

Sprachkonzepte

Teil 3: Programmierparadigmen
imperative / deklarative Programmierung,
Anwendungs- / Systemprogrammierung,
Scripting, Textgenerierung

Programmierparadigmen

Ein Paradigma ist eine grundsätzliche Denkweise (*Wikipedia, 2021*).

Bei Programmierparadigmen bezieht sich die Denkweise darauf, was man als die fundamentalen Bausteine, Prinzipien und Zwecke von Programmen ansieht. Beispiele sind etwa die imperative und die deklarative Denkweise:

- imperative Programmierung

Programme bestehen aus einer Folge von *Anweisungen*, die festlegen, wie ein Anfangszustand schrittweise in einen gesuchten Endzustand überführt wird
die prozedurale und die objektorientierte Programmierung sind imperative Paradigmen

- deklarative Programmierung

Programme bestehen aus *Ausdrücken*, die eine gesuchte Lösung beschreiben, ohne dabei den Lösungsweg detailliert festzulegen
die funktionale und die logische Programmierung sind deklarative Paradigmen

Viele gängige Programmiersprachen unterstützen mehrere Paradigmen zugleich.

Imperative Programmierung: Motivation

Die Rechnerhardware legt die imperative Programmierung nahe:

- der Rechner hat einen **Zustand**, bestehend aus den Inhalten von Prozessorregistern, Caches, Hauptspeicher, Dateien usw.
- der Prozessor ändert beim Ausführen von Befehlen diesen Zustand

Imperative höhere Programmiersprachen abstrahieren von den technischen Details der Rechnerhardware, behalten aber das Verarbeitungsprinzip der schrittweisen Zustandsänderung bei:

- der Zustand ist über **Variablen** und Ein-/Ausgabekanäle zugänglich
eine Variable ist hier ein Name für einen Speicherbereich
- die Befehle erzeugt der Compiler aus **Anweisungen**
grundlegende Anweisung ist die Variablenzuweisung
- die Menge der Variablen und Anweisungen kann je nach Sprache auf unterschiedliche Weise **strukturiert** werden

Imperative Programmierung: prozedural

Die prozedurale Programmierung stellt die Strukturierung der Anweisungen mittels Prozeduren in den Mittelpunkt:

- Leitfrage: Was muss das Programm tun?
- Programmieraufgaben werden mittels **Prozeduren** schrittweise in überschaubare Teile zerlegt
bzw. bei Bottom-up-Vorgehen: wiederholt auftretende Anweisungsfolgen werden zu Prozeduren zusammengefasst
- aus den Prozeduren ergibt sich die Strukturierung der **Variablen**
Variablen treten als Aufrufparameter und lokale Variablen von Prozeduren auf oder als globale Variablen außerhalb der Prozeduren
- Prozeduren liefern Ergebnisse als Rückgabewert oder als **Seiteneffekt**
Zuweisungen an Ausgabeparameter und globale Variablen sind Seiteneffekte die Ergebnisse können vom gesamten erreichbaren Programmzustand abhängen, nicht nur von den Parameterwerten

Imperative Programmierung: objektorientiert

Die objektorientierte Programmierung stellt die Strukturierung der Variablen mittels Objekte in den Mittelpunkt:

- Leitfrage: Womit muss das Programm umgehen?
- der Programmzustand wird mit **Objekten** modelliert
neue Objekte entstehen bei Klassen-basierten Sprachen durch Klasseninstanziierung und bei Prototyp-basierten Sprachen durch Ableitung von bestehenden Objekten
Variablen treten zuvorderst gekapselt als Instanzvariablen in Objekten auf
- die **Methoden** der Objekte bzw. Klassen strukturieren die Anweisungen
die Anweisungen in den Methoden legen das Verhalten von Objekten fest und sorgen für jederzeit konsistente Objektzustände
- kontrollierte **Seiteneffekte**
in Methoden Zuweisungen nur an die Variablen des Aufrufobjekts

Deklarative Programmierung: Motivation

Die Mathematik legt die deklarative Programmierung nahe:

- **Formeln** beschreiben, welche Eigenschaften eine gesuchte Lösung haben muss, aber nicht unmittelbar, wie die Lösung berechnet werden kann oder soll
z.B. beschreiben Formeln mit verschachtelten Funktionen die Abbildung von Eingangsgrößen auf Ausgangsgrößen
z.B. beschreiben Formeln mit prädikatenlogischen Ausdrücken Einschränkungen einer Lösungsmenge

Deklarative Programmiersprachen übernehmen das Prinzip der mathematischen Formel und ergänzen automatische Lösungsverfahren:

- Variablen sind Platzhalter für **Ausdrücke**, nicht Namen von Speicherbereichen
Variablen sind nicht änderbar (immutable)
- der Compiler / Interpreter legt die Auswertungsreihenfolge der Ausdrücke fest
es sind viele automatische Optimierungen möglich, weil es keine Seiteneffekte in Form von Zustandsänderungen gibt

Deklarative Programmierung: funktional

Bei der funktionalen Programmierung besteht ein Programm aus Ausdrücken mit verschachtelten Funktionen:

- Leitfrage: Wie hängen die Programmausgaben von den Eingaben ab?
- **Funktionen** sind Prozeduren, die keine Seiteneffekte haben und deren Rückgabewert nur von den Aufrufparametern abhängt
Funktionen höherer Ordnung haben Funktionen als Parameter und / oder Rückgabewert
- **Lambdas** (= *anonyme Funktionen*) können dynamisch erstellt werden
zusammen mit dem Erstellungskontext bilden sie Closures
häufig als Argumente oder Rückgabewerte von Funktionen höherer Ordnung
- **Rekursion** statt Schleifen
bevorzugt als Endrekursion (tail recursion), bei der die Rekursion als letzte Operation vor dem return auftritt

Funktionale Programmierung: Currying

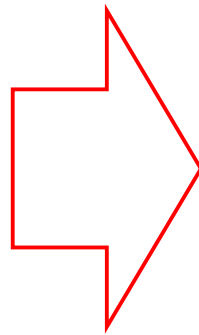
Currying erlaubt es, eine Funktion mit mehreren Parametern durch mehrere Funktionen mit je einem Parameter zu ersetzen

funktionale Programmiersprachen brauchen deshalb im Prinzip keine Funktionen mit mehreren Parametern zu unterstützen (ist z.B. bei Haskell der Fall)

- Beispiel in Python (*entnommen aus de.wikipedia.org/wiki/Currying*):

```
#!/usr/bin/python3
def uncurried_add(x, y):
    return x + y

n = uncurried_add(3, 5)
```



```
#!/usr/bin/python3
def curried_add(x):
    return lambda y: x + y

n = curried_add(3)(5)

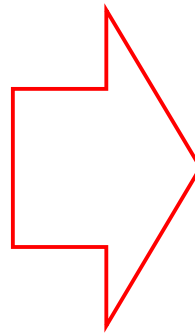
add_three = curried_add(3)
n = add_three(5)
```

curried_add ist eine Funktion höherer Ordnung mit einem Parameter, die eine Funktion mit wiederum einem Parameter zurückliefert

Funktionale Programmierung: Endrekursion

Endrekursion kann ein optimierender Compiler leicht durch eine Schleife ersetzen und so die Laufzeit und den Stack-Speicher der Funktionsaufrufe einsparen:

```
R f(P p) {  
    if (condition) {  
        return value;  
    }  
    ...  
    return f(expression);  
}
```



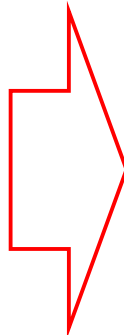
```
R f(P p) {  
    while (true) {  
        if (condition) {  
            return value;  
        }  
        ...  
        p = expression;  
        continue;  
    }  
}
```

Funktionale Programmierung: Continuations

Continuations sind Funktionen, die verbleibende Berechnungen repräsentieren. An die Stelle der Funktionsrückkehr tritt der Aufruf der übergebenen Continuation. (siehe de.wikipedia.org/wiki/Continuation-Passing_Style)

- eine rekursive Funktion kann per Continuation endrekursiv gemacht werden:

```
#!/usr/bin/python3
def fakultaet(n) :
    if (n == 0) :
        return 1
    return n * fakultaet(n - 1)
print(fakultaet(5))
```



```
#!/usr/bin/python3
def fakultaet(c, n) :
    if (n == 0) :
        return c(1)
    fakultaet(lambda f : c(n * f), n - 1)
fakultaet(lambda f: print(f), 5)
```

Funktionale Sprachen mit Tail-Call Optimization können in der endrekursiven Version das dynamische Wachsen des Stacks unterbinden.

Bei Python sind die Continuations hier kontraproduktiv, weil deren Aufrufe die Zahl der Stack-Frames verdoppelt. Python hat keine Tail-Call Optimization.

Deklarative Programmierung: logisch

Bei der logischen Programmierung ist ein Programm eine Liste von speziellen prädikatenlogischen Ausdrücken (Horn-Klauseln):

- Leitfrage: Was ist über die Lösungsmenge bekannt?
- **Fakten** nennen Eigenschaften konkreter Objekte und Relationen zwischen konkreten Objekten
- **Regeln** beschreiben, unter welchen Bedingungen beliebige Objekte bestimmte Eigenschaften haben oder Relationen zwischen beliebigen Objekten bestehen
- auf Grundlage der Fakten und Regeln werden **Anfragen** beantwortet

Fragen nach Eigenschaften von oder Relationen zwischen konkreten Objekten werden mit ja (oder nein) beantwortet

Fragen nach der Existenz von Objekten, die bestimmte Eigenschaften haben oder in bestimmten Relationen stehen, werden mit einer Liste konkreter Objekte beantwortet

Weitere Programmierparadigmen (1)

Denkweisen, die sich mehr auf den Charakter der zu erstellenden Software als auf den Charakter der Programmiersprachen beziehen:

- **Anwendungsprogrammierung**

Umsetzen der Anforderungen von Endanwendern

*als Programmiersprachen kommen eine Vielzahl von GPLs und DSLs in Frage
weit verbreitet sind etwa C#, Java, Python, HTML / CSS / JavaScript*

- **Systemprogrammierung**

Verwaltung und Steuerung von Ressourcen in Rechensystemen

Bereitstellen von Plattformen für Anwendungsprogramme

dominierende höhere Programmiersprachen sind die GPLs C und C++

Die beiden Denkweisen lassen sich nicht scharf voneinander abgrenzen, sie treten mitunter vermischt auf oder sind eine Frage des Blickwinkels

Ist ein Webbrowser ein Anwendungs- oder ein Systemprogramm?

Weitere Programmierparadigmen (2)

Unter dem Begriff Programmierparadigmen werden zahllose weitere Denkweisen beschrieben (*siehe die deutsche und englische Wikipedia*)

Hier nur ein paar weitere Beispiele:

- **Scripting**
*für Kommandoprozeduren, die die Bedienung von Rechnern automatisieren
für Gluecode, der z.B. GUI-Bedienelemente mit Programmlogik verknüpft
Sprachen: z.B. sh, JavaScript, Python, VBScript, ...*
- **Textgenerierung**
*von formatierten Programmausgaben bis zur Generierung von Quellcode und Webseiten
Sprachen: z.B. printf, Stringtemplate, ...*
- Textauszeichnung und -präsentation
Sprachen: z.B. TeX / LaTeX, Html / CSS, XML
- Datenabfrage
Sprachen: z.B. SQL, XQuery / XPath
- ...

Scripting: Eigenschaften

Bei der **Script-Programmierung** werden neue Anwendungen durch einfaches Zusammenfügen bereits vorhandener Anwendungen und Komponenten erstellt.

Mit mächtigen GPLs kann man komplexe Anwendungen und Komponenten sicher und effizient von Grund auf implementieren. Will man solche Anwendungen und Komponenten lediglich verwenden, reichen flexiblere und leichter zu lernende Scriptsprachen.

Einige Typische Eigenschaften von Scriptsprachen:

- deklarationsfreie Syntax
Namen werden implizit deklariert und ihre Typen dynamisch bestimmt
- ausgefeilter Umgang mit Strings
Pattern-Matching mit regulären Ausdrücken
Strings als Array-Index
Strings als Code ausführen
- zeilenweise Interpretation des Quelltexts statt Übersetzung
dadurch auch interaktiv nutzbar

Scripting: Beispiele (1)

Mit einer Unix-Shell wie **sh** lassen sich Anwendungen bequem aus vorhandenen ausführbaren Programmen und Systemressourcen zusammensetzen.

- Umlenkung der Standard-Ein-/Ausgabe

Programm1 **|** *Programm2*

Pipe als Standard-Ausgabe für Programm1 und Standard-Eingabe für Programm2

Programm **<** *Datei1* **>** *Datei2* **2>&1**

Datei1 als Standard-Eingabe, Datei2 als Standard-Ausgabe und Fehlerausgabe (2) auf Standardausgabe (1) umleiten

- sequentielle Ausführung

Programm1 **;** *Programm2* *unbedingte Sequenz*

Programm1 **&&** *Programm2* *Programm2 nur, wenn Programm1 mit Exitcode 0*

Programm1 **||** *Programm2* *Programm2 nur, wenn Programm1 mit Exitcode nicht 0*

- Dateien mit regulären Ausdrücken auswählen

***.[ch]** *alle Dateien mit Endung .c und .h im aktuellen Arbeitsverzeichnis*

Scripting: Beispiele (2)

Mit einer Scriptsprache wie Python lassen sich zusammengesetzte Datentypen deklarationsfrei und flexibel nutzen.

- Listen können Elemente verschiedener Typen enthalten und sind zugleich wie Arrays nutzbar:

```
liste = [123, 456, 890]    # neue Liste
liste[1] = 4.56            # zweites Element bekommt neuen Typ und Wert
liste.insert(1, 'Hallo')   # zusätzliches Element hinter dem ersten
liste.append(liste.pop(0)) # erstes Element wird ans Ende verschoben
print(liste)               # Ausgabe: ['Hallo', 4.56, 890, 123]
```

- weitere zusammengesetzte Typen:

```
tupel = (10, 20, 10)      # im Gegensatz zur Liste nicht änderbar
menge = {'Hallo', 'Hi'}   # kein Index und keine doppelten Werte
dictionary = {'x':1, 'y':2} # Schlüssel-Wert-Paare, Schlüssel als Index
```


Textgenerierung: Eigenschaften

Bei DSLs für die **Textgenerierung** werden Lückentexte mit Daten gefüllt.

*spielt vor allem bei der Generierung von HTML-Seiten eine große Rolle,
aber auch bei der Generierung von Programmcode aus Modellen*

- **Formatierungssprachen** sind in eine Wirtssprache eingebettete DSLs:

der Lückentext ist aus Sicht der Wirtssprache ein gewöhnlicher String

Bibliotheksfunktionen bzw. -klassen interpretieren zur Laufzeit bestimmte Zeichenfolgen im String als Lücken und setzen dort formatierte Daten ein

Beispiele: C printf, Java String.format, Python str.format

- **Templatesprachen** sind eigenständige DSLs:

die Lückentexte werden abgesetzt in eigenen Dateien erstellt

es gibt syntaktische Mittel, um zum Füllen der Lücken durch ein Datenmodell zu iterieren, die Syntax wird von einer Templateengine interpretiert

Beispiele: VTL (Velocity Template Language), StringTemplate

Textgenerierung: Beispiel (1)

StringTemplate ist eine Templatesprache und Java Template-Engine von Terence Parr (*siehe auch ANTLR in Teil 2*).

Lückensyntax (*statt der spitzen Klammern sind auch andere Begrenzer möglich*):

- Referenzen erlauben den Zugriff auf Java-Objekte eines Modells:

`<attribute.property>`

Attribute sind sozusagen Parameter von Templates und werden javaseitig mit Objekten eines Modells initialisiert.

Properties werden auf Instanzvariablen oder getter-Methoden der Objekte abgebildet.

- funktionaler Stil für die Iteration über Modellelemente:

`<attribute.property:Template()>`

`<attribute.property:Template(); separator="...">`

auf alle Elemente der Property das angegebene Template anwenden

- Bedingte Template-Abschnitte:

`<if(Bedingung)> ... <elseif(Bedingung)> ... <else> ... <end>`

Textgenerierung: Beispiel (2)

- Template-Gruppe mit zwei Templates für die Generierung einer HTML-Seite:

`delimiters "$", "$"`

spitze Klammern als Begrenzer bei HTML ungeeignet

`notenspiegel(n) ::= <<`

`<!DOCTYPE html>`

`<html lang="de">`

`<body>`

`NOTENSPIEGEL`

`<table>`

`$n:fachnote(); separator="\n"$`

`</table>`

`</body>`

`</html>`

`>>`

*Template **notenspiegel** mit Attribut **n**
und mehrzeiligem Rumpf von << bis >>*

*wendet das Template **fachnote** auf
alle Elemente aus **n** an und fügt nach
jedem Element einen Zeilenwechsel ein*

`fachnote(f) ::= <<`

`<tr>`

`<td>$f.fach$</td>`

`<td>$if(f.benotet)$L$else$$endif$</td>`

`<td>$f.note$</td>`

`</tr>`

`>>`

*Template **fachnote** mit Attribut **f***

*greift auf **fach**, **benotet** und
note im Datenmodell zu*

Textgenerierung: Beispiel (3)

- Datenmodell Fachnote.java:

```
public final class Fachnote {  
    public final String fach;  
    public final boolean benotet;  
    public final String note;  
  
    public Fachnote(String fach, double note) {  
        this.fach = fach;  
        this.benotet = true;  
        this.note = String.format("%.1f", note);  
    }  
  
    public Fachnote(String fach, boolean bestanden) {  
        this.fach = fach;  
        this.benotet = false;  
        this.note = bestanden ? "be" : "nb";  
    }  
}
```

Textgenerierung: Beispiel (4)

- Modellinstanziierung und Aufruf der Template-Engine:

```
import org.stringtemplate.v4.STGroupFile;
```

```
import org.stringtemplate.v4.ST;
```

```
...
```

```
Fachnote[] fachnoten = new Fachnote[] {  
    new Fachnote("Sprachkonzepte", 1.3),  
    new Fachnote("Sprachkonzepte Uebungen", true)  
};
```

```
ST template  
    = new STGroupFile("notensiegel.stg").getInstanceOf("notenspiegel");  
template.add("n", fachnoten);
```

```
String htmlSeite  
    = template.render();
```

```
...
```

*Initialisierung des
Templates-Attributs*

*Name des Templates
in der Gruppen-Datei*