Grundlagen der Informatik und Programmiersprachen

Prof. Dr. Malte Weiß

Klassifikation von Programmiersprachen

HOCHSCHULE RUHR WEST
UNIVERSITY OF APPLIED SCIENCES

INSTITUT **INFORMATIK**

Klassifikation von Programmiersprachen

- Programmiersprachen lassen sich auf verschiedene Weisen klassifizieren.
- Einteilung nach Generationen:
  - Einteilung erfolgt nach Abstraktionsniveau (je höher die Generation, desto abstrakter).
  - Generation 1-2: Niedere Programmiersprachen (nah an der Hardware)
    - Befehlssatz und Datentypen sind genau auf den verwendeten Rechner abgestimmt.
  - Generation 3-5: Höhere Programmiersprachen (problemorientiert)
    - Abstrakter, maschinenunabhängig. Werden entweder nach der Programmierung in Maschinencode übersetzt oder während der Laufzeit interpretiert.
- Einteilung nach Paradigmen:
  - Ein Programmierparadigma bezeichnet das Model, mit dem ein Entwickler ein Programm beschreibt.

Klassifikation von Programmiersprachen

Generationen

1. Generation: Maschinencode

- Binäre Befehle, z.B. für
  - Daten aus dem Speicher in ein Prozessorregister holen (oder umgekehrt).
  - Einfache Arithmetik mit den Inhalten der Register.
- Beispiel (Intel x86):
  - `10110000 01100011` entspricht Hexadezimal B0 63
    - Bedeutet: Lade in das AL Register den Operand $(63)_{16}$
- Vor- und Nachteile
  - \+ sehr schnell arbeitende Programme
  - – schwer zu lernen und zu lesen
  - – stark abhängig vom Prozessor

2. Generation: Assemblercode

- Symbolische Befehle (Mnemonics) für die Binärcodes
- Bezeichner für Speicherstellen (inkl. Positionen im Programm)
  - erleichtert die Programmierung enorm und hilft Fehler zu vermeiden
- Beispiel (Assemblercode für den Binärcode aus vorherigem Beispiel)
  - `mv al, 63h` (vorher: `10110000 01100011`)

- Mittels eines Assemblers werden symbolische Befehle in Maschinencode übersetzt.
  - Einsatzgebiet: hochgradig optimierte Routinen, systemnahe Programmierung, Mikrocontroller, für die keine Hochsprache verfügbar ist.
- Vor- und Nachteile
  - \+ sehr schnell arbeitende Programme
  - – viel Zeit zur Programmierung notwendig anstatt zur eigentlichen Problemlösung
  - – abhängig vom Prozessor

3. Generation: Höhere Programmiersprachen

- Höheres Abstraktionsniveau, damit näher am zu lösenden Problem
  - Verwendung von Variablen
  - Komplexe Datentypen
  - Blockweiser Aufbau mit klaren Kontrollstrukturen (kein wildes Hin- und Herspringen)
  - Prozeduren und Funktionen erlauben einfache Mehrfachverwendung von Code-Teilen
- Verschiedene Paradigmen (z.B. prozedural, objektorientiert)

gleich mehr dazu…

- Beispiele: C, C++, Java, Python, Fortran, Algol, Pascal
- Vorteile
  - \+ leicht erlernbar und lesbar
  - \+ schnell
  - \+ geringe Abhängigkeit von der Hardware
  - – teilweise Unterschiede zwischen Betriebssystemen, Rechnertypen, Übersetzern

```
printf("Hallo Bottrop!");

ergebnis = 2*a + 5*(b-3);
```

Auszug aus einem C-Programm

Beispiel: 1. bis 3. Generation

1. Generation 2. Generation 3. Generation

Maschinencode (hexadezimal)	zugehöriger Assemblercode	zugehöriger C-Code	Erläuterung
55 48 89 E5	`push rbp` `mov rbp, rsp`	`int main() {`	Sichere Register RBP auf dem Stack und setze RBP auf den Wert von Register RSP, dem Stackpointer (gehört nicht zur eigentlichen Berechnung). Diese Vorbereitung ist notwendig, um die Werte der Variablen *a*, *b* und *c* auf dem Stack speichern zu können.
C7 45 FC 02	`mov DWORD PTR [rbp-4], 2`	`int a = 2;`	Setze Variable *a*, die durch Register RBP adressiert wird, auf den Wert 2.
C7 45 F8 03	`mov DWORD PTR [rbp-8], 3`	`int b = 3;`	Setze Variable *b*, die durch Register RBP adressiert wird, auf den Wert 3.

8B 45 F8
8B 55 FC
01 D0
89 45 F4
	`mov eax, DWORD PTR [rbp-8]`
	`mov edx, DWORD PTR [rbp-4]`
	`add eax, edx`
	`mov DWORD PTR [rbp-12], eax`
		`int c = a + b;`	Setze Register EAX auf den Wert von Variable *b*.
			Setze Register EDX auf den Wert von Variable *a*.
			Addiere den Wert von EDX zum Wert von EAX.
			Setze Variable *c*, die durch RBP adressiert wird, auf den Wert von EAX.

8B 45 F4	`mov eax, DWORD PTR [rbp-12]`	`return c;`	Setze Register EAX auf den Wert von Variable *c*. Weil Register EAX diesen Wert bereits enthält, könnte diese Anweisung in einem optimierten Programm entfallen.
5D C3	`pop rbp` `ret`	`}`	Setze RBP wieder auf seinen ursprünglichen Wert. Springe z

Quelle: https://de.wikipedia.org/wiki/Maschinensprache, 04.10.2018

4. Generation

- Das „Was“ steht im Vordergrund, nicht das „Wie“.
- Domänenspezifische Sprachen, d.h. für bestimmte Einsatzgebiete
- Beispiele
  - SQL (Structured Query Language): Sprache zur Datenbankabfrage, es wird beschrieben, welche Daten benötigt werden, aber nicht wie sie gefunden werden sollen
  - Deskriptive Sprachen: HTML beschreibt, wie eine Web-Seite aussehen soll, nicht aber wie der Browser sie zeichnen soll
  - Matlab, Mathematica: Sprachen für mathematische Berechnungen
  - Skriptsprachen, die andere Programme steuern (z.B. VBA)
  - Werkzeuge zur graphischen Programmierung (GUI-Design, Ablaufpläne)
- Vor- und Nachteile
  - \+ problemnah
  - \+ keine Definition von Einzelschritten notwendig
  - – wenig flexibel

Beispiel: 4. Generation

SQL

```
select * from users where age > 17 and gender = 'f';
```

HTML

```
<button type="submit" name="action">Zahlungspflichtig bestellen</button>
```

MATLAB

```
colorImage = imread('photo');
I = rgb2gray(colorImage);

% Detect MSER regions.
[regions, connectedComponents] = detectMSERFeatures(I, 'RegionAreaRange', [200 8000], 'ThresholdDelta', 4);
```

5. Generation

- Keine Lösung des Problems wird angegeben, sondern Beschreibung von Rand- und Zwangsbedingungen (Constraints)
- Einsatzgebiete: künstliche Intelligenz, Forschung
- Beispiele
  - Prolog: basiert auf Prädikatenlogik, bei der Programmierung werden Fakten und Regeln festgelegt, anhand derer das System später Fragen beantwortet
  - Lisp: basiert auf dem Lambda-Kalkül, alles wird als Funktion definiert, alle Objekte sind Listen, die nach bestimmten Regeln miteinander verknüpft werden
- Vor- und Nachteile
  - \+ allgemein
  - \+ maschinenunabhängig
  - \+ kurze Programme für komplexe Probleme möglich
  - – langsam

Beispiel

Prolog

```
% Fakten
studiert(stefan, GdI).
studiert(annika, SoftwareTechnik).
studiert(bernd, GdI).
studiert(sandra, GdI).

unterrichtet(weiss, GdI).
unterrichtet(weiss, SoftwareTechnik).
unterrichtet(geisler, MTI).

% Regeln
professor(X, Y) :- unterrichtet(X, FACH), studiert(Y, FACH).

% Anfragen
?- studiert(arthur, GdI).
?- professor(weiss, annika).
```

AUSGABE

```
false
true
```

Klassifikation von Programmiersprachen

Paradigmen

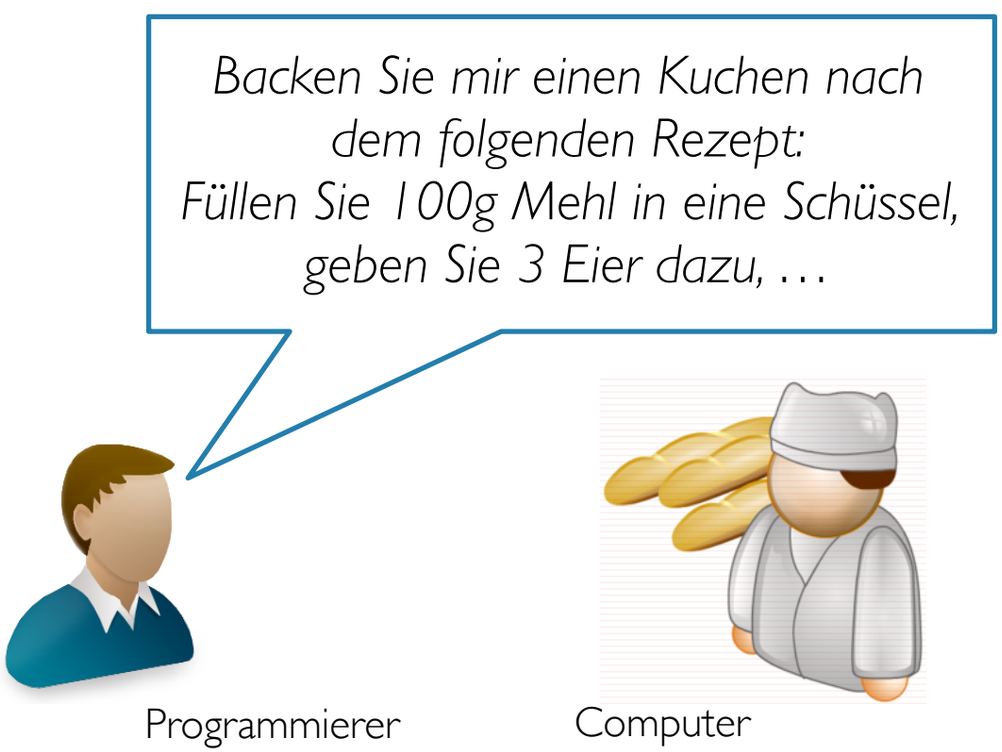
Programmierparadigmen

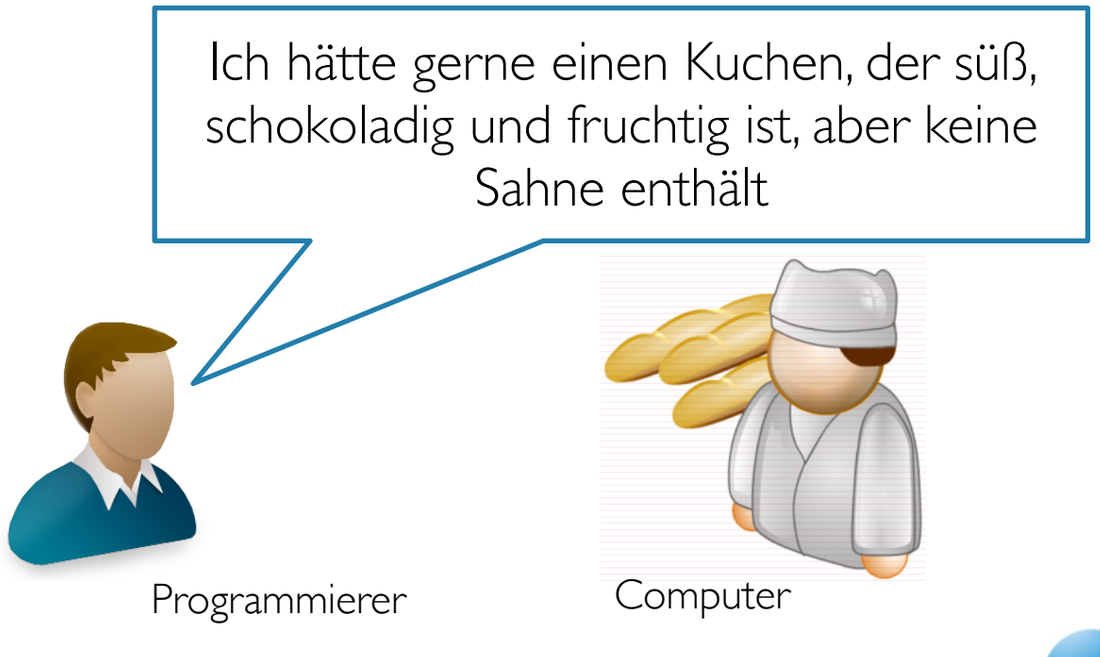
- Ein Programmierparadigma ist ein fundamentaler Programmierstil.
  - Er beschreibt das Modell, mit dem ein Programmierer ein Programm auffasst.
- Programmiersprachen folgen einem oder mehreren Paradigmen.
- Übliche Paradigmen:
  - Imperativ
  - Deklarativ
    - Funktional
    - Logisch
  - Objektorientiert

Programmierparadigmen: Imperativ vs. deklarativ

Imperativ	Deklarativ
Programm besteht aus Anweisungen die beschreiben, **wie** das Programm seine Ergebnisse erzeugt	Programm besteht aus Bedingungen (dem **Was**), welche die Ausgabe des Programms erfüllen muss
Beispiele: C, Pascal, Fortran,…	Beispiele: Prolog (logisch), Haskell (funktional), SQL (mengen-orientierte Abfragesprachen)

Fokus der Vorlesung





Paradigma: Objektorientiert

Objektorientiert: Daten und die darauf arbeitenden Routinen werden zu Einheiten („Objekten“) zusammengefasst. Ein Computerprogramm ist eine Menge miteinander interagierender Objekte.

Daten (Eigenschaften):

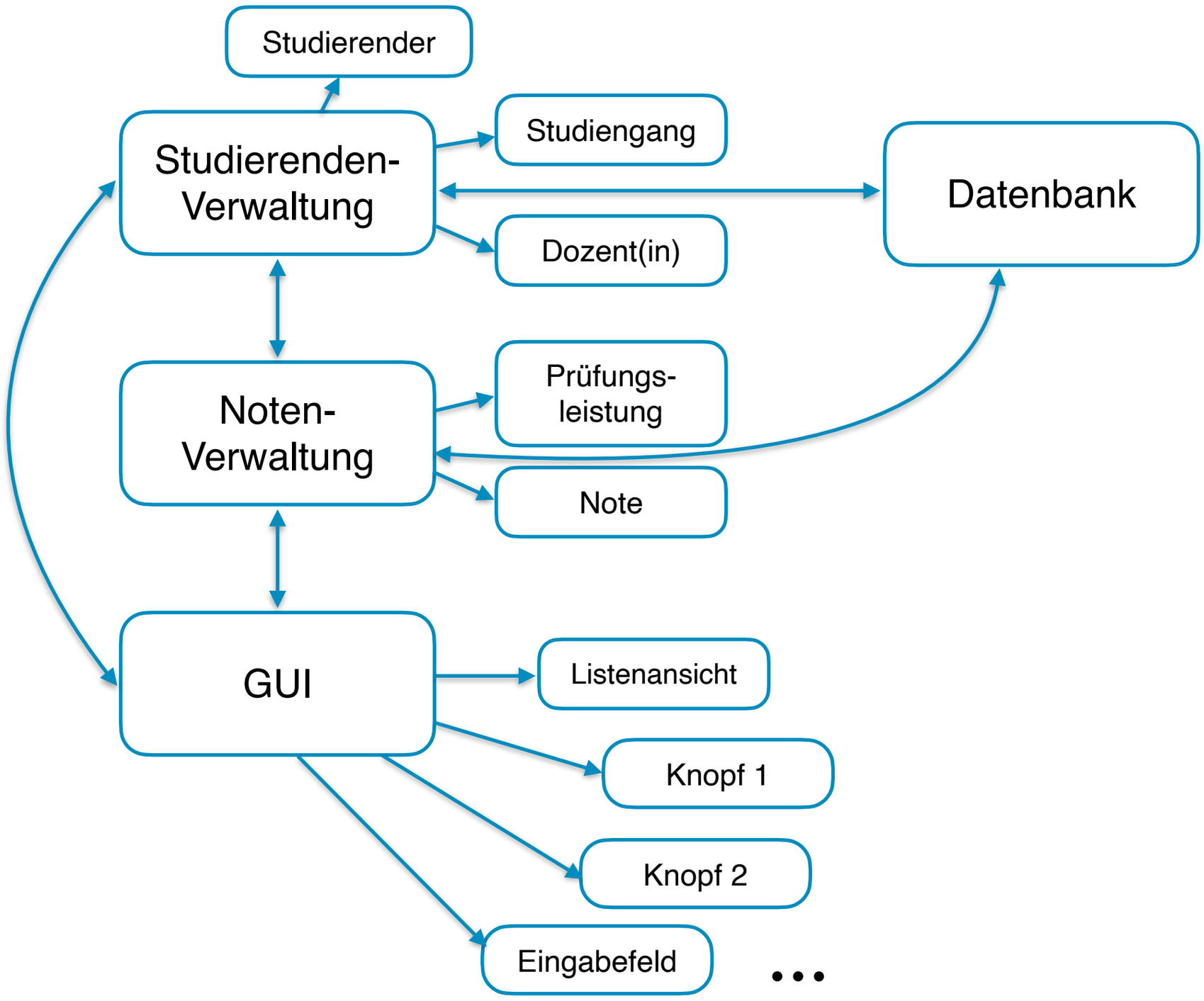
- Farbe
- Anzahl Türen
- PS
- Höchstgeschwindigkeit
- …

Routinen (Methoden):

- Starte Motor
- Beschleunige
- Bremse
- …
-

```
car = new Car("green", 4, 120, 250);
car.startEngine();
car.accelerate();
car.break();
```

Bildquelle: https://de.wikipedia.org/wiki/Automobil#/media/File:1986_Porsche_911_SC.jpg, Download am 02.03.2018


Studierender
Studierenden-
Verwaltung
Studiengang
Datenbank
Dozent(in)
Noten-
Verwaltung
Prüfungs-
leistung
Note
GUI
Listenansicht
Knopf 1
Knopf 2
Eingabefeld
...

Paradigma: Funktional

Das Programm wird als **funktionalen Ausdruck** aufgefasst. Das selbstständige Anwenden von **Funktionsersetzung** und **Auswertung** seitens des Interpreters / Compilers löst dann die Aufgabenstellung.

Beispiel: Berechnung der Fakultät in Haskell

```
fact :: Int -> Int
fact 0 = 1
fact n = n * fact(n-1)
```

Funktionsersetzung & Auswertung:

fact (4) = 4 ∗ fact (3)

fact(3) = 3 ∗ fact (2)

fact(2) = 2 ∗ fact (1)

fact(1) = 1 ∗ fact (0)

fact (0) = 1

fact (4) = 4 ∗ 3 ∗ 2 ∗ 1 ∗ 1 = 24

Paradigma: Logisch

Logisch: Die Aufgabenstellung und ihre Prämissen werden als **logische Aussagen** (Regeln) formuliert. Der Interpreter versucht, die gewünschte Lösungsaussage **herzuleiten** / zu **beweisen**.

Beispiel: Familienbeziehungen in Prolog

Faktenbasis:

```
parent(iris,beatrix).
parent(beatrix,hugo).

ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(X,Z),ancestor(Z,Y).
```

Anfragen:

```
?- parent(iris, beatrix).
true.

?- ancestor(iris, X).
X = beatrix ;
X = hugo ;
false.
```