Data structures programs

(Daten strukturieren Programme)

Patrick Cornelißen

Inhaltsverzeichnis

1. Einleitung

Thema dieses Vortrages ist es, dem Zuhörer nahe zubringen, dass es sinnvoll ist, sein Programm nach den Daten zu strukturieren und nicht umgekehrt. Des weiteren werden einige Beispiele erbracht, anhand derer man solche Techniken sehen kann.

2. Umfragen auswerten – ein erstes Beispiel

Inhalt dieses Beispiels ist eine fiktive Umfrage unter Studierenden an vier Universitäten. Erfasst wird u.a. die Universität, das Geschlecht, die Fakultät und vieles mehr. Die Daten werden in Strukturen in dem folgenden Format eingelesen:

- Universität (Werte: 0, 1, 2, 3)
- Geschlecht (Werte: 0, 1)
- Fakultät (Werte: 0, 1, ..., 19)

Eine Möglichkeit der Verarbeitung wäre es, sehr viele einzelne Variablen zu haben, in denen man entsprechend die Anzahl der Antworten zählt und dazu jeweils noch eine Reihe von Variablen für Enthaltungen.

Nachteil:

- sehr langes Programm, unübersichtlich, schwer zu warten

Eine erste Vereinfachung besteht daraus Arrays zu benutzen, ein naiver Ansatz wäre dann evtl. folgender:

Zwei mehrdimensionale Tabellen mit dem Aufbau:
Count[Uni][Frage][Antwort]=Anzahl der Antworten zu der Frage
Declined[Uni][Frage]=Anzahl der Bögen mit der Frage ohne Antwort

Nachteil:

- unnötig hoher Speicherplatzverbrauch, da die letzte Dimension von der Frage mit der höchsten Antwortanzahl abhängt! Eine Ausweitung der Umfrage auf eine weitere Universität würde den Platzbedarf um #Fragen*max(#Antworten_aller_Fragen) steigern. Dies ist offensichtlich nicht unerheblich.

Durch diese Lösung erkauft man sich eine geringere Komplexität durch einen erhöhten Speicherplatzverbrauch. Dies ist aber nicht immer erwünscht; eine weitere wesentlich flexiblere Lösung besteht darin, eine Offsettabelle zu benutzen. Die Funktionsweise dieser Methode wird jetzt im Folgenden erläutert.

Man definiert sich drei Tabelle mit den Dimensionen:

int [#Universitäten][$\sum_{i=1}^{\#Fragen} \#Antworten\_von\_Frage\_i$] Count

int [#Fragen] Offset

int [#Universität][#Fragen] Declined

Der Speicherverbrauch von Declined hat sich nicht verändert, Count wurde aber gesplittet in Count und Offset. Eine Gegenüberstellung der Platzkomplexität erfolgt nach Besprechung des dazugehörigen Algorithmus.

Nachteil:

- komplex

Den Aufbau von Count und Offset ist wie in der unten stehenden Tabelle zu verstehen:

Count

Uni 1	#(Frage1 Antwort1)	#(Fr1A2)	#(Fr2A1)	#(Fr2A2)	#(Fr2A3)	...
Uni 2	s.o.	s.o.	s.o.	s.o.	s.o.	s.o.

Ebenso für Universität 3 und 4 ...

Offset bekommt bei Programmstart folgende Werte:

0	0	2	22	22 + #Antworten_Frage4	22 + #Antworten_Frage4+#Antworten_Frage5	...

Der Algorithmus zum Einlesen der Antworten würde für einen Datensatz nun folgendermaßen aussehen:

```
For i=0 to EndIndex do
        If Entry[i]=refused then
                declined[entry[0]][i]++
        Else count[entry[0]][Offset[i]+Entry[i]]++
```

Daraus wird auch offensichtlich, warum in Offset die ersten beiden Stellen mit 0 initialisiert wurden, da im ersten Feld von Entry die Universität codiert ist und das zweite Feld ist 0, weil die Antworten für die erste Frage an Stelle 0 beginnen.

Jetzt möchte ich diese drei Lösungsmöglichkeiten einmal in Hinsicht auf die genannten Kriterien vergleichen.

	Einzelne Variablen	**1. Variante mit Arrays**	**2. Variante**
Erweiterbarkeit	Hinzufügen der neuen Variablen (u.U. schwer in den Code zu integrieren)	Redefinition der Arrays, Code ist komplett zu überprüfen	OffsetTabelle und Count sind anzupassen, bei entsprechender Programmierung ist keine weitere Anpassung nötig
Codelänge	lang und unübersichtlich	kurz und übersichtlich	kurz und übersichtlich
Codekomplexität	gering	mittel	höher
Platzkomplexität	#Universitäten * $\sum_{i=1}^{\#Fragen} \#\,Antworten\_Frage\_i$	#Universitäten * max(#Antworten)	#Universitäten * $\sum_{i=1}^{\#Fragen} \#Antworten\_Frage\_i$

Man sieht, dass die mittlere Lösung aufgrund der hohen Platzkomplexität bei ungünstigen Frage-Antwort Konstellationen suboptimal ist. Die erste Lösung ist zu unflexibel und aufgrund ihres Aufbaus eher als „Hack“ denn als optimale Lösung zu bezeichnen. Die dritte Lösung ist sehr flexibel, braucht wenig Platz, aber sie benötigt mehr Konzentration beim Programmieren, da der Code vergleichsweise komplex ist.

Die Laufzeit, des ersten und letzten Beispiels, sollte bei entsprechend guter Programmierung ungefähr gleich sein.

Die Überlegenheit der beiden Array-Lösungen ist offensichtlich, da auch vor allem die Erweiterbarkeit nicht leidet.

3. Zählen von Elementen

Eine weitere Aufgabe, vor die man immer wieder gestellt wird, ist das Zählen von bestimmten Elementen. Anfänger werden dies evtl. mit vielen Variablen erledigen. Eine Prozedur, die diese Aufgabe löst, könnte dann so aussehen:

```
void zähl(int k){
        if (k=0) c000++
        if (k=1) c001++
        if (k=2) c002++
        if (k=3) c003++
        …
        if (k=42) c042++
        …
}
```

Diese Lösung ist, wie man schnell sieht, suboptimal. Erweitern ist schwer und die Auswertung ist komplex. Verwendet man kein „Else“ wie hier dann ist auch die Laufzeit O(#Elemente) anstatt O(1).

Besser wäre die Verwendung eines Arrays mit „#Elemente“ Elementen. Vorteile sind die geringe Codelänge (Zählen mittels c[k]++ anstatt der langen Prozedur) und leichte Erweiterbarkeit. Nachteilig ist nur, dass man auf einen Datentyp festgelegt ist (außer bei Verwendung von Variant) und dass man auf die Grenzen des Arrays besonders aufpassen muss.

4. Serienbriefe

Serienbriefe der Art:

```
Hallo vorname nachname,
Du hast gewonnen! Wir senden dir deinen Gewinn in die
strasse in PLZ stadt, wenn du jetzt bestellst...
```

hat jeder schon mehr als einmal bekommen. Die Programmierung eines solchen Serienbriefgenerators ist nicht sonderlich kompliziert. Man kann jedoch viele Fehler im Konzept machen. Im Folgenden werde ich wieder zwei Varianten der Programmierung vorstellen. Zuerst den naiven Ansatz:

```
Read nachname, vorname, strasse, stadt, PLZ
print „Hallo „ + vorname + “ „ + nachname
print „Du hast gewonnen! Wir senden dir deinen Gewinn
in die“
```

```
print strasse+“ in „+ PLZ + „ „ + stadt + „, wenn du
jetzt bestellst...“
```

Dieser Ansatz produziert zu gelesenem Namen und Adresse einen Serienbrief, der dem Leser auf ihn zugeschnitten erscheint, aber leider ist der Text des Briefes nicht, bzw. nur durch neu übersetzen des Programmes veränderbar. Das ist eine, für den alltäglichen Einsatz, im Normalfall nicht hinnehmbare Einschränkung.

Die Anforderungen an einem normalen Serienbrief sind:
Parametrisierbar oder Datenbankanbindung
Text soll veränderbar bleiben

Diese Anforderungen werden von Serienbriefgeneratoren erfüllt, die sich Schemata, bzw. Vorlagen, bedienen. Sie parsen einen vom Benutzer erstellten Text und setzen an den vorgegebenen Positionen die entsprechenden Felder des aktuellen Datensatzes ein.

Eine mögliche Implementation sieht so aus:

```
Read fields from database
Loop from start to end of schema
     c = next char in schema
     if c != ‘$‘ printchar c
     else
          c = next char in scheme
          case c of
               ‘$‘: printchar c
               ‘0‘..‘9‘:printstring field[c]
               default: error(,Fehler in Vorlage‘)
```

Das Programm ist äußerst flexibel, simpel und kurz.

5. Arrays sinnvoll nutzen

5.1. Menüeinträge organisieren

In vielen Programmen sieht man Menüleisten, in denen man ein Element aktiviert (mit einem Haken versieht) und damit ein anderes Element deaktiviert. Es ist also immer nur eines gleichzeitig aktiv.

Sind diese Menüeinträge in einzelnen Variablen organisiert, sieht ein OnClick eines Menüelementes wahrscheinlich so aus:

```
Sub menuitem0_click()
     Menuitem0.checked = 1
     Menuitem1.checked = 0
     Menuitem2.checked = 0
     Menuitem3.checked = 0
     Menuitem4.checked = 0
```

```
        Menuitem5.checked = 0
        Menuitem6.checked = 0
```

Dies sind meist gewachsene Lösungen, die mal aus zwei bis drei Menüeinträgen bestanden und danach erweitert wurden. Dies ist nicht nur unelegant, sondern auch unnötig lang. Wird außerdem mal ein Menüeintrag umbenannt, muss man alle Prozeduren verändern.

Eleganter kann man das natürlich wieder mit Arrays lösen. Dadurch reicht es, wenn man eine parametrisierte Prozedur erstellt, die alle anderen deaktiviert und das entsprechende Element aktiviert.

```
Sub Menuitem_click(int i)
    for i = 0 to numchoices
        Menuitem[i].checked = 0
    Menuitem[i].checked = 1
```

Wieder ist es sehr einfach, ein weiteres Element zu den Menüs hinzuzufügen und die „Komplexität“ liegt in einer Prozedur.

5.2. Sukzessive Mittelwertbildung

Im Wintersemester 00/01 habe ich bei einer Gruppe von Mathematik-LehramtsstudentInnen eine Programmierberatung durchgeführt. Eine Problemstellung war die sukzessive Mittelwertbildung von Funktionswerten zur Bestimmung von Näherungswerten.

Alle erkannten, dass man diese Aufgabe am besten mit Arrays lösen kann. Doch die meistgewählte Lösung war bei weitem nicht optimal.
Es wurde ein sehr großes Array initialisiert und in dem wurde dann gerechnet. Wenn das Abbruchkriterium erfüllt war, wurde die Lösung ausgegeben und das Programm terminierte. Nachteile dieser Methode werden einem schnell klar, wenn man sich ansieht, wie gerechnet wird:
In der ersten Spalte stehen Werte einer Funktion, die in der n-ten Zeile n-mal iteriert wird. Dann wird aus jeweils zwei untereinander stehenden Werten der Mittelwert berechnet, dies wird dann in der nächsten Spalte abgespeichert. Ist man mit der aktuellen Spalte fertig, wird in der nächsten Spalte genauso verfahren. Dadurch ist dieses Array nur halb gefüllt, der Rest bleibt leer. (Es sieht dann wie eine obere Dreiecksmatrix aus, da in der nächsten Spalte immer ein Wert weniger als in der aktuellen Spalte steht) Das Abbruchkriterium ist die Differenz des ersten Elementes der letzten beiden berechneten Spalten. Abgebrochen wird, wenn diese kleiner ist, als ein vorher festgelegter Grenzwert.

An einem kleinen Beispiel erläutere ich eine Lösungsvariante, die einem nicht unerhebliche Laufzeit- und Platzersparnis bringt. Die aktuelle Lösung braucht $|\text{Arraygröße}|^2$ Platz und die Laufzeit ist analog $O(|\text{Arraygröße}|^2)$.

Die unten stehende Tabelle gibt das Array nach drei Mittelwertbildungen wieder.

F1	MW1(F1,F2)	MW12(MW1,MW2)	MW1(MW12,MW23)	0
F2	MW2(F2,F3)	MW23(MW2,MW3)	MW2(MW23,MW34)	0
F3	MW3(F3,F4)	MW34(MW3,MW4)	MW3(MW34,MW45)	0
F4	MW4(F4,F5)	MW45(MW4,MW5)	MW4(MW45,MW56)	0

F5	MW5(F5,F6)	MW56(MW5,MW6)	MW5(MW56,MW67)	0
F6	MW6(F6,F7)	MW67(MW6,MW7)	MW6(MW67,MW78)	0
…	…	…	…	…

Die beiden grünen Felder sind in diesem Schritt relevant für das Abbruchkriterium.

Bei genauer Betrachtung wird man feststellen, dass man nur die in der unten stehenden Tabelle grau markierten Felder benötigt, um das nächste Feld an Position (5,1) zu berechnen.

F1	MW1(F1,F2)	MW12(MW1,MW2)	MW1(MW12,MW23)	0
F2	MW2(F2,F3)	MW23(MW2,MW3)	MW2(MW23,MW34)	0
F3	MW3(F3,F4)	MW34(MW3,MW4)	MW3(MW34,MW45)	0
F4	MW4(F4,F5)	MW45(MW4,MW5)	MW4(MW45,MW56)	0
F5	MW5(F5,F6)	MW56(MW5,MW6)	MW5(MW56,MW67)	0
F6	MW6(F6,F7)	MW67(MW6,MW7)	MW6(MW67,MW78)	0
…	…	…	…	…

Aus Feld F4 und F5 kann man MW4(F4,F5) berechnen, aus MW3(F3,F4) und dem eben berechneten MW4 kann man MW34 berechnen, usw.

Man sieht also, dass es reicht, wenn man nur die Diagonale abspeichert und dann die nächste Iteration von F berechnet. Dies reicht vollkommen aus, um bei dem Verfahren schnell zu einer Lösung zu gelangen.
Der Platzbedarf ist also nur bestimmt durch die Diagonale. Der maximale Platzbedarf wird also durch die Zeile bestimmt, in der das Abbruchkriterium erfüllt wird. Bis dahin steigt er linear um ein Feld pro Schritt an. Temporär wird das alte Array zum Neuberechnen des nächsten Arrays benötigt, also beträgt der Platzbedarf im Schritt n:

$$P_n=2*P_{n-1}+1$$

wobei $P_0=1$ ist.

Die Laufzeit ist auch wesentlich kürzer, da nicht erst bis zur unteren Grenze eines viel zu großen Arrays gerechnet wird, sondern nur das, was wirklich benötigt wird. Die Laufzeit beträgt dann in etwa O(n), wobei n die Anzahl der Iterationen ist.

An diesem Beispiel sieht man, dass der Einsatz von Arrays nicht sofort eine Laufzeitverbesserung oder eine Arbeitserleichterung bringt. Ohne das Problem genau durchdacht zu haben, wird man nicht die optimale Lösung finden.

6. OOP

6.1. Einleitung

Objektorientierte Programmierung ist eine „Programmier-Philosphie“ oder auch ein Programmier-Paradigma. Ziel dieser Herangehensweise ist, Daten und Code zu vereinen und damit logische Blöcke zu bilden. Es werden i.d.R. Prozesse oder Objekte der Umwelt abgebildet, zum Beispiel ein Objekt Auto, das bestimmte Eigenschaften (also Daten) hat und bestimmte Aktionen ausführen kann (code).

Java ist zum Beispiel eine konsequent objektorientierte Sprache, in der es außer den primitiven Datentypen boolean, integer, long, float und double nur Objekte gibt, wobei es von den Primitiven auch noch OOP Versionen gibt.
KDE (ein Fenstermanager unter Linux) basiert auf QT, einer objektorientierten Sammlung von Fensterelementen (Knöpfe, Listboxen, Fenster,...). Daher ist der innere Aufbau von KDE gemessen an der Komplexität der Software sehr strukturiert und übersichtlich. Im Gegensatz zu Gnome, der auf C aufsetzt und daher kein OOP unterstützt.

6.2. Vererbung

Vererbung beschreibt die Möglichkeit, von einem Objekt ein anderes abzuleiten. Dieses „erbt“ alle Fähigkeiten und Eigenschaften des Vorgängers und man hat die Möglichkeit, dieses dann entsprechend zu erweitern. (Dies werde ich gleich in einem Beispiel verdeutlichen)

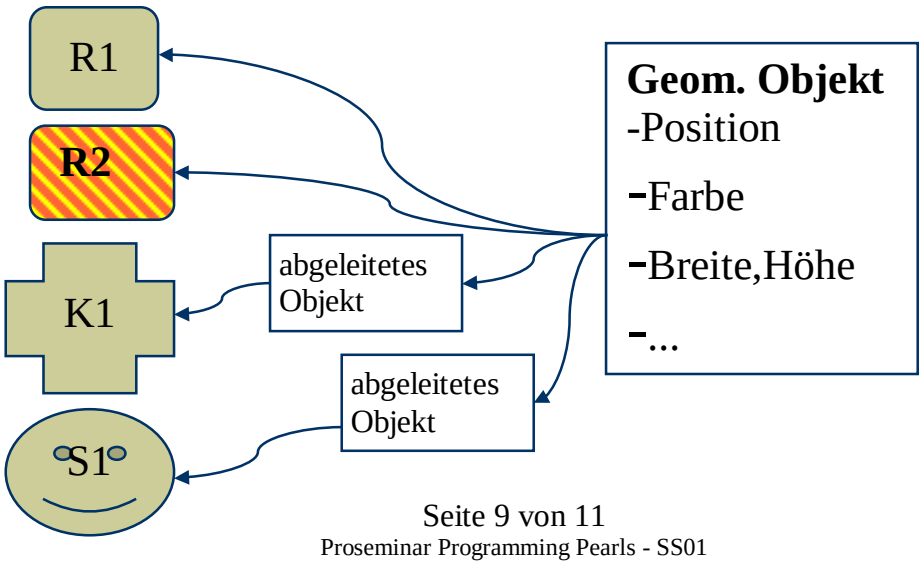
Mittels Vererbung kann man auch elegant auf eine große Zahl von unterschiedlichen Objekten zugreifen, sie müssen nur von einem bestimmten Typ abgeleitet worden sein, dann kann man in einer Liste dieses Typs alle diese Objekte abspeichern.

6.3. Beispiel

Geom. Objekt
- Position
- Farbe
- Breite,Höhe
- ...
- setFarbe
- getFarbe
- ...

Ein beliebtes Beispiel für OOP sind geometrische Objekte. Diese haben bestimmte Attribute (Eigenschaften) und einige Methoden (Fähigkeiten), mit denen man die Attribute verändert.
Dieses Objekt hat für geometrische Objekte typische Attribute und Methoden. Wenn man davon einige Versionen ableitet, können diese z.B. eine andere Form haben, oder die Möglichkeit sich zu bewegen und vieles mehr. Je nachdem, was man gerade benötigt.

Das sieht dann in etwa so aus:



R1 und R2 sind Objekte desselben Typs. In ihren Attributen (Füllfarbe, Position,...) stehen verschiedene Werte, sonst sähen sie gleich aus.
K1 und S1 haben ein anderes Aussehen und stammen trotzdem wie R1 und R2 vom selben Objekt ab. Sie wurden lediglich um wenige Attribute erweitert; die Grund-Attribute haben sie von Geom. Objekt übernommen.

Ein weiterer Vorteil ist, dass man diese vier Objekte in einer Liste von Typ Geom. Objekt abspeichern kann und dort dann direkt die allen Objekten gemeinsame Funktionalität von Geom. Objekt nutzen kann.

6.4. Fazit

Dies ist nur ein ganz kleiner Einblick in die Welt der „Objektorientierten Programmierung“, eine weitergehende Einführung würde diesen Rahmen sprengen. Ich kann deshalb da nur auf weiterführende Literatur verweisen. Dennoch sollte jeder Programmierer sich mit OOP vertraut machen, weil sie viele Probleme des alltäglichen Programmiererlebens (und das der Projektleiter) erheblich vereinfachen, da man alles sehr genau strukturieren kann.

7. Tools

Es gibt viele Tools, die einem Programmierer viel Arbeit abnehmen. Einige möchte ich kurz vorstellen.

7.1. IDEs

IDEs wie Delphi, CBuilder, JBuilder, VisualC, Forte, um nur einige zu nennen, helfen dem Programmierer effektiv zu arbeiten und seinen Code sauber zu strukturieren und zu analysieren. Integrierte Debugger sind bei der Fehlersuche hilfreich, usw.

7.2. Spezialisierte Tools benutzen

- **HTML für Dokumentationen oder Menüstrukturen von CDs benutzen**
  Es gibt kaum Gründe, warum man nicht so verfahren sollte, da es mittlerweile auf fast jedem System HTML-Viewer gibt und man sich so arbeitsintensive und fehlerträchtige Arbeit spart.
- **Tabellenkalkulation nutzen, wenn sinnvoll**
  Wenn das eigene Programm Zahlenkolonnen produziert, die der eine oder andere Benutzer gerne in Charts oder andere Diagramme integrieren möchte, schreibt man einen Exportfilter für gängige Tabellenkalkulationen (KSpread, Excel,...) und der Benutzer kann dies selber machen. Es ist auch nicht immer einfach, einen Formelparser selbst zu schreiben. Tabellenkalkulationen erfüllen diesen Zweck aber sehr gut, also steuert man diese Anwendung einfach fern (DCOP, CORBA, DCOM,...) und hat sich so dieser Verantwortung entzogen.

- **Datenbanken**
  Datenbanken sind aus dem heutigen Informatikerleben nicht mehr wegzudenken. Es gibt große Datenbanksysteme, die auf noch größeren Servern liegen und es gibt die kleine Desktop-Datenbank, mit der man über seine DVD Sammlung die Übersicht behält. Soll man so etwas programmieren, sollte man nicht das Rad neu erfinden und die Datenbank-Funktionalität selber nachbilden. Dafür gibt es sehr viele Tools und freie Datenbanken, die einem sehr viel Arbeit abnehmen und von denen man sicher sein kann, dass sie funktionieren und schon im Alltag getestet sind.
- **...**

Es gibt noch sehr viele Tools, die alle einen Blick wert sind. UML-Tools sind in diesem Kontext natürlich auch zu nennen, aber es ist nicht möglich, eine vollständige Liste anzugeben. Wird man vor eine Aufgabe gestellt sollte man sich nach Programmen oder Techniken umsehen, die einem die gestellte Aufgabe vereinfachen.

8. Fazit

Abschließend fasse ich noch einmal die Leitsätze zusammen.

- **Programme (bzw. Funktionen) klein und übersichtlich halten**
  Mammutfunktionen sind schwer zu warten und tragen nicht zum Verständnis des Programms bei. Besser sind kleine Funktionen, die Teilaufgaben lösen.
- **Manche Probleme lassen sich besser erst allgemein lösen und dann auf den Spezialfall anwenden**
  Bevor man sechs oder sieben Funktionen schreibt, die dasselbe machen, nur mit verschiedenen Parametern, sollte man besser den allgemeinen Fall lösen und dann die Spezialfälle durch Aufrufe mit den richtigen Parametern lösen.
- **Kapseln von komplexen Strukturen (OOP)**
  Wie im Bereich OOP schon angesprochen, kann man damit seine Programme sauber planen und übersichtlich halten, auch wenn noch Erweiterungen anstehen. Außerdem ist CodeReUse ein häufig geforderter Ansatz, der durch OOP gefördert wird.
- **Durch geänderte Randbedingungen (oder Hauptaufgaben) ist ein ReDesign im Normalfall immer angebracht**
  Es hilft nicht, wenn man eine funktionierende Lösung so oft an geänderte Bedingungen anpasst, dass diese dann hinterher nicht mehr zu durchschauen ist und (evtl.) sogar Fehler enthält, die man nicht sieht, weil man nur noch die notwendigen Stellen ändert. Man sollte dann das Gesamtkonzept überprüfen, ob der gewählte Ansatz noch gut für das Problem ist.
- **Das Programm nach den zu verarbeitenden Daten planen und nicht umgekehrt**
  Eine verständliche Forderung, da man i.d.R. vor die Aufgabe gestellt wird, Daten zu verarbeiten und nicht besonders schöne Algorithmen für nicht vorhandene Probleme zu schreiben.

9. Weiterführende Literatur

- **Code Complete**
  Steve McConnell
  Microsoft Press 1993
- **Rapid Development**
  S. McConnell
  Microsoft Press 1996
- **Software Projekt Survival Guide**
  S. McConnell
  Microsoft Press 1998

Persönliche Empfehlungen

Writing Solid Code
Steve Maguire
Microsoft Press 1993