation (siehe Info I - Vorlesung 14)



Wo stehen wir?

- 3 wichtige Konzepte der Objekt-Orientierung
  - √ Vererbung
  - ➠ Dynamische Bindung
  - – Polymorphismus
- Anwendung der Konzepte:
  - – Generische Klassen
  - – Entwurfsmuster (Brückenmuster, Strategiemuster)

Wiederholung: Bindung und Bindungsbereich

- **Definition Bindung:** Sei x ein beliebiger Bezeichner (Variable, Konstante), t ein Typ und E ein Ausdruck dieses Typs. Dann heißt
  - t x = E;

  die **Deklaration des Bezeichners** x.
  Durch die Deklaration ist x an den Wert von E **gebunden**.
- **Definition Bindungsbereich:** Eine Bindung bezieht sich immer auf einen Bereich, den sogenannten Bindungsbereich, in Java durch geschweifte Klammern "**{**" und "**}**" gekennzeichnet.
- **Beispiele von Bindungsbereichen:**
  - Formaler Parameter: Methodenrumpf
  - Lokale Variable: Methodenrumpf ab Stelle der Definition
  - Instanz- und Klassenvariable: Innerhalb des Klassenrumpfes
  - Zählvariable: Schleifenkörper

Bindungsbereiche in Java

```
public class ZeichenbaresRechteck extends Rechteck
   implements  Zeichenbar {
   private Farbe f;     ...

   …
  public void setFarbe(Farbe f)
  {
   ….
  }

  private foo () {
    for (int f = 0; f < 1000; f++) {
     ….
    }
  }
….
} //ZeichenbaresRechteck
```

Bindungsbereich der Instanzvariable f

Bindungsbereich des formalen Parameters f

Bindungsbereich der Schleifenvariable f

Lebensdauer und Gültigkeitsbereich einer Bindung

- In einem Java-Programm kann ein Bezeichner x mehrfach vereinbart werden, gebunden sein, und mit verschiedenen Werten belegt sein. Wir unterscheiden deshalb zwischen der Lebensdauer und dem Gültigkeitsbereich einer Bindung.
- **Lebensdauer einer Bindung**:
  - Der Bindungsbereich des Bezeichners.
- **Gültigkeitsbereich einer Bindung (scope)**:
  - Die Lebensdauer eines Bezeichners x abzüglich aller Bindungsbereiche, in denen der Bezeichner erneut gebunden wird.

Statische vs Dynamische Bindung

- **Definition Statische Bindung:** Die Bindung zwischen einem Bezeichner x und dem im zugeordneten Wert wird zur Compilationszeit festgelegt und kann zur Laufzeit nicht verändert werden.

- **Definition Dynamische Bindung:** Die Bindung zwischen einem Bezeichner x und dem ihm zugeordneten Wert wird erst zur Laufzeit festgelegt und kann jederzeit verändert werden.

Bindung bei Operationen

- In objekt-orientierten Modellierungs- und Programmiersprachen interessieren uns besonders Bindungen, bei denen der Bezeichner x eine Signatur oder ein Methodenaufruf ist.

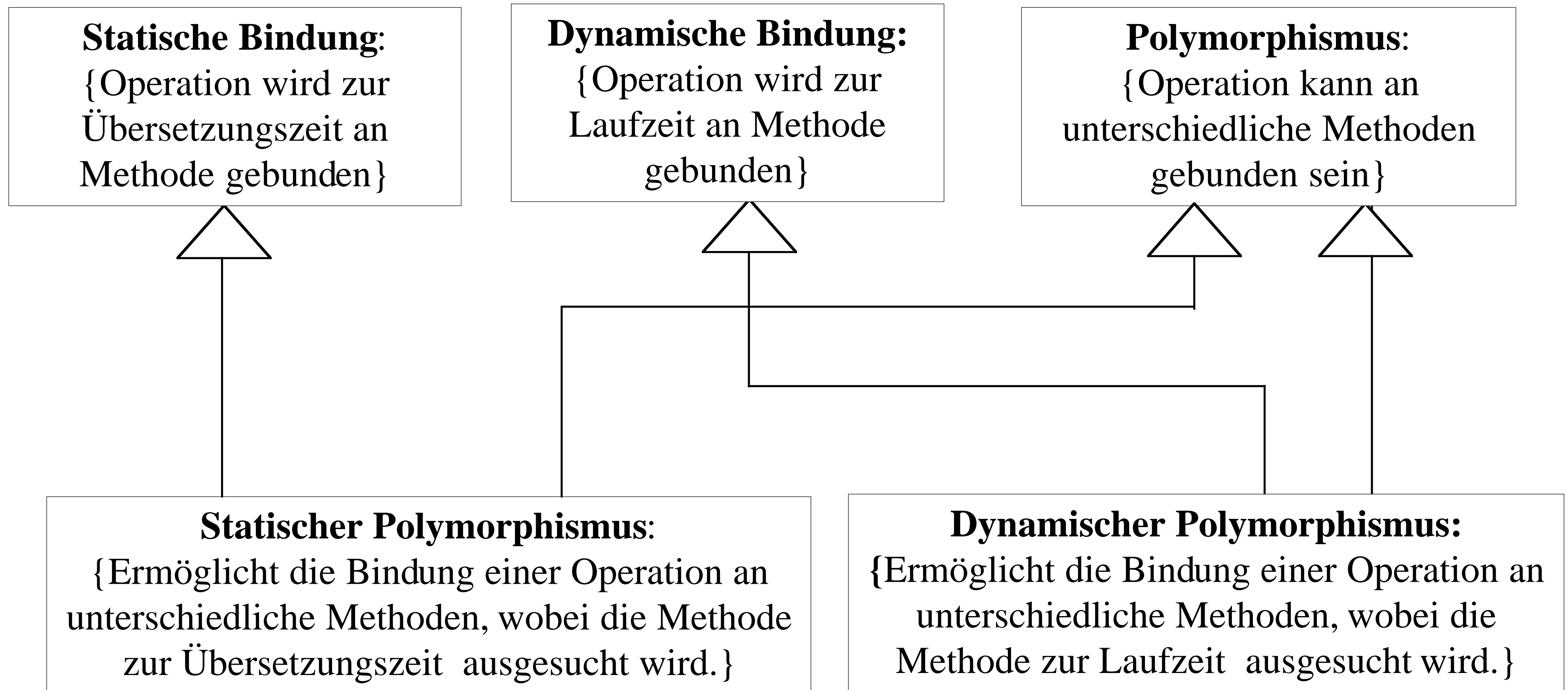
- **(Zur Wiederholung) Definition Operation:** Deklaration oder Aufruf eines Algorithmus.
Synonyme: Methodendeklaration bzw. Methodenaufruf
- **Definition Methode:** Implementation eines Algorithmus.
Synonym: Methodenrumpf

- **Statische Bindung**: Bindet einen Methodenaufruf (Operation) *zur Übersetzungszeit (compile time)* an *einen* Methodenrumpf.
- **Dynamische Bindung:** Bindet einen Methodenaufruf (Operation) *zur Laufzeit (run time)* an *einen* Methodenrumpf.

Polymorphismus

- **Polymorphismus**: Ermöglicht die Bindung einer Operation an *unterschiedliche* Methoden
- **Statischer Polymorphismus:** Ermöglicht die Bindung einer Operation an *unterschiedliche* Methoden, wobei die Methode allerdings *zur Übersetzungszeit* ausgesucht wird.
- **Dynamischer Polymorphismus:** Ermöglicht die Bindung einer Operation an *unterschiedliche* Methoden, wobei die Methode erst *zur Laufzeit* ausgesucht wird.

Modellierung von Bindung und Polymorphismus



Wann setzen wir Polymorphie ein?

- **Warum Polymorphismus?**
  - **=> Erlaubt Flexibilität und Allgemeinheit der Lösung**

- **Drei Einsatzgebiete:**
  - **1. Trennung zwischen Gebrauch und Realisierung**
  - **2. Einheitlicher Gebrauch von verschiedenen Objekten**
  - **3. Erhöhung der Wiederverwendbarkeit**

- Ziel 1 kann man mit statischem Polymorphismus realisieren
- Ziele 2 und 3 lassen sich nur mit dynamischen Polymorphismus realisieren.

Wann setzen wir Polymorphie ein?

- **Einsatzgebiet 1: Trennung zwischen Gebrauch und Realisierung:**
  - Sobald die Schnittstelle oder die reine abstrakte Klasse festgelegt ist, können wir uns den Gebrauch der Klasse überlegen (d.h. wir können die Implementationsrolle "Klassen-Benutzer" spielen).
  - Die Implementation kann dann in Form einer konkreten Klasse nachgeliefert werden (Rolle: "Klassen-Implementierer").
- **Einsatzgebiet 2: Einheitlicher Gebrauch verschiedener Objekte.**
  - Unterschiedliche Objekte haben dieselbe Schnittstelle, die Auswahl, welches Objekt verwendet wird, kann sich ändern. Beispiel: Drucken auf verschiedenen Druckern (Schwarz-Weiß, Farbe)
- **Einsatzgebiet 3: Erhöhung der Wiederverwendbarkeit**:
  - Wir ersetzen in einem System ein altes Stück Code durch neuen Code. Ist der neue Code in einer neuen Unterklasse und bezieht sich der Rest des Systems auf die unveränderte abstrakte Klasse oder Schnittstellenklasse, muss dieser Rest nicht recompiliert werden.

Beispiel: Einheitlicher Gebrauch von verschiedenen Objekten

```
public class Test {
 public static void main(String[] argv) {
  Chiffre  message = new Caesar();
  …...
  String secret = message.encrypt(plain);
  …….
  System.out.println(message.decrypt(secret));

   messsage = new Transpose();
   …...
   secret = message.encrypt(plain);
   …...
  System.out.println(message.decrypt(secret));
 } // main()
} // Test
```

Geheimnisprinzip

- Der eben skizzierte Einsatz von Polymorphie verlangt, dass wir das sogenannte **Geheimnisprinzip** beachten.
- **Definition Geheimnisprinzip:** Die Schnittstelle eines Subsystems/Klasse ist so gekapselt, dass die Umgebung in keiner Form - weder beabsichtigt noch unbeabsichtigt - die korrekte Arbeitsweise des Subsystems oder der Klasse stören kann.
- Ein kooperierendes Objekt hat außer den Kenntnissen über die Schnittstellenklasse keine weiteren Kentnisse über seinen Kooperationspartner.

Was macht eine gute Klasse aus?

- **Anforderungen an die Eigenschaften eines Subsystems (hier benutzen wir Subsystem synonym mit Klasse):**
  - ***Rekursivität (recursion)**:* Ein Subsystem kann sich wiederum aus Subsystemen zusammensetzen.
  - ***Unterscheidung Schnittstelle/Implementierung (separation of interface and implementation):*** Die Schnittstelle ist die Außenansicht eines Subsystems. Sie kann auf verschiedene Arten implementiert sein.
  - ***Geheimnisprinzip (information hiding):*** Die Einzelheiten der Implementierung einer Schnittstelle sind so gekapselt, dass die Umgebung weder beabsichtigt noch unbeabsichtigt die korrekte Arbeitsweise des Subsystems stören kann.
  - ***Zusammengehörigkeit (cohesion):*** Die Funktionen der Schnittstelle bilden ein logisch zusammenhängendes und vollständiges Ganzes.

Wo stehen wir?

Wichtige Konzepte der Objekt-Orientierten Programmierung:

✓ **Vererbung**

✓ **Dynamische Bindung**

✓ **Polymorphismus**

Anwendung der Konzepte auf

✴ **Generische Klassen**

✴ **Entwurfsmuster (Brücken-Muster, Strategie-Muster**)

➠ Wir wenden uns jetzt zunächst der Anwendung der Konzepte auf **Generische Klassen** zu.

Motivation von Generischen Klassen (Modellierung)

- Bei Modellierung von Wirklichkeiten gibt es oft ein Objekt, das mit vielen anderen in Beziehung steht. Wir benutzen dann eine Assoziation mit Multiplizitäten 1-*, um diese Beziehung darzustellen
  - Ein Auftrag besteht aus 100 Auftragspositionen
  - Das Studentenverzeichnis für Info II besteht aus 900 Studenten
- Bei der Implementierung übersetzen wir diese 1-* Multiplizität in eine Instanzvariable, wobei wir eine Datenstruktur wie Reihung, Liste oder Baum deklarieren und den Elementtyp festlegen.
  - `auftragsposition[] array = new auftragsposition[100];`
  - `int[] array = new int[900];`
- Die Implementation der Klasse ist dann oft spezifisch auf den Elementtyp zugeschnitten. Sobald wir einen anderen Elementtyp benötigen (float, bool,…) müssen wir sehr viel neuen Code schreiben.

Copyright 2001 Bernd Brügge

Motivation von Generischen Klassen (Datenstrukturen, Algorithmen)

- In Info I, Vorlesung 13 hatten wir Datenstrukturen wie Listen und Bäume benutzt, deren Knoten nur applikationspezifische Daten vom Typ `int` speichern konnten.
- Was uns auch interessierte, war die Frage, ob wir derartige Datenstrukturen auch für allgemeinere Elementtypen, nämlich Klassen als Abstraktionen aus der Applikationsdomäne (Personen, Autoteile, Flugzeugreservierungen, …) nehmen können.
- Und zwar so, dass wir den Code nur einmal schreiben müssen:
  – Man möchte also ein und denselben Code für Daten verschiedener Typen verwenden.
- Das machen wir mit dem Konzept der generischen Klasse.
  –

Beispiel: Bubblesort

Sortieren von Zahlen vom Typ int:

```
public void bubbleSort(int [] v) {
    for (int pass = 1; pass < v.length; pass++)
      for (int pair = 1; pair < v.length; pair++)
          if (v[pair-1] > v[pair])
              swap(pair-1, parr);
```

Sortieren von Elementen beliebigen Typs:

```
public void bubbleSort(Object [] v) {
     for (int pass = 1; pass < v.length; pass++)
       for (int pair = 1; pair < v.length; pair++)
          if v[pair-1].vergleiche(v[pair])
              swap(pair-1, pair);
```

Generischer Vergleichsoperator (Wichtig: Diese Implementation ist inkorrekt, da `vergleiche`() noch nicht definiert ist!)

Definitionen

- **Definition Generische Klasse:** Eine Klasse, die die Gemeinsamkeiten von Datenstrukturen und/oder Algorithmen über mehreren Klassen charakterisiert. Eine generische Klasse ist ein *generischer Datentyp*. Über Parametrisierung werden die in der Klasse verarbeitbaren Typen festgelegt.
- **Definition Generischer Datentyp:** Ein strukturierter Typ (z.b. Liste, Baum, Reihung, Keller), dessen Elementtyp noch nicht definiert ist.
  - **Beispiel:** .Ein Keller von Elementen, wobei der Typ der Elemente nicht festgelegt ist (**`int`**, **`bool`**, **`Student`**, **`Auftrag`**)
- **Definition Generische Methode:** Eine Menge von Methoden, die sich nur durch einen Aspekt ("Parameter") unterscheiden:
  - Durch den **Elementtyp**, auf dem sie arbeiten (z.B. **`int`**, **`student`**)
  - Durch eine bestimmte **Berechnung**, die in jeder Methode anders ausgeführt wird (z.B. die Berechnung der Summe, des Maximums oder des Minimums bei einer Reihung von **`int`**-Zahlen).

Generische Klassen, Methoden und Datentypen

- Generische Klassen erlauben uns, das Navigieren in höheren Datenstrukturen wie Reihung, Liste, Baum unabhängig vom Typ der gespeicherten Daten zu behandeln.
  - Andere Möglichkeit: Besucher-Muster (visitor pattern) (wird nicht in Info II behandelt).
- Generische Methoden und Generische Datentypen gehören zusammen:
  - Die generischen Methoden arbeiten auf allen Instanzen von generischen Datenstrukturen, müssen allerdings oft zur Übersetzungszeit oder Laufzeit spezialisiert werden.
    - Dies wird durch eine Kombination von Vererbung und Polymorphismus erreicht.

Umsetzung generischer Klassen in Java

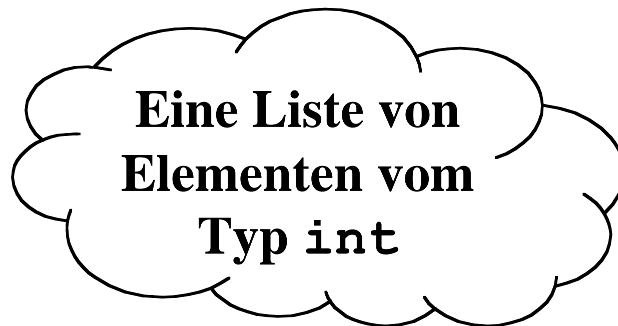
- Generische Klassen erlauben einfache Wiederverwendung von (typunabhängigen) Code (z.B. Listen-Klasse)
- **Problem**: Java unterstützt die Erstellung generischer Klassen nicht
- **Frage**: Wie kann man Algorithmen und Datenstrukturen in Java so implementieren, dass sie trotzdem für verschiedene Datentypen funktionieren?
- **Antwort**: Implementierung einer allgemeinen Lösung für eine Oberklasse und Wiederverwendung dieser Lösung für alle Unterklassen.

Generische Datentypen in Java

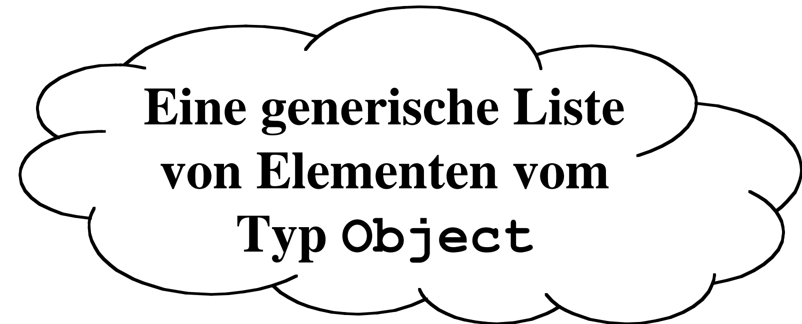
- In Java ist jede Klasse automatisch eine Unterklasse der Klasse **`Object`**, ohne dass diese mit dem Schlüsselwort **`extends`** gekennzeichnet werden muss.
- Ist ein formaler Parameter einer Methode vom Typ **`Object`**, so kann beim Aufruf der Methode für diesen Parameter jedes beliebige Objekt als Argument angegeben werden.
- Als Beispiel nehmen wir die Liste und schauen an, wie wir eine generische Liste erstellen können
- Spezialisierung ist auch möglich:
  – Statt **`Object`** kann auch eine beliebige Oberklasse einer Klassenhierarchie oder eine Schnittstelle als Elementtyp angegeben werden.

Beispiel: Generischer Datentyp `Liste` in Java

```
Class List {
private Node head;
…
}

class Link {
 private Link next;
 private int data;

...

}
```




```
Class List {
private Node head;
…
}

class Link {
  private Link next;
  private Object data;

...
}
```



Beispiel einer trivialen generischen Methode

- Durchlauf durch eine Liste:

```
public class List   {
private Node head;
…
public void scan () {
  Node cursor = head;
  while (cursor!= null) {
    // Aktion
    cursor = cursor.next;
  }
}}
```

- Die Methode **`scan()`** ist *generisch*, denn sie funktioniert für alle Listen mit Knoten unterschiedlicher Datentypen.
- Und sie ist *trivial*, weil in **`scan()`** die Daten in den Knoten gar nicht benutzt werden, d.h. ihr Datentyp ist irrelevant :-)

Eine etwas interessantere generische Methode

Drucken aller Elemente unabhängig vom Datentyp des Knotens:

```
public class List   {
private Node head;
….
public void scan () {
  Node cursor = head;
  while (cursor != null) {
   System.out.println(
  cursor.data.toString());
   cursor = cursor.next;
}
```

Eine generische Aktion

- Warum kann man so etwas machen?
- In der Klasse **`Object`** ist **`toString()`** eine Methode, die die Objekt-Referenz als **`String`** ausgibt.
- Java-Subtypen von **`Object`** überschreiben **`toString(),`** um sich selbst als **`String`** ausdrucken zu können.
- Alle Java-Subtypen haben also eine **`toString()`**-Methode mit derselben Signatur, aber unterschiedlicher Implementation.
- Welche Implementation zur Laufzeit aufgerufen wird, hängt vom Typ des Datenelements im Knoten ab.
- **Erweiterbarkeit**: Beim Schreiben dieser Methode braucht man noch nicht einmal alle Typen zu kennen, auf denen **`toString()`** aufgerufen wird.

Warum generische Methoden?

- **Konzept: Such- und Sortieralgorithmen** können als generische Methoden implementiert werden, die mit einem allgemeinen **Vergleichsoperator** parametrisiert sind.

- Generische Methoden arbeiten auf allen Spezialisierungen des generischen Datentyps, müssen allerdings oft zur Übersetzungs- oder Laufzeit erst angepasst werden (dynamische Bindung, Polymorphismus).

Ein generischer Vergleichsoperator

- Für generische Sortier- und Suchalgorithmen brauchen wir einen generischen Vergleichsoperator, der auf alle Typen anwendbar ist.
- Wir spezifizieren diesen Vergleichsoperator als Java Schnittstelle mit dem Namen **`Geordnet`** und einer öffentlichen Methode mit dem Namen **`vergleiche()`**:

```
public interface Geordnet {
        // -1: wenn this kleiner ist als wert
        //  0: wenn this gleich wert ist
        // +1: wenn this grösser ist als value
        public int vergleiche(Geordnet wert);
}
```

- Datentypen, für die der generische Vergleich verwendet werden soll , müssen die Schnittstelle **`Geordnet`** durch Implementation der Schnittstellenmethode **`vergleiche`**() bereitstellen.
- Im folgenden zeigen wir die Implementation von **`vergleiche`**() für die Klasse **`Student`**. Zunächst definieren wir **`Student`** als Unterklasse von **`Object`**.

Wiederholung: Die Java-Klasse Object (Info I - Vorlesung 13)

Vollständige Definition von Object
=> Java Referenz-Manual

```
public class Object {
    ….
    // Instanzmethoden
    public boolean equals(Object obj);
        // true, wenn beide Objekte gleich sind.
    ….
    public String toString();
        // Konvertiert die Werte der Felder eines Objektes in eine
    Zeichenkette
    ….
}
```

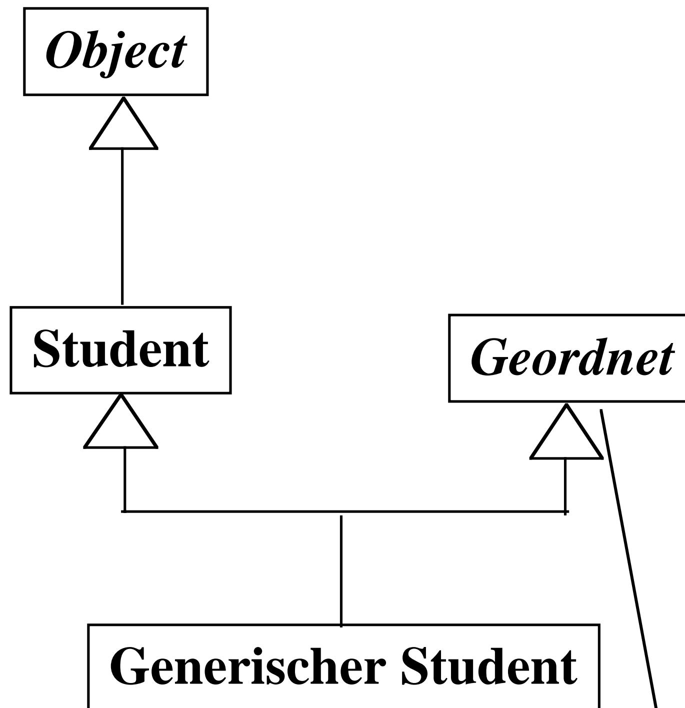
`Object` ist eine konkrete Klasse: `equals()` und `toString`() sind überschreibbare Methoden

Beispiel: Klasse `Student` (als Erweiterung von `Object`)

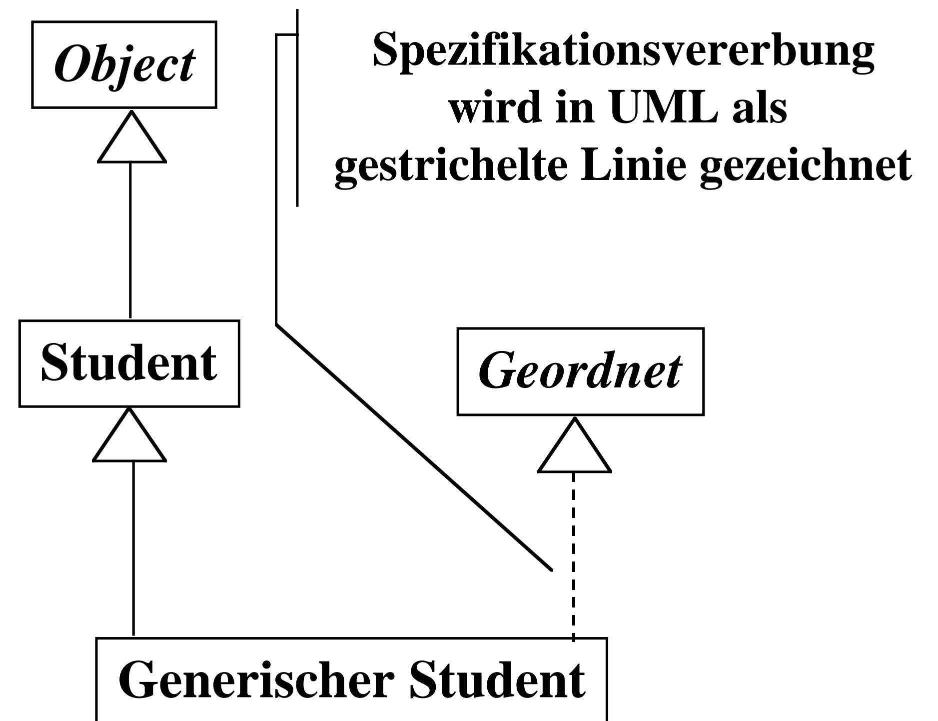
```
class Student extends Object {
  private String name;
  private double note;
  private int matrikelnr;
  // Konstruktoren
  Student () { } // Standard-Konstruktor
  Student (String n, double zensur, int mat) {
    name = n;  note = zensur; matrikelnr = mat;
  }
  // Abfragen/Setzen von Attributwerten
  String getName () { return name; }
  int getMatrikelNr () {return matrikelnr; }
  boolean hatDiplom () { return note <= 4.0; }
  double getNote () { return note;}
  void setName (String name) {this.name = name;}
  void setMatrikelNr (int matrikelnr) {  this.matrikelnr = matrikelnr; }
}
```

Modellierung des Generischen Studenten

Analysemodell



Implementationsmodell für Java



Spezifikationsvererbung wird in UML als gestrichelte Linie gezeichnet

Mehrfachvererbung in Java nicht möglich, also implementieren wir Geordnet als Schnittstelle

Generischer Student

- Mit der Definition von **`Student`** als Unterklasse von **`Object`** können wir jetzt eine Klasse **`GenerischerStudent`** als Unterklasse von **`Student`** definieren, die die Schnittstelle **`Geordnet`** implementiert:

```
public class GenerischerStudent extends Student implements Geordnet {
  GenerischerStudent (String Name, int matrikelnr) {
    this.setName(Name);
    this.setMatrikelNr(matrikelnr);
  }
  public int vergleiche (Geordnet wert) {
    GenerischerStudent andererStudent = (GenerischerStudent) wert;
    if (this.getMatrikelNr() < andererStudent.getMatrikelNr())  return -1;
    else if (this.getMatrikelNr() == andererStudent.getMatrikelNr()) return 0;
    else return 1;
  } // vergleiche
} // GenerischerStudent
```

Diese Typkonvertierung kann fehlschlagen! Typ-Überprüfung oder Fehlerbehandlung nötig (→ später)

Generischer Bubblesort

```
public class BubbleSort {
  private void swap (Geordnet[] v, int index1, int index2) {
    Geordnet temp;
    temp = v[index1];
    v[index1] = v [index2];
    v[index2] = temp;
  }
  public void sort (Geordnet[] v) {
    for (int pass = 1; pass < v.length; pass++)
      for (int pair = 1; pair < v.length; pair++)
        if ( v[pair-1].vergleiche( v[pair]) == 1)
          swap(v, pair-1, pair);
  }
} // end BubbleSort
```

Mit der Anweisung
`GenerischerStudent andererStudent = (GenerischerStudent) wert;`
(siehe Folie 73) konvertiert **`vergleiche`**() die Variable **`v[pair]`** vom Typ **`Geordnet`** in eine Variable vom Typ **`GenerischerStudent`**.

Bei Eingabe einer Reihung von **`GenerischerStudent`**-Objekten wird diese Reihung nach der Matrikelnummer sortiert.

Copyright 2001 Bernd Brügge

Wo stehen wir?

Wichtige Konzepte der Objekt-Orientierten Programmierung:

✓ **Vererbung**

✓ **Dynamische Bindung**

✓ **Polymorphismus**

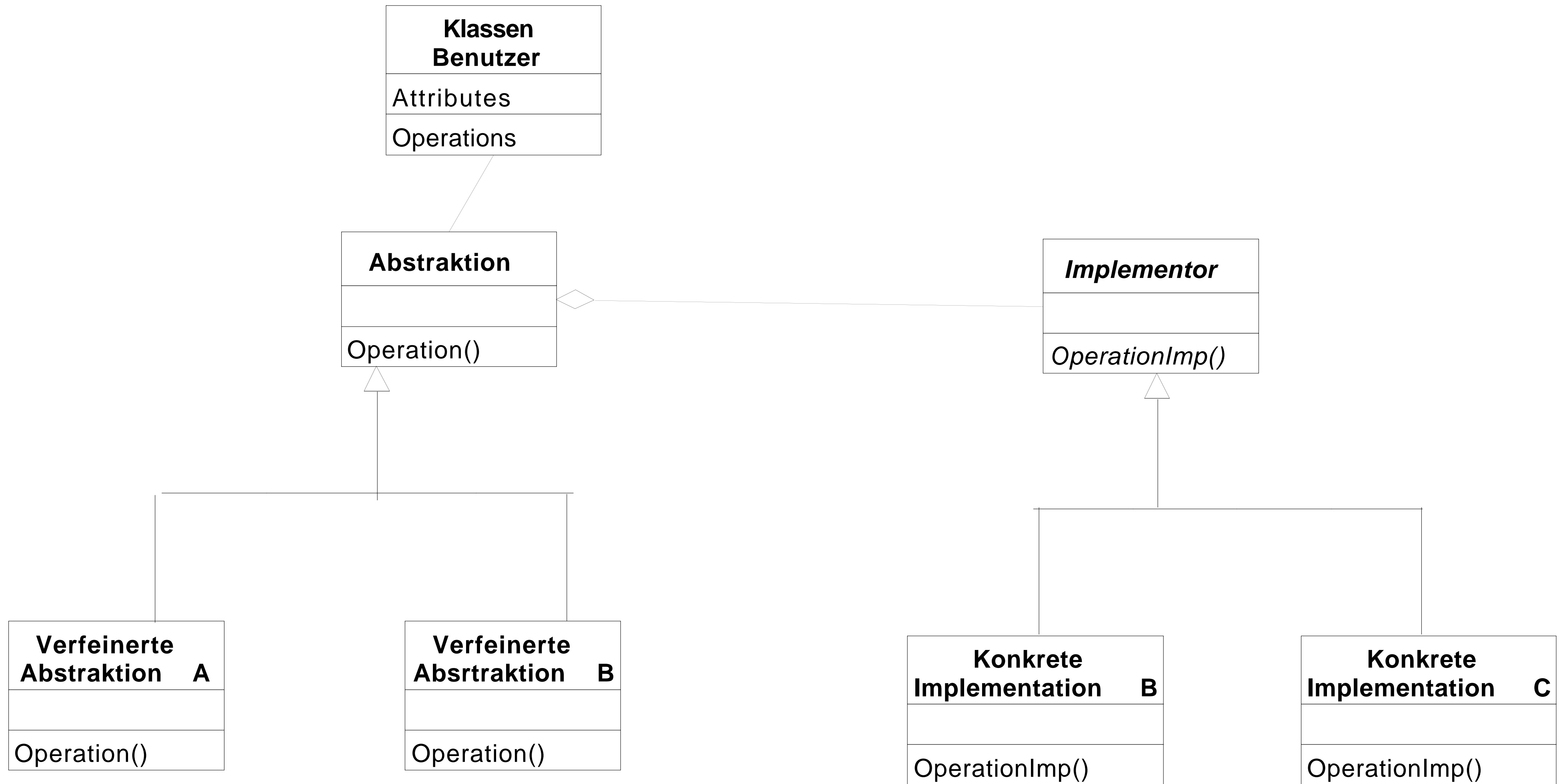
Anwendungen

✓ **Generische Klassen**

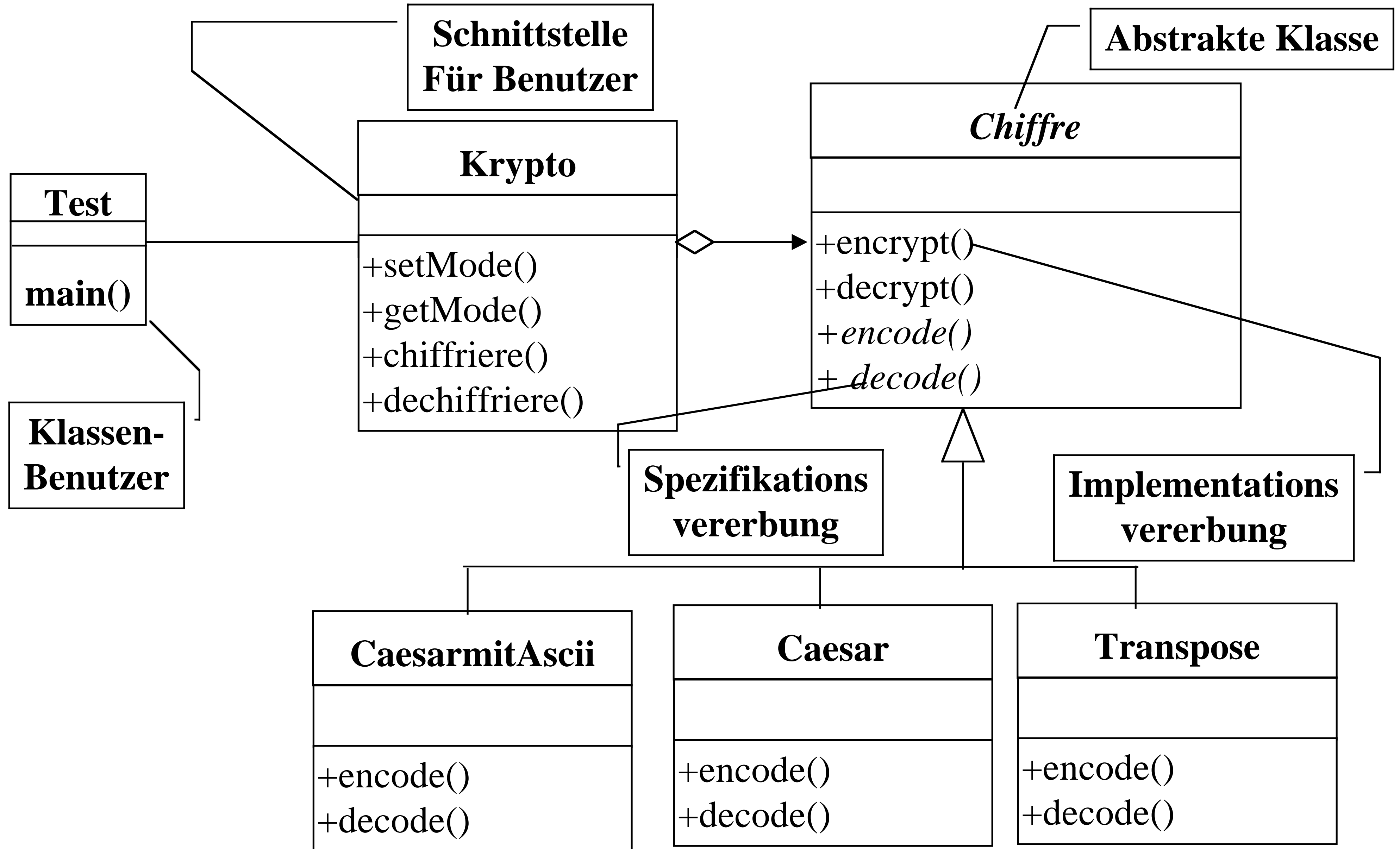
➡ **Entwurfsmuster (Brücken-Muster, Strategie-Muster**)

➠ Wir wenden uns jetzt der Anwendung der Konzepte auf **Entwurfsmuster** zu.

Brücken-Muster



Beispiel aus Info I



Wann setzen wir das Brückenmuster ein?

3 von vielen Gründen, das Brückenmuster zu benutzen:

1. Eine Abstraktion hat verschiedene mögliche Implementationen, wir wollen aber eine permanente Bindung zwischen Abstraktion und Implementation vermeiden

   ⇒ hier kommt der dynamische Polymorphismus rein

2. Sowohl die Abstraktion als auch Implementation sollen erweiterbar sein

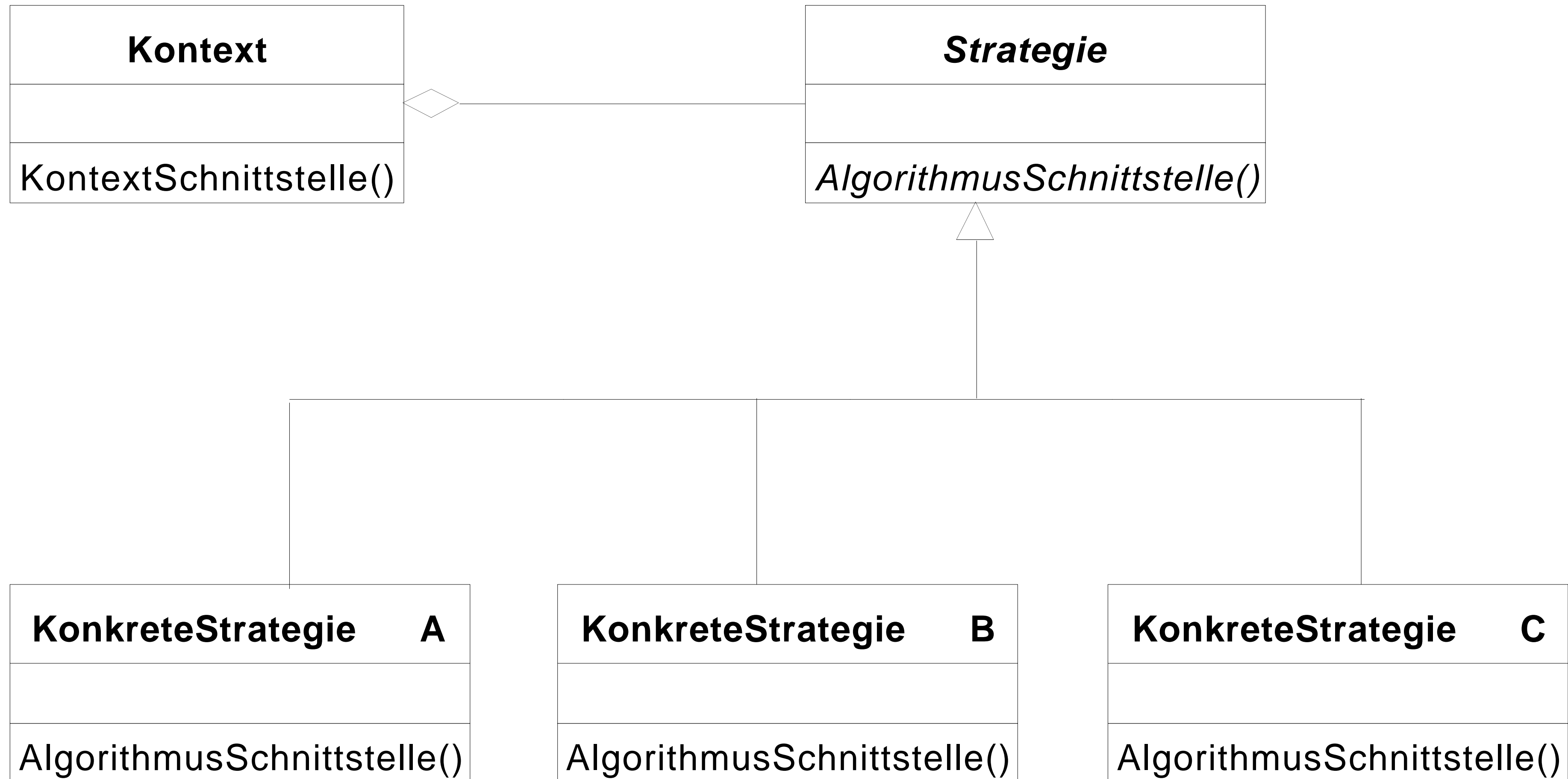
   ⇒ hier kommt die Vererbung rein

3. Veränderungen in den Implementationen haben keinen Einfluss auf den Klassen-Benutzer.
   - Wir wollen sowohl Benutzer unterstützen, die ein Alt-System (legacy system) verwenden als auch Benutzer, die das neueste System benutzen wollen.

Grenzen des Brücken-Musters

- Die Entscheidung zwischen den einzelnen Implementationen ist zwar dynamisch, aber manchmal möchte der Klassen-Benutzer in Abhängigkeit von einem Kontext unterschiedliche Algorithmen auswählen, ohne spezifisch diese Algorithmen benennen zu wollen.
- Beispiele:
  - Mobile Benutzer haben oft unterschiedliche Bandbreite zur Verfügung (GSM, GPRS, UMTS). Abhängig von der Bandbreite sollen unterschiedliche Implementationen von **`send()`** und **`receive()`** aufgerufen werden.
  - Ein System bietet eine Online-Hilfe-Funktion für Software-Installation an. Abhängig von der Erfahrung des Benutzers sollen unterschiedliche Funktionen aufgerufen werden, die dem Benutzer bei der Installation helfen (Hilfe für blutigen Anfänger, Hilfe für Experte, Hilfe für Anfänger, der aber Experte auf einem anderen System ist).

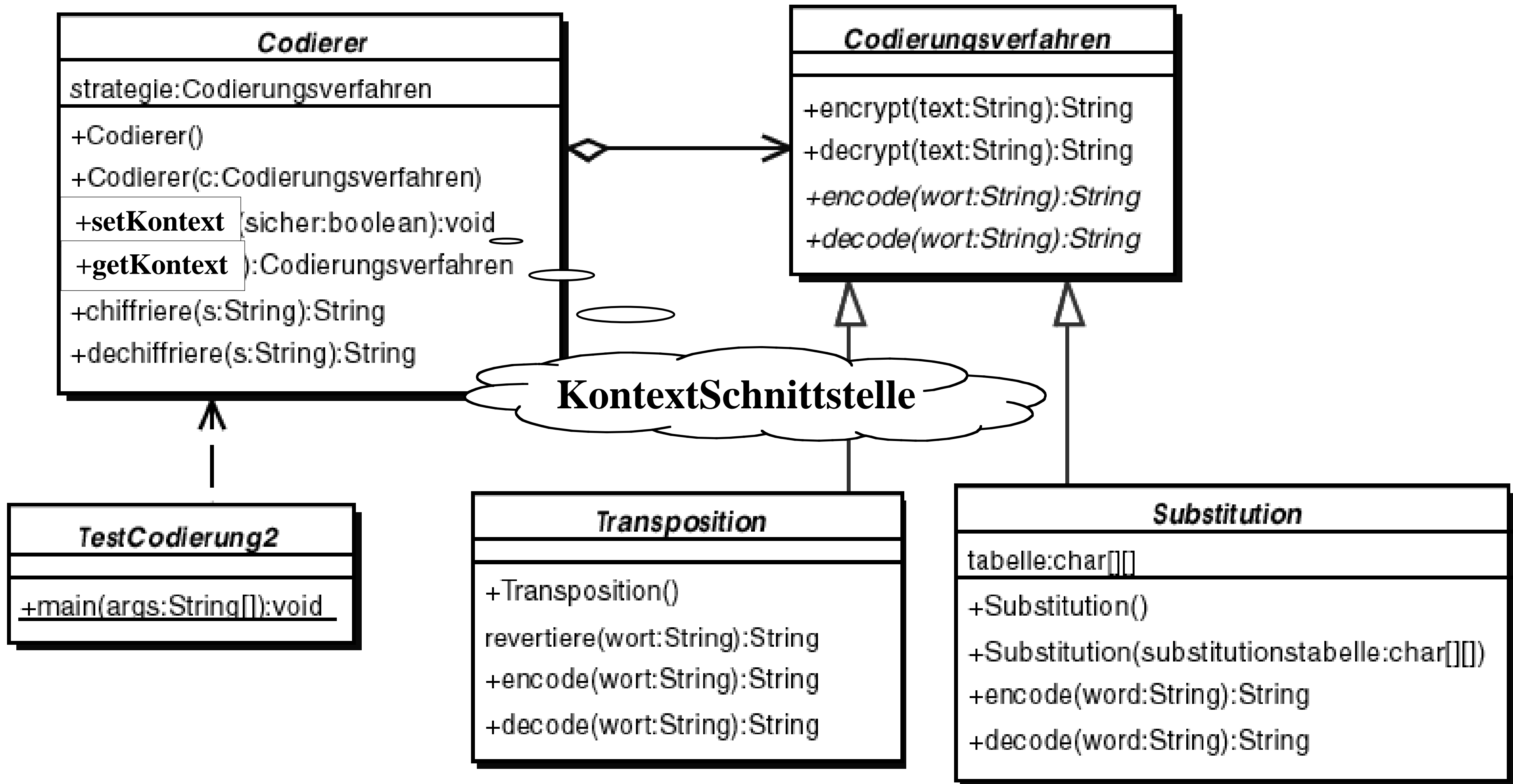
Strategie-Muster



Wann nimmt man das Strategie-Muster?

- **Familie von Algorithmen:** Man hat viele Varianten (Algorithmen) derselben Operation mit unterschiedlichen Laufzeit und Speicherplatzkomplexitäten. Abhängig von einem Kontext (wenig Speicherplatz, wenig Zeit, schneller Prozessor, usw.) möchte man einen bestimmten Algorithmus zur Laufzeit auswählen, ohne den Aufruf zu verändern.
- **Wahl von unterschiedlichen Implementationen:** Verschiedene Algorithmen benutzen unterschiedliche Datenrepräsentationen, die vor dem Klassenbenutzer versteckt werden sollen.
- **Elimination von verschachtelten Anweisungen:** Im Benutzer-Code erscheinen viele verschachtelte `if`-Anweisungen.
  – In diesem Fall ist es oft besser, für jeden Typ von Verzweigung eine eigene Strategie-Klasse zu definieren.

Strategie Muster im Kryptologie-Beispiel



Copyright 2001 Bernd Brügge

Implementierung von Codierer

```
public class Codierer {
   private Codierungsverfahren strategie;
   public Codierer (Codierungsverfahren c) {
     strategie = c;
   }
   public void setKontext (boolean sicher) {
     if (sicher) strategie = new Transposition();
     else strategie = new Substitution();
   }
   public Codierungsverfahren getKontext () {
     return strategie;
   }
  public String chiffriere (String s)  {
     return strategie.encrypt(s);
  }
  public String dechiffriere (String s) {
     return strategie.decrypt(s);
  }
} // Codierer
```

Implementierung von Codierungsverfahren

```
import java.util.StringTokenizer;

public abstract class Codierungsverfahren {

  public String encrypt(String text) {
    String result = new String();
    StringTokenizer words = new
      StringTokenizer(text);
   // Segmentiere text in Worte
    while (words.hasMoreTokens()) {
      // Für jedes Wort w in text:
      result = result + " " +
        encode(words.nextToken()); // Verschlüssele
     }
    return result;
  } // encrypt()
```

```
public String decrypt(String text) {
  String result = new String();
  StringTokenizer words = new
    StringTokenizer(text);
  // Segmentiere text in Worte
  while (words.hasMoreTokens()){// Für jedes
    Wort w in text:
   result = result + " " +
    decode(words.nextToken());// Entschlüssele
   }
  return result;
  } // decrypt()

// Abstrakte Methoden
  public abstract String encode(String wort);
  public abstract String decode(String wort);
} // Codierungsverfahren
```

Implementierung von Substitution

```
public class Substitution extends
    Codierungsverfahren{
  private char[][] tabelle;
  public Substitution() {
    char[][] standard =
     {'A','B','C','D','E','F','G','H','I','J','K','L','M'
     ,'N','O',
      'P','Q','R','S','T','U','V','W','X','Y','Z'},
     {'D','E','F','G','H','I','J','K','L','M','N','O','P'
     ,'Q','R',
     'S','T','U','V','W','X','Y','Z','A','B','C'}};
    tabelle = standard;
  }
public Substitution(char[][]
    substitutionstabelle) {
   // Vorbedingung Tabelle muss 2
    gleichlange Zeilen haben
    tabelle = substitutionstabelle;
  }
```

```
public String encode(String word) {
   String result = new String();
   for (int k = 0; k < word.length(); k++) {
            // wir wandern durch das Wort
      char ch = word.charAt(k);
      char subst = ch;
      for (int i = 0; i < tabelle[0].length; i++) {
         if (ch == tabelle[0][i]) subst =
    tabelle[1][i];
         // schlagen in der Tabelle das
    Substitutionszeichen nach
      }
      result = result + subst;    // und
    konkatenieren es mit result
   }
   return result;
 } // encode()
```

Implementierung von Transposition

```
public class Transposition extends
    Codierungsverfahren {
  private String revertiere(String wort) {
    String result = "";
    for (int i=wort.length()-1; i>=0; i--)
        result = result + wort.charAt(i);
    return result;
  }
  public String encode(String wort) {
    return revertiere(wort);
  } // encode
  public String decode(String wort) {
    return revertiere(wort); // oder:  return
     encode(wort);
  } // decode
} // Transposition
```

Implementierung von TestProgramm

```
public class TestCodierung2 {
  public static void main(String[] args) {
    boolean sicher = true;
    Codierer code = new Codierer();
    // Festlegung des Algorithmus:
    code.setKontext(sicher);

    //Anwendung des Algorithmus:
    String original = "DIES IST EINE GEHEIME NACHRICHT";
    String codiert = code.chiffriere(original);
    System.out.println(" ** Codierungsverfahren: " + code.getKontext() + " **");
    System.out.println("Normal: " + original);
    System.out.println("Verschlüsselt: " + codiert);
    System.out.println("Entschlüsselt: " + code.dechiffriere(codiert));
  } // main
} // TestCodierung2
```

Brücken-Muster versus Strategie-Muster

- **Brückenmuster**:
  – Die dynamische Auswahl ist nur für den Implementierer der Klasse interessant. Oft werden mit dem Brückenmuster zwei Implementationen unterstützt, die "alte" und die "neue" (bessere).
  – Für den Klassenbenutzer ist es egal, welche Implementation gewählt wird, solange sie "funktioniert". Er will sich nicht um die Auswahl kümmern.
- **Strategie-Muster**:
  – Das "Funktionieren" der Methode ist dem Klassen-Benutzer nicht genug. Abhängig von einem Kontext soll eine bestimmte Methode gewählt werden.
  – Der Klassenbenutzer muss verstehen, wie sich die Strategien unterscheiden, bevor er eine wählt (kann ein Problem sein).

Zusammenfassung der wichtigsten Konzepte

- Wichtigte Ziele bei der Erstellung von Informatik-Systemen:
  - **Organisation, Flexibilität, Wiederverwendbarkeit und Allgemeinheit**
- 3 wichtige Konzepte der Objekt-Orientierung helfen uns, diese Ziele zu erreichen
  - **Vererbung**
  - **Dynamische Bindung**
  - **Polymorphismus**
- Wichtige Arten der Vererbung: **Spezialisierung, Generalisierung, Spezifikationsvererbung, Implementationsvererbung**
- Viele andere Konzepte lassen sich auf diese 3 Grundkonzepte zurückführen
  - Beispiel: **Generische Programmierung, Brücken-Muster, Strategie-Muster**