Paradigmen der Programmierung

- Motivation
- Inhalte
- Werkzeuge
- Begriff „Paradigma“
- Prozedurale versus objektorientierte Programmierung
- Erste Beispiele

Motivation

„Conventional programming languages are growing ever more enormous, but not stronger. Inherent defects at the most basic level cause them to be both fat and weak: their primitive word-at-a-time style of programming inherited from their common ancestor – the von Neumann computer, their close coupling of semantics to state transitions, their division of programming into a world of expressions and a world of statements, their inability to effectively use powerful combining forms for building new programs from existing ones, and their lack of useful mathematical properties for reasoning about programs.“
(J. Backus, 1978)

„But whereas machines must be able to execute programs (without understanding them), people must be able to understand them (without executing them).“
(E. Dijkstra, 1978)

„And we must study through reading, listening, discussing, observing and thinking. We must not neglect any of those ways of study. The trouble with most of us is that we fall down on the latter – thinking – because it's hard work for people to think.“
(Th. Watson, IBM)

Inhalte

- Logikprogrammierung (Prolog)
- Funktionale Programmierung (Scala)
- Teil 2 (Prof. Dr. Kohls)
- Praktikum (5 Termine à 3h, ab 23.10., Staffelplan demnächst)

Werkzeuge

Prolog:
www.swi-prolog.org
Entwicklungsumgebung:
Am einfachsten SWI-Prolog. Das Eclipse-Plugin bietet kaum Vorteile.

Scala:
www.scala-lang.org
Entwicklungsumgebung:
Gutes Eclipse-Plugin. Separates Download von Scala ist möglich, aber nicht erforderlich.

Paradigmen der Programmierung

par·a·digm \'per-ə-"dīm, 'pa-rə- *also* -"dim\ *n*

[LL *paradigma,* fr. Gk *paradeigma,* fr. *paradeiknynai*

to show side by side, fr. *para-* + *deiknynai* to show — more at diction] (15c)

1 : example pattern ; *esp*: an outstandingly clear or typical example or archetype

2 : an example of a conjugation or declension showing a word in all its inflectional forms

3 : a philosophical and theoretical framework of a scientific school or discipline within which theories, laws, and generalizations and the experiments performed in support of them are formulated ; *broadly*: a philosophical or theoretical framework of any kind

Ein verwandter Begriff ist „Weltbild“. Ein Weltbild betrifft die Einstellung zur Welt insgesamt; ein Paradigma ist in aller Regel auf einen bestimmten Anwendungsbereich beschränkt.

Hintergrund für den Begriff Paradigmen

Thomas S. Kuhn, *The Structure of Scientific Revolutions*, 1962
(deutsch: *Die Struktur wissenschaflicher Revolutionen*)

Das Buch stellt die These auf, dass sich im Wissenschaftsbetrieb „normale“ Phasen von „revolutionären“ Umbrüchen unterscheiden lassen.

Laut Kuhn bedeuten diese Umbrüche einen Paradigmenwechsel.

Standardbeispiel: *die kopernikanische Revolution*

Vorher: alle Beobachtungen werden im geozentrischen Weltbild gedeutet, was nicht passt (Planetenbewegung) wird als unwichtig abgetan.

Nachher: alle Phänomene müssen im heliozentrischen Weltbild neu verstanden werden (neue Physik durch Galilei und Newton).

Verallgemeinerung: Menschliches Denken und Handeln orientiert sich an Paradigmen als einem Denkrahmen.

Was ist zu dem Beispiel zu sagen?

Umwandlung eines C- in ein Java-Programm:

- Wandle `struct` in `class` um
- Wandle C-Datei in Java-Datei um
- Wandle C-Funktion in Java-Klassenfunktion um.
- Wandle globale Variable in `public static` Variable um.
- Wandle `#define` in `public static final` um.
- `...`

Die Regeln stammen aus einer Diplomarbeit. Das Programm hatte Klassen wie: `Global`, `Konstanten`, `Vektor, Vektorfunktionen, Matrix, Maxtrixfunktionen`

Der Vorteil der Vorgehensweise liegt darin, dass der Aufwand gering ist und das das entstehende Java-Programm vermutlich genauso funktioniert wie das C-Programm. Man erhält aus dem C-Programm damit schnell eine portable Lösung.

Hat dieses Vorgehen auch Nachteile?

Liegt hier ein Paradigmenwechsel vor?

Paradigma 1:

Ein Programm wird aufgeteilt nach dem Motto: *Programm = Algorithmen + Datenstrukturen.* Typdeklarationen, Konstanten und globale Variable werden in Headerdateien deklariert und Funktionen werden sinnvoll auf verschiedene Implementierungsdateien verteilt. Man beachtet bei der Bildung von Funktionen das *Prinzip der schrittweisen Verfeinerung*.

Ein C-Programm beschreibt einen Ablauf, der durch ein *Flussdiagramm* oder ein *Struktogramm* graphisch dargestellt werden kann.

Paradigma 2:

Ein Programm besteht aus Klassen, die für die im Programm auftretenden *Konzepte und Entitäten* stehen. Mittels Klassen erzeugt das Programm *Objekte*, die durch Nachrichten miteinander kommunizieren.

Klassen bilden untereinander ein *Netz von Beziehungen und Assoziationen*, die durch ein *UML-Klassendiagramm* graphisch dargestellt werden.

Martin Odersky (Scala with Style):

- Mathematische Anwendungen haben wenige Datentypen aber viele Unterschiedliche Prozeduren.

- Simula 67, die erste OO-Sprache entstand für Simulation:
  Viele unterschiedliche Simulationsobjekt, wenige und gleichartige Prozeduren

- Smalltalk 80, die nächste große OO-Sprache entstand für graphische Benutzeroberflächen:
  Viele unterschiedliche Widgets, wenige gleichartige Prozeduren

Fazit: Neue Paradigmata kommen auf, wenn Sie gebraucht werden.

Vorläufiges Fazit

Ein Paradigma ist für sich genommen nicht schon gut oder schlecht.

Bestimmte Paradigmen können für einen Anwendungsbereich angemessen oder weniger angemessen sein (vgl. html-Java).

Die Wahl eines Paradigmas ist stark von subjektiven Vorlieben bestimmt.

Vorsicht: wer in einem bestimmten Paradigma denkt, wird andere Denk- und Vorgehensweisen leicht allzu negativ beurteilen (Religionen).

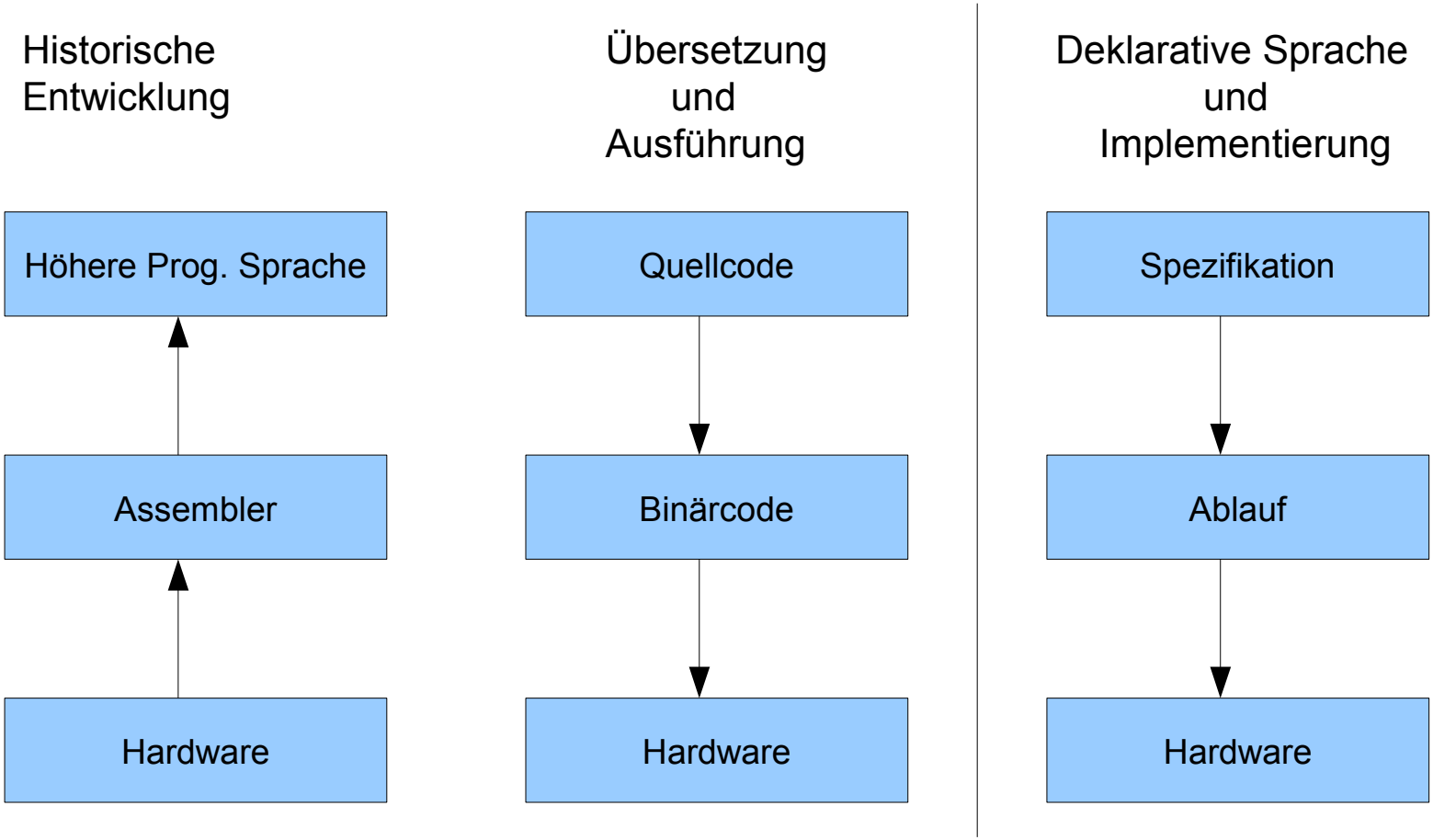
In Programmierparadigmen lenken den Programmierstil.

Programmiersprachen erzwingen nicht die Verwendung eines bestimmten Programmierparadigmas. Sie können jedoch einzelne Paradigmen mehr oder weniger gut unterstützen.

Ein Ziel der Vorlesung ist, durch die Diskussion unterschiedlicher Programmierparadigmen den Horizont zu erweitern.

Imperative und deklarative Sprachen

(J. Backus: Can Programming be liberated from the von Neumann Style?)



Was ist, wenn wir eine nicht-vN Hardware (z.B. Multicore) haben?

Was für Vorlagen für die Programmierung gibt es?

Befehlsatz eines einfachen Computers

Verknüpfung von (programmierbaren) Elementen

Mathematik:

Funktionen
Gleichungen
Logik

Sprache:

Befehlssatz
Frage
Aussagesatz

Grundlegende Unterscheidung:

- Imperative Programmierung
- Deklarative Programmierung

Imperative Paradigmen und Programmiersprachen

Maschinensprache
Assembler
Prozedurale Sprachen
Objektorientierung
Aspektorientierte Programmierung
Nebenläufigkeit

Deklarative Paradigmen und Programmiersprachen

Funktionale Programmierung
Logikprogrammierung
Expertensysteme
Seitenbeschreibungssprachen
Datenbanken
Konfigurationsdateien

Beispiel prozedural (Java / Scala)

Java:

```
public static <T> boolean contains(Node<T> node, T x) {
    Node<T> p = node;
    while (p != null && ! p.value.equals(x)) p = p.next;
    return p != null;
}
```

Scala:

```
def contains[T](liste: List[T], x: T): Boolean = {
   var p = liste
   while (p != Nil && p.head != x) p = p.tail
   p != Nil
}
```

Unterschiede:

Scala bevorzugt unveränderliche Variablen (**`val`**), Hier steht aber **`var`** (= veränderlich).
Eine Funktion liefert einen Wert. Keine Return-Anweisung nötig!
Typdeklarationen werden, wenn möglich, automatisch „geraten“.
In Scala vergleicht man mit **==** (ruft `equals()` auf)

Beide Beispiele sind prozedural!

Beispiel funktional (Scala)

Fallunterscheidung:

```
def contains[T](liste: List[T], x: T): Boolean =
    if (liste == Nil) false
    else if (liste.head == x) true
    else contains(x, liste.tail)
```

Mustererkennung:

```
def contains[T](liste: List[T], x: T): Boolean = liste match {
    case `x`::_ => true
    case _::xs  => contains(xs, x)
    case Nil    => false
}
```

Objektorientierte Bibliothek:

```
if (namensListe.contains("Einstein")) ...
```

Besonderheiten:

Keine Variable !
Keine Anweisungen, nur Ausdrücke.
Mustererkennung mit Prüfung / Zuweisung
(effiziente) Rekursion anstelle von while

Hinweis: Mein Thema ist *funktionale* Programmierung nicht *objekt-funktionale* Programmierung !

Beispiel Logikprogrammierung (Prolog)

= vordefiniertem member:

```
contains(X, [X|_]).     % das Kopfelement ist in der Liste
contains(X, [_|Xs]):-   % ein Element, das in der Restliste ist,
    contains(X, Xs).    %   ist in der Liste
```

Besonderheiten:

Programm = Menge von Fakten und Regeln
Fakt: Anfangselement ist in der Liste
Regel: Jedes Element der Restliste in in der Liste
Keine Aussage für leere Liste; Antwort automatisch = false.
Musterkennung für Listen. Aufbau ähnlich zu Scala.
Prolog kann auch (mehrere) Elemente finden.

Fragen und Antworten:

```
?- contains(2, [1,2,3]).
true
?- contains(7, [1,Y,3]).
Y = 7
?- contains(X, [1,2,3]).
X = 1;
X = 2
```

Deklarative Bedeutung / Ausführung

Die Bedeutung eines imperativen Programms ergibt sich durch die Programmausführung

Die Bedeutung eines Logikprogramms ist durch die Prädikatenlogik definiert.

Die Bedeutung eines funktionalen Programms ergibt sich durch Einsetzen der Funktionsdefinitionen.

Natürlich werden letztlich auch funktionale / Logikprogramme ausgeführt. Die abstraktere Formulierung ermöglich aber häufig eine bessere Optimierung (z.B. Multicore).