Fachhochschule Kaiserslautern
University of Applied Sciences
Informatik und Mikrosystemtechnik
Zweibrücken

Grundlagen der Informatik
– Grundlagen der Programmierung –

Prof. Dr. Bernhard Schiefer

basierend auf Unterlagen von Prof. Dr. Duque-Antón
einige Abbildungen aus [Herold,Lurz,Wohlrab: GDI, 2012)]

bernhard.schiefer@fh-kl.de
http://www.fh-kl.de/~schiefer

Inhalt

- Programmiersprachen
- Formale Sprachen
- Vom Problem zum Programm

Programmiersprachen

- Ein Programm oder genauer ein Programmtext muss nach genau festgelegten Regeln formuliert werden, welche durch die Grammatik der entsprechenden Programmiersprache definiert werden.
  - ⇨ Frühe Programmiersprachen waren maschinennah, d.h. die verfügbaren Operationen des Rechners standen im Vordergrund.
  - ⇨ Aktuelle, höhere Programmiersprachen sind abstrakt und orientieren sich an dem zu lösenden Problem.
- Es existieren unterschiedliche Klassifikationen zur Einteilung von Programmiersprachen. Zum Beispiel nach Programmierparadigmen:

Live-Beispiele: Bash, Java, ...

Pascal	imperativ, prozedural
Java	imperativ, objektorientiert
Haskell	deklarativ, funktional
Prolog	deklarativ, logisch

Achtung: Hat hier nichts mit "Funktionen" zu tun! -> Haskell-Folie aus SWT einfügen.

Begriff: Formale Sprachen

- Die konzeptionelle Lösung eines Problems wird i.d.R. durch einen *Algorithmus* beschrieben.

  `= "Schrittweise Beschreibung", "Rezept".`

- Zu einem gegebenen Algorithmus stellt das *Programm* dann eine Lösung dar, die auf einem Rechner ausgeführt werden kann.

- Die Formulierung eines Programms in einer Programmiersprache erfolgt mit Hilfe *formaler Sprachen.* `Unterschied zur "natürlichen Sprache"?`

- Eine *formale Sprache* legt die Struktur (Syntax/Grammatik) der Sprache und die Bedeutung (Semantik) der Worte und Sätze aus der Sprache eindeutig fest. `... so dass der Computer sie auf Gültigkeit überprüfen kann.`

`Quiz: Wer erfand den ersten formal notierten Algorithmus / das erste Programm?`

Vom Programm zur Maschine

- Programme in einer höheren Programmiersprache können nicht unmittelbar auf einem Rechner ausgeführt werden:
  - ⇨ Sie sind anfangs meist in einer Textdatei gespeichert
- Verarbeitung durch *Compiler* (Übersetzer):
  - ⇨ Ein Compiler überführt das Programm in eine Folge von Maschinenbefehlen (Maschinencode), die der Prozessor des Rechners dann ausführen kann.
- Alternative: *Interpreter*
  - ⇨ zur Laufzeit wird immer nur ein kleiner Ausschnitt betrachtet, analysiert und durch Maschinenbefehle ausgeführt
- Ein Interpreter kann den Eingabetext "sofort ausführen", ~~ist leichter zu schreiben~~ aber nicht so effizient wie ein Compiler.
- Grundsätzlich können alle Sprachen compilierend oder interpretierend implementiert werden.

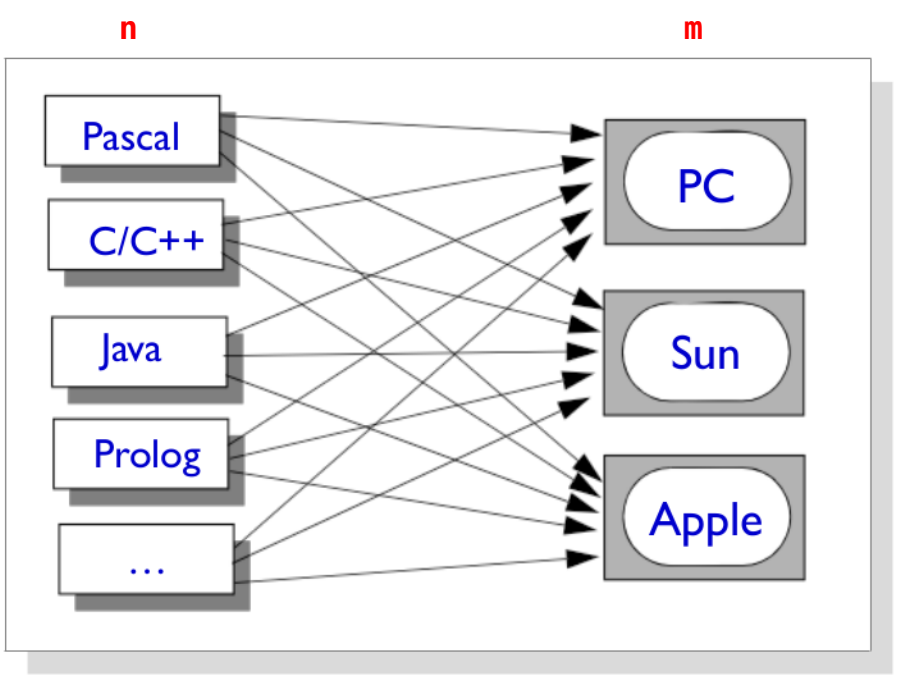
Compiler vs. Interpreter

- Übersetzung des Programmes in Maschinencode, bevor es ausgeführt wird
- Schleifen/mehrfach auftretende Befehle müssen nur einmal übersetzt werden
  → Zeitersparnis
- Fehler können teilw. bereits während des Programmierens angezeigt werden
- Beispiele: C, C++, Pascal

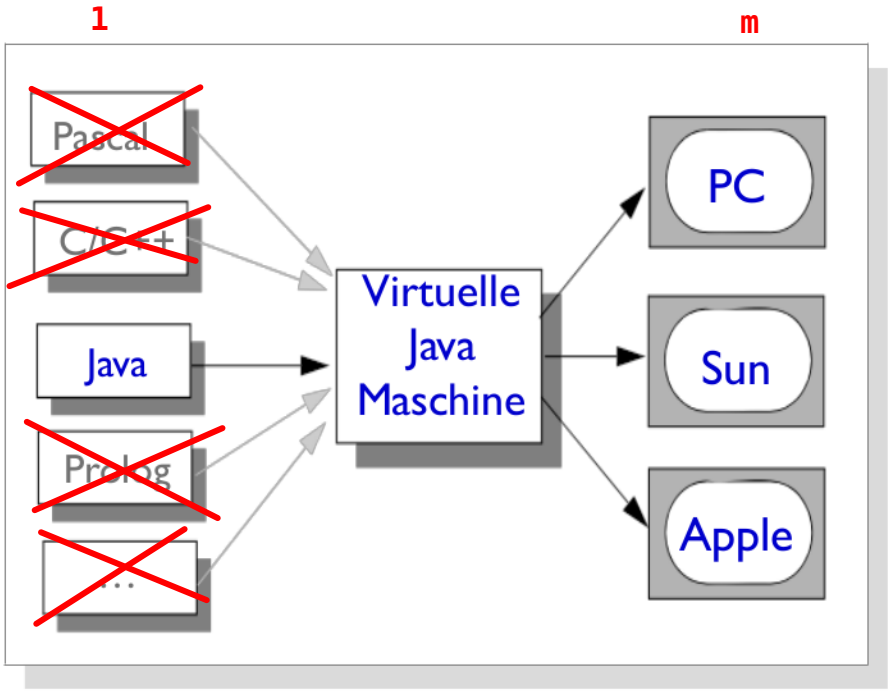
- Wandelt den Programmcode Anweisung für Anweisung während der Laufzeit in Mikrobefehle um
- Unterprogramme/Schleifen etc. werden möglicherweise mehrfach übersetzt
  → Zeitverlust
- Fehler werden erst zur Laufzeit bemerkt
- Beispiele: Basic, Prolog, PHP, `Python, Bash`

Programmiersprachen mit Mischformen: z.B. Java, Smalltalk, ABAP

Virtuelle Maschinen



n*m viele Compiler nötig, damit der Code auf der Plattform läuft



Java Code ~~wird~~ ist plattformunabhängig, ~~wenn~~ da eine gemeinsame virtuelle Maschine genutzt wird.

Beispiel Java

- Nutzung von Compiler und Interpreter

  wie z.B. bei C/C++ üblich

  ⇨ Ein Java-Programm wird vom Java-Compiler nicht direkt in Maschinensprache, sondern in Code für eine virtuelle Java-Maschine (JVM) übersetzt.

  ⇨ Dieser sogenannte Java Byte-Code wird anschließend durch die virtuelle Java-Maschine interpretiert und ausgeführt.

- Die JVM – ein Interpreter – ist im allgemeinen in Software realisiert und auf sehr vielen Rechnerplattformen und Betriebssystemen verfügbar

- Durch die Einschaltung dieses Zwischenschrittes wird Java plattformunabhängig.

  ⇨ Ein übersetztes Java-Programm ist ohne Anpassungen auf jeder Maschine lauffähig, auf der die JVM installiert ist.

  "Compile once, run everywhere."

- Java Byte-Code kann auch mittels eines Web-Browsers über das Internet geladen und von der JVM des Browsers interpretiert werden.

Beispiele für Sprachen: Pascal

Ggf. weglassen

- **Pascal** wurde Ende der 60er von Niklaus Wirth an der ETH Zürich für Unterrichtszwecke entwickelt:
  - ⇨ Pascal hat klare und übersichtliche Kontrollstrukturen, Blockstrukturen, Rekursion und Unterprogramme und eignet sich daher besonders für das Erlernen des algorithmischen Denkens
  - ⇨ Pascal verwendet das Prinzip der Orthogonalität: Für jede Aufgabe bietet sich genau nur ein Sprachkonzept als Lösung an
  - ⇨ Pascal gilt inzwischen als veraltet.
    Insbesondere fehlen moderne Konzepte wie Module und Objekte.
  - ⇨ Der klare und saubere Programmierstil von Pascal hat auch Nachteile.
    Insbesondere bei systemnahen Programmen kann dies störend oder gar effizienzhemmend sein.

Weitere Beispiele für Sprachen

- Mit **C** und **C++** können sehr effiziente und systemnahe Programme geschrieben werden.
  - ⇨ Genügend Selbstdisziplin vorausgesetzt kann man damit auch saubere Programme schreiben.
- **Java** ist eine Weiterentwicklung von C++, enthält jedoch weniger Freiheitsgrade.
  - ⇨ Java ist durch die virtuelle Maschine plattformunabhängig
  - ⇨ Die Sprache enthielt von Anbeginn geeignete Sicherheitskonzepte zur Verarbeitung von Programmen über das Internet `-> Browser-Plugins, SSL, ...`
- **Prolog** ist eine logische Programmiersprache:
  - ⇨ Ein Prolog-Programm besteht aus einer Menge von Fakten und Regeln
  - ⇨ Lösungen für Anfragen werden mit Hilfe der Resolution gesucht
  - ⇨ Typischer Anwendungsbereich: KI, insbes. Expertensysteme

Definition: Formale Sprache

- Bereits bekannt: Informationen werden durch Daten repräsentiert
  - ⇨ Die Daten werden mittels Symbolen codiert, die aus einer Menge verfügbarer Symbole wie etwa {0, 1} oder {a, b, ... } stammen.
  - ⇨ Diese Menge definiert ein **Alphabet** A als eine endliche Menge von Symbolen.

Woher weiß dies der Computer? -> Grammatik!

- Unser Ziel ist es letztendlich korrekte Programme zu formulieren, die bestimmte Probleme lösen. Dazu benötigen wir noch folgende Definitionen:
  - ⇨ Ein **Wort** über dem Alphabet A ist eine Aneinanderreihung endlich vieler Symbole aus A. Das leere Wort ε , das aus keinem Buchstaben besteht stellt ein besonderes Wort dar. Die Menge aller möglichen Wörter über einem Alphabet bezeichnen wir als A^*.
  - ⇨ Eine (**formale**) **Sprache** L ist eine Menge von Worten über einem Alphabet A, welche insbesondere das leere Wort ε enthält: $L \subseteq A^*$.

Codieren

- Wie entsteht der Programmcode?
  - ⇨ Unter Codieren versteht man die Formulierung der Lösung in einem konkreten Programmcode, also in den Symbolen eines Alphabets.
  - ⇨ Es ist zugleich auch die Umsetzung von einer Sprache in eine andere. Also z.B. die Programmierung eines Algorithmus in Java.

- Zur Definition korrekter Programme benötigen wir neben dem Begriff der formalen Sprache nun noch die Begriffe:
  - ⇨ **Lexikalische Analyse:** Definiert gültige Zeichen und Wörter einer Sprache. Wird mit Hilfe eines **Scanners** gelöst und basiert auf dem Lexikon der Sprache
  - ⇨ **Syntax**: Definiert die Grammatik der Sprache – die Regeln für den Aufbau von Sätzen aus Grundbausteinen
    Die syntaktische Überprüfung wird durch den **Parser** realisiert. Korrektheit
  - ⇨ **Semantik**: Definiert den Sinn der Worte und Sätze. Bedeutung
    Die semantische Analyse ist i.A. schwer zu realisieren und kann i.d.R. nicht vollständig formalisiert werden.

Syntaktische Analyse

- Durch die Syntax (Grammatik) werden aus den Wörtern der Programmiersprache Programme:
  - ⇨ Die Syntax beschreibt die Regeln für den Aufbau von Sätzen aus Grundbausteinen.
  - ⇨ Diese Regeln können z.B. in Form von Syntaxdiagrammen dargestellt werden.
- Ein Beispiel für eine Menge von Syntaxregeln aus dem Bereich der natürlichen Sprache ist etwa:
  - ⇨ Satz = Subjekt Prädikat
  - ⇨ Subjekt = Clara | Hugo
  - ⇨ Prädikat = redet | hört zu
- Formal: Satz S = AB, und A = a | b sowie B = c | d,
  - ⇨ wobei a Clara, b Hugo, c redet und d hört zu abkürzt.
  - ⇨ Das Symbol | hat die Bedeutung „oder“.
- Wie sieht die Menge der möglichen Sätze aus, formal und umgangssprachlich?
  - ⇨ L = ?

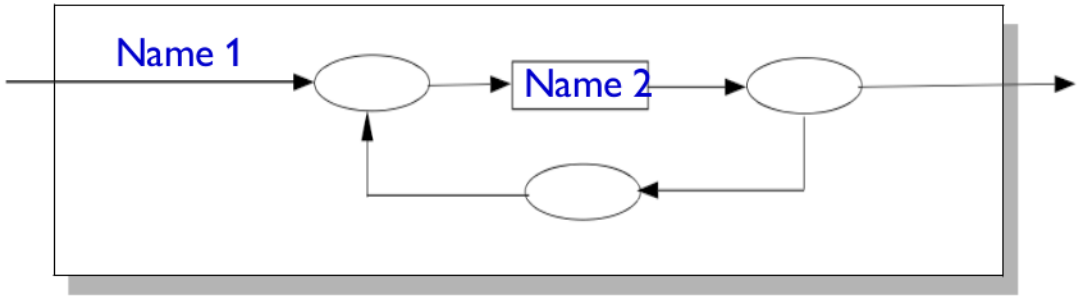
Begriffe

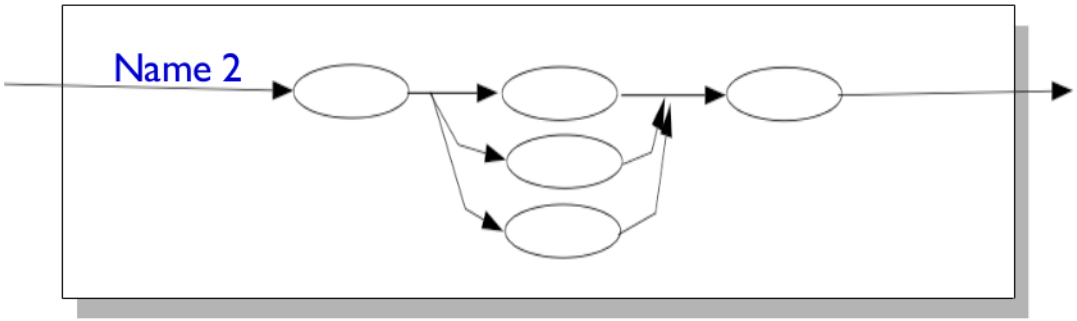
Übung 3!!!

- Eine Sprache L kann mit diesen Komponenten konstruiert werden:
  - ⇨ Menge von Terminalsymbolen T,
  - ⇨ Menge von Nonterminalsymbolen N ($N \cap T = \{\}$),
  - ⇨ Menge von Produktionen (Syntax-Regeln) P
  - ⇨ einem Startsymbol $S \in N$ aus der Menge der Nonterminale.
- Alle Zeichen, die nur auf der rechten Seite von Produktionen vorkommen, gehören zu den *Terminalen*. (Sie sind nicht substituierbar)
- Alle *Nichterminale* kommen also mindestens einmal auf der linken Seite einer Produktion vor.

Syntaxdiagramme

- Mit Hilfe eines Syntaxdiagramms wird eine Produktion (Syntax-Regel) grafisch dargestellt:

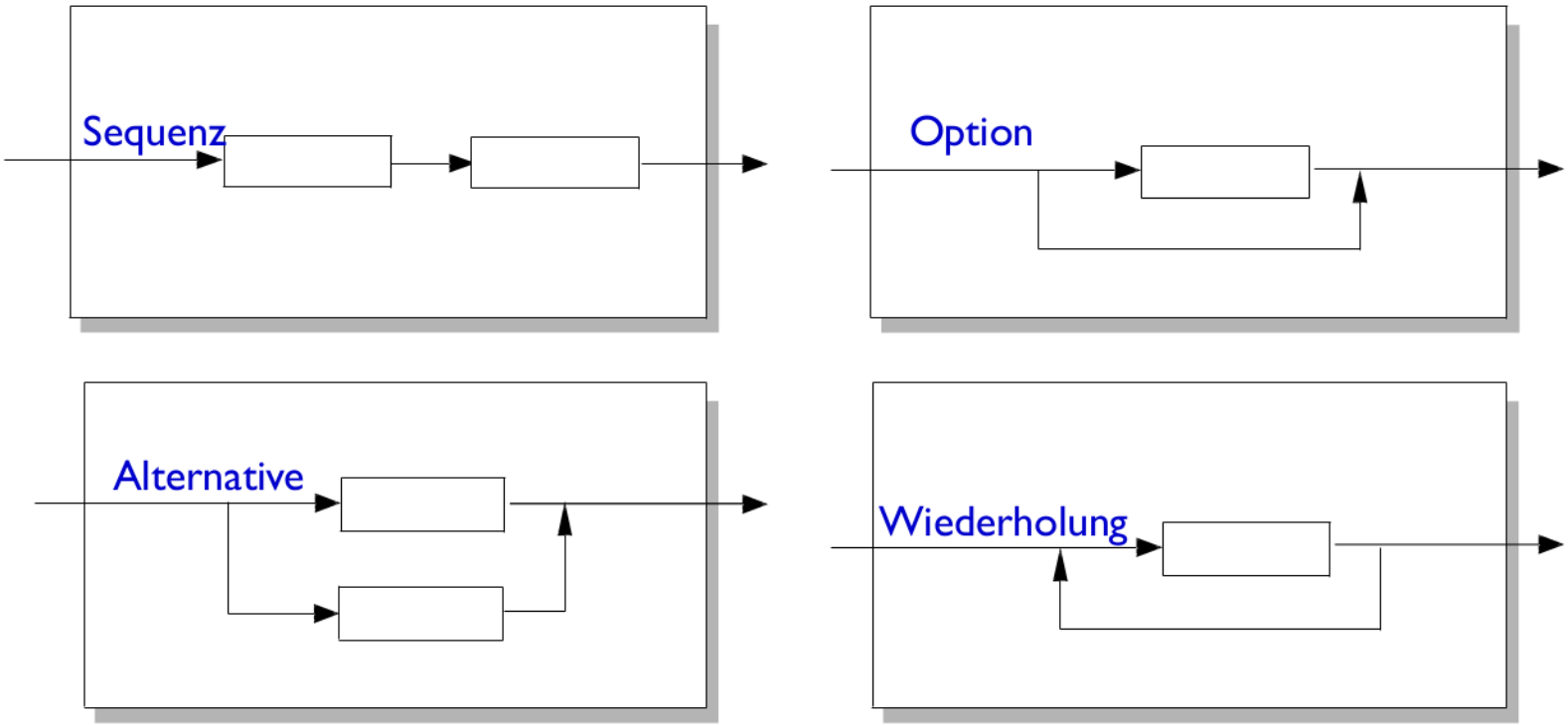




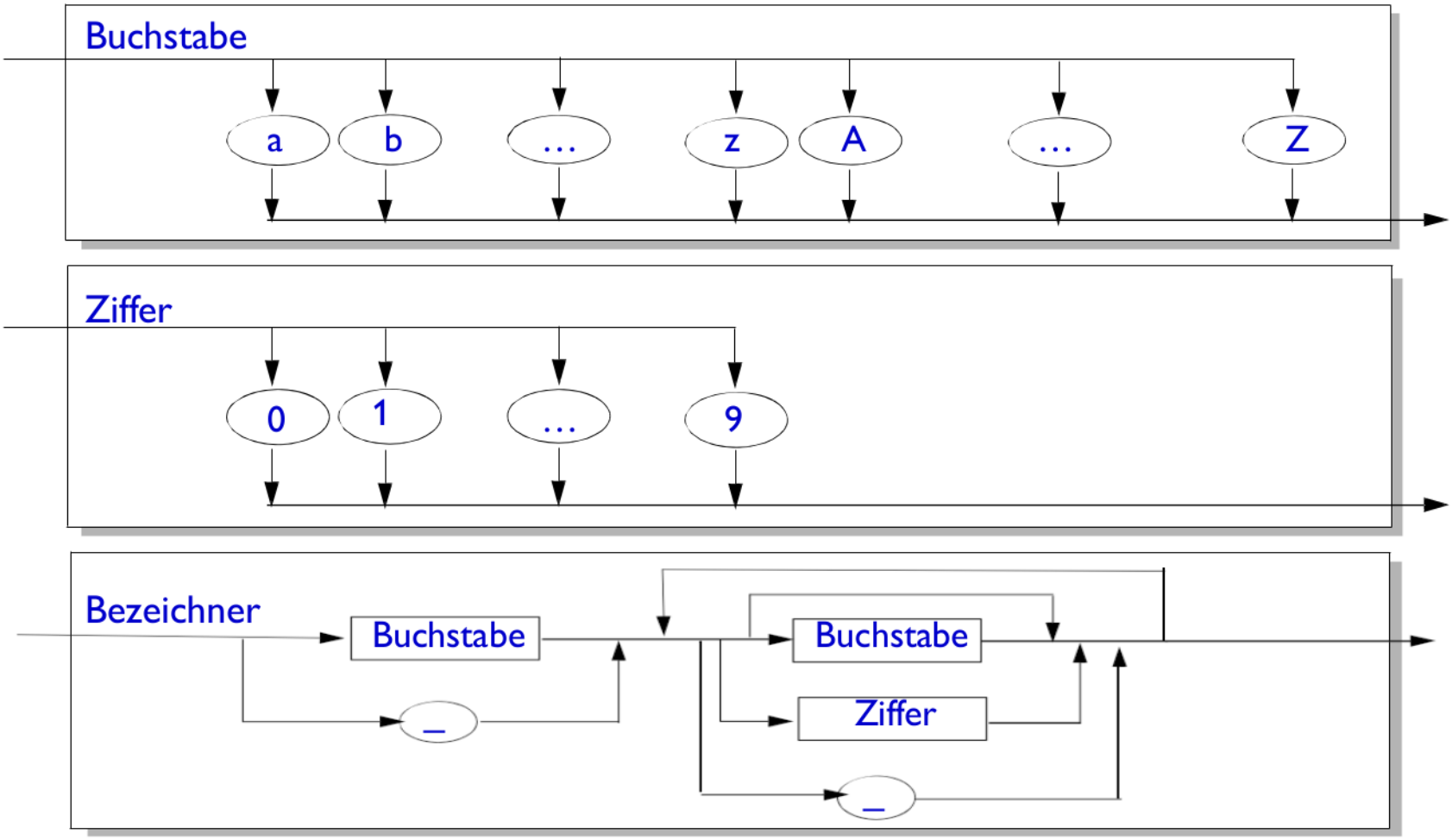
- Ovale: Terminale
- Rechtecke: Nonterminale
- Gerichtete Pfeile: Sind die eigentlichen Regeln und verbinden die einzelnen Elemente

- Jedes Syntaxdiagramm hat genau einen Ein- und einen Ausgang. Der entsprechende Name steht i.d.R. oben links.

Syntaxdiagramme: Grundtypen



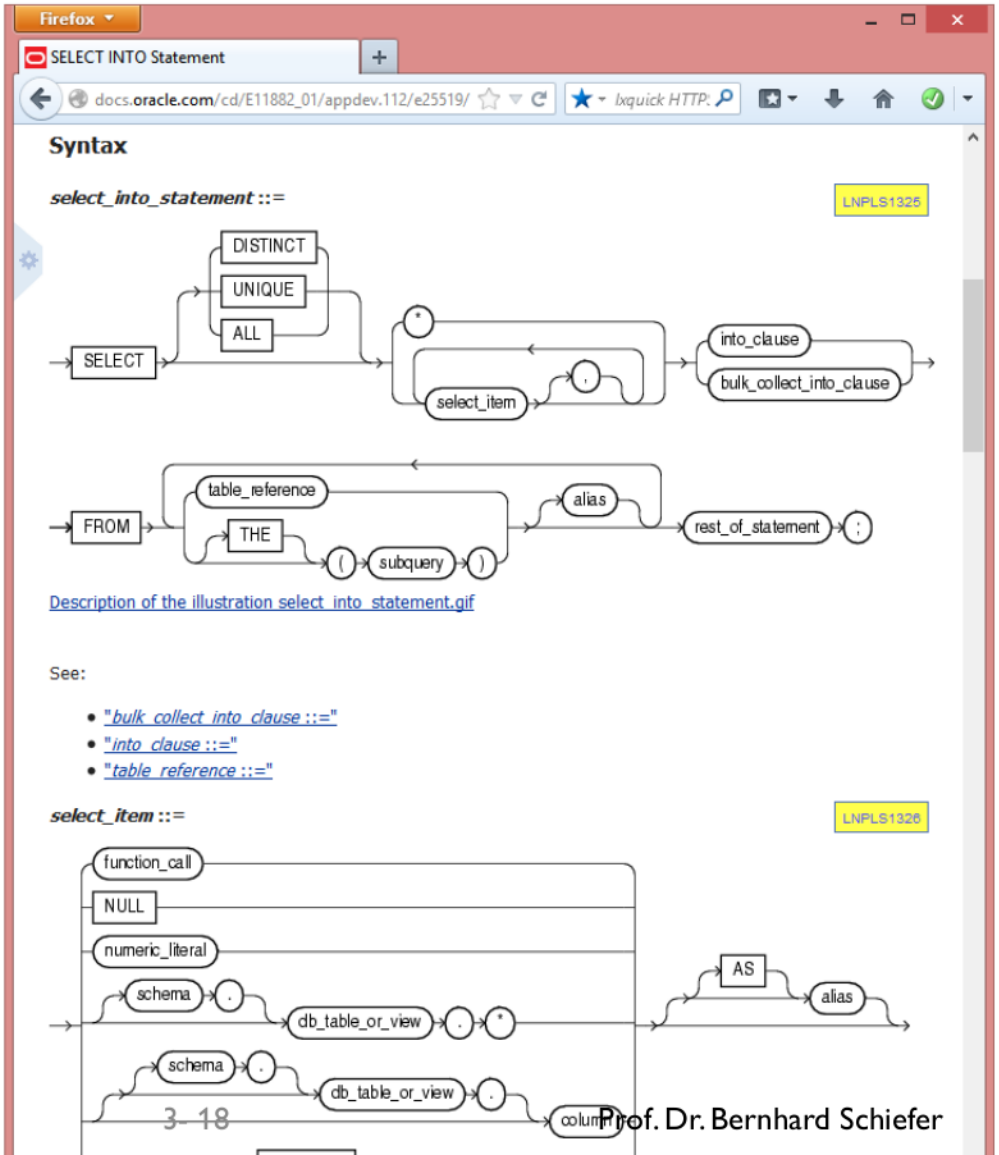
Syntaxdiagramme: Beispiele



Syntaxdiagramm: Beispiel SQL

Weglassen.

- Oracle SQL SELECT-Statement:
  - ⇨ Achtung: Abweichende Bedeutung der Symbole!



Syntaxdiagramme: Übung

- Geben Sie ein Syntaxdiagramm an, zur Beschreibung von geraden Zahlen im Dualsystem.

Backus-Naur-Formen: BNF und EBNF

- Die Backus-Naur-Form (BNF) ist eine Möglichkeit der formalen Beschreibung einer Syntax.
  - ⇨ Ziel ist die direkte Beschreibung von Syntax-Regeln.

besonders kurz notierbare,

- Eine äquivalente, aber besser strukturierte Variante stellt die Erweiterte BNF (EBNF) dar:
  - ⇨ Ursprünglich zur Darstellung der Syntax von Pascal eingeführt
  - ⇨ Eine Sprache besteht wie zu Beginn definiert weiterhin aus Terminalen, Nonterminalen, Produktionen und einem Startsymbol.
  - ⇨ Terminalen werden in Anführungszeichen „ eingeschlossen. ("Literale")
  - ⇨ Die rechte Seite kann geklammerte Ausdrücke haben, neben den ursprünglichen Terminalen, Non-Terminalen und Alternativen (Symbol |).
- Die linke Seite besteht weiterhin nur aus einem Nonterminal
  - ⇨ kontextfrei

EBNF

- Man unterscheidet insgesamt drei Arten von Klammern:
  - ⇨ ( ) Inhalt der Klammer ist gruppiert.
  - ⇨ [ ] Inhalt der Klammer ist optional.
  - ⇨ { } Inhalt der Klammer kann n-fach stehen mit $n \geq 0$.

- Manchmal zusätzliche Zeichen der Wiederholung und Optionalität:
  - ⇨ * beliebig viele (auch keines)
  - ⇨ + beliebig viele, mind. eins
  - ⇨ ? eins oder keins

EBNF: Beispiele

```
Buchstabe = "a" | "b" | . . . | "z";

Ziffer = "0" | "1" | "2" | . . . | "9";

Start = Buchstabe | "_";

Bezeichner = Start { Ziffer | Start };

GPZ = Ziffer { Ziffer };

GZ = ["-"] GPZ;

Zuweisung = Bezeichner ":=" ( GZ | Bezeichner ) ";" ;

Programm = Zuweisung  { Zuweisung };
```

EBNF: Übung

- Beschreiben Sie durch 2 ohne Rest teilbare Dualzahlen mit der EBNF.

Beispiel: Java Notation

Weglassen, eher Syntaxdiagramm

- Anmerkungen
  - ⇨ Modifikation der EBNF

Productions from §8 (Classes)

ClassDeclaration:
  NormalClassDeclaration
  EnumDeclaration

NormalClassDeclaration:
  {*ClassModifier*} class *Identifier*
  [*TypeParameters*] [*Superclass*]
  [*Superinterfaces*] *ClassBody*

ClassModifier:
  Annotation public protected private
  abstract static final strictfp

TypeParameters:
  < *TypeParameterList* >

TypeParameterList:
  TypeParameter {, *TypeParameter*}

Superclass:
  extends *ClassType*

Chomsky-Grammatiken

Übung 3!!!

- Chomsky definierte 1959 ein Regelwerk zur Beschreibung einer formalen Sprach- Hierarchie. Dazu wird der Begriff der Grammatik zusammen mit einem Ableitungsbegriff eingeführt.

- In der allgemeinsten Form ist eine Chomsky-Grammatik als Viertupel **G = (N, T, P, S)** wie folgt definiert:
  - ⇨ N eine endliche, nichtleere Menge von Nonterminalen (Symbolen),
  - ⇨ T eine endliche, nichtleere Menge von Terminalen (Symbolen) mit $T \cap N = \{\}$.
  - ⇨ $S \in N$ das Startsymbol.
  - ⇨ P ein endliche Menge von Regeln der Form (p, q) mit $p, q \in (T \cup N)^*$ und p enthält mindestens ein nonterminales Symbol.
    Eine Regel (p, q) wird auch Produktion genannt.

Die Chomsky Hierarchie

- In der Chomsky-Hierarchie werden vier Typen von Grammatiken unterschieden.
- Die allgemeinste Form stellt eine **Typ-0** Grammatik dar und besitzt die größte Mächtigkeit.
  - ⇨ Beliebige Produktionen sind zulässig
  - ⇨ Problem: Akzeptor hält bei Worten außerhalb von L(G) evtl. nie

    http://de.wikipedia.org/wiki/Akzeptor_%28Informatik%29
- Zur Beschreibung von Programmiersprachen werden normalerweise **Typ-2** Grammatiken verwendet:
  - ⇨ In diesem Fall gilt ε $\forall(p,q) \in P$: $p \in N$ und $|p| = 1$
- Typ-2-Grammatiken definieren die Menge der ***kontextfreien Sprachen***

Der Ableitungs-Begriff

- Wie können syntaktisch korrekte Wörter generiert werden ?
- Ist $x \in (N \cup T)^*$ und $(p,q) \in P$ mit $x = x'px''$, wobei x' und $x'' \in (N \cup T)^*$,
  - ⇨ dann wird die Anwendung von (p,q) auf x definiert als: $x \rightarrow_{(p,q)} y$
  - ⇨ bzw. abgekürzt: $x \rightarrow y$ mit $y = x'qx''$.
- Die Hintereinander-Anwendung der Produktionen auf x_0 $(N \cup T)^*$ erzeugt
  - ⇨ eine Folge von Wörtern x_1, x_2 bis $x_n \in (N \cup T)^*$ mit
  - ⇨ $x_0 \rightarrow x_1$, $x_1 \rightarrow x_2$ bis $x_{n-1} \rightarrow x_n$ mit $n \geq 0$.
  - ⇨ In diesem Fall wird x_n die Ableitung von x_0 nach G genannt: $x_0 \rightarrow_G x_n$
- Die von G erzeugt Sprache L(G) definiert die Menge aller gültigen Wörter, die aus dem Startsymbol ableitbar sind:
  - ⇨ $L(G) := \{w \mid S \rightarrow_G w \text{ und } w \in T^*\}$

Beispiel 1

- G = (N, T, P, S) mit
  - ⇨ N = { S }
  - ⇨ T = { a, b}
  - ⇨ P = { (S, b), (S, aSa) }
- Charakterisierung der Grammatik?
- Geben Sie die erzeugte Sprache L(G) an

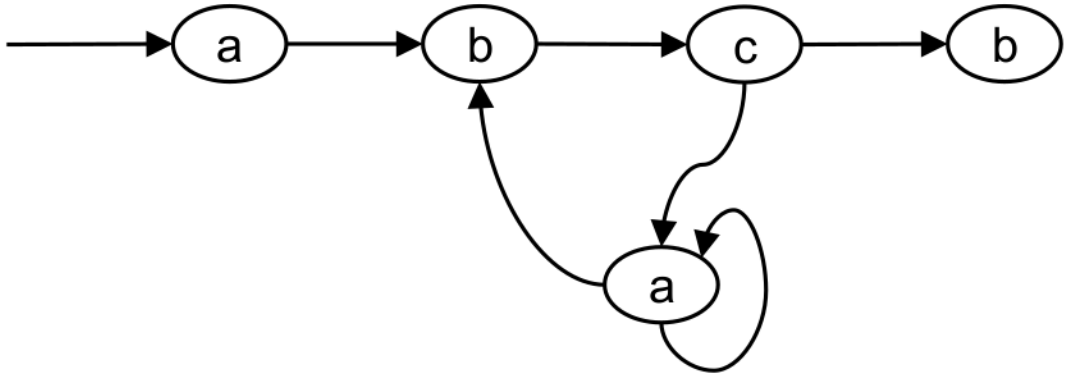
Beispiel 2

- G = (N, T, P, S) mit
  - ⇨ N = { S }
  - ⇨ T = { a, b }
  - ⇨ P = { (aaS, b), (S, aSb) }
- Charakterisierung der Grammatik?
- Geben Sie die erzeugte Sprache L(G) an

Mächtigkeit der Notationen

- Syntaxdiagramme, BNF, EBNF, Chomsky Grammatiken und Java Notation sind gleichmächtig !
  - ⇨ Es können jeweils die gleichen Sprachen beschrieben werden

- Beispiel zur Illustration:
  - ⇨ Wandeln Sie das folgende Diagramm in eine Grammatik und in EBNF um:



Vom Problem zum Programm

- Vor dem Beginn der Programmierung muss das zu lösende Problem exakt beschrieben (*spezifiziert*) werden.
  - ⇨ Anforderungen an die Eingabegrößen **Definitionsbereich**
  - ⇨ Zusicherung von Eigenschaften der berechneten Ergebnisse **Wertebereich**
- Danach muss ein Ablauf von Aktionen entworfen werden, der insgesamt zur Lösung des Problems führt.
  - ⇨ Ein solcher Ablauf wird *Algorithmus* genannt.
- Erst dann kann der Algorithmus in einen Programmcode umgewandelt werden, der maschinell (auf einem Rechner) ausgeführt werden kann.

Spezifikation

- Eine Spezifikation muss vollständig, detailliert und unzweideutig das Problem beschreiben:
  - ⇨ vollständig, wenn alle Anforderungen und relevanten Rahmenbedingungen angegeben sind.
  - ⇨ detailliert, wenn alle Hilfsmittel insbesondere Basis-Operationen, die zur Lösung zugelassen sind, bekannt sind.
  - ⇨ unzweideutig, wenn es klare Kriterien gibt, wann eine vorgeschlagene Lösung akzeptabel ist.
- Informelle Spezifikation
  - ⇨ oft umgangssprachlich und daher eher unpräzise formuliert.
  - ⇨ Sie erlauben, von weniger wichtigen Dingen zu abstrahieren – nicht jedes Detail wird explizit spezifiziert
- Formale Spezifikation
  - ⇨ Benutzt formale Spezifikationssprache (oft mit Logik-Ausdrücken)

Spezifikation: Ein Beispiel

- Wir betrachten die Aufgabe, den größten gemeinsamen Teiler (ggT) zweier Zahlen zu finden. Eine informelle Spezifikation könnte lauten:
  - ⇨ Für beliebige Zahlen n und m berechne den größten gemeinsamen Teiler ggT(n,m), also die größte Zahl, die sowohl n als auch m teilt.
- Diese (informelle) Spezifikation lässt viele Fragen offen:
  - ⇨ Vollständigkeit: Welche Zahlen n und m sind zugelassen? Dürfen n und m nur positive Zahlen oder auch negative oder gar rationale Zahlen sein. Ist 0 erlaubt?
  - ⇨ Detailliertheit: Welche Operationen sind erlaubt?
  - ⇨ Unzweideutigkeit: Was heißt berechnen? Soll das Ergebnis in einer bestimmten Variablen gespeichert werden, oder einfach nur ausgedruckt werden?

Algorithmen

- Auf der Basis der Spezifikation geht es nun darum, einen Lösungsweg zu entwerfen. Da der Lösungsweg von einem Rechner durchgeführt wird, muss jeder Schritt exakt vorgeschrieben werden.

- Ein **Algorithmus** ist eine detaillierte und explizite Vorschrift zur schrittweisen Lösung eines Problems:
  - ⇨ Die Ausführung des Algorithmus erfolgt in einzelnen Schritten.
  - ⇨ Jeder Schritt besteht aus einer einfachen und offensichtlichen Grundaktion.
  - ⇨ Zu jedem Zeitpunkt muss klar sein, welcher Schritt als nächstes auszuführen ist.

- Wichtige Eigenschaften eines Algorithmus:
  - ⇨ *Korrektheit*, *Terminierung*, *Determinismus*, *Determiniertheit*

Algorithmen: Eigenschaften

- Ein Algorithmus muss realisierbar sein, das bedeutet:
  - ⇨ Es muss eine endliche Beschreibung geben, die endliche Ergebnisse liefert und
  - ⇨ alle notwendigen Aktionen (bzw. die entsprechenden Funktionen) sind berechenbar (effektiv)
- Ein (realisierbarer) Algorithmus kann:
  - ⇨ effizient oder ineffizient gelöst werden.
  - ⇨ korrekte oder nicht korrekte Ergebnisse liefern.
  - ⇨ terminieren oder auch nicht.
- Bemerkung: effektiv ≠ effizient
  - ⇨ Effektiv: Ziel wird erreicht.
  - ⇨ Effizient: Ziel wird mit minimalem Aufwand (z.B. möglichst schnell) erreicht.

Algorithmen: Terminierung

- Ein Algorithmus, der für alle den Eingabespezifikationen entsprechenden Eingaben nach endlich vielen Schritten abbricht, heißt terminierend.
- Beispiele:
  - ⇨ Terminiert ein Primzahltest?
  - ⇨ Terminiert die Berechnung der Eulerschen Zahl e = 2,71828?
  - ⇨ Terminiert die Addition zweier positiver ganzer Zahlen?
  - ⇨ Terminiert die Sortierung einer unsortierten Kartei?
  - ⇨ Terminiert der Ulam-Algorithmus?
    - Beginne mit einer beliebigen Zahl n.
    - Ist n ungerade, multipliziere sie mit 3 und addiere 1, ansonsten halbiere sie.
    - Fahre so fort, bis 1 erreicht ist.

Algorithmen: Korrektheit

- Partielle Korrektheit:
  - ⇨ Jedes berechnete Ergebnis entspricht der Ausgangsspezifikation, sofern die Eingaben der Eingangsspezifikation entsprechen.
- Totale Korrektheit:
  - ⇨ Der Algorithmus ist partiell korrekt und terminiert mit jeder der Eingangsspezifikation entsprechenden Eingabe.
  - ⇨ D.h. er hält bei gültiger Eingabe nach endlich vielen Schritten mit dem richtigen Ergebnis an.

Algorithmen: Determiniertheit

- Ein Algorithmus heißt determiniert, wenn er bei gleichen Eingabewerten (die den Eingabespezifikationen entsprechen) stets das gleiche Ergebnis liefert.

- Beispiel:
  - ⇨ (1) Wählen Sie eine Zahl zwischen 0 und 10. („Eingabe“)
  - ⇨ (2) Addieren Sie 7.
  - ⇨ (3) Schreiben Sie 4 auf. („Ausgabe“)

Algorithmen: Determinismus

- Ein Algorithmus heißt deterministisch, wenn zu jeder Zeit der weitere Ablauf des Algorithmus eindeutig bestimmt ist. Der Ablauf des Algorithmus wird also nur von den Eingaben bestimmt.

- Beispiel:
  - ⇨ (1) Wählen Sie eine Zahl zwischen 0 und 10. („Eingabe“)
  - ⇨ (2) Addieren Sie 2 hinzu.
  - ⇨ (3) Schreiben Sie das Ergebnis auf. („Ausgabe“)

- Bemerkung:
  Durch den Rechner erzeugte Zufallszahlen sind i.d.R. sog. Pseudozufallszahlen (sind keine echten Zufallsereignisse).
  - ⇨ Streng genommen bleiben im Rechner umgesetzte Algorithmen mit Pseudozufallszahlen deterministisch.

- Beispiel nicht-deterministischer Algorithmus: Münzwurf

Beispiel: Test für Zahl z

- Welche Eigenschaften weist folgender Algorithmus auf?
  (1) Wähle eine beliebige Zahl n mit 1 < n < z, die bisher noch nicht getestet wurde
  (2) Falls n Teiler von z: Ausgabe „Keine Primzahl“. Gehe zu (5).
  (3) Falls alle Zahlen n mit 1 < n < z getestet: Ausgabe „Primzahl“. Gehe zu (5).
  (4) Gehe zu (1)
  (5) Fertig

Darstellung von Algorithmen

- Zur Darstellung von Algorithmen haben sich verschiedene Darstellung bewährt:
  - ⇨ Mathematische Notation
  - ⇨ Umgangssprachliche Beschreibung
  - ⇨ Flussdiagramme
  - ⇨ Pseudocode
  - ⇨ Programmiersprachen

Algorithmen: Mathematische Notation

- erlaubt eine sehr kompakte und knappe Beschreibung des Algorithmus, die zugleich eindeutig ist.
- ist naturgemäß besonders geeignet zur Beschreibung von mathematischen Problemstellungen.
- Beispiele:
  - ⇨ Addition zweier ganzer Zahlen a, b ≥ 0 allein mit Hilfe der Operation additiver Nachfolger $f_{+1}(x) = x + 1$ bzw. der Umkehrfunktion $f_{-1}(x) = x - 1$.
  - ⇨ Berechnung des Divisionsrests r von a dividiert durch b: r = a mod b für ganze Zahlen a ≥ 0 und b > 0.
  - ⇨ Berechnung des größten gemeinsamen Teilers zweier natürlicher Zahlen a und b mit ggT (a,b) mit a > 0 und b ≥ 0.

Beispiele: Mathematische Notation

Häufig: Einsatz von Rekursion

- Addition:

$$sum(a, b) = \begin{cases} a & , \text{falls } b = 0 \\ sum(a, b-1)+1 & , \text{falls } b > 0 \end{cases}$$

- Rest:

$$a \bmod b = \begin{cases} a & , \text{falls } a < b \\ (a-b) \bmod b & , \text{sonst} \end{cases}$$

- ggT: (mit a,b > 0)

$$ggT(a, b) = \begin{cases} a & , \text{falls } b = 0 \\ ggT(b, a) & , \text{falls } b > a \\ ggT(b, a \bmod b) & , \text{sonst} \end{cases}$$

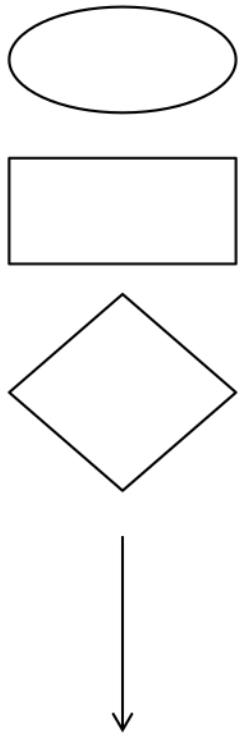
Algorithmen: Umgangsspr. Beschreibung ggT

- Berechne ggT (a,b) mit a > 0 und b ≥ 0 (Euklidischer Algorithmus):
  - ⇨ Schritt 1: Falls b > a, dann vertausche a und b.
  - ⇨ Schritt 2: Teile a durch b und finde den Rest r := a mod b.
  - ⇨ Schritt 3: Falls r = 0, dann endet der Algorithmus mit der Lösung b.
  - ⇨ Schritt 4: Sonst, wird a durch b (b → a) und b durch r (r → b) ersetzt und bei Schritt 2 fortgefahren.
  - ⇨ Ausgabe: b enthält den gesuchten Rest.
- Wie sehen die entsprechenden Algorithmen für
  - ⇨ die Addition für a, b ≥ 0 bzw.
  - ⇨ die Bestimmung des Rests (a mod b) für a > 0 und b ≥ 0 aus?

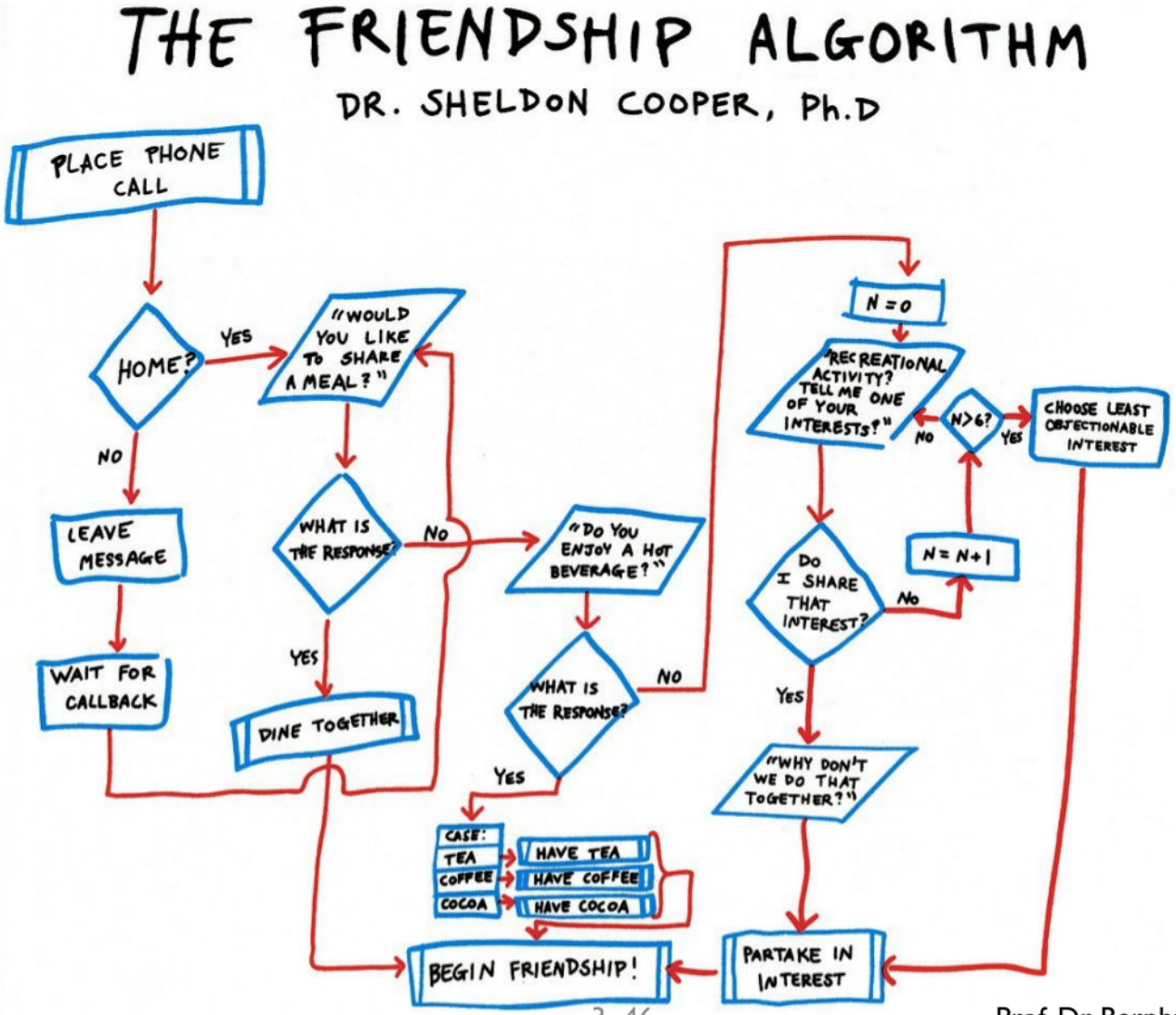
Algorithmen: Flussdiagramm

- grafische Repräsentation eines Algorithmus
  - ⇨ i.d.R. zum Detailentwurf eingesetzt
  - ⇨ Mit Hilfe grafischer Symbole werden informell Aktionen beschrieben.
  - ⇨ Mit Hilfe von Flusslinien werden die Symbole verbunden und damit der Kontrollfluss festgelegt.
- Vorteile:
  - ⇨ Flussdiagramme veranschaulichen den Kontrollfluss
  - ⇨ Schleifenkonstrukte lassen sich besonders deutlich hervorheben, so dass auf einen Blick alle Anweisungen einer Schleife grafisch erfasst werden können.

- Symbole von Flussdiagrammen (Auswahl)

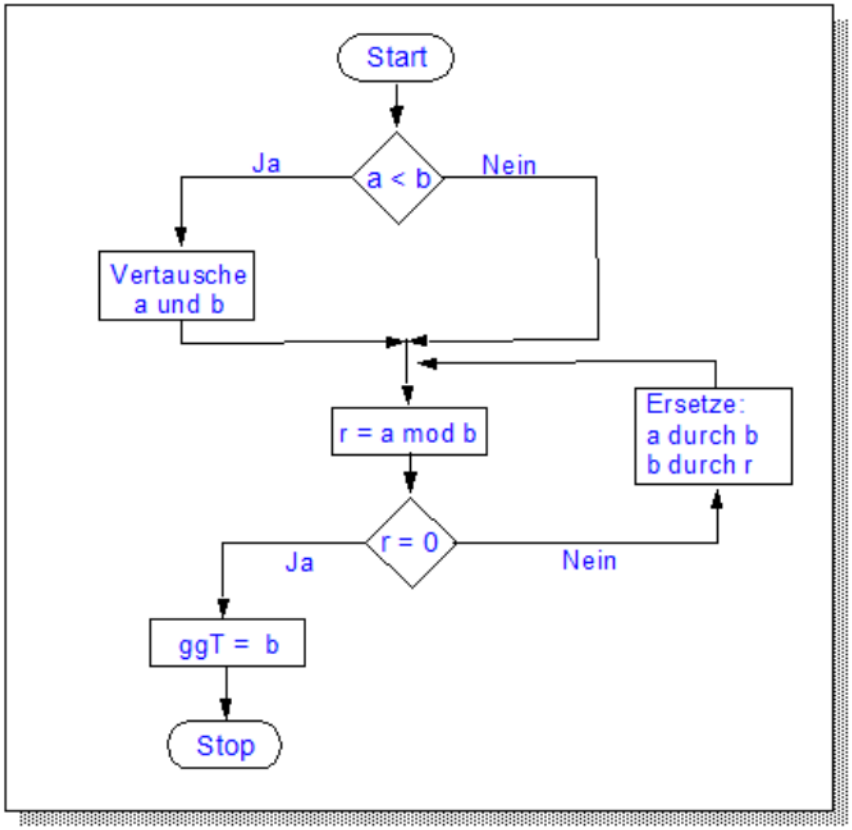


Algorithmen: Flussdiagramm aus „The Big Bang Theory“

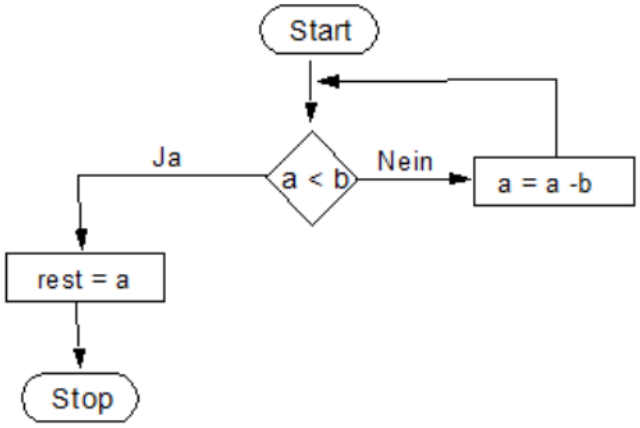


Algorithmen: Flussdiagramm für ggT, Rest

- Flussdiagramm für ggT



- Flussdiagramm für Rest



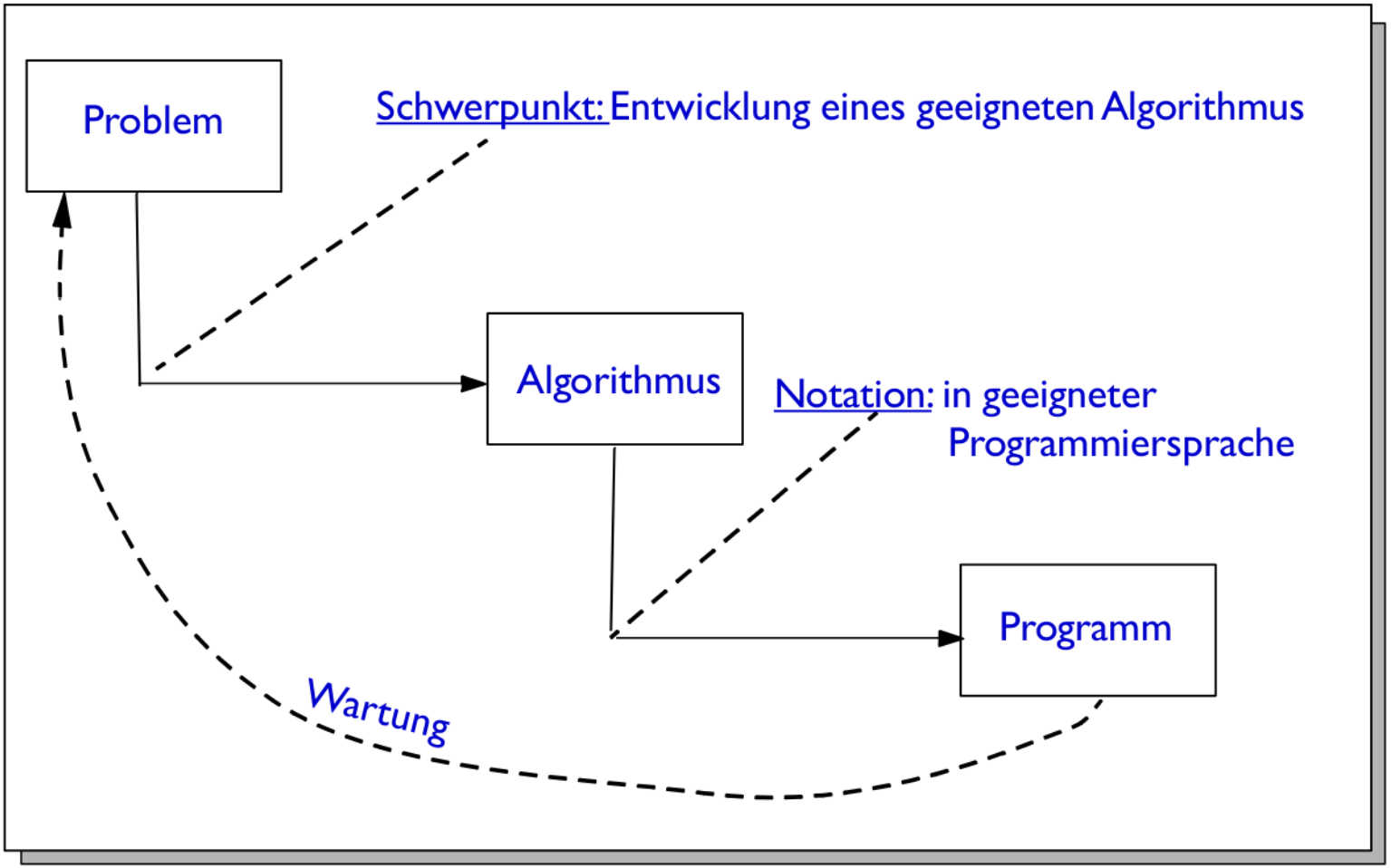
Algorithmen: Pseudonotation

- In konkreten Programmiersprachen formulierte Algorithmen sind i.d.R. schwer verständlich.
  - ⇨ Daher formuliert man Algorithmen häufig in Pseudocode, der besser verständlich ist.
  - ⇨ Pseudocode stellt eine *abstrakte* Programmiersprache dar, die eine Basis für eine präzise Beschreibung von Algorithmen bildet.
- Pseudocode verwendet allgemeine Konzepte wie Verzweigung oder Iteration, die i.d.R. in den meisten Programmiersprachen mit derselben Semantik verwendet werden.
  - ⇨ Daher kann Pseudocode leicht in eine *konkrete* Programmiersprache abgebildet werden.

Algorithmen: Pseudonotation ggT

```
/* Euklidischer Algorithmus */
begin
    /* Einlesen der Werte */
    read (a, b)
    / * Vorbedingung testen */
    wenn (a < b), dann vertausche a und b
    r = a mod b
    /* Algorithmusdurchlauf bis Abbruch-Bedingung r = 0 erfüllt wird */
    solange (r ist nicht gleich 0)
        ersetze a durch b
        ersetze b durch r
        r = a mod b
    ausgabe:  ggT = b
end
```

Software-Entwicklung



Programmieren und Testen

- Man unterscheidet verschiedene Fehler-Varianten:

  - ⇨ **Syntaxfehler** sind mit Rechtschreib- oder Grammatikfehlern vergleichbar: man hat sich bei einem Wort vertippt (Scanner) oder einen unzulässigen Satzbau (Befehls- Syntax) verwendet.
  - ⇨ **Semantische Fehler** entstehen, wenn nicht die Programm-Bedeutung realisiert wurde, die eigentlich gemeint war.
    - Dabei unterscheidet man Fehler, die noch zur Compile-Zeit erkannt werden können und **Laufzeitfehler**.
  - ⇨ **Denkfehler** werden sichtbar, wenn ein Programm problemlos abläuft, aber nicht das (vom Programmierer) gewünschte Ergebnis liefert.

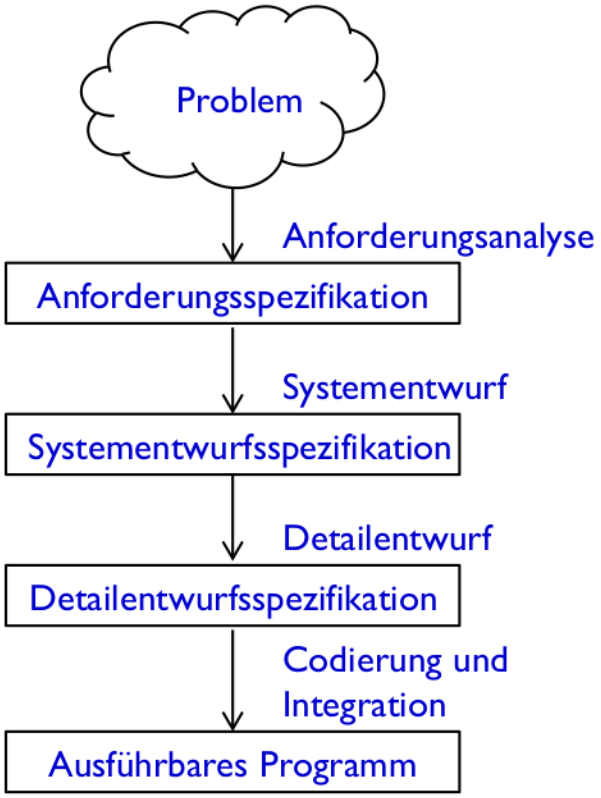
Programmieren und Testen

- Programmierung ist immer mit einer ausführlichen Testphase verbunden. Das Suchen und Verbessern von Fehlern (Bugs) in der Testphase nennt man *debugging*.
  - ⇨ Laufzeitfehler und Denkfehler können bei einem Testlauf sichtbar werden
  - ⇨ Prinzipiell gilt immer die Aussage von Dijkstra: Durch Testen kann die Anwesenheit, aber nie die Abwesenheit von Fehlern gezeigt werden!
- Die praktische Entwicklung eines Programms findet daher im Zusammenspiel von verschiedenen Vorgängen statt:
  - ⇨ Editieren, (Compilieren), Debuggen
- Für fast alle Sprachen gibt es heute dazu sogenannte IDEs (Integrated Development Environments), die Werkzeuge wie Editor, Compiler und Debugger zusammenfassen.

Software Engineering

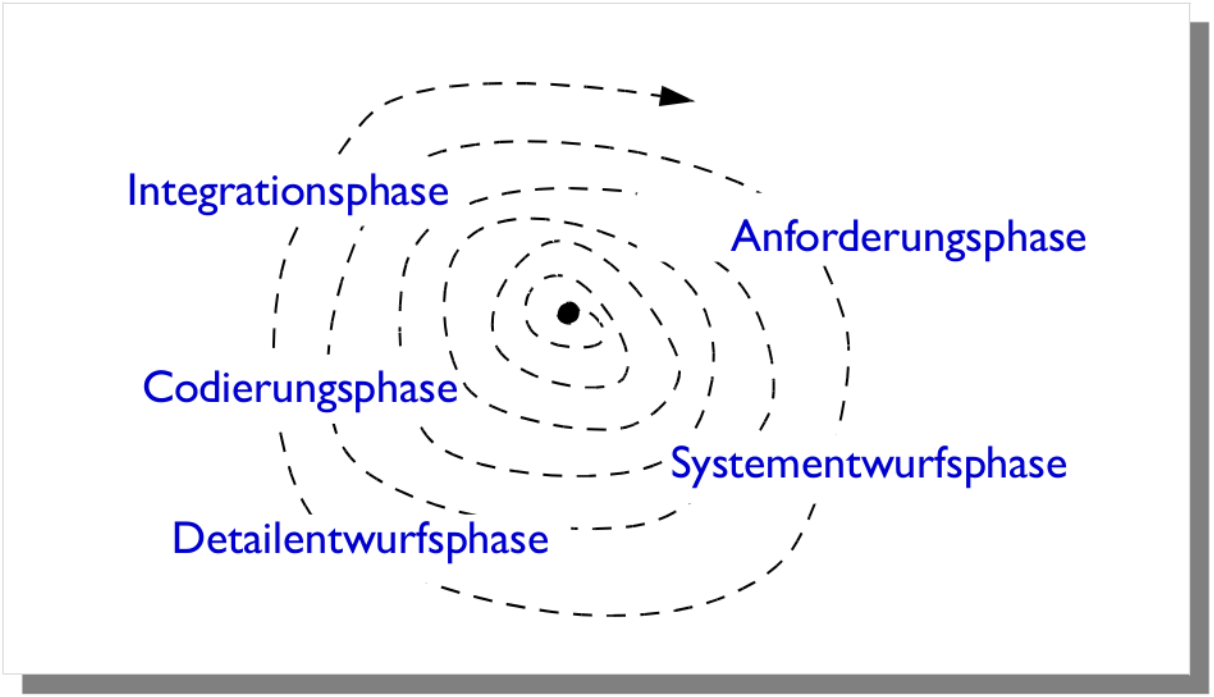
- ... bietet Entwicklungsmethoden, die insbesondere auf große Softwaresysteme skalierbar sind und die geeignet sind,
  - ⇨ Qualitätssoftware zu niedrigen, vorhersagbaren Kosten und
  - ⇨ in einer kurzen, vorhersagbaren Entwicklungszeit zu produzieren.
- Der Schlüssel liegt in der Definition eines geeigneten, modularen Entwicklungs-Prozesses.
- In der einfachsten Variante wird das Phasenmodell verwendet.

- Phasenmodell



Spiralmodell

- In der Praxis hat sich das Wasserfallmodell als zu unflexibel und daher nicht praktikabel erwiesen:
  - ⇨ Tatsächlich laufen die einzelnen Phasen nicht strikt hintereinander ab
  - ⇨ Meistens möchte man am Anfang recht schnell zu ersten, groben Ergebnissen kommen.



Agile Software-Entwicklung

- Gefahr klassischer Vorgehensmodelle:
  - ⇨ Viel Zeit wird für die Spezifikation der Anforderungen zu Beginn des Projektes investiert.
  - ⇨ Änderungen zur Laufzeit können nur schwer eingebracht werden
  - ⇨ Produkte stehen dem Kunden erst sehr spät zur Verfügung
  - ⇨ Verzögerungen und erhöhte Kosten können den Erfolg des gesamten Projektes gefährden
- Das agilen Vorgehensmodell sieht eine sehr enge und direkte Zusammenarbeit mit dem Kunden vor
  - ⇨ Die Spezifikation erfolgt sukzessive während der Umsetzung
  - ⇨ Bei Schwierigkeiten kann direkt gegengesteuert werden – z.B. durch Verzicht auf weniger wichtige Features
  - ⇨ Der Kunde bekommt, was er braucht, nicht was er irgendwann vor langer Zeit einmal spezifiziert hat!

Das Manifest für Agile Softwareentwicklung

- Von einigen renommierten Software-Entwicklern wurde im Februar 2001 folgende Werte als *Agiles Manifest* (englisch *Manifesto for Agile Software Development*) formuliert:

- Wir erschließen bessere Wege, Software zu entwickeln, indem wir es selbst tun und anderen dabei helfen.
  Durch diese Tätigkeit haben wir diese Werte zu schätzen gelernt:
  - ⇨ **Individuen und Interaktionen** mehr als Prozesse und Werkzeuge
  - ⇨ **Funktionierende Software** mehr als umfassende Dokumentation
  - ⇨ **Zusammenarbeit mit dem Kunden** mehr als Vertragsverhandlung
  - ⇨ **Reagieren auf Veränderung** mehr als das Befolgen eines Plans