

Wiederholung: Klassen und Objekte

Hands-On: Modellieren Sie ein Auto mit Motor!

- Der `Motor` besitzt eine public-Methode `power()`, die auf dem Bildschirm “TuckTuckTuck” ausgibt.
- Jedes `Auto` hält als private-Attribut einen `Motor`.
- Der `Motor` jedes Autos wird bei Konstruktion des Autos initialisiert.
- Das `Auto` hat eine public-Methode `drive()`, welche den Motor startet (= `power()` aufruft) und “Let us go!” ausgibt.

Wiederholung: Klassen und Objekte

Hands-On: Modellieren Sie ein Auto mit Motor!

- Der `Motor` besitzt eine public-Methode `power()`, die auf dem Bildschirm "TuckTuckTuck" ausgibt.
- Jedes `Auto` hält als private-Attribut einen `Motor`.
- Der `Motor` jedes Autos wird bei Konstruktion des Autos initialisiert.
- Das `Auto` hat eine public-Methode `drive()`, welche den Motor startet (=power() aufruft) und "Let us go!" ausgibt.

```
class Motor:
    def power(self):
        print "TuckTuckTuck"

class Auto:
    def __init__(self):
        self.__motor=Motor()

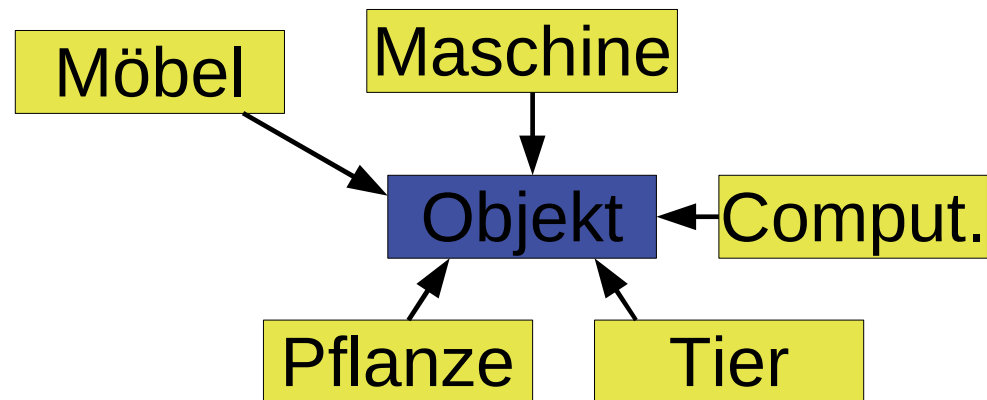
    def drive(self):
        self.__motor.power()
        print "Let us go!"
```

```
a1 = Auto()
a1.drive()
```

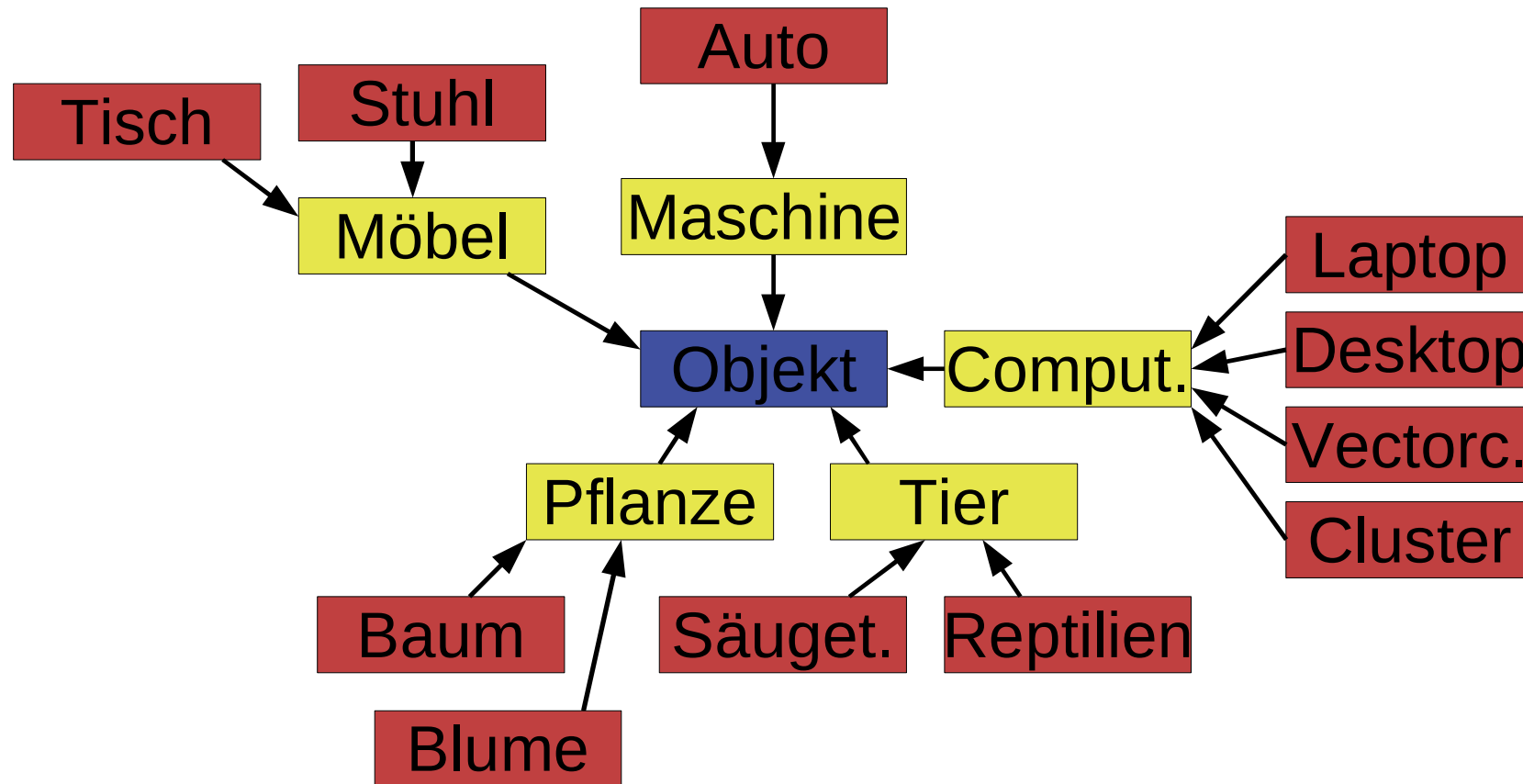
Vererbung in der Realität

Objekt

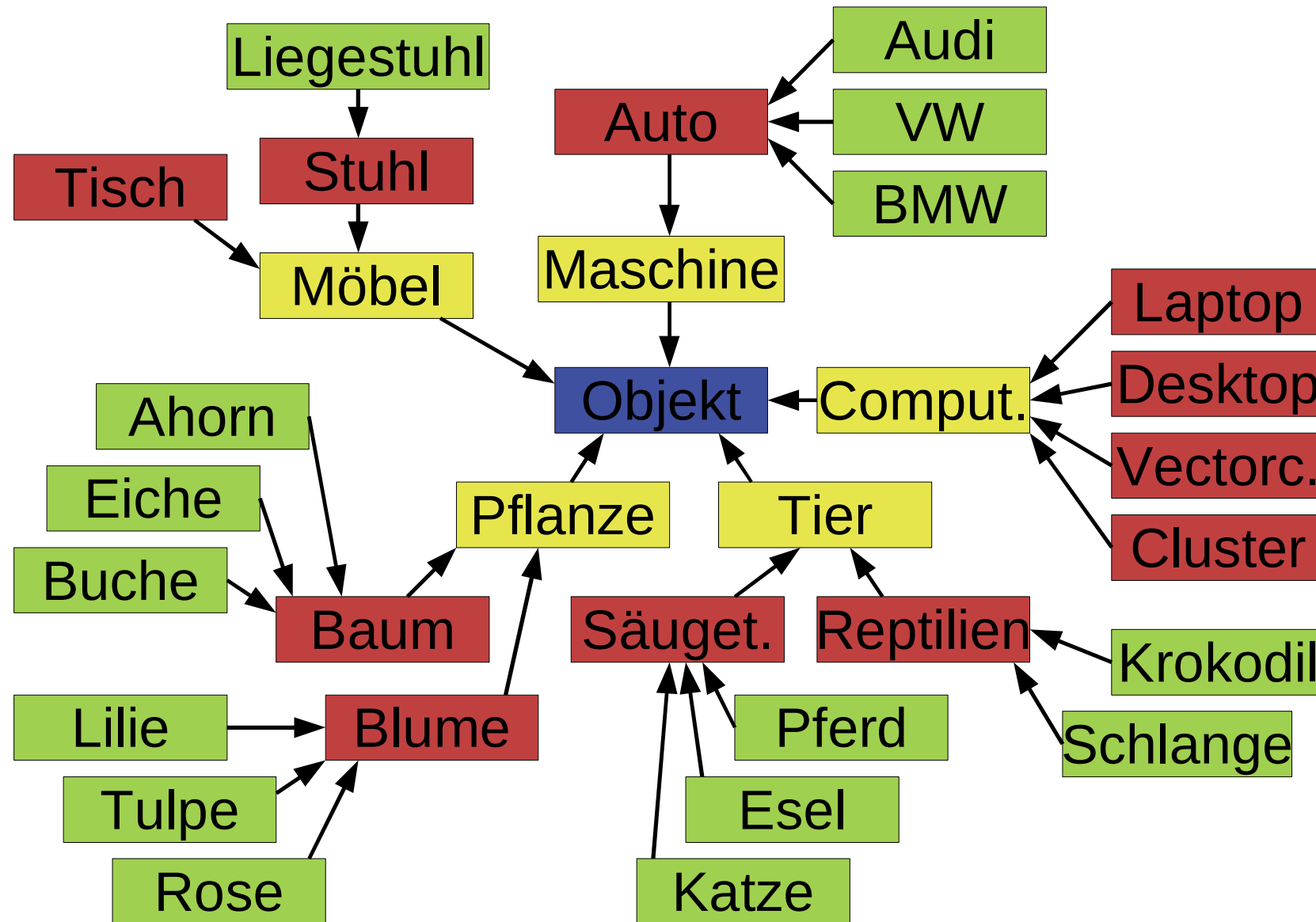
Vererbung in der Realität



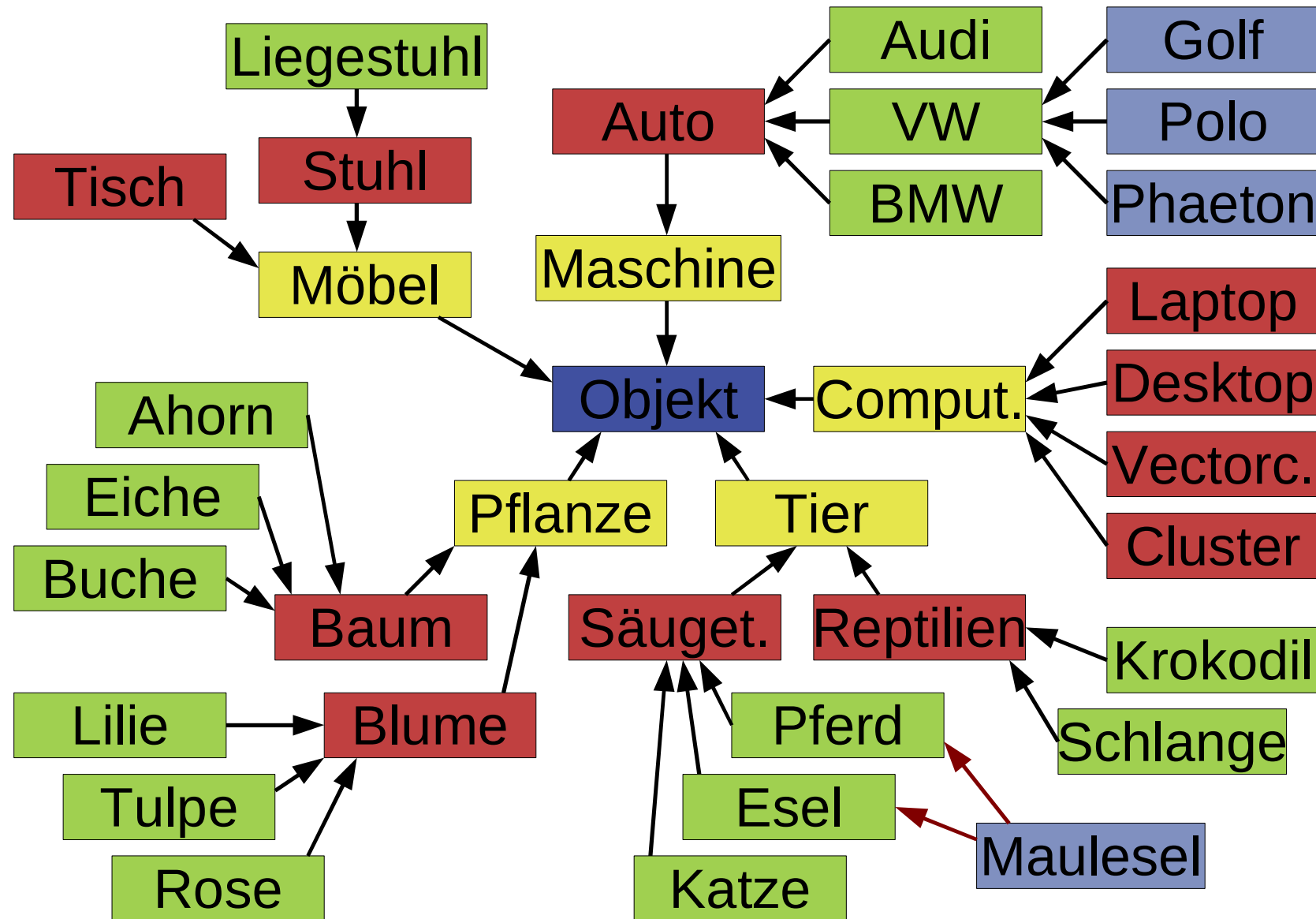
Vererbung in der Realität



Vererbung in der Realität



Vererbung in der Realität



Rückblick: Erstes Klassenbeispiel

```
class Student:
    "Einfache Beschreibung eines Studierenden"
    def __init__(self, name): # Konstruktor
        self.__name = name

    def getName(self):        # Methode
        return self.__name

person1 = Student("Alex")
person2 = Student("Michaela")
print "Hier kommt " + person1.getName()
```


Vererbung

- Klassen können von einer *Basisklasse / Oberklasse* erben
- Dadurch sind sie *abgeleitete Klasse / Unterklasse* der Basisklasse
- Alle Klassen können spezialisiert werden
- Alle Klassen zusammen bilden eine Klassenhierarchie
- Methoden der Oberklasse können übernommen oder neu definiert (überschrieben) werden

```
class Werkstudent(Student):
    def __init__(self, name, thema):
        Student.__init__(self, name)
        self.thema = thema
    def printInfo(self):
        print "%s arbeitet an: %s" % \
            (self.getName(), self.thema)

person3 = Werkstudent("Stefan", "Tabellenkalkulation")
person3.printInfo()
```

Vererbung: “Ist ein”

- Vererbung impliziert ein **“Ist ein”**-Verhältnis zwischen abgeleiteter und Basisklasse
 - Werkstudent “Ist ein” Student
 - Auto “Ist ein”e Maschine

Vererbung: “Ist ein”

- Vererbung impliziert ein **“Ist ein”**-Verhältnis zwischen abgeleiteter und Basisklasse
 - Werkstudent “Ist ein” Student
 - Auto “Ist ein”e Maschine
- Saubere Klassenmodellierung vs. intuitive/verbale Beschreibung

Vererbung: “Ist ein”

- Vererbung impliziert ein “**ist ein**”-Verhältnis zwischen abgeleiteter und Basisklasse
 - Werkstudent “ist ein” Student
 - Auto “ist ein”e Maschine
- Saubere Klassenmodellierung vs. intuitive/verbale Beschreibung
- Beispiel: Vögel

```
class Vogel:
    def fly(self):
        print "Ich_kann_fliegen"

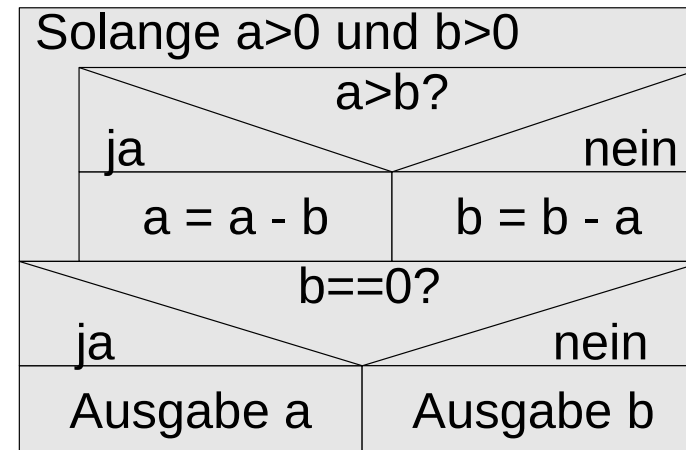
class Pinguin(Vogel):
    def swim(self):
        print "Ich_schwimme"

p = Pinguin()
### argh, Pinguine koennen nicht fliegen!!!
p.fly()
```

Modellierung mit UML

Flussdiagramme

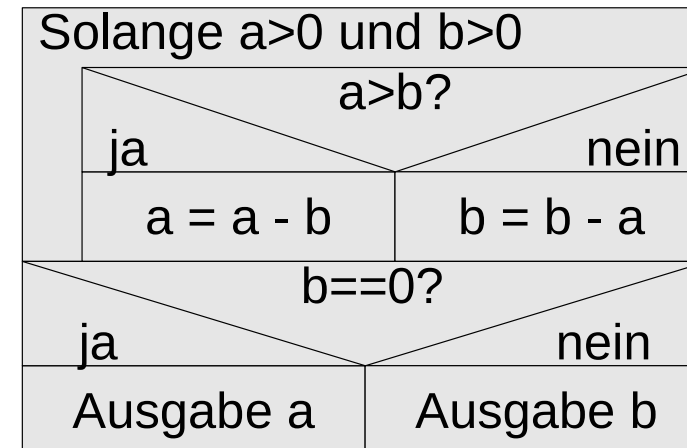
- Nur geeignet für kleine Algorithmen
- Keine Darstellung von Klassenhierarchien
- Keine Darstellung von Datenabhängigkeiten
- ...



Modellierung mit UML

Flussdiagramme

- Nur geeignet für kleine Algorithmen
- Keine Darstellung von Klassenhierarchien
- Keine Darstellung von Datenabhängigkeiten
- ...



UML: Unified Modeling Language

- „visuelle“ Modellierungssprache
- Sprache unterstützt unterschiedliche Diagrammtypen
- Software wird über diese Diagramme/Modelle spezifiziert
- Kann als Basis für die Dokumentation dienen
- Automatische Code-Generierung möglich
- Auch für sonstige betriebliche Abläufe geeignet

Strukturdiagramme

- Klassendiagramm
- Objektdiagramm
- Kompositionsstrukturdiagramm
- Komponentendiagramm
- Verteilungsdiagramm
- Paketdiagramm
- Profildiagramm

Verhaltensdiagramme

- Aktivitätsdiagramm
- Anwendungsfalldiagramm (Use-Cases)
- Interaktionsübersichtsdiagramm
- Kommunikationsdiagramm
- Sequenzdiagramm
- Zeitverlaufdiagramm
- Zustandsdiagramm

Strukturdiagramme

- **Klassendiagramm**
- Objektdiagramm
- Kompositionsstrukturdiagramm
- Komponentendiagramm
- Verteilungsdiagramm
- Paketdiagramm
- Profildiagramm

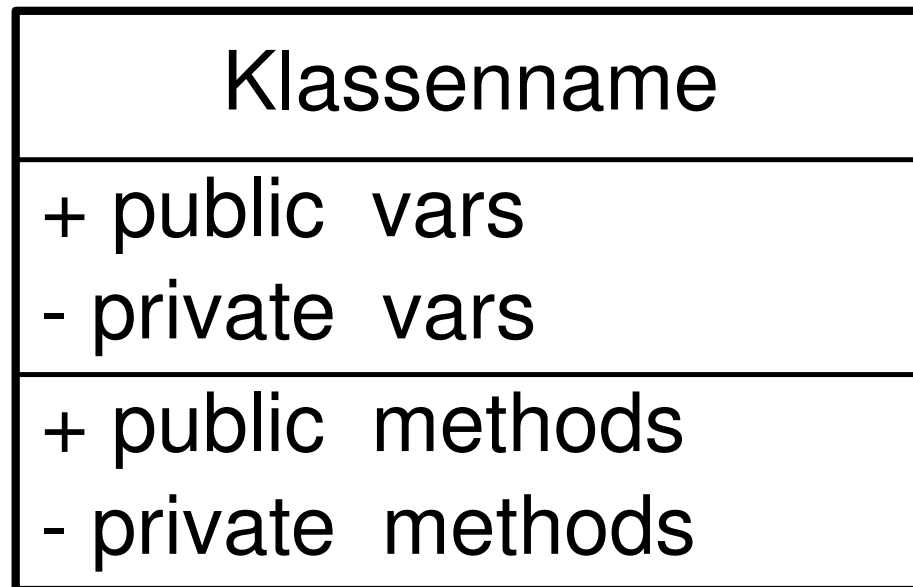
Verhaltensdiagramme

- Aktivitätsdiagramm
- Anwendungsfalldiagramm (Use-Cases)
- Interaktionsübersichtsdiagramm
- Kommunikationsdiagramm
- Sequenzdiagramm
- Zeitverlaufdiagramm
- Zustandsdiagramm

Klassendiagramm

Einzelne Klasse

- Klasse dargestellt durch Rechteck
- Horizontale Unterteilung in drei Bereiche
- Oberster Bereich: Klassenname
- Mittlerer Bereich: Attribute/Variablen
- Unterer Bereich: Methoden



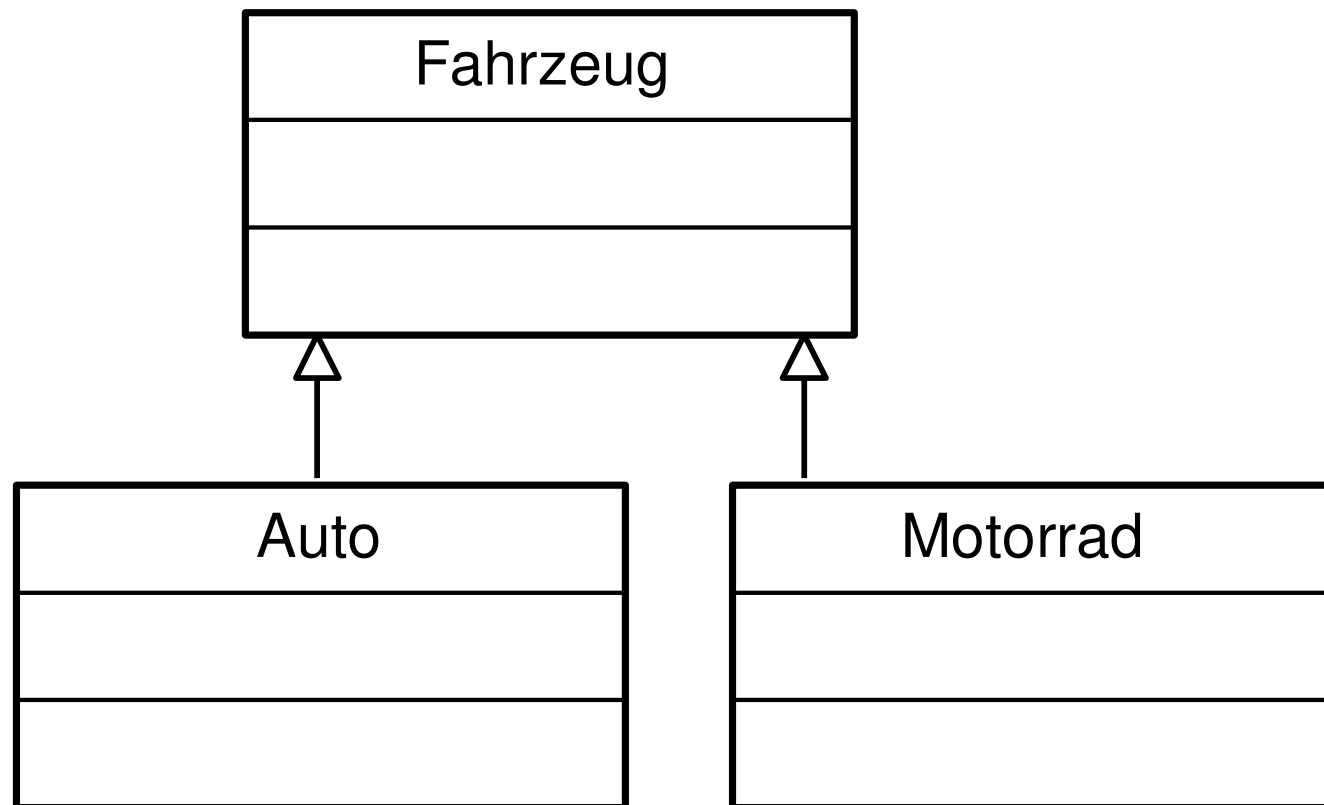
Syntax der Klassenbeschreibung

- Name von Klasse/Variablen/Methoden frei wählbar
- +: public Variable/Methode (optional)
- -: private Variable/Methode (optional)
- Für Variablen kann der Typ angegeben werden (nach :)
- Für Methoden kann der Rückgabewert angegeben werden (nach :)

Auto
+ farbe: str - nummer: int
+ drive()

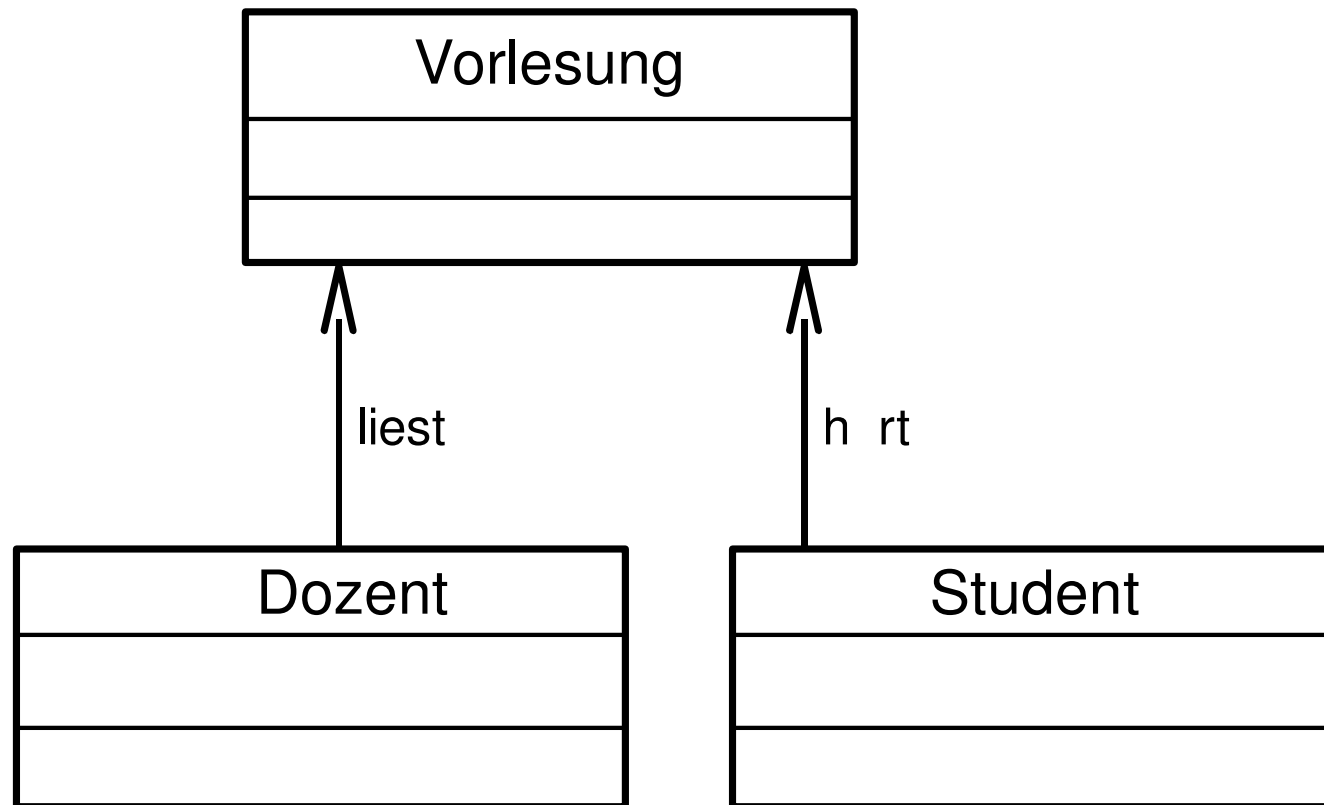
Vererbung

- Darstellung durch Pfeil mit „geschlossener“ Spitze
- „ist ein“- Beziehung
- Pfeil von spezialisierter Klasse zur Vaterklasse



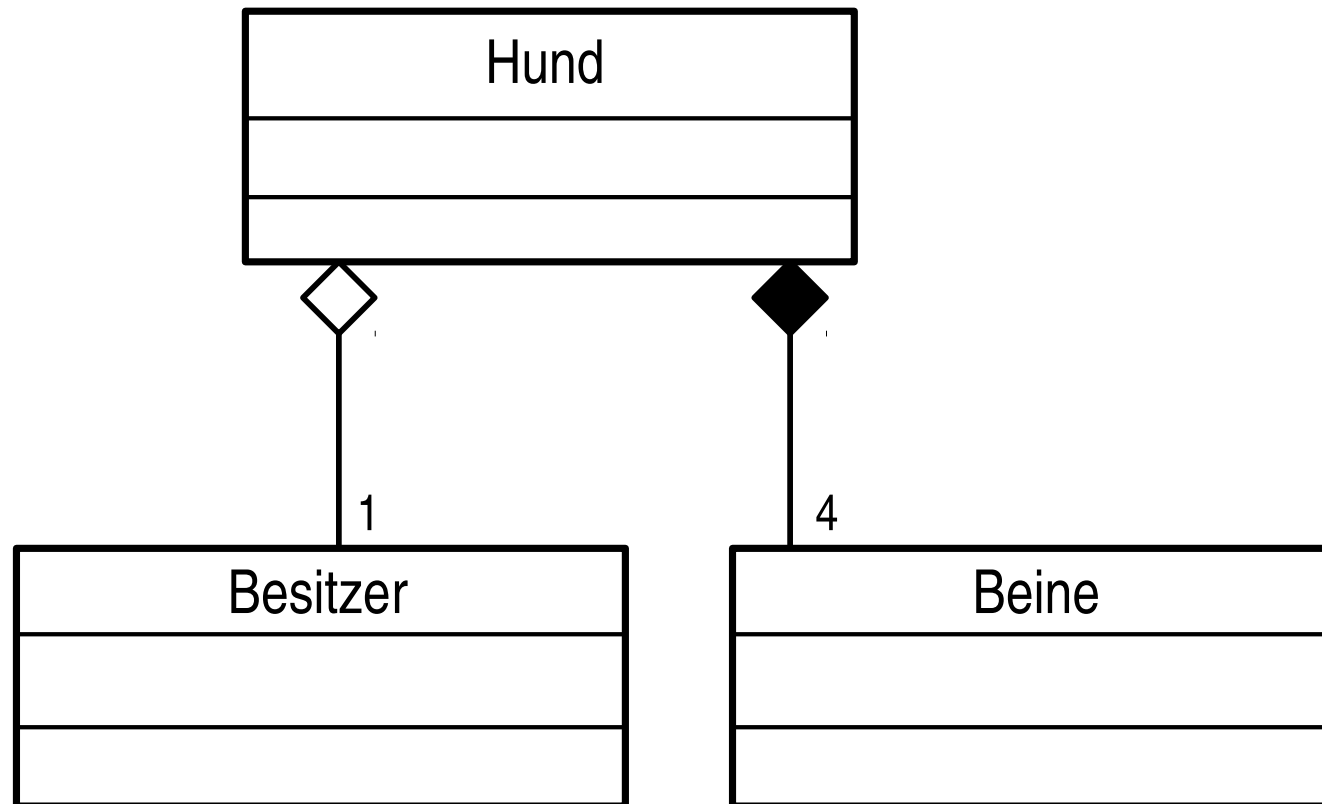
Assoziation

- Semantischer Zusammenhang zwischen Klassen
- Beschreibung der Assoziation neben dem Pfeil



Aggregation und Komposition

- Aggregation (leere Raute): „hat ein“
- Komposition (volle Raute): „enthält ein“
- Angabe der Anzahl möglich



Hands-On: Stellen Sie das Verhältnis von `Auto`/`Motor` in UML dar!

```
class Motor:
    def power(self):
        print "TuckTuckTuck"

class Auto:
    def __init__(self):
        self.__motor=Motor()

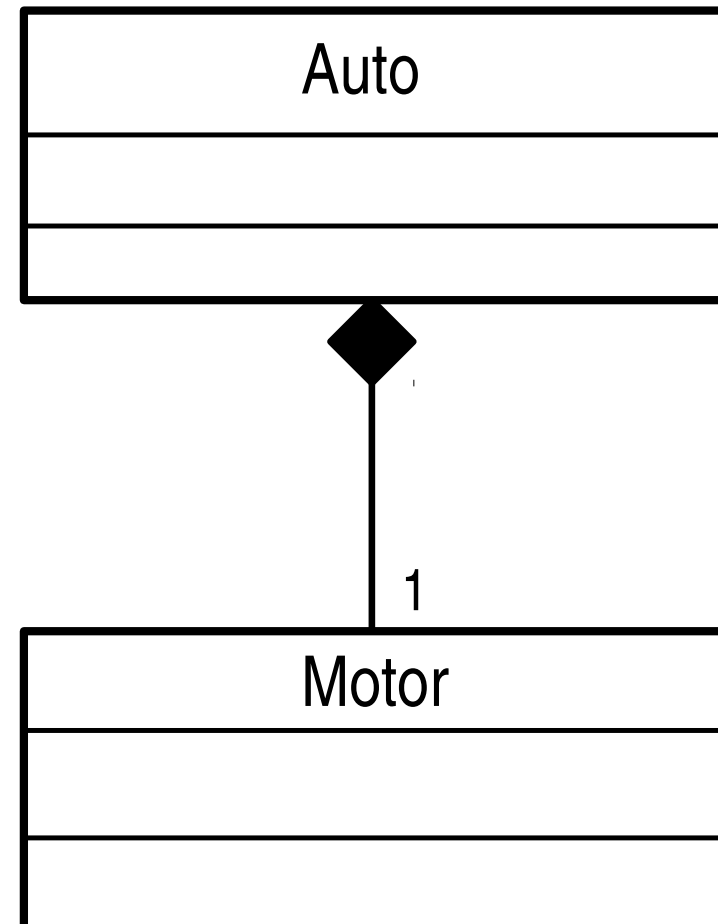
    def drive(self):
        self.__motor.power()
        print "Let us go!"
```

Hands-On: Stellen Sie das Verhältnis von `Auto`/`Motor` in UML dar!

```
class Motor:
    def power(self):
        print "TuckTuckTuck"

class Auto:
    def __init__(self):
        self.__motor=Motor()

    def drive(self):
        self.__motor.power()
        print "Let us go!"
```



Teil VII

Reguläre Ausdrücke

Was wir uns (vielleicht) schon immer gefragt haben:

- Wie funktioniert Suche nach einer Zeichenfolge in einem Text?
- Wie wird Auto-Vervollständigung in Entwicklungsumgebungen realisiert?
- Woher weiß ein email client, ob eine Emailadresse valide ist?

Was wir uns (vielleicht) schon immer gefragt haben:

- Wie funktioniert Suche nach einer Zeichenfolge in einem Text?
- Wie wird Auto-Vervollständigung in Entwicklungsumgebungen realisiert?
- Woher weiß ein email client, ob eine Emailadresse valide ist?

⇒ **Da sind reguläre Ausdrücke im Spiel**

reguläre Ausdrücke = regular expressions = regex

Was wir uns (vielleicht) schon immer gefragt haben:

- Wie funktioniert Suche nach einer Zeichenfolge in einem Text?
- Wie wird Auto-Vervollständigung in Entwicklungsumgebungen realisiert?
- Woher weiß ein email client, ob eine Emailadresse valide ist?

⇒ **Da sind reguläre Ausdrücke im Spiel**

reguläre Ausdrücke = regular expressions = regex

Unterbau: Formale Sprachen (Theoretische Informatik)

Formale Sprachen

Syntax

- Jede Sprache (natürliche und formale) hat Regeln.
- Für einen Text lässt sich feststellen, ob er zu einer Sprache gehört:
 - Deutsch: Die Fans befinden sich im Stadion.
 - Python: `print "Hello_World"`
- Eine Grammatik beschreibt, nach welchen Regeln eine Sprache syntaktisch aufgebaut ist.
- Bei einer Programmiersprache prüft der Compiler/Interpreter die Syntax.

Formale Sprachen

Syntax

- Jede Sprache (natürliche und formale) hat Regeln.
- Für einen Text lässt sich feststellen, ob er zu einer Sprache gehört:
 - Deutsch: Die Fans befinden sich im Stadion.
 - Python: `print "Hello World"`
- Eine Grammatik beschreibt, nach welchen Regeln eine Sprache syntaktisch aufgebaut ist.
- Bei einer Programmiersprache prüft der Compiler/Interpreter die Syntax.

Semantik

- Auch bei grammatikalisch korrektem Aufbau kann ein Text sinnlos sein
 - Deutsch: Das Stadion befindet sich in den Fans.
 - Python: `print "sin(x) = %f" % cos(x)`
- Eine Semantik-Korrektur für Programmiersprachen gibt es noch nicht.

Formale Grammatik

- Eine formale Grammatik ist ein 4-Tupel $G = (V, \Sigma, P, S)$

Formale Grammatik

- Eine formale Grammatik ist ein 4-Tupel $G = (V, \Sigma, P, S)$
- V ist eine endliche Menge, das Vokabular

Formale Grammatik

- Eine formale Grammatik ist ein 4-Tupel $G = (V, \Sigma, P, S)$
- V ist eine endliche Menge, das Vokabular
- $\Sigma \subset V$ ist das Alphabet, die Elemente heißen Terminalsymbole („Student“, „Vorlesung“; a, b, ...)

Formale Grammatik

- Eine formale Grammatik ist ein 4-Tupel $G = (V, \Sigma, P, S)$
- V ist eine endliche Menge, das Vokabular
- $\Sigma \subset V$ ist das Alphabet, die Elemente heißen Terminalsymbole („Student“, „Vorlesung“; a, b, \dots)
- $N = V \setminus \Sigma$ sind die Nichtterminalsymbole/Variablen ($\langle \text{Subjekt} \rangle$, $\langle \text{Verb} \rangle$; A, B, \dots)

Formale Grammatik

- Eine formale Grammatik ist ein 4-Tupel $G = (V, \Sigma, P, S)$
- V ist eine endliche Menge, das Vokabular
- $\Sigma \subset V$ ist das Alphabet, die Elemente heißen Terminalsymbole („Student“, „Vorlesung“; a, b, \dots)
- $N = V \setminus \Sigma$ sind die Nichtterminalsymbole/Variablen ($\langle \text{Subjekt} \rangle$, $\langle \text{Verb} \rangle$; A, B, \dots)
- X^* ist die Kleensche Hülle der Menge X . Enthält beliebige Konkatenationen von Elementen aus der Menge.

Formale Grammatik

- Eine formale Grammatik ist ein 4-Tupel $G = (V, \Sigma, P, S)$
- V ist eine endliche Menge, das Vokabular
- $\Sigma \subset V$ ist das Alphabet, die Elemente heißen Terminalsymbole („Student“, „Vorlesung“; a, b, \dots)
- $N = V \setminus \Sigma$ sind die Nichtterminalsymbole/Variablen ($\langle \text{Subjekt} \rangle$, $\langle \text{Verb} \rangle$; A, B, \dots)
- X^* ist die Kleensche Hülle der Menge X . Enthält beliebige Konkatenationen von Elementen aus der Menge.
- $P \subset (V^* \setminus \Sigma^*) \times V^*$ ist die Menge der Produktionsregeln. Überführung eines Wortes/Texts R , das mindestens ein Nichtterminal enthält ($R \in V^* \setminus \Sigma^*$) in ein beliebiges Wort $Q \in V^*$

$\langle \text{Subj} \rangle \langle \text{Verb} \rangle \langle \text{Obj} \rangle \quad \rightarrow \quad \text{Der Student} \langle \text{Verb} \rangle \langle \text{Obj} \rangle$
 $AB \quad \rightarrow \quad AaB$

Formale Grammatik

- Eine formale Grammatik ist ein 4-Tupel $G = (V, \Sigma, P, S)$
- V ist eine endliche Menge, das Vokabular
- $\Sigma \subset V$ ist das Alphabet, die Elemente heißen Terminalsymbole („Student“, „Vorlesung“; a, b, \dots)
- $N = V \setminus \Sigma$ sind die Nichtterminalsymbole/Variablen ($\langle \text{Subjekt} \rangle$, $\langle \text{Verb} \rangle$; A, B, \dots)
- X^* ist die Kleensche Hülle der Menge X . Enthält beliebige Konkatenationen von Elementen aus der Menge.
- $P \subset (V^* \setminus \Sigma^*) \times V^*$ ist die Menge der Produktionsregeln. Überführung eines Wortes/Texts R , das mindestens ein Nichtterminal enthält ($R \in V^* \setminus \Sigma^*$) in ein beliebiges Wort $Q \in V^*$

$\langle \text{Subj} \rangle \langle \text{Verb} \rangle \langle \text{Obj} \rangle \quad \rightarrow \quad \text{Der Student} \langle \text{Verb} \rangle \langle \text{Obj} \rangle$
 $AB \quad \rightarrow \quad AaB$

- $S \in (V \setminus \Sigma)$ ist das Startsymbol

Chomsky-Hierarchie

- Eingeführt von Noam Chomsky
- Eine Hierarchie von Klassen formaler Grammatiken

Typ 0: unbeschränkte Grammatik

- enthält alle formalen Grammatiken
- zugehörige Sprache wird von Turingmaschine akzeptiert

Typ 1: kontextsensitive Grammatik

- Produktionsregeln der Form $\alpha B \gamma \rightarrow \alpha \beta \gamma$ (B Nichtterminal, griechische Buchstaben Worte aus V^*)
- Sprache wird von linear beschränkter Turingmaschine erkannt

Typ 2: kontextfreie Grammatik

- Produktionsregeln der Form $A \rightarrow \alpha$
- erkannt von Kellerautomat, Programmiersprachen sind Typ 2

Typ 3: reguläre Grammatik

- Produktionsregeln der Form $A \rightarrow a$ und $A \rightarrow aB$
- erkannt durch endliche Automaten / reguläre Ausdrücke

Reguläre Grammatik der Exponentialzahlen

- Einschränkende Annahmen:
 - Beide Vorzeichen (Anfang+Exponent) zwingend anzugeben
 - Genau eine Vor- und mindestens eine Nachkommastelle
- $G = (V, \Sigma, P, S)$
- $V = \Sigma \cup N$
- $N = S, M, N1, P, N2, E1, E2$
- $\Sigma = -, +, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ., e$
- Platzhalter: $z \in \{0 - 9\}$

Produktionsregeln (rechts Beispiel: -3.81e-11)

S	->	-M	S	->	+M		S	->	-M		
M	->	zP					->	-3P			
P	->	.N1					->	-3.N1			
N1	->	zN2					->	-3.8N2			
N2	->	zN2	N2	->	eE1		->	-3.81N2	->	-3.81eE1	
E1	->	-E2	E1	->	+E2		->	-3.81e-E2			
E2	->	zE2	E2	->	z		->	-3.81e-1E2	->	-3.81e-11	

Reguläre Sprachen und Ausdrücke???

Die Menge der regulären Sprachen über einem Alphabet Σ und die zugehörigen regulären Ausdrücke sind rekursiv definiert:

- Die leere Sprache \emptyset ist eine reguläre Sprache und der zugehörige reguläre Ausdruck ist \emptyset .
- Der leere String $\{\wedge\}$ ist eine reguläre Sprache und der zugehörige reguläre Ausdruck ist \wedge .

Reguläre Sprachen und Ausdrücke???

Die Menge der regulären Sprachen über einem Alphabet Σ und die zugehörigen regulären Ausdrücke sind rekursiv definiert:

- Die leere Sprache \emptyset ist eine reguläre Sprache und der zugehörige reguläre Ausdruck ist \emptyset .
- Der leere String $\{\wedge\}$ ist eine reguläre Sprache und der zugehörige reguläre Ausdruck ist \wedge .
- Für jedes a in Σ , ist die einelementige Sprache $\{ a \}$ eine reguläre Sprache und der zugehörige reguläre Ausdruck ist a

Reguläre Sprachen und Ausdrücke???

Die Menge der regulären Sprachen über einem Alphabet Σ und die zugehörigen regulären Ausdrücke sind rekursiv definiert:

- Die leere Sprache \emptyset ist eine reguläre Sprache und der zugehörige reguläre Ausdruck ist \emptyset .
- Der leere String $\{\wedge\}$ ist eine reguläre Sprache und der zugehörige reguläre Ausdruck ist \wedge .
- Für jedes a in Σ , ist die einelementige Sprache $\{a\}$ eine reguläre Sprache und der zugehörige reguläre Ausdruck ist a
- Wenn A und B reguläre Sprachen sind mit zugehörigen regulären Ausdrücken r_1 and r_2 , dann
 - ist $A \cup B$ (Vereinigung) eine reguläre Sprache und der zugehörige reguläre Ausdruck ist $(r_1|r_2)$
 - ist AB (Verknüpfung) eine reguläre Sprache und der zugehörige reguläre Ausdruck ist (r_1r_2)
 - ist A^* (Kleene star) eine reguläre Sprache und der zugehörige reguläre Ausdruck ist (r_1^*)

Regulärer Ausdruck für Exponentialzahlen

- Es gibt unendlich viele Sprachen über jedem endlichen nichtleeren Alphabet
- z.B. für $\Sigma = \{a\}$ sind Sprachen: $\{\}$, $\{a\}$, $\{aa\}$, $\{a, aa\}$, $\{aaa\}$, ...

Regulärer Ausdruck für Exponentialzahlen

- Es gibt unendlich viele Sprachen über jedem endlichen nichtleeren Alphabet
- z.B. für $\Sigma = \{a\}$ sind Sprachen: $\{\}$, $\{a\}$, $\{aa\}$, $\{a, aa\}$, $\{aaa\}$, ...
- Wir wollen zeigen, dass sich die Sprache der Exponentialzahlen und der zugehörige reguläre Ausdruck rekursiv herleiten lassen

Regulärer Ausdruck für Exponentialzahlen

- Es gibt unendlich viele Sprachen über jedem endlichen nichtleeren Alphabet
- z.B. für $\Sigma = \{a\}$ sind Sprachen: $\{\}, \{a\}, \{aa\}, \{a, aa\}, \{aaa\}, \dots$
- Wir wollen zeigen, dass sich die Sprache der Exponentialzahlen und der zugehörige reguläre Ausdruck rekursiv herleiten lassen
- Alphabet: $\{-, +, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ., e\}$
- Einelementige Sprachen: $S_- = \{-\}, S_+ = \{+\}, S_0 = \{0\}, S_1 = \{1\}, \dots, S_. = \{.\}, S_e = \{e\}$
- zugehörige regulären Ausdrücke:
 $r_- = -, r_+ = +, r_0 = 0, r_1 = 1, \dots, r_. = ., r_e = e$

Regulärer Ausdruck für Exponentialzahlen

- Es gibt unendlich viele Sprachen über jedem endlichen nichtleeren Alphabet
- z.B. für $\Sigma = \{a\}$ sind Sprachen: $\{\}, \{a\}, \{aa\}, \{a, aa\}, \{aaa\}, \dots$
- Wir wollen zeigen, dass sich die Sprache der Exponentialzahlen und der zugehörige reguläre Ausdruck rekursiv herleiten lassen
- Alphabet: $\{-, +, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ., e\}$
- Einelementige Sprachen: $S_- = \{-\}, S_+ = \{+\}, S_0 = \{0\}, S_1 = \{1\}, \dots, S_. = \{.\}, S_e = \{e\}$
- zugehörige regulären Ausdrücke:
 $r_- = -, r_+ = +, r_0 = 0, r_1 = 1, \dots, r_. = ., r_e = e$
- Vereinigung zweier Sprachen:
 $\{a, b\} \cup \{0, 1, 2\} = \{a, b, 0, 1, 2\} \Rightarrow |S_1| + |S_2|$ Elemente
- Verknüpfung zweier Sprachen:
 $\{a, b\}\{0, 1, 2\} = \{a0, a1, a2, b0, b1, b2\} \Rightarrow |S_1| \cdot |S_2|$ Elemente
- Kleensche Hülle einer Sprache:
 $\{0, 1\}^* = \{\emptyset, 0, 1, 00, 01, 10, 11, 000, \dots\} \Rightarrow \infty$ Elemente

Erzeugung weiterer Sprachen aus den einelementigen

Operation	Sprache	Ausdruck
$S_- \cup S_+$	$S_A = \{-, +\}$	$r_A = (- +)$
$S_0 \cup S_1 \cup \dots$	$S_B = \{0, 1, \dots, 9\}$	$r_B = (0 1 \dots 9) = (0 - 9)$
$S_A S_B S_.$	$S_C = \{-0., \dots, -9., \dots\}$	$r_C = (- +)(0 - 9).$
$S_C S_B$	$S_D = \{-0.0, -0.1, \dots\}$	$r_D = (- +)(0 - 9).(0 - 9)$
$S_D S_B^*$	$S_E = \{-0.0, -0.00, \dots\}$	$r_E = (- +)(0 - 9).(0 - 9)(0 - 9)^*$ $r_E = (- +)(0 - 9).(0 - 9)^+$
$S_E S_e$	$S_F = \{-0.0e, \dots\}$	$r_F = (- +)(0 - 9).(0 - 9)^+ e$
$S_F S_A$	$S_G = \{-0.0e-, \dots\}$	$r_G = (- +)(0 - 9).(0 - 9)^+ e(- +)$
$S_F S_B$	$S_H = \{-0.0e - 0, \dots\}$	$r_H = (- +)(0 - 9).(0 - 9)^+ e(- +)(0 - 9)$
$S_H S_B^*$	$S_I = \{-0.0e - 00, \dots\}$	$r_I = (- +)(0 - 9).(0 - 9)^+ e(- +)(0 - 9)^+$

Erzeugung weiterer Sprachen aus den einelementigen

Operation	Sprache	Ausdruck
$S_- \cup S_+$	$S_A = \{-, +\}$	$r_A = (- +)$
$S_0 \cup S_1 \cup \dots$	$S_B = \{0, 1, \dots, 9\}$	$r_B = (0 1 \dots 9) = (0 - 9)$
$S_A S_B S_.$	$S_C = \{-0., \dots, -9., \dots\}$	$r_C = (- +)(0 - 9).$
$S_C S_B$	$S_D = \{-0.0, -0.1, \dots\}$	$r_D = (- +)(0 - 9).(0 - 9)$
$S_D S_B^*$	$S_E = \{-0.0, -0.00, \dots\}$	$r_E = (- +)(0 - 9).(0 - 9)(0 - 9)^*$ $r_E = (- +)(0 - 9).(0 - 9)^+$
$S_E S_e$	$S_F = \{-0.0e, \dots\}$	$r_F = (- +)(0 - 9).(0 - 9)^+ e$
$S_F S_A$	$S_G = \{-0.0e-, \dots\}$	$r_G = (- +)(0 - 9).(0 - 9)^+ e(- +)$
$S_F S_B$	$S_H = \{-0.0e - 0, \dots\}$	$r_H = (- +)(0 - 9).(0 - 9)^+ e(- +)(0 - 9)$
$S_H S_B^*$	$S_I = \{-0.0e - 00, \dots\}$	$r_I = (- +)(0 - 9).(0 - 9)^+ e(- +)(0 - 9)^+$

- S_I ist genau die Sprache der Exponentialzahlen, für die bereits eine Grammatik erstellt wurde
- $r_I = (-|+)(0 - 9).(0 - 9)^+ e(-|+)(0 - 9)^+$ ist der zugehörige reguläre Ausdruck

Erzeugung weiterer Sprachen aus den einelementigen

Operation	Sprache	Ausdruck
$S_- \cup S_+$	$S_A = \{-, +\}$	$r_A = (- +)$
$S_0 \cup S_1 \cup \dots$	$S_B = \{0, 1, \dots, 9\}$	$r_B = (0 1 \dots 9) = (0 - 9)$
$S_A S_B S_.$	$S_C = \{-0., \dots, -9., \dots\}$	$r_C = (- +)(0 - 9).$
$S_C S_B$	$S_D = \{-0.0, -0.1, \dots\}$	$r_D = (- +)(0 - 9).(0 - 9)$
$S_D S_B^*$	$S_E = \{-0.0, -0.00, \dots\}$	$r_E = (- +)(0 - 9).(0 - 9)(0 - 9)^*$ $r_E = (- +)(0 - 9).(0 - 9)^+$
$S_E S_e$	$S_F = \{-0.0e, \dots\}$	$r_F = (- +)(0 - 9).(0 - 9)^+ e$
$S_F S_A$	$S_G = \{-0.0e-, \dots\}$	$r_G = (- +)(0 - 9).(0 - 9)^+ e(- +)$
$S_F S_B$	$S_H = \{-0.0e - 0, \dots\}$	$r_H = (- +)(0 - 9).(0 - 9)^+ e(- +)(0 - 9)$
$S_H S_B^*$	$S_I = \{-0.0e - 00, \dots\}$	$r_I = (- +)(0 - 9).(0 - 9)^+ e(- +)(0 - 9)^+$

- S_I ist genau die Sprache der Exponentialzahlen, für die bereits eine Grammatik erstellt wurde
- $r_I = (-|+)(0 - 9).(0 - 9)^+ e(-|+)(0 - 9)^+$ ist der zugehörige reguläre Ausdruck
- in Python: `rI = r'[-+] [0-9]\. [0-9]+e [-+] [0-9]+'`

Erzeugung weiterer Sprachen aus den einelementigen

Operation	Sprache	Ausdruck
$S_- \cup S_+$	$S_A = \{-, +\}$	$r_A = (- +)$
$S_0 \cup S_1 \cup \dots$	$S_B = \{0, 1, \dots, 9\}$	$r_B = (0 1 \dots 9) = (0 - 9)$
$S_A S_B S_.$	$S_C = \{-0., \dots, -9., \dots\}$	$r_C = (- +)(0 - 9).$
$S_C S_B$	$S_D = \{-0.0, -0.1, \dots\}$	$r_D = (- +)(0 - 9).(0 - 9)$
$S_D S_B^*$	$S_E = \{-0.0, -0.00, \dots\}$	$r_E = (- +)(0 - 9).(0 - 9)(0 - 9)^*$ $r_E = (- +)(0 - 9).(0 - 9)^+$
$S_E S_e$	$S_F = \{-0.0e, \dots\}$	$r_F = (- +)(0 - 9).(0 - 9)^+ e$
$S_F S_A$	$S_G = \{-0.0e-, \dots\}$	$r_G = (- +)(0 - 9).(0 - 9)^+ e(- +)$
$S_F S_B$	$S_H = \{-0.0e - 0, \dots\}$	$r_H = (- +)(0 - 9).(0 - 9)^+ e(- +)(0 - 9)$
$S_H S_B^*$	$S_I = \{-0.0e - 00, \dots\}$	$r_I = (- +)(0 - 9).(0 - 9)^+ e(- +)(0 - 9)^+$

- S_I ist genau die Sprache der Exponentialzahlen, für die bereits eine Grammatik erstellt wurde
- $r_I = (-|+)(0 - 9).(0 - 9)^+ e(-|+)(0 - 9)^+$ ist der zugehörige reguläre Ausdruck
- in Python: `rI = r'[-+] [0-9]\. [0-9]+e [-+] [0-9]+'`
- noch kürzer: `rI = r'[-+] \d\.\d+e [-+] \d+'`

Reguläre Ausdrücke!!!

What does all this mean to you, as a user? Absolutely nothing. As a user, you don't care if it's regular, nonregular, unregular, irregular, or incontinent. So long as you know what you can expect from it, you know all you need to care about. — Jeffrey Friedl

Reguläre Ausdrücke!!!

What does all this mean to you, as a user? Absolutely nothing. As a user, you don't care if it's regular, nonregular, unregular, irregular, or incontinent. So long as you know what you can expect from it, you know all you need to care about. — Jeffrey Friedl

Wo fast jeder sie schon verwendet hat:

- Suche nach einer Zeichenfolge in einem Text
- Tabulator-Vervollständigung
- Mit * eine Gruppe von Dateien auszuwählen (*.txt)

Reguläre Ausdrücke!!!

What does all this mean to you, as a user? Absolutely nothing. As a user, you don't care if it's regular, nonregular, unregular, irregular, or incontinent. So long as you know what you can expect from it, you know all you need to care about. — Jeffrey Friedl

Wo fast jeder sie schon verwendet hat:

- Suche nach einer Zeichenfolge in einem Text
- Tabulator-Vervollständigung
- Mit * eine Gruppe von Dateien auszuwählen (*.txt)

Einschränkungen

- Für Anfänger sehr kryptisch
- Nach reiner Lehre „Zählen“ nicht möglich (z.B. $a^n b^n$ geht nicht)

Reguläre Ausdrücke!!!

What does all this mean to you, as a user? Absolutely nothing. As a user, you don't care if it's regular, nonregular, unregular, irregular, or incontinent. So long as you know what you can expect from it, you know all you need to care about. — Jeffrey Friedl

Wo fast jeder sie schon verwendet hat:

- Suche nach einer Zeichenfolge in einem Text
- Tabulator-Vervollständigung
- Mit * eine Gruppe von Dateien auszuwählen (*.txt)

Einschränkungen

- Für Anfänger sehr kryptisch
- Nach reiner Lehre „Zählen“ nicht möglich (z.B. $a^n b^n$ geht nicht)

Vorteile

- Sehr nützliches Werkzeug zum Finden von Pattern
- Vergleichbare „manuelle“ Implementierung viel aufwändiger
- Python hat Erweiterungen, die sogar „Zählen“ ermöglichen

Reguläre Ausdrücke in Python

Pattern Syntax

- Reguläre Ausdrücke immer als „raw string“
- Z.B. `r'([0-9]*\.[0-9]*|[0-9]*)'`

Reguläre Ausdrücke in Python

Pattern Syntax

- Reguläre Ausdrücke immer als „raw string“
 - Z.B. `r'([0-9]*\.[0-9]*|[0-9]*)'`
- | | |
|------------|---|
| . | entspricht beliebigem Zeichen außer newline |
| ^ | entspricht dem Beginn eines strings |
| \$ | entspricht dem Ende eines strings |
| * | voriger Ausdruck beliebig oft (incl. Null mal) (gierig) |
| + | voriger Ausdruck beliebig oft (excl. Null mal) (gierig) |
| ? | voriger Ausdruck Null- oder einmal (gierig) |
| *?, +?, ?? | nicht gierige Versionen von *, +, ? |
| {m} | voriger Ausdruck genau m Mal |
| {m,n} | voriger Ausdruck m bis n Mal (gierig) |
| {m,n}? | nicht gierige Version von {m,n} |
| [...] | ein beliebiges Zeichen aus der Menge (...) |
| [^...] | ein beliebiges Zeichen, das nicht in der Menge ist |
| A B | A oder B (A und B sind reguläre Ausdrücke) |
| (...) | speichert den gefundenen Inhalt der Klammern |

Reguläre Ausdrücke - Maskierungszeichen

- Zeichen mit einer speziellen Bedeutung (`.`, `*`, `...`) können mit ihrer eigentlichen Bedeutung durch voranstellen eines `\` verwendet werden, z.B. `\.`, `*`

Reguläre Ausdrücke - Maskierungszeichen

- Zeichen mit einer speziellen Bedeutung (`.`, `*`, `...`) können mit ihrer eigentlichen Bedeutung durch voranstellen eines `\` verwendet werden, z.B. `\.`, `*`
- Normale Maskierungszeichen (`\n`, `\t`, `...`) funktionieren wie erwartet, z.B. `r'\n+'` entspricht einem oder mehreren Zeilenumbrüchen

Reguläre Ausdrücke - Maskierungszeichen

- Zeichen mit einer speziellen Bedeutung (`.`, `*`, ...) können mit ihrer eigentlichen Bedeutung durch voranstellen eines `\` verwendet werden, z.B. `\.`, `*`
- Normale Maskierungszeichen (`\n`, `\t`, ...) funktionieren wie erwartet, z.B. `r'\n+'` entspricht einem oder mehreren Zeilenumbrüchen

<code>\number</code>	entspricht dem n-ten zuvor gefundenen Text (von 1 beginnend)
<code>\d</code>	entspricht <code>[0-9]</code>
<code>\D</code>	entspricht <code>[^0-9]</code>
<code>\s</code>	entspricht beliebigem whitespace (= <code>[\t\n\r\f\v]</code>)
<code>\S</code>	alles außer whitespace
<code>\w</code>	beliebiges alphanumerisches Zeichen (= <code>[a-zA-Z0-9_]</code>)
<code>\W</code>	beliebiges nicht-alphanumerisches Zeichen
<code>\A</code>	entspricht dem Beginn eines strings
<code>\Z</code>	entspricht dem Ende eines strings

Beispiele (1)

Modul `re`:

`findall(patt, string)` – finde alle nicht überlappenden Vorkommen
von `patt` in `string`

`\d` entspricht `[0-9]`

`+` voriger Ausdruck beliebig oft (excl. Null mal) (gierig)

`A|B` A oder B (A und B sind reguläre Ausdrücke)

```
import re
regstr = r'\d+\.\d+|\d+'
s = "12.3/5/4.4;5.7;6"
re.findall(regstr,s)
```

Beispiele (1)

Modul `re`:

`findall(patt, string)` – finde alle nicht überlappenden Vorkommen
von `patt` in `string`

`\d` entspricht `[0-9]`

`+` voriger Ausdruck beliebig oft (excl. Null mal) (gierig)

`A|B` A oder B (A und B sind reguläre Ausdrücke)

```
import re
regstr = r'\d+\.\d+|\d+'
s = "12.3/5/4.4;5.7;6"
re.findall(regstr, s)
```

→ Finde $n \geq 1$ Ziffern, einen “.” und weitere $n \geq 1$ Ziffern
oder
nur $n \geq 1$ Ziffern

Beispiele (1)

Modul `re`:

`findall(patt, string)` – finde alle nicht überlappenden Vorkommen
von `patt` in `string`

`\d` entspricht `[0-9]`

`+` voriger Ausdruck beliebig oft (excl. Null mal) (gierig)

`A|B` A oder B (A und B sind reguläre Ausdrücke)

```
import re
regstr = r'\d+\.\d+|\d+'
s = "12.3/5/4.4;5.7;6"
re.findall(regstr,s)
```

→ Finde $n \geq 1$ Ziffern, einen "." und weitere $n \geq 1$ Ziffern

oder

nur $n \geq 1$ Ziffern

Hands-On: Was passiert für `regstr=r'\d+|\d+\.\d+'`?

Beispiele (1)

Modul `re`:

`findall(patt, string)` – finde alle nicht überlappenden Vorkommen
von `patt` in `string`

`\d` entspricht `[0-9]`

`+` voriger Ausdruck beliebig oft (excl. Null mal) (gierig)

`A|B` A oder B (A und B sind reguläre Ausdrücke)

```
import re
regstr = r'\d+\.\d+|\d+'
s = "12.3/5/4.4;5.7;6"
re.findall(regstr, s)
```

→ Finde $n \geq 1$ Ziffern, einen "." und weitere $n \geq 1$ Ziffern

oder

nur $n \geq 1$ Ziffern

Hands-On: Was passiert für `regstr=r'\d+|\d+\.\d+'`?

Hands-On: Was passiert für `regstr=r'\d+\.\d+|\d+'`?

Beispiele (2)

Modul `re`:

`sub(patt, repl, string)` – ersetze Vorkommen durch `repl`

`\d` entspricht `[0-9]`

`+` voriger Ausdruck beliebig oft (excl. Null mal) (gierig)

`(...)` speichert den gefundenen Inhalt der Klammern

`\number` entspricht dem `n`-ten zuvor gefundenen Text (von 1 beginnend)

```
regstr = r'(\d+\.\d+|\d+)'
re.sub(regstr, r'\1xxx', s)
```

Reguläre Ausdrücke - gierig/greedy

- Was bedeutet gierig/nicht-gierig (bzw. greedy/non-greedy)?
- Beispiel:

```
reg_greedy = r'<.*>'           #match as many chars as possible
s = "<H1>title</H1>"
findall(reg_greedy, s)          #whole '<H1>title</H1>' matched
reg_nongreedy = r'<.*?>'       #match as few chars as possible
findall(reg_nongreedy, s)      #only '<H1>' matched
```

- gierig/greedy: so viele Zeichen wie möglich gematched; nur, wenn das schiefgehen würde, gibt es ein backtracking durch die regex engine

Reguläre Ausdrücke - Match-Objekte

- Manche Funktionen aus `re` geben keine strings, sondern „Match-Objekte“ (m) zurück
- `match(patt, string)` Sucht nach Übereinstimmung am Beginn des strings
- `search(patt, string)` Sucht die erste Übereinstimmung im string
- `finditer(patt, string)` wie `findall`, gibt aber einen iterator zurück
- `m.group(i)` Gibt Übereinstimmungsgruppe i zurück ($i \geq 1$)
- `m.groups()` Gibt Tupel mit allen Übereinstimmungsgruppen zurück

```
import re
s = "Ferien_12/24/2010_-_01/06/2011"
patt = r'(\d+)/(\d+)/(\d+)'
for m in re.finditer(patt, s):
    print("%s.%s.%s" % (m.group(2), m.group(1), m.group(3)))
```

Reguläre Ausdrücke - Rätsel zu Primzahlen

Hands-On: Wieso “berechnet” folgender Code Primzahlen ¿1?

```
import re

def isPrime(p):
    m = re.search(r'^1?$|^(11+?)\1+$', p)
    if(m):
        return False
    else:
        return True

for i in xrange(2,1024):
    if(isPrime('1' * i)):
        print i
```

Reguläre Ausdrücke - Rätsel zu Primzahlen

Hands-On: Wieso “berechnet” folgender Code Primzahlen ¿1?

```
import re

def isPrime(p):
    m = re.search(r'^1?$|^(11+?)\1+$', p)
    if(m):
        return False
    else:
        return True

for i in xrange(2,1024):
    if(isPrime('1' * i)):
        print i
```

→ Hausaufgabe :-)