Musterlösungen zu Kapitel 17

aus H.Mössenböck: Sprechen Sie Java?

Aufgabe 1: Würfelspiel

Ebene 1

Auf oberster Ebene ist das Programm eine Folge von Wurfserien. Wir definieren also eine Methode

```
points = diceSeries(n);
```

die eine Serie mit n Würfen durchführt und die erzielten Punkte als Ergebnis zurückliefert. Damit können wir bereits die Steuerlogik des Programms hinschreiben:

```
int n = In.readInt();
int totalPoints = 0;
for (int i = 1; i < 1000; i++)
    totalPoints += diceSeries(n);
Out.println("Durchschnittliche Punktezahl bei " + n + "-Serien ist " + totalPoints / 1000);
```

Ebene 2

Nun verfeinern wir diceSeries. Jede Serie besteht aus n Würfen. Das Ergebnis eines Wurfes soll uns die Methode

```
points = throwDice();
```

liefern. Mit ihr können wir diceSeries folgendermaßen implementieren (wenn eine 1 gewürfelt wird, ist laut Spezifikation das Ergebnis der gesamten Wurfserie 0):

```
static int diceSeries(int n) {
    int points = 0;
    for (int i = 1; i <= n; i++) {
        int res = throwDice();
        if (res == 1) return 0;
        points += res;
    }
    return points;
}
```

Ebene 3

Die Implementierung von throwDice ist einfach: Wir brauchen nur den Java-Zufallszahlengenerator Math.random() aufzurufen, der eine double-Zufallszahl im Intervall [0..1[liefert. Wenn wir diesen Wert mit 6 multiplizieren, erhalten wir eine Zufallszahl im Intervall [0..6[. Der ganzzahlige Teil davon liegt im Bereich 0..5. Addieren wir 1, so erhalten wir eine Zahl im Bereich 1..6:

```
static int throwDice() {
    return 1 + (int)(Math.random() * 6);
}
```

Aufgabe 2: Textzentrierung

Ebene 1

Wir müssen die Datei Wort für Wort lesen und können dazu die existierende Lese-Methode

```
word = In.readWord();
```

verwenden. Um die Wörter auszugeben, verwenden wir einen Puffer, in den wir einfach die Wörter in der Reihenfolge ihres Auftretens schreiben. Wenn der Puffer voll ist, soll er sich selbst entleeren, indem er seinen Inhalt in einer Zeile mit n Zeichen zentriert ausgibt. Zum Schluß muß ein eventuell noch halb gefüllter Puffer ausgegeben werden. Schließlich brauchen wir noch eine Operation zum Anlegen eines Puffers der Länge n. Dies führt zu folgenden Operationen:

Puffer anlegen:	`buf = new Buffer(n);`
Wort in Puffer einfügen:	`buf.append(word);`
Restpuffer ausgeben:	`buf.flush();`

Somit können wir die 1. Ebene des Programms wie folgt implementieren:

```
Buffer buf = new Buffer(n);
String word = In.readWord();
while (In.done()) {
    buf.append(word);
    word = In.readWord();
}
buf.flush();
```

Ebene 2

Auf dieser Ebene muß die Pufferklasse implemetiert werden. Als Datenstruktur können wir einen Java-StringBuffer verwenden.

```
class Buffer {
    StringBuffer b;
    int lineLength;

    Buffer(int len) {
        b = new StringBuffer(len);
        lineLength = len;
    }
    void append(String word); {...}
    void flush() {...}
    …
}
```

Um ein Wort mit append anzufügen, müssen wir seine Länge sowie die Länge des Puffers bestimmen, das Wort anhängen und gegebenenfalls den Puffer zentriert ausgeben. Die dafür nötigen Operationen (die zum Teil bereits existieren) sind:

Wortlänge bestimmen:	`len = word.length();`
Pufferlänge bestimmen:	`len = b.length();`
Wort anfügen:	`b.append(word);  // eventuell mit " " davor`
Puffer zentriert ausgeben:	`flush();`

Die Implementierung von append sieht mit diesen Operationen so aus:

```
void append(String word) {
    if (b.length() == 0) // still empty
        if (word.length() <= lineLength)
            b.append(word);
        else { // word longer than line
            b.append(word.substring(0, lineLength));
            flush();
            append(word.substring(lineLength));
        }
    else // buffer already contains words
        if (b.length() + 1 + word.length() <= lineLength)
            b.append(" " + word);
        else {
            flush();
            append(word);
        }
}
```

Um den Puffer zentriert auszugeben, muß man die Differenz d zwischen Pufferlänge und Zeilenlänge berechnen, dann d/2 Leerzeichen und schließlich den Pufferinhalt ausgeben. Ist der Puffer leer, wird nichts ausgegeben.

```
void flush() {
    if (b.length() > 0) {
        int d = lineLength - b.length();
        for (int i = 1; i <= d/2; i++) Out.print(" ");
        Out.println(b.toString());
        b.delete(0, b.length());
    }
}
```

Aufgabe 3: Tic-Tac-Toe

Ebene 1

Um das Spiel bequem implementieren zu können, würden wir uns folgenede Operationen wünschen:

Zug einlesen	`move = readMove();`
Zug spielen	`play(move, player);`
Prüfen, ob ein Zug gültig ist	`isValid(move)`
Prüfen, ob ein Spieler gewonnen hat:	`hasWon(player)`

Einen Zug stellen wir dabei als Klasse

```
class Move { int x, y; }
```

dar und das Spielfeld als 3 x 3-Matrix

```
static int[][] board = {{0,0,0}, {0,0,0}, {0,0,0}};
```

wobei die Felder die Werte 1 (Spieler 1 hat hier gesetzt), 2 (Spieler 2 hat hier gesetzt) und 0 (noch kein Spieler hat hier gesetzt) annehmen können. Somit können wir das Hauptprogramm des Spiels bereits codieren:

```
class TicTacToe {
    public static void main(String[] arg) {
        In.open("input.txt");
        int player = 1;
        while (true) {
            Move move = readMove();
            if (move == null) { Out.println(" -- Too few moves to win the game"); break; }
            if (isValid(move)) {
                play(move, player);
                if (hasWon(player)) { Out.println("Player " + player + " has won!"); break; }
            } else {
                Out.println("-- Invalid move: " + move); break;
            }
            player = 3 - player;
            move = readMove();
        }
        In.close();
    }
}
```

Ebene 2: readMove

Wir nehmen an, daß die Spielzüge in der Form A2 C0 A0 ... vorliegen, also durch Leerzeichen getrennt sind. Abgesehen davon, daß wir beim Lesen eines Spielzugs eine Plausibilitätsprüfung vornehmen sollten, ist readMove nicht besonders kompliziert, so daß wir es sofort implementieren können:

```
static Move readMove() {
    String s = In.readWord();
    if (!In.done() || s.length() != 2) return null;
    char x = s.charAt(0);
    char y = s.charAt(1);
    if (x >= 'A' && x <= 'C' && y >= '0' && y <= '3')
        return new Move(x-'A', y-'0');
    else
        return null;
}
```

Ebene 2: isValid

Diese Methode muß nur prüfen, ob das entsprechende Feld noch frei ist:

```
static boolean isValid(Move move) {
    return board[move.x][move.y] == 0;
}
```

Ebene 2: play

Diese Methode setzt auf das entsprechende Feld die Nummer des momentanen Spielers

```
static void play(move, player) {
    board[move.x][move.y] = player
}
```

Sie kann davon ausgehen, daß bereits geprüft wurde, ob dieses Feld leer ist. Eigentlich könnte man play und isValid wegen ihrer Kürze direkt an der Aufrufstelle einsetzen. Wir lassen die beiden Methoden aber bestehen, weil das Programm dadurch lesbarer wird.

Ebene 2: hasWon(player)

Die Gewinnprüfung können wir in folgende Einzelprüfungen zerlegen

Prüfe, ob der Spieler eine Zeile gefüllt hat:	lineFilled(int line, int player)
Prüfe, ob der Spieler eine Spalte gefüllt hat:	columnFilled(int col, int player)
Prüfe, ob der Spieler eine Diagonale gefüllt hat:	diagonalFilled(player)

Damit sieht die Gewinnprüfung wie folgt aus:

```
static boolean hasWon(player) {
    for (int i = 0; i < 3; i++) {
        if (lineFilled(i, player)) return true;
        if (columnFilled(i, player)) return true;
    }
    return diagonalFilled(player)
}
```

Ebene 3

Die Prüfung, ob eine Zeile, eine Spalte oder eine Diagonale zur Gänze vom Spieler gefüllt wurde, ist einfach und kann wie folgt geschrieben werden:

```
static boolean lineFilled(int line, int player) {
    for (int i = 0; i < 3; i++) if (board[line][i] != player) return false;
    return true;
}

static boolean columnFilled(int col, int player) {
    for (int i = 0; i < 3; i++) if (board[i][col] != player) return false;
    return true;
}

static boolean diagonalFilled(player) {
    return board[0][0] == player && board[1][1] == player && board[2][2] == player
    || board[0][2] == player && board[1][1] == player && board[2][0] == player;
}
```

Auch diese drei Methoden könnte man aus Effizienzgründen direkt in hasWon einsetzen. Wir lassen sie aber aus Lesbarkeitsgründen wieder als eigenständige Methoden bestehen.

Aufgabe 4: Galton-Brett

Ebene 1

Zuerst überlegen wir uns eine geeignete Datenstruktur für die Töpfe, in die die Kugeln fallen können. Bei n Ablenkungen brauchen wir n+1 Töpfe; wird die Kugel immer nach links abgelenkt, fällt sie in den Topf 0, wird sie immer nach rechts abgelenkt, fällt sie in den Topf n, sonst in einen der dazwischen liegenden Töpfe. Die Töpfe können wir als Array modellieren:

```
static int[] bin = new int[n+1];
```

Bei jedem Wurf brauchen wir dann nur zu zählen, wie oft die Kugel nach rechts abgelenkt wird und den Inhalt des Topfes, der dieser Zählung entspricht, um 1 zu erhöhen.

Die gröbste Zerlegung könnte aus folgenden beiden Operationen bestehen

Wirf die Kugel mit n Ablenkungen:	`throwBall(int n);`
Gib den Inhalt der Töpfe als Histogramm aus:	`printBins();`

Somit ergibt sich folgendes Hauptprogramm

```
public static void main(String[] arg) {
    Out.print("Anzahl der Ablenkungen: ");
    int levels = In.readInt();
    bin = new int[n+1];
    for (int i = 0; i <= n; i++) bin[i] = 0;
    for (int i = 0; i < 100; i++) throwBall(levels);
    printBins();
}
```

Ebene 2: throw

Wir müssen n mal die Ablenkung der Kugel ermitteln und sie dann in den richtigen Topf fallen lassen. Jede Ablenkung bestimmen wir durch die Operation

```
(int)(Math.random()*2)
```

die mit gleicher Wahrscheinlichkeit 0 oder 1 liefert. Somit sieht die Methode throwBall wie folgt aus:

```
static void throwBall(int n) {
    int x = 0;
    for (int i = 0; i < n; i++)
        x += (int)(Math.random() * 2);
    bin[x]++;
}
```

Ebene 2: printBins

Für jeden Topf müssen wir seinen Inhalt als Balken aus Sternen visualisieren. Dazu verwenden wir die Operation

```
printBar(len);
```

Die Methode printBins lautet dann

```
static void printBins() {
    for (int i = 0; i < bin.length; i++) {
        Out.print(i + ": "); printBar(bin[i]); Out.println();
    }
}
```

Ebene 3: printBar

Einen Balken mit i Sternchen zu drucken ist einfach:

```
static void printBar(int len) {
    for (int i = 0; i < len; i++) Out.print("*");
}
```

Diese Methode setzen wir am besten an der Aufrufstelle ein.

Aufgabe 5: Stichwortverzeichnis

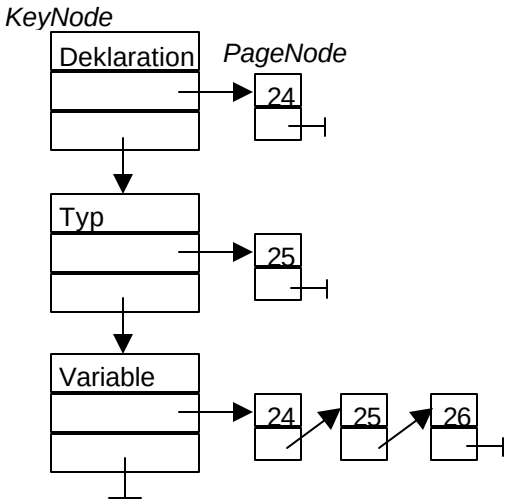
Ebene 1

Auf äußerster Ebene besteht das Programm aus dem wiederholten Lesen und Verarbeiten von Einträgen aus Seitennummer und Stichworten. Anschließend muß das Stichwortverzeichnis ausgedruckt werden. Das erfordert folgende beiden Operationen

Lies und Verarbeite Eintrag

Gib Stichwortverzeichnis aus

Bevor wir uns diese beiden Operationen näher vornehmen, überlegen wir uns, in welcher Form wir die Daten speichern könnten. Die Stichwörter speichern wir am besten in einer sortierten verketteten Liste. Für jedes Stichwort werden die Seiten, auf denen es vorkommt, wieder in einer sortierten verketteten Liste gespeichert. Die Datenstruktur könnte also z.B. so aussehen:



Die beiden Listen halten wir in einer Klasse Index, die folgende einfache Schnittstelle hat:

```
class Index {
    void enter(String keyword, int page)
    void print();
}
```

Die beiden oben genannten Operationen können wir nun unter Verwendung des Index wie folgt spezifizieren:

Lies und Verarbeite Eintrag: `ready = processEntry(index);`

Gib Stichwortverzeichnis aus: `index.print();`

Somit ergibt sich folgendes Hauptprogramm:

```
public static void main(String[] arg) {
    Index index = new Index();
    boolean ready;
    In.open("input.txt");
    do { ready = processEntry(index); } while (! ready);
    index.print();
    In.close();
}
```

Ebene 2: processEntry

Um einen Eintrag zu verarbeiten, müssen wir zuerst eine Seitennummer und anschließend ihre Stichwörter lesen. Jedes Stichwort/Seitennummern-Paar tragen wir mit enter in den Index ein, also:

```
static boolean processEntry(Index index) {
    int page = In.readInt();
    if (!In.done()) return true; // ready
    String keyword = In.readWord();
    while (In.done()) {
        index.enter(keyword, page);
        keyword = In.readWord();
    }
    return false; // maybe more entries
}
```

Ebene 2: Methode enter

Um ein Stichwort einzutragen, müssen wir es zuerst in der Liste suchen und es dort einfügen, falls es noch nicht vorhanden ist. Wenn wir den Stichwort-Knoten haben, müssen wir in seiner Seitennummern-Liste die entsprechende Seitennummer einsortieren. Wir benötigen also folgende Operationen

Suche Knoten und füge ihn nötigenfalls ein:	`node = find(keyword);`
Sortiere Seitennummer ein:	`addPage(keyNode, page);`

Beide Operationen implementieren wir als private Methoden von Index. Die Methode enter sieht folgendermaßen aus:

```
void enter(String keyword, int page) {
    KeyNode p = find(keyword);
    addPage(p, page);
}
```

Ebene 3: find

Das Suchen und Einfügen eines Knotens in eine sortierte Liste ist eine Standardoperation. Wir implementieren sie wie folgt:

```
private KeyNode find(String keyword) {
    KeyNode p = head, last = null;
    while(p != null && keyword.compareTo(p.keyword) > 0) {
        last = p; p = p.next;
    }
    // p == null || keyword is less or equal to p.keyword
    if (p == null || keyword.compareTo(p.keyword) < 0) {
        KeyNode q = new KeyNode(keyword);
        q.next = p;
        if (p == head) head = q; else last.next = q;
        p = q;
    }
    // p points to Node containing keyword
    return p;
}
```

Ebene 3: addPage

Auf ähnliche Weise sortieren wir die Seitennummer in die PageNode-Liste ein, die an einem Stichwort-Knoten hängt:

```
private void addPage(KeyNode node, int page) {
    PageNode p = node.firstPage, last = null;
    while (p != null && page > p.page) {
        last = p; p = p.next;
    }
    // p == null || page <= p.page
    if (p == null || page < p.page) {
        PageNode q = new PageNode(page);
        q.next = p;
        if (p == node.firstPage) node.firstPage = q; else last.next = q;
    }
}
```

Ebene 2: print

Um das Stichwortverzeichnis auszugeben, laufen wir durch die Stichwortliste. Zur Ausgabe der Seitennummern eines Stichworts verwenden wir eine Hilfsmethode printPages(keyNode);

```
void print() {
    for (KeyNode p = head; p != null; p = p.next) {
        Out.print(p.keyWord + "  ");
        printPages(p);
        Out.println();
    }
}
```

Ebene 3: printPages

Die Ausgabe der Seitennummern eines Stichworts ist einfach. Wir müssen nur darauf achten, daß wir die Kommas richtig setzen:

```
private void printPages(KeyNode node) {
    boolean first = true;
    for (PageNode p = node.firstPage; p != null; p = p.next) {
        if (first) first = false; else Out.print(", ");
        Out.print(p.page);
    }
}
```

Aufgabe 6: Wortliste

Ebene 1

Zum Lesen von Wörtern bietet die Klasse In zwar die Methode In.readWord() an. Wir wollen diesmal aber das Lesen der Wörter im Sinne der schrittweisen Verfeinerung selbst entwerfen und implementieren. Wir brauchen also Operationen, um Wörter zu lesen und Zwischenräume zu überspringen. Da mehrfach vorkommende Worte in der Ausgabe nur einmal aufscheinen sollen, müssen wir uns die Worte, die wir schon gesehen haben, in einer Liste merken. Diese Liste können wir anschließend ausgeben. Wir brauchen also folgende Operationen:

word = readWord();	überliest Leerzeichen und liefert das nächste Wort (oder null, wenn es keines mehr gibt)
list.enter(word);	Trägt das Wort in eine Liste ein
list.print();	Gibt die Liste der gespeicherten Wörter aus

Damit implementieren wir das Hauptprogramm wie folgt:

```
public static void main(String[] arg) {
    In.open("input.txt");
    List list = new List();
    String word = readWord();
    while (word != null) {
        list.enter(word);
        word = readWord();
    }
    list.print();
    In.close();
}
```

Ebene 2: readWord

Zum Lesen einzelner Zeichen verwenden wir die Methode In.read(). Wir müssen zuerst Trennzeichen überlesen und anschließend Buchstaben zu einem Wort zusammenfassen. Für diesen Zweck verwenden wir einen StringBuffer.

```
static String readWord() {
    char ch = In.read();
    while (In.done() && !Character.isLetter(ch)) ch = In.read();
    // !In.done() || ch is a letter
    if (!In.done()) return null;
    StringBuffer b = new StringBuffer();
    while (In.done() && Character.isLetter(ch)) {
        b.append(ch);
        ch = In.read();
    }
    return b.toString();
}
```

Ebene 2: list.enter

Der Einfachheit halber halten wir die Wörter in einer unsortierten Liste. Wir müssen zuerst prüfen, ob das Wort schon in der Liste steht und es nötigenfalls dort eintragen. Das ergibt wieder zwei Operationen, nämlich:

```
list.contains(word)

list.add(word)
```

Die Methode enter lautet:

```
void enter(String word) {
    if (!this.contains(word)) this.add(word);
}
```

Ebene 3: contains und add

Diese Methoden sind so einfach, daß wir sie gleich implementieren können:

```
boolean contains(String word) {
    for (Node p = head; p != null; p = p.next)
        if (word.equals(p.word)) return true;
    return false;
}

void add(String word) {
    Node p = new Node(word);
    p.next = head; head = p;
}
```

Die beiden Methoden sind allerdings so kurz, daß wir sie gleich an der Aufrufstelle einsetzen, was folgende Implementierung von enter ergibt:

```
void enter(String word) {
    for (Node p = head; p != null; p = p.next)
        if (word.equals(p.word)) return;
    Node p = new Node(word);
    p.next = head; head = p;
}
```

Ebene 2: list.print

Auch diese Aufgabe ist einfach. Wir müssen nur die Liste durchgehen und alle Wörter ausgeben:

```
void print() {
    for (Node p = head; p != null; p = p.next)
        Out.println(p.word);
}
```

Aufgabe 6: Auswertung von Lottoscheinen

Ebene 1

Das Programm muß Sechsergruppen von Zahlen lesen, also braucht man eine Operation

```
int[] group = readGroup();
```

die so eine Gruppe liest und in einem Zahlenarray zurückgibt. Die erste Sechsergruppe stellt die Ergebnisse der Lottoziehung dar, mit denen die anderen Tipps verglichen werden müssen. Wir brauchen also eine Operation

```
int n = hits(tip, result);
```

die die Anzahl der Treffer in der Sechsergruppe tip ermittelt, indem sie sie mit dem Ergebnis der Ziehung result vergleicht. Im Prinzip könnte man result ebenfalls als Sechsergruppe von Zahlen speichern. Effizienter ist allerdings, das Ergebnis in ein Boolesches Array umzuwandeln:

```
boolean[] valid = new boolean[50];
```

in dem valid[i] == true ist, wenn die Zahl i gezogen wurde und sonst false. Um eine Sechsergruppe in das valid-Array umzuwandeln brauchen wir eine Operation

```
valid = convert(group);
```

Mit diesen Operationen können wir die Methode check bereits wie folgt implementieren:

```
static void check() {
    In.open("input.txt");
    int[] results = readGroup();
    boolean[] valid = convert(results);
    int[] tip = readGroup();
    while (tip != null) {
        Out.println(hits(tip, valid) + " Treffer");
        tip = readGroup();
    }
    In.close();
}
```

Ebene 2: readGroup

Das Lesen von 6 Zahlen ist so einfach, daß wir es gleich implementieren:

```
int[] readGroup() {
    int[] a = new int[6];
    for (int i = 0; i < 6; i++) a[i] = In.readInt();
    if (In.done()) return a; else return null;
}
```

Ebene 2: convert

Auch diese Methode ist so einfach, daß sie sofort implementiert werden kann:

```
boolean[] convert(int[] group) {
    boolean[] b = new boolean[50];
    for (int i = 0; i < b.length; i++) b[i] = false;
    for (int i = 0; i < group.length; i++) {
        int val = group[i];
        if (val < 0 || val >= 50) { Out.println("-- invalid number: "+ val); return null; }
        b[val] = true;
    }
    return b;
}
```

Ebene 2: hits

Um die Anzahl der Treffer zu zählen, braucht man nur nachzusehen, wieviele Zahlen von tip in valid vorkommen:

```
int hits(int[] tip, boolean[] valid) {
    int res = 0;
    for (int i = 0; i < tip.length; i++)
        if (valid[hit[i]]) res++;
    return res;
}
```