# Modellierung der Breitensuche

Die Entwicklung des Programms zur Breitensuche ist eine gute Möglichkeit, eine der Grundlagen der Entwicklung von Software zu demonstrieren.

Es geht um das Beachten der Kohäsion. Angewandt auf die funktionale Programmierung fordert man:

> **Jede Funktion ist für genau eine Aufgabe zuständig.**
>
> **Jede Funktion sollte möglichst wenig Strukturebenen haben.**

## *Analyse*

Das hat zur Folge, dass man mit der Untersuchung der Problemlösung auf die unterschiedlichen Aufgaben hin beginnen sollte.

Einige Teilaufgaben kennen wir bereits von der Tiefensuche, die Funktionen **`exakt-voll?`** und **`zu-voll?`**:

```
(define (exakt-voll? container kapazitaet)
   (= kapazitaet (fuellung container)))
```

und

```
(define (zu-voll? container kapazitaet)
   (< kapazitaet (fuellung container)))
```

Beide greifen auf eine weitere, hier ausgegliederte Funktion zu, die Funktion fuellung:

```
(define (fuellung container)
   (apply + container))
```

Ob das zusätzliche Einführen der Funktion **`fuellung`** einen Zugewinn bringt, mag man bezweifeln, es zeigt aber das Prinzip des Vorgehens.

Mit diesen Funktionen sind wir auf der untersten Programmebene, setzen jetzt aber auf der oberen Programmebene fort:

### *Weitere Modellierungsschritte*

Die Aufgabe lässt sich global in vier Teilaufgaben zerlegen:

- die Bearbeitung aller Fälle in einer Ebene
- die Expansion einer neuen Ebene
- die Steuerung des Ablaufs
- die Aufrufhülle

Mit dieser Gliederung wird – beginnend am Kopf – nach dem Prinzip der schrittweisen Entwicklung begonnen:

Die Definition einer Aufrufhülle

```
(define  (suche stuecke kapazitaet)
  ; realisiert den passenden Aufruf und Ausgaben
  "noch nicht realisiert")
```

dazu die interne Steuerungsfunktion

```
(define  (steuerung stuecke kapazitaet warteschlange akku)
  ; steuert die anderen Funktionen
  "noch nicht realisiert")
```

und die Bearbeitung einer Ebene. Hier wird auf Erfolg (exakt voll) und Misserfolg (zu voll

oder keine Stücke mehr) getestet und

- entweder die Loesung oder **#f** zurueckgegeben oder
- aus der Warteschlange das zu volle Element entfernt und
- schließlich die verbliebene Warteschlange (WS) zurueckgegeben.

```
(define  (bearbeite kapazitaet warteschlange akku)
  "noch nicht realisiert")
```

Die Expansion einer Ebene nimmt die vorhandene Warteschlange und erzeugt alle Alternativen, mit dem neuen Stueck und ohne. Wegen der Endrekursion wird eine Akkumulatorliste benötigt.

```
(define  (expandiere stueck warteschlange akku)
  "noch nicht realisiert")
```

### *Wo geht es weiter?*

Das ist bei diesem Vorgehen prinzipiell beliebig. Oft fällt das Testen der entwickelten Teilfunktionen einfacher, wenn man auf der unteren Programmebene beginnt.

#### expandiere

Wir wählen daher die Funktion expandiere. Sie bekommt als Parameter die **warteschlange** übergeben und erzeugt die neue, die danach alle bisherigen Elemente, einmal mit und einmal ohne das neue **stueck** enthält. Da wir endrekursiv arbeiten, benötigen wir für die aufzubauende WS einen **akku**. Abbruchbedingung ist die leere WS, dann wird der akku zurückgegeben.

```
(define
  (expandiere stueck warteschlange akku)
  (cond
    ((null? warteschlange)
     akku)
    (else
     (expandiere
      stueck
      (rest warteschlange)
      (cons                                    ; in den akku
       (cons stueck (first warteschlange))     ; in die Containerliste
       (cons (first warteschlange) akku)       ; in den akku[1]
      )))))
```

Ein Testaufruf

```
(expandiere 20 '((30 30 40) (30 40) (30) (40)) '())
```

ergibt

```
((20 40) (40) (20 30) (30) (20 30 40) (30 40) (20 30 30 40) (30 30 40))
```

#### bearbeite

Die Funktion bearbeite stellt eine Filterfunktion dar. Im Erfolgsfall wird eine Liste mit allein dieser Lösung zurückgegeben.

```
(define  (bearbeite kapazitaet warteschlange akku)
  (cond
    ((exakt-voll? (first warteschlange) kapazitaet)
     (list (first warteschlange)))
    (else
```

1 Siehe Hinweis am Schluss dieses Textes

```
     "noch nicht realisiert")))
```

Allerdings zeigt der Zugriff auf den Kopf von **warteschlange**, dass wegen der Rekursion mit Abbau dieser Liste vorher geprüft werden muss, ob sie leer ist. In dem Fall ist die Ebene vollständig bearbeitet und es erfolgt ein Rücksprung mit dem **akku** als Wert.

```
(define
  (bearbeite kapazitaet warteschlange akku)
  (cond
    ((null? warteschlange)
     akku)
    ((exakt-voll? (first warteschlange) kapazitaet)
     (list (first warteschlange)))
    (else
     "noch nicht realisiert")))
```

Außerdem ist der Fall zu bearbeiten, dass das aktuelle Element am Kopf von **warteschlange** bereits zu voll ist. In dem Fall wird es nicht in den **akku** übernommen ...

```
(define
  (bearbeite kapazitaet warteschlange akku)
  (cond
    ((null? warteschlange)
     akku)
    ((exakt-voll? (first warteschlange) kapazitaet)
     (list (first warteschlange)))
    ((zu-voll? (first warteschlange) kapazitaet)
     (bearbeite kapazitaet (rest warteschlange) akku))
    (else
     "noch nicht realisiert")))
```

… nun fehlt nur noch der Standardfall, das Element wird übernommen.

```
(define
  (bearbeite kapazitaet warteschlange akku)
  (cond
    ((null? warteschlange)
     akku)
    ((exakt-voll? (first warteschlange) kapazitaet)
     (list (first warteschlange)))
    ((zu-voll? (first warteschlange) kapazitaet)
     (bearbeite kapazitaet (rest warteschlange) akku))
    (else
     (bearbeite kapazitaet
                (rest warteschlange)
                (cons (first warteschlange) akku)))
    )
  )
```

#### steuerung

Sie steuert die anderen Funktionen. Es treten die Fälle auf:

- das erste Element ist die Lösung, sie wird zurückgegeben
  Anmerkungen:
  - "erstes Element" wegen der Rückgabe der Funktion bearbeite
  - Da die WS sicher zumindest die leere Liste enthält, die stets zulässig ist, kann sie nicht leer sein
- die Liste **stuecke** ist leer, es gab keine Lösung
- sonst wird eine neuer Durchlauf gestartet.

```
(define
  (steuerung stuecke kapazitaet warteschlange)
  (cond
    ((exakt-voll? (first warteschlange) kapazitaet)
     (first warteschlange))
    ((null? stuecke)
     #f)
    (else
     (steuerung
      (rest stuecke)
      kapazitaet
      (bearbeite
       kapazitaet
       (expandiere (first stuecke) warteschlange '())
       '()))))
    ))
```

#### suche

Sie ist allein eine Aufrufhülle, die einen Aufruf ohne unverständliche Übergaben sowie eine verständliche Ausgabe ermöglicht.

```
(define
  (suche stuecke kapazitaet)
    (write-line
     "Suche mit " stuecke
    " Kapazitaet " kapazitaet
    " ergibt "
    (steuerung stuecke kapazitaet '(((((())
```

Ein Testaufruf mit

```
(suche '(60 30 30 20 20 20)  100)
```

liefert

```
Suche mit (60 30 30 20 20 20) Kapazitaet 100 ergibt (20 20 60)
```

#### Hinweis zur Funktion expandiere

Die cons-Schachtelung

```
(cons
 (cons stueck (first warteschlange))
 (cons (first warteschlange) akku))
```

ist vielleicht etwas unübersichtlich, wird möglicherweise verständlicher durch append:

```
(append
 (list
  (cons stueck (first warteschlange))
  (first warteschlange))
 akku)
```

Das Ausgliedern in eine eigene Funktion könnte sinnvoll sein:

```
(define (mit-und-ohne stueck container)
  (list
   (cons stueck container)
   container))
```

aufgerufen dann durch die Zeilen

```
(append
 (mit-und-ohne stueck (first warteschlange))
 akku)
```