Programmierparadigmen

Die Informatik kennt vier wesentliche Grundkonzepte von höheren Programmiersprachen, die man als Programmierparadigmen bezeichnet. Es sind das imperative (prozedurale), das funktionale (applikative), das prädikative (logische) und das objektorientierte Programmierparadigma[1].
Obwohl bei konkreten Programmiersprachen durchaus fließende Grenzen auftreten, lassen sich die Sprachen nach dieser Unterscheidung ordnen. Im Unterricht ist beim Wechsel von JAVA zu Scheme deutlich geworden, dass nicht nur anders programmiert wird, sondern dass auch ein anderes Denken gefordert ist.

- imperativ (prozedurales Konzept)

Man versteht ein Programm als eine Befehlsfolge. Dem Computer wird also der Ablauf beschrieben, in dem er Aktionen ausführen soll. Neben dem eigentlichen Programm ist der Datenbestand zu verwalten, der in Variablen gehalten wird.
Ein typisches Beispiel solcher Sprachen ist PASCAL und Delphi zeigt, dass anderen Sprachen heute oft ein objektorientierter Aufsatz mitgegeben wird. Typische Wiederholungsstuktur ist die Schleife.

- applikativ (funktionales Konzept)

Da Funktionen prinzpiell die Aufgabe haben, auf der Grundlage von vorgegebenen Ausgangswerten neue Werte zu berechnen, wird hier die Datenbasis nicht als statisch betrachtet. Das Programm besteht also aus dem Auswertungsverfahren, das auf beliebige (aber natürlich passende) Werte anwendbar sein soll. Typische Wiederholungsstuktur ist die Rekursion.
Besonderes Merkmal von funktionaler Programmierung ist der geschachtelte Aufruf: Gemäß der reinen Lehre funktionaler Programmierung besteht die Auswertung allein aus einer Schachtelung von Auswertungen, d.h. eine Teilfunktion liefert einen Teilwert, der an die aufrufende Funktion weiter gereicht wird, von der weiter mit anderen Teilergebnissen zu einem eigenen Ergebnis führt, das dann ...
Eine Auswertung, die keinen verwendeten Wert (void) liefert, wäre dabei nicht zulässig. Sie wissen von unserer Arbeit mit Scheme, dass wir solche Nebeneffekte durchaus einsetzen und damit gegen dieses strenge Konzept verstoßen.
Die Parameter von Funktionen haben eine besondere Bedeutung. Sie können in der Regel beliebigen Typ haben. So sind auch wieder Funktionen selbst als Parameter zulässig und ermöglichen so in einem hohen Maße Generalisierungen: Als Beispiel sei eine Funktion zur Sortierung einer Datenmenge genannt, der man eine zum speziellen zu sortierenden Datentyp passende Vergleichsfunktion mitgibt.
Funktionale Sprachen ermöglichen damit auch in einem besonderen Maße Kapselung von Funktionen, da – betrachten wir das o.a. Beispiel – für die Lösung nicht bekannt sein muss, wie intern die Sortierung realisiert wird; man achte nur darauf, ihr die richtige Vergleichsfunktion zu den Daten zu übergeben.

- prädikativ

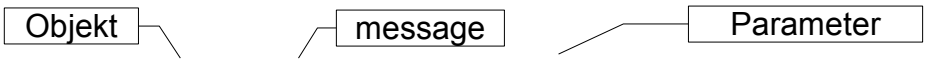
Programme bestehen aus Fakten und Regeln und eine solche Sprache muss die

1 Von diesen vier Programmierparadigmen haben wir drei kennen gelernt, auf das prädikative haben wir verzichtet und werden das gemäß Rahmenplan auch weiterhin.
Die Ereignis – gesteuerte Programmierung wird meines Wissens nach im deutschen Sprachraum nicht als Programmierparadigma bezeichnet, obwohl das sicher sinnvoll ist.

Möglichkeit bieten, Beweise von neuen Aussagen auf dieser Basis zu liefern[2].

- objektorientiert

Alles, was zur Problemlösung notwendig ist, wird als Objekt aufgefasst. Objekte mit gleichen prinzipiellen Eigenschaften und Fähigkeiten werden zu Klassen zusammengefasst.
Ein Programm besteht dann aus einer Menge von Objekten, die miteinander Botschaften austauschen. Dieser „message – passing – style“ ist also ein typisches Merkmal von OO Sprachen.
Wir haben das einmal an der Umsetzung des streng objektorientierten Konzeptes bei SMALLTALK erläutert. Will man dort zwei Zahlen subtrahieren, dann schickt man der ersten Zahl die message subtrahiere und übergibt ihr dazu als notwendigen Parameter die zweite Zahl. In JAVA würden wir schreiben:




```
ersteZahl.subtrahiere(zweiteZahl)
```

Man erhält als Wert dann wiederum ein neues Zahlobjekt.
Dabei ist die Einheit von Daten und Methoden ein besonderes Merkmal, das Lösung in einem objektorientierten Programm von einer prozeduralen unterscheidet. Das ermöglicht dann auch das weitere Merkmale Kapselung, die uns allerdings auch schon bei der funktionalen Programmierung begegnet ist. Die Kapselung bezieht sich hier aber in viel stärkerem Maße auf die Daten. Jedes Objekt hat seine eigenen (möglichst privaten) Daten, auf die von außen nur durch die dafür zuständigen Methoden zugegriffen werden kann. So hat jedes Objekt selbst, wenn es nicht aktiv ist, einen eigenen Zustand, der sich in den aktuellen Werten der Attribute ausdrückt.

Man ändert den Zustand eines Objektes, indem man ihm eine dafür zuständige message schickt.

Objektorientierte Programmierung kennt als ein besonders über die anderen Sprachen hinaus gehendes Konzept die Vererbung. Durch sie gelingt es objektorientierten Sprachen zu generalisieren, indem gemeinsame Eigenschaften von zunächst verschiedenen Objekten in einer neuen generalisierenden Klasse ausmodelliert werden.
Dazu gehört dann auch die Möglichkeit der Polymorphie: Durch Überschreiben können Klassen aus einer Vererbungskette verschiedene Varianten der selben Methode (mit derselben Signatur) definieren. Je nach der Klasse des erzeugten Objektes führt es dann genau die zu ihm gehörende Variante der Methode aus. Das geschieht selbst dann, wenn der Aufruf dieser Methode von einer anderen Methode aus erfolgt, die nicht innerhalb seiner eigenen Klasse, sondern innerhalb einer Klasse implementiert wurde, von der sie erbt.

- Ereignis – gesteuerte Programmierung[1]

Eines der wichtigsten Konzepte moderner Programme mit grafischer Oberfläche (**g**raphical **u**ser **i**nterface) ist die Steuerung des Programmablaufes durch Ereignisse. Das bedeutet, dass unabhängig vom gewählten Programmierparadigma der Ablauf eines Programms nicht allein vom vorher festgelegten Programmcode abhängt. Zusätzlich zu den im Code festgelegten Abläufen reagiert das Programm auf Ereignisse „von außen“.

2 Beispielsprache ist Prolog.
1 event-driven programming or event-based programming

Ereignisse

Diese Ereignisse können Mausereignisse oder Tastenereignisse sein, sie können aber auch andere Quellen haben.

nicht nur von außen

Damit ein Programm seinen Ablauf „von außen“ steuern lässt, muss es diese Eigenschaft aber zwingend selbst bereitstellen[1]. Es muss von Zeit zu Zeit seinen eigenen sonstigen Ablauf unterbrechen, um die angemeldeten Steuerungselemente abzufragen.
Für den Fall, dass ein Ereignis auftritt, muss das Programm auch den event-handler bereitstellen (eine Methode, Prozedur, Funktion,...), in dem die Reaktionen des Programmes auf die Aktion beschrieben sind.
Ein typisches Szenario Ereignis – gesteuerter Programmierung ist, das ein Programm nach seinem Start eine Initialisierungsphase durchläuft und dann in einen Wartezustand geht, in dem nun nur noch nacheinander die registrierten Ereignissender abgefragt werden, ob bei ihnen ein Ereignis aufgetreten ist. Tritt es auf, kann abgefragt werden, welcher Typ von Ereignis es war und in der Regel wird das Programm nun selbstständig den geplanten Ablauf durchführen. Anschließend geht das Programm dann wieder in einen Ruhezustand.

Zustandsänderung durch Ereignisse

Ereignissteuerung hat immer eine Zustandsänderung zur Folge. Das System ändert seinen Zustand in vielen Fällen so umfangreich, dass dasselbe Ereignis zu einem späteren Zeitpunkt andere Folgen hat. Gerade bei funktionaler Programmierung ist unbedingt darauf zu achten, dass dies berücksichtigt wird.

1 Wenn Programme auf Betriebssystemen basieren (Windows, linux, ...), stellen auch diese eine Ereignissteuerung zur Verfügung. Verschiedene Programme laufen in verschiedenen threads und wenn nicht das Programm selbst, dann kann (hoffentlich!) das Betriebssystemen auf Ereignisse reagieren und stoppt dann die laufenden threads.