Objektorientierte Abstraktionen

Fragt man drei Vertreter der Objektorientierung nach den Grundkonzepten wird man wahrscheinlich drei verschiedene Antworten erhalten. Aber zweifellos wird dazu gerechnet:

1. Bündelung von Daten und Methoden
2. Datenabstraktion
3. Klassen als Objektschablonen
4. Vererbung
5. Polymorphismus

Objekte & Klassen

Wert versus Objekt

Betrachten wir zunächst ein ganz einfaches Konstrukt – die Deklaration mit Zuweisung, etwa

```
int i = 987;
```

Was sind die Bestandteile?

Wert Die ganze Zahl `987` ist ein *Wert*, eine individuelle Konstante, die keinen Ort in Raum und Zeit hat; sie ist unveränderlich, es gibt diese Zahl nur einmal.

Der Wert kann im Speicher eines Computers repräsentiert werden, aber nicht verändert werden.

Variable Eine Variable, in unserem Beispiel mit Namen `i`, ist ein Platzhalter für einen Wert; im Computer genauer für die Repräsentation eines Wertes.

Eine Variable kann sich ändern, sie kann zu verschiedenen Zeitpunkten verschiedene Werte enthalten. Addiert man etwa die Zahl `1` durch die Anweisung `i = i + 1;` wird der Wert `987` ersetzt durch den Wert `988`.

Wertetyp Die Angabe `int` bezeichnet den Wertetyp der Variablen, d.h. die Menge der Werte, aus der die Variable ein Element enthalten kann, in unserem Fall etwa $[-2^{31}, +2^{31})$.

Nun machen wir dasselbe mit einem Objekt. Wir konstruieren uns eine Klasse `myInteger` und wir erzeugen ein Objekt dieser Klasse, das den Wert 987 in der Membervariable `m_Inhalt` enthält.

```
myInteger oi = new myInteger(987);
```

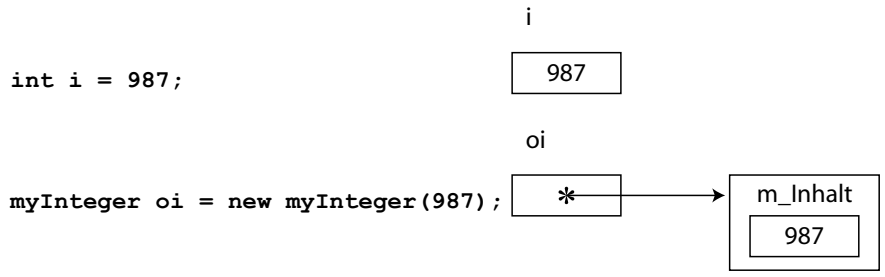
Was sind nun die Bestandteile?

Objekt Auf der rechten Seite der Zuweisung erzeugt der Konstruktor `new` ein Objekt. Im Unterschied zum Wert *kann* dieses nun veränderlich sein, weil es nämlich kein Wert ist, sondern ein Paar von Variable *und* Wert. Das Objekt hat also einen Zustand, nämlich den aktuellen Wert seiner Variablen.

Objektvariable Die Variable `oi` ist ein Platzhalter, der aber nicht direkt einen Wert enthält, sondern eine *Referenz* auf das Objekt. [1]

Klasse des Objekts `myInteger` ist der Typ des Objekts, die Klasse, aus der dieses Exemplar erzeugt wurde. Es handelt sich um einen bentuzerdefinierten Typ, denn die objektorientierten Sprachen erlauben uns eigene Klassen zu definieren.

Der Unterschied kann so verbildlicht werden.



Untersuchen wir nun weiter, was passiert, wenn man eine Methode hat, um Objekte vom Objekttyp myInteger zu verändern:

Angenommen, wir haben in unsere Klasse die Methode `add( int i )` so programmiert[2]:

[1] Die obige Syntax ist Java; in C++ wäre die Referenz auf das Objekt direkt sichtbar als Pointer. Es müsste heißen: `myInteger* oi = new myInteger(987)`. Uns interessieren aber nicht die Sprachspezifika, sondern das Konzept!

[2] Achtung: die Klasse Integer in Java hat keine Methode `add`. Die Klasse Integer in Java ist nämlich eine *wertorientierte* Klasse. Unser Beispiel von myInteger ist hingegen eine *zustandsorientierte* Klasse, denn das Beispiel dient hier zur Illustration des Unterschieds von Werten und Objekten.

```
myInteger add( int i ) {
  m_Inhalt += i;
  return this;
}
```

`oi.add( 1 )` hat zur Folge, dass `m_Inhalt` den Wert `988` enthält – der Wert von `oi`, die *Objektidentität* hat sich *nicht* verändert.

Das Ganze kommt einem möglicherweise etwas umständlich vor – wozu diese zusätzliche Indirektion?

Betrachten wir das Beispiel eines Kontos:

Heute:

```
Konto Nummer     1 334 789 65
      Besitzer   Hans Meier
      Saldo      1246,-- Euro
```

Morgen:

```
Konto Nummer     1 334 789 65
      Besitzer   Hans Meier
      Saldo      746,-- Euro
```

Der Kontostand, der Saldo hat sich geändert; das Konto ist aber dasselbe geblieben.

Oder erinnern wir uns an den Stapel aus dem vorherigen Skript, auch dort war das Prinzip der Datenabstraktion, dass sich der Zustand des Stapels ändern kann, die interne Struktur aber nach außen verborgen ist.

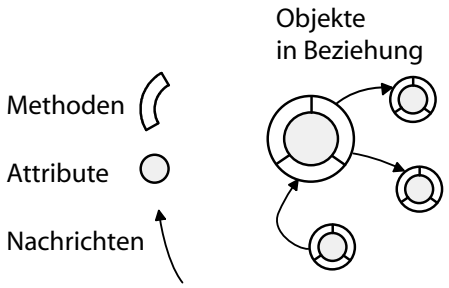
Hinweis: Ganz unberührt von dieser Diskussion ist die Tatsache, dass man Objekte wert- oder zustandsorientiert konzipieren kann – wie dies in der Veranstaltung „Objektorientierte Programmierung“ erläutert wird. Wir werden auf die Bedeutung dieses Unterschieds zurückkommen, wenn es um Polymorphismus geht.

Datenabstraktion bei Objekten

Fassen wir unsere bisherigen Überlegungen zusammen:

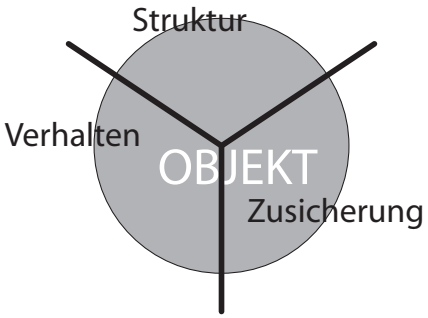
Ein *Objekt* enthält seinen Zustand als gekapselte Daten (Attribute). Nach außen stellt ein Objekt seine Dienste durch Methoden zur Verfügung. Der Aufruf einer Methode (Nachricht) verwendet die Objektidentität, um das Objekt zu referenzieren.

Datenabstraktion bei Objekten



Eine *Klasse* ist eine Fabrik, eine Schablone zum Erzeugen von Objekten.

Das Tripel des Objekts

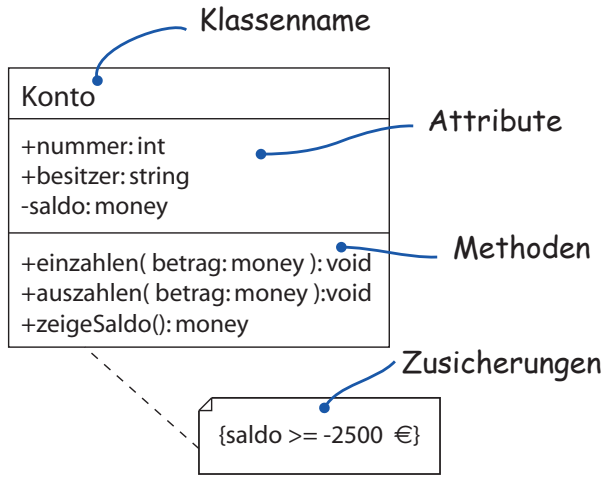


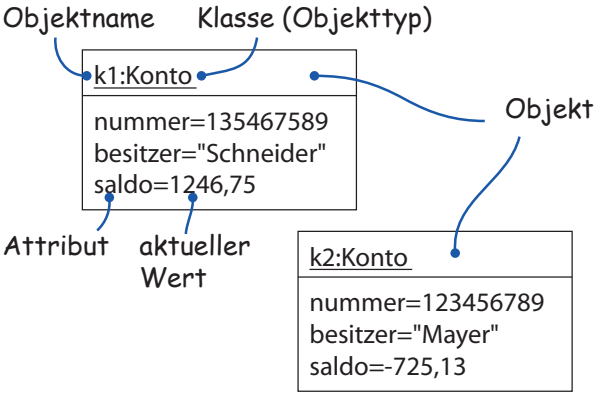
Das Tripel des Objekts

- Struktur: Eigenschaften = Attribute und Beziehungen
- Verhalten: Methoden, implementiert durch Funktionen, ausgelöst durch Nachrichten/Methodenaufruf
- Zusicherungen: Constraints = Vorbedingungen, Nachbedingungen, Invarianten — oft dargestellt durch OCL *Object Constraint Language*

Klasse als Implementierungskonzept

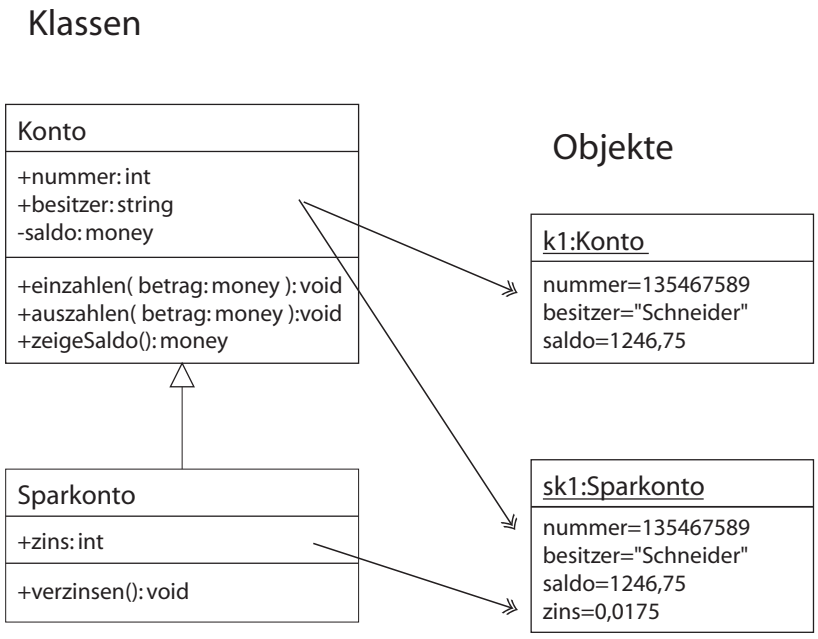
Schablone und Exemplar





Vererbung

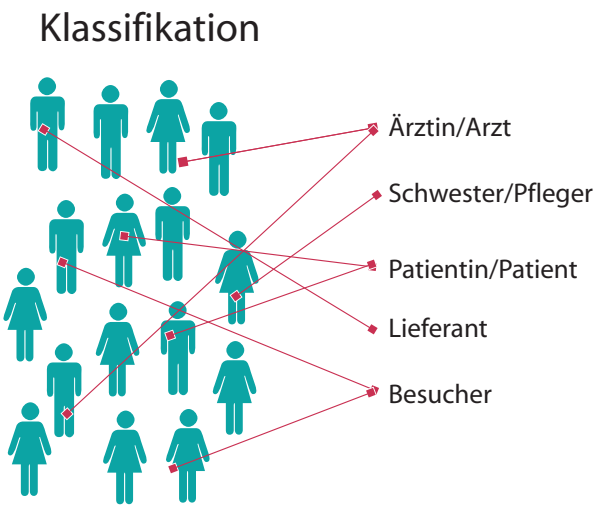
Die Unterklasse erbt die Eigenschaften, Methoden und Zusicherungen. Sie kann weitere Eigenschaften, modifizierte und weitere Methoden und weitere Zusicherungen haben.



Objekttyp als Analyse- und Modellierungskonzept

Klassifikation

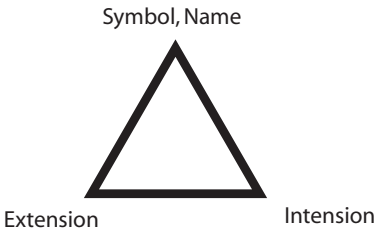
Übertragung der Konzepte aus der Implementierung in die Analyse des Problemfelds, der Domäne: Wir betrachten die Entitäten (Dinge, Personen, Konzepte usw.) als Objekte, die nach ihren wesentlichen Gemeinsamkeiten klassifiziert werden. Man teilt die Objekte in Objekttypen (Klassen) ein.



Klassifikation = den Begriff eines Objekts finden, es auf das Wesentliche konzentrieren — ein Schritt der Abstraktion.

Symbol, Extension & Intension

Eigenschaften eines Begriffs:

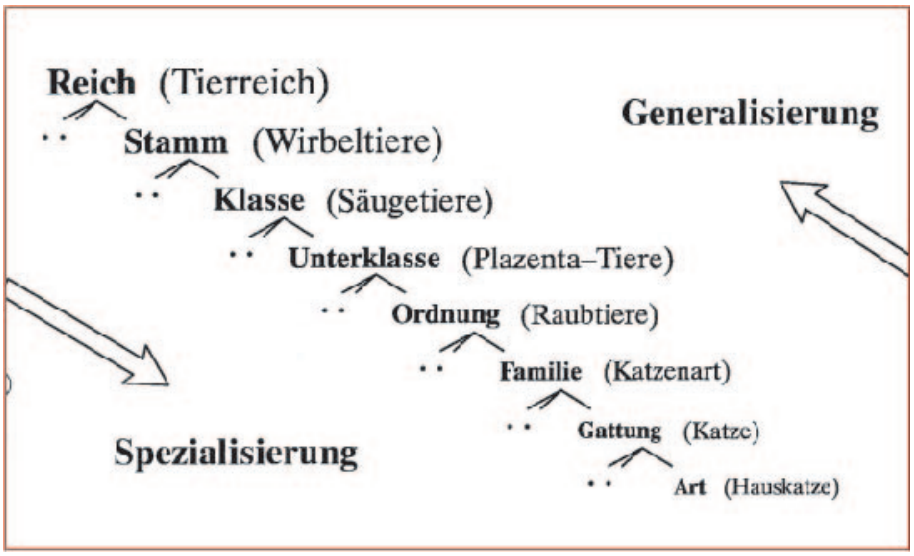


- Symbol/Name: Treffende, begriffliche Bezeichnung
- Extension: Die Menge der Objekte, die dem Begriff entsprechen
- Intension: Die Eigenschaften, die den Begriff charakterisieren, und die die Objekte haben (Test auf Zugehörigkeit)[3]

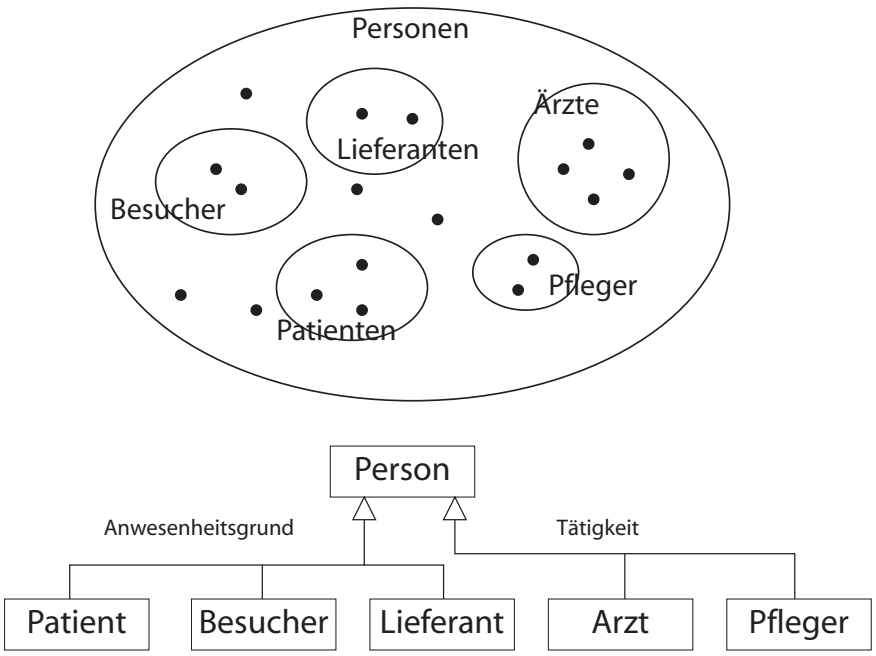
[3] Die Begriffe Extension und Intension kommen auch bei Datenbanksystemen vor: Mit *Intension* bezeichnet man das Schema einer Relation, mit *Extension* die Menge der Tupel einer Relation, also den Zustand der Relation.

Generalisierung & Spezialisierung

Eine „is-a“-Beziehung: ist von dieser Art, bloß spezieller; ist ersetzbar.
Generalisierung = Erweiterung der Extension
Spezialisierung = Einschränkung der Intension



Ein Beispiel für Klassifizierung:



Unterschied Beschreibung – Codestruktur

- In der Analyse betrachten wir die Objekte der realen Welt und klassifizieren sie, wir finden ihren Begriff. Generalisierung und Spezialisierung ist die Taxonomie der Objekte der Welt.
- In der Implementierung erzeugen wir Codestrukturen, die Objekte der realen Welt simulieren, die vielleicht aber auch allein durch unsere Entwurfsentscheidungen entstehen (und keine Entsprechung in der Welt haben). Vererbung ist ein Mechanismus der Strukturierung und Wiederverwendung von Code.

Die Struktur des Codes wird in irgendeiner Weise die Klassifikation und Modellierung der wirklichen Welt widerspiegeln. Sonst könnte das zu entwickelnde System seinen Zweck nicht erfüllen.

Wir müssen aber stets unterscheiden, worüber wir sprechen: über die Welt oder über die Codestruktur. Beide Gesichtspunkte unterscheiden sich grundlegend – wie wir noch sehen werden.

Dynamischer Polymorphismus

Dynamischer Polymorphismus ist das Konzept der Objektorientierung, das den eigentlichen Nutzen für die Wiederverwendung von Klassenbibliotheken erbringt: man kann vorhandene Klassen um neue Eigenschaften erweitern und vorhandenen (!) Code mit diesen neuen Unterklassen verwenden, ohne ihn verändern zu müssen.

Man spricht wegen dieser Eigenschaft objektorientierter Sprachen auch vom sogenannten *Open-Closed-Prinzip*:

> Modules should be both open and closed.
> The contradiction between the two terms is only apparent as they correspond to goals of a different nature:
>
> - A module is said to be open if it is still available for extension. For example, it should be possible to expand its set of operations or add fields to its data structures.
> - A module is said to be closed if it is available for use by other modules. This assumes that the module has been given a well-defined, stable description (its interface in the sense of information hiding). At the implementation level, closure for a module also implies that you may compile it, perhaps store it in a library, and make it available for others (its clients) to use ...
>
> – Bertrand Meyer

Oder paraphrasiert: Eine Klasse soll offen für Erweiterungen sein, aber geschützt gegenüber Veränderungen.

Dies klingt paradox – ist aber wegen dynamischem Polymorphismus möglich. Wir werden sehen, wie. Aber Obacht: Das Open-Closed-Prinzip funktioniert nur, wenn man Klassen und Vererbung korrekt einsetzt, sie müssen dem Substitutionsprinzip entsprechen!

Motivation

Stellen wir uns vor, wir entwickeln einen Bildschirmschoner, der farbige Quadrate verschiedener Größe an immer wechselnden Positionen am Bildschirm in wechselnden Zeitabständen anzeigt.

Grob skizziert könnte der Code für unseren Bildschirmschoner so aussehen (Skizze der Idee, keine echter Code):

```
// in der Klasse Application
initApp() {
```

```
  ...
  Quadrat myQuadrat = new Quadrat();
  ...
  while ( !UserEvent() )
  {
    display( myQuadrat );
  }
  ...
}

display( Quadrat quadrat )
{
  // bestimme Farbe, Groesse und Position per Zufallsgenerator
  ...
  // zeichne
  quadrat.display();
  ...
}
```

Unser Bildschirmschoner funktioniert gut, aber kaum wird er von den ersten Anwendern eingesetzt, da kommen schon die Wünsche auf. Viele Anwender möchten keine Quadrate (zu eckig!) auf dem Bildschirm sehen, sondern Kreise. Was tun?

Eine erste Erweiterung könnte so aussehen:

```
initApp()
{
  ...
  if ( typ == 'Q' )
    Quadrat myQuadrat = new Quadrat();
  else if ( typ == 'K' )
    Kreis myKreis = new Kreis();

  ...
  while ( !UserEvent() )
  {
    if ( typ == 'Q' )
      displayQuadrat( myQuadrat );
    else if ( typ == 'K' )
      displayKreis( myKreis );
  }
  ...
}

displayQuadrat( Quadrat quadrat )
{
  // analog zu oben
}

displayKreis( Kreis kreis )
```

```
{
  // analog
}
```

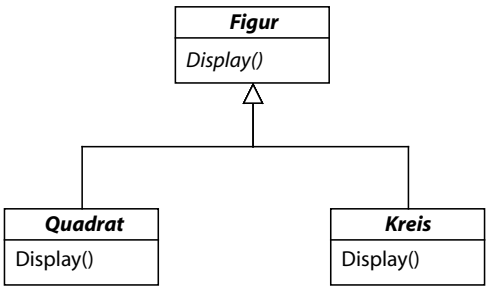
Da ist nur zu hoffen, dass unsere Anwender nicht auch noch andere Figuren vom Bildschirmschoner angezeigt bekommen wollen!

Es geht aber auch anderes:

```
initApp()
{
  ...
  Figur myFigur;
  ...
  if ( typ == 'Q' )
    myFigur = new Quadrat();
  else if ( typ == 'K' )
    myFigur = new Kreis();
  ...
  while ( !UserEvent() )
  {
    display( myFigur );
  }
  ...
}

display( Figur figur )
{
  // analog zu oben
  figur.display();
}
```

In welcher Beziehung stehen Figur, Kreis und Quadrat?



Und was haben wir dadurch erreicht? An der Stelle, wo Kreis oder Quadrat gezeichnet werden, muss gar nicht bekannt sein, dass wir einen Kreis oder ein Quadrat zeichnen wollen, sondern nur, dass eine Figur gezeichnet werden soll. Je nachdem, ob die Figur für einen Kreis oder

ein Quadrat steht, wird die passende Methode aufgerufen. Das nennt man „spätes Binden“.

Wenn man als das *Erzeugen* von Objekten von ihrer *Verwendung* trennt, dann kann man den Code der Verwendung unverändert lassen, und ihm neue, andersartige Objekte unterschieben, die sich nur analog verhalten müssen, also in unserem Beispiel etwa Dreiecke, Sterne etc.

Dynamisches Binden

Jetzt begrifflicher.

Unter *Polymorphie* (aus dem Griechischen: Vielgestaltigkeit) versteht man:

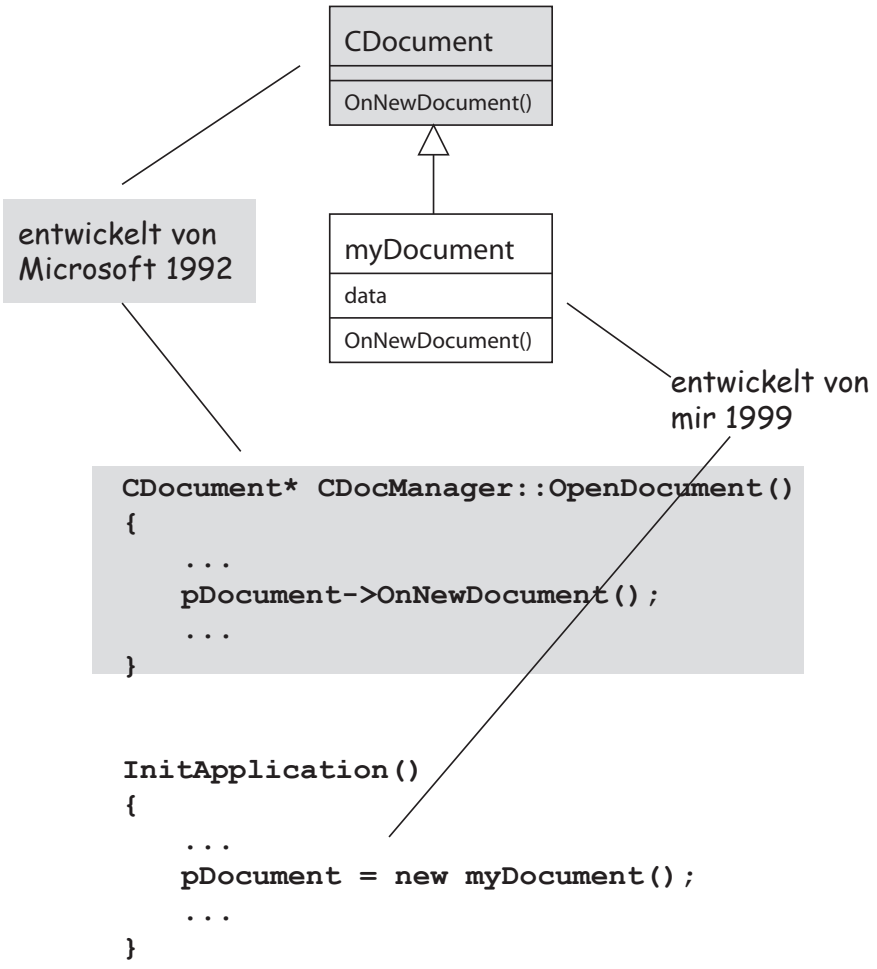
- Eine Variable eines Typs K kann Objekte der Klasse K oder von Unterklassen von K enthalten
- Eine Methode mit einem formalen Parameter des Typs K kann mit einem aktuellen Objekt der Klasse K oder von Unterklassen von K aufgerufen werden.

Und *dynamisches Binden* bedeutet

- Enthält eine Variable vom Typ K eine Objekt der Unterklasse U, welche Methode wird zur Laufzeit verwendet? Diejenige von K oder die von U?
- Beim dynamischen Binden wird die Methode der aktuellen Klasse des Objekts, also der Unterklasse U verwendet.

Weil zur Laufzeit Objekte der Unterklasse an die Stelle der Objekte der Oberklasse treten können, spricht man von *Substituierbarkeit*. Und genau in diesem Sinne wollen wir das Vererbungssymbol der UML interpretieren!

Durch dynamisches Binden wird das Open-Closed-Prinzip erreicht. Nochmal ein illustratives Beispiel von einer Klassenbibliothek, den Microsoft Foundation Classes:



Substituierbarkeit bedeutet also:

> *Alter Code ruft neuen Code auf!*

Substituierbarkeit

Wert versus Objekt

Man unterscheidet: Wertesubstituierbarkeit und Objektsubstituierbarkeit.

- Als Wert ist ein Kreis gegen eine Ellipse substituierbar, denn jeder Kreis *ist* eine spezielle Ellipse; die Kreise sind eine Teilmenge der Ellipsen.

- Als Objekt ist ein Kreis nicht gegen die Ellipse substituierbar, wenn die Ellipse Methoden `setA()` und `setB()` hat, die angewandt auf einen Kreis diesen in eine Ellipse verwandeln.

Deshalb ist es wichtig, dass wir beim Design von Klassen uns darüber Gedanken machen, ob wir wert- oder zustandsorientiert vorgehen. (Siehe die Veranstaltung OOP)

Was tun?

- Unterscheiden zwischen Klassifikation in der Analyse – hier sind Teilmengen sinnvoll – und Klassenhierarchien in Design und Implementierung – hier kommen oft zustandsorientierte Klassen vor.
- Beim Design der Codestruktur stets auf Liskovs Substitutionsprinzip achten.

Liskovs Substitutionsprinzip

- **Regel über die Signaturen**
  Unterklassen müssen alle Methoden der Oberklasse haben und die Signaturen müssen kompatibel sein.
- **Regel über die Methoden**
  Die Methode der Unterklasse darf keine stärkere Vorbedingung und keine schwächere Nachbedingung haben.
- **Regel über die Eigenschaften**
  Alle Eigenschaften (Invarianten) der Oberklasse müssen von der Unterklasse eingehalten werden.

Zusammengefasst verlangt das Substitutionsprinzip:

> *Verlange nicht mehr, garantiere nicht weniger!*

Das Substitutionsprinzip wird *nicht* durch die objektorientierten Sprachen erzwungen, man kann also dagegen verstoßen, ohne dass der Compiler darauf aufmerksam macht. Es muss also dadurch eingehalten werden, dass Klassen und Vererbung entsprechend entworfen und eingesetzt werden.

Interface & Implementierung

Arten der Vererbung

Man unterscheidet zwei Arten der Vererbung:

Whitebox-Vererbung

- Klassenvererbung, Implementierungsvererbung.
- Definiert eine neue Klasse auf Basis einer anderen Klasse. Die Unterklasse erbt Schnittstelle und Implementierung der Oberklasse
- Whitebox (Innensicht): Die Unterklasse erweitert die Interna der Oberklasse, eventuell überschreibt sie diese.

Blackbox-Vererbung

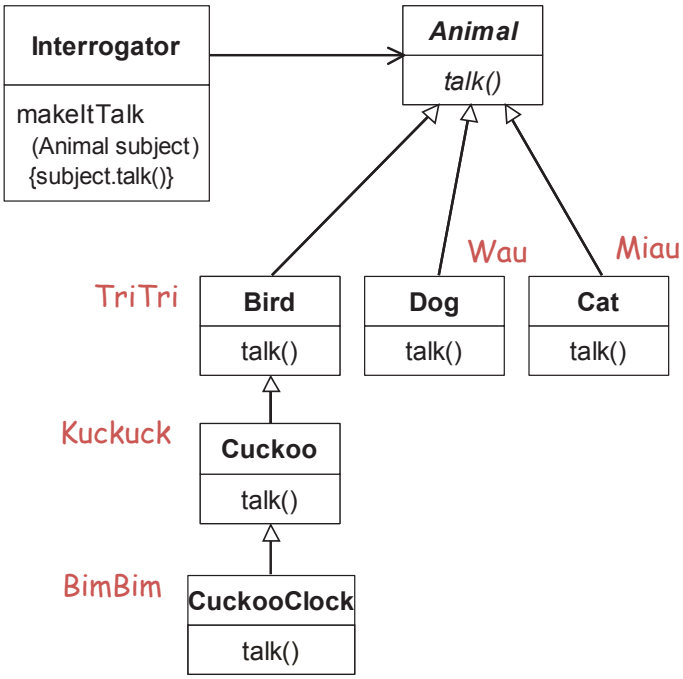
- Schnittstellenvererbung
- Definiert eine neue Schnittstelle auf Basis einer anderen Schnittstelle
- Blackbox (Außensicht): Die neue Schnittstelle erbt Verhalten, keine Implementierung!

Beispiel

Folgendes Beispiel stammt von Bill Venners: *Designing with Interfaces* JavaWorld 1998

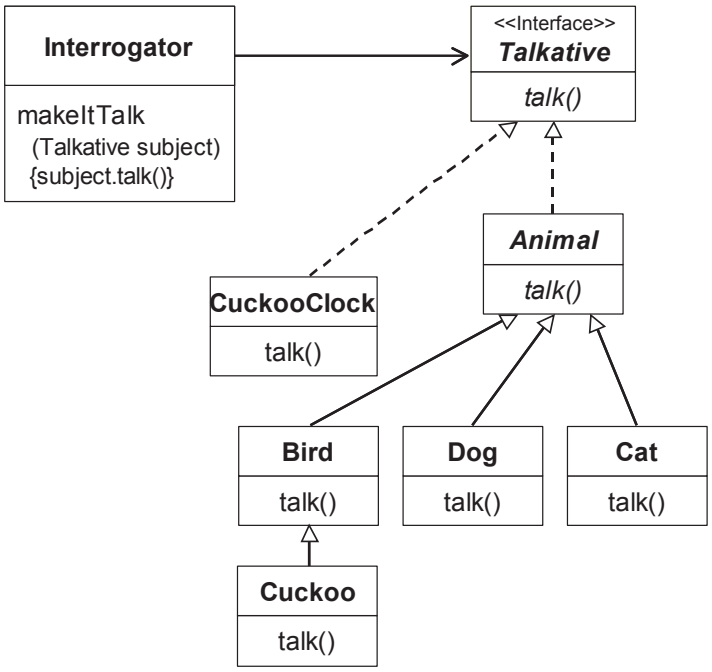
Im Internet zu finden unter `http://www.javaworld.com/javaworld/jw-12-1998/jw-12-techniques.html`.

Stellen wir uns vor, wir haben eine Hierarchie von Tieren: Tier, Vogel und Kuckuck; und einen Befrager, der Tiere sprechen lassen kann. Wenn nun eine weitere Sorte hinzukommt, die Kuckucksuhr, die sich einerseits ähnlich verhält wie ein Kuckuck, andererseits aber gar kein Tier ist – dann haben wir ein Problem. Soll der Befrager auch die Kuckucksuhr befragen können, anders gesagt: soll die Kuckucksuhr den Kuckuck substituieren können, dann bleibt uns fast nichts übrig als so zu entwerfen:



Wenn wir nicht die Klassenvererbung, sondern die Schnittstellenvererbung verwenden, dann sieht die Situation so aus:

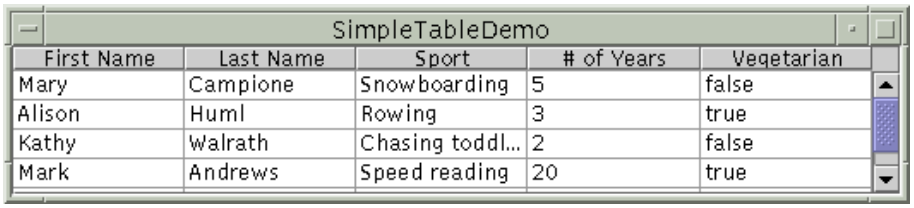
Alle Tiere erfüllen (und die Klassen implementieren) die Schnittstelle `Talkative` (= geschwätzig, gesprächig, redselig) und der Befrager verwendet diese Schnittstelle, um die Tiere sprechen zu lassen. Kommt nun die Kucucksuhr hinzu, besteht nicht der Zwang, die „unnatürliche“ Vererbung zu machen, sondern wir müssen nur in der Kuckucksuhr auch die Schnittstelle `Talkative` implementieren. Nun kann man die Kuckucksuhr an der Stelle eines Kuckucks *verwenden*, obwohl eine Kuckucksuhr *nicht* eine Art Kuckuck ist!



Entwurfsprinzip

In Java ist diese Trennung von Interface und Implementierung zu einem grundlegenden Prinzip geworden, das durchgängig angewandt wird.

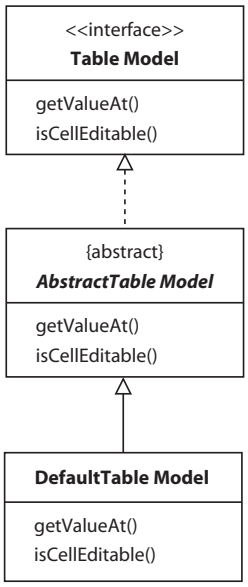
Dazu noch ein Beispiel: eine Tabelle in Java Swing



Eine solche Tabelle wird so aufgeteilt:

- Eine Schnittstelle, die die Methoden zur Verwendung der Tabelle definiert
- Eine abstrakte Klasse, die dieses Interface implementiert und eine Defaultimplementierung oder Teile davon enthält, die von anderen Implementierungen durch Klassenvererbung genutzt werden können

- Eine konkrete Klasse, die eine komplette Implementierung hat, die man „out of the box“ verwenden kann.



Interface
definiert API

Abstrakte Klasse
bietet
Defaultimplementierung

Konkrete Klasse
bietet
benutzbare Implementierung
(out of the box)

Peter Coad hat folgende Regeln aufgestellt in Bezug auf die Frage, ob man eher Klassenvererbung (man sagt auch *Subclassing*) oder eher Schnittstellenvererbung (auch *Subtyping*) verwenden soll.

Subclassing soll man nur verwenden, wenn die folgenden Bedingungen alle erfüllt sind:

- Unterklasse „ist eine spezielle Art von“ Oberklasse und **nicht** Klasse „spielt die Rolle von“ ...
- Das Objekt muss niemals in ein Objekt einer anderen Klasse umgewandelt werden.
- Eine Oberklasse soll erweitert und nicht überschrieben oder eingeschränkt werden.

Subtyping soll man verwenden, wenn die obigen Bedingungen nicht erfüllt sind. Insbesondere die Situation „Rolle“ tritt häufig auf, weshalb in der Regel die Schnittstellenvererbung gewählt werden sollte.

Beispiel: Kunde als eine „Art“ von Person? Oder besser: Eine Person tritt in der „Rolle“ eines Kunden auf?

Als Fazit eine Gegenüberstellung:

- Klassenvererbung
  - kostenlose Implementierung
  - A-priori-Wiederverwendung (statisch)
  - fokussiert auf Struktur & Verhalten
  - wird leicht überstrapaziert
  - Seiteneffekte :Aufweichung der Kapselung (Problem der zerbrechlichen Basisklasse)
- Schnittstellenvererbung
  - schafft Flexibilität
  - Ad-hoc-Wiederverwendung (dynamisch)
  - delegiert Dienste (Verhalten)
  - fördert den Entwurf mit Schnittstellen - Abstraktion
  - Prinzip vieler Design Patterns
  - flache Klassenhierarchien
  - kaum Seiteneffekte: Erhalt der Kapselung
  - höherer Lernaufwand: dynamische Objektstrukturen sind schwieriger zu verstehen

Zum Abschluss dieser Vorlesung wieder ein Lektüretipp: Robert C. Martin „The Liskov Substitution Principle“, http://www.objectmentor.com/resources/articles/lsp.pdf

Burkhardt Renz
Technische Hochschule Mittelhessen
Fachbereich MNI
Wiesenstr. 14
D-35390 Gießen

Rev 3.0 – 10. April 2012