Christian-Albrechts-Universität zu Kiel
Sommersemester 2000
Fortgeschrittenenpraktikum (Java)

Übung 4:

Ausgabetermin: **22. Mai 2000**

Abgabe: **29. Mai 2000**

Aufgabe 1: [GUIs, Netzwerk]

In dieser Aufgabe soll ein Programm zur Abgabe der Lösungen erstellt werden. Das Programm soll eine graphische Oberfläche (GUI) verwenden und folgende Merkmale haben:

- Das Programm soll den User fragen, ob das aktuelle Verzeichnis tatsächlich das gewünschte (die Lösung enthaltende) ist.
- Optional (nicht Pflicht): Das Verzeichnis aufräumen (frühere Aufgabe).
- Erfragung der Gruppennummer (default: die eigene Gruppe) und Seriennummer, daraus automatische Generierung eines Subjects (gemäß Handout).
- Zusammenpacken der Dateien des Verzeichnisses. Dieses Archiv wird mit der erzeugen Subject-Zeile per Mail an `unix01` versandt (ACHTUNG: nicht die Test-Emails!!!).
- Es soll außerdem eine einfache Exception-Behandlung für mögliche auftretende Exceptions programmiert werden.

Für die Versendung von Email aus Java-Programmen heraus empfiehlt es sich, das Paket `java.net` anzusehen (z.B. `URL u = new URL("mailto:" + addr)`).

Aufgabe 2: [Ausdrücke]

Bemerkung: Diese Aufgabe sieht länger aus als sie ist. Das meiste ist Hinführung, einiges werde ich auf dem Netz zur Verfügung stellen, und überdies ist vieles aus den vorangegangenen Aufgaben bekannt.

Nehmen Sie an, Sie wollten oder müßten einen Compiler bauen. Eine syntaktische Kategorie, die in sogut wie jeder Sprache vorkommt, ist die der *Ausdrücke* (expressions). In der Aufgabe wollen wir uns auf boolesche Ausdrücke[1] beschränken und betrachten daher Ausdrücke nach folgender BNF-Definition:

$$expr_B ::= Const \quad | \quad expr_B \vee expr_B \quad | \quad expr_B \wedge expr_B \tag{1}$$

Konstanten (in diesem Fall boolesche Konstanten für Wahrheit und Falschheit) sind Ausdrücke, und man kann durch die angegebenen Operationen neue Ausdrücke aus booleschen Ausdrücken zusammensetzen. Mit anderen Worten, Ausdrücke sind *induktiv* definiert. Die BNF-Notation ist eine präzise Schreibweise zu ihrer Spezifikation.

Programmieren Sie eine Klasse `B_Expr_D` für boolesche Ausdrücke, sodaß man[2] Ausdrücke oder vielmehr Syntaxbäume für Ausdrücke wie folgt konstruieren kann:

```
B_Expr_D b_expr = new Oder(new Const(true),
                           new Und(new Const(false),
                                   new Const(true)));
```

Die Aufgabe bietet außer der Tatsache, daß wir es nun mit *verzweigten* statt linearen Strukturen zu tun haben, im Vergleich zu den Pizza- und den Listenaufgaben der vergangenen Zettel nichts Neues, d.h. wir können die Klassen `Const`, `Und` und `Oder` wieder als Unterklassen der abstrakten Klasse `B_Expr_D` für Ausdrücke darstellen.[3]

Nun soll wieder auf die so konstruierten Daten zugegriffen werden. Die Aktionen, die wir auf den Ausdrücken ausführen wollen, seien

1. Auswerten des booleschen Ausdruckes
2. Bestimmung der Tiefe des Baumes[4]

[1] Java besitzt selbstverständlich auch Ausdrücke, auf der lexikalischen Ebene sind die Operatorsymbole `&&` und `||` ... in konkreter Syntax für boolesche Operatoren reserviert.

[2] In der Regel macht man es nicht selbst, sondern der Parser konstruiert einem derartige Datenstrukturen, sogenannte Parsebäume.

[3] Um nochmals auf die alte Pizza-Geschichte und die in den damals abgegebenen Lösungen vorgeschlagenen Alternativrepräsentierungen zurückzukommen: Auch in dieser Aufgabe böte es sich auf den ersten Blick an, die booleschen Ausdrücke nicht als verzeigerten Baum aufzubauen, sondern beim Konstruieren des Ausdruckes diesen gleich auszuwerten und sich als Ergebnis nur den sich ergebenden booleschen Wert in der Instanz zu merken. In realistischen Anwendungen, also echten Parsern, deren Aufgabe es ist, Instanzen von Klassen wie `Expr_D` zu konstruieren, ist dies nicht möglich. Alleine die Anwesenheit von booleschen Variablen in derartigen Ausdrücken läßt den Ansatz scheitern, weiter auch Funktionen in Ausdrücken, die einen booleschen Wert zurückliefern. All dies kann nicht bereits beim *Aufbau* des Baumes berechnet werden.

[4] In dem Code, den ich zur Verfügung stellen werde, ist noch eine drittes Interface zum Ausdrucken mit `String` als Rückgabetyp definiert.

Wir kennen mittlerweile eine Methode, wie man das machen kann: mittels *Visitoren*. Nun, analog der entsprechenden Aufgabe über Listen wäre die Aufgabe mittels der beiden Visitoren-Interfaces

```
interface Bool_B_ExpressionVisitor {
  boolean  forConst (boolean b) ;
  boolean  forOder(B_Expr_D e1, B_Expr_D e2);
  boolean  forUnd(B_Expr_D e1, B_Expr_D e2);
  boolean  forXOder(B_Expr_D e1, B_Expr_D e2);
}

interface Int_B_ExpressionVisitor {
  int      forConst (boolean b);
  int      forOder(B_Expr_D e1, B_Expr_D e2);
  int      forUnd(B_Expr_D e1, B_Expr_D e2);
  int      forXOder(B_Expr_D e1, B_Expr_D e2);
}
```

zu lösen. Das Bedenkliche daran ist, daß wir *zwei* Visitoren-Interfaces definieren müssen, die sehr ähnlich sind und die wir mit Namenskonvention unterscheiden. Offensichtlich brauchen wir für jeden möglichen Ergebnistyp ein eigenes Interface, und da es im Prinzip unendlich viele mögliche Ergebnistypen gibt, ist die Situation unbefriedigend, um so mehr wenn wir uns daran erinnern, wofür die ganze Visitorenschinderei überhaupt gedacht war: wir wollten — dies war das Ziel, welches wir mit Übungszettel 4 erreichen zu haben glaubten — die Definition der Pizzas/Listen/Expressions *unabhängig* von den Operationen machen, die darauf arbeiten. Insbesondere sollten nachträglich neue Datenmanipulationen programmiert werden können, *ohne* die Datenklassen überhaupt anzufassen. Wie wir sehen, ist dies immer noch nicht gelungen, denn wir können natürlich nicht, nur um auf alle (unendlich vielen) eventuellen Rückgabetypen vorbereitet zu sein, zur Vorbeugung unendlich viele Visitoren in die Daten-Klasse mit aufnehmen:

```
abstract class B_Expr_D {
  abstract int     accept(Int_B_ExpressionVisitor ask);
  abstract boolean accept(Bool_B_ExpressionVisitor ask);
  abstract String  accept(String_B_ExpressionVisitor ask);
  abstract Pizza_D accept(PizzaD_B_ExpressionVisitor ask);
  // und unendlich lange so weiter
};
```

Die **Aufgabe** besteht nun darin, dieses Problem zu lösen, indem man ein einziges *allgemeines* Visitoren-Interface für boolesche Ausdrücke `B_ExpressionVisitor` programmiert, welches alle speziellen Interfaces umfaßt. Als Hinweis: Denken Sie dabei an die im Vorlesungsteil vorgestellten Begriffe von Type Casts und an Wrapperklassen. Damit Sie nicht alles neu abtippen brauchen, werde ich, sobald die Aufgaben des vorangegangenen Zettels eingegangen sein werden, den gesamten Code für die oben beschriebene nicht-allgemeine Lösung mittels unterschiedlicher Interfaces auf der Netzseite unseres Kurses zur Verfügung stellen, die Sie geeignet zusammenfassen sollen.

Exkurs für Interessierte: Diese Aufgabe ist neben der Tatsache, daß Typcasts geübt werden sollen, aus einem weiteren — und damit zusammenhängenden — Grund interessant: Die Aufgabe zeigt zum ersten Mal eine unerwünschte Grenze der Ausdrucksfähigkeit von Java. Natürlich nicht im Sinne von Berechenbarkeit, denn Java ist berechbarkeitsvollständig, sondern im Sinne, daß gewisse natürliche Abstraktionen oder Vereinheitlichungen nicht oder nur etwas unschön hinschreibbar sind. Wenn wir auf das Ziel der Aufgabe zurückschauen, so bestand es darin, gewissermaßen *unendlich viele* Visitoren, einen für boolesche Werte, einen für Integer, und so weiter, zu programmieren, um auf alle unendlich vielen eventuellen Rückgabetypen vorbereitet zu sein.

Was in Java nicht geht, ist ein Interface wie

```
interface B_Expression_Visitor(X) {
  X  forBottom () ;
  X  forOder(B_Expr_D e1, B_Expr_D e2);
  X  forUnd(B_Expr_D e1, B_Expr_D e2);
  X  forXOder(B_Expr_D e1, B_Expr_D e2);
}
```

hinzuschreiben, wobei `X` für einen beliebigen Typ/Klasse steht, `X` spielt also hier die Rolle einer Typvariable. Stünde einem dies zur Verfügung, so könnte man umstandslos alle erforderlichen Interfaces bekommen, indem man ihnen den Typ als *Argument* übergibt:

- `B_Expression_Visitor(bool)`,
- `B_Expression_Visitor(int)`,
- `B_Expression_Visitor(Pizza_D)`,

etc. Sprachen, die dies erlauben, nennt man *parametrisch polymorph.* Es ist nicht klar, warum die Designer von Java parametrische Polymorphie nicht in die Sprache mit aufgenommen haben, insbesondere wo es seit fast 20 Jahren Sprachen gibt, die dies unterstützen. In der Tat wird das Fehlen parametrischer Polymorphie von vielen als *der* Mangel im Kern der Sprachdefinition von Java betrachtet und man diskutiert die Erweiterung von Java um parametrische Polymorphie (*"... this is being seriously considered for future versions."*, Java-FAQ's, 1998). Die Erweiterung würde eine fundamentale Änderung der Sprache bedeuten, und es steht zu befürchten, daß, bis man sich zu diesem Schritt entschließt, Java bereits in jedem Videorekorder und jeder Smart-Card eingebaut ist, so zumindest Suns mittelfristige Ziele, sodaß der Umstieg ökonomisch nicht machbar sein wird. Damit wäre die Chance, daß ein innovatives Sprachdesign[5] weite Verbreitung findet, vertan.

Was einem Java anstelle dessen anbietet, das genau sollte aus der Lösung der Aufgabe deutlich werden und beinhaltet die Verwendung von Type Casts. Im wesentlichen sind die Casts eine Erblast aus C/C++-Zeiten seligen Angedenkens. Nicht nur sind sie wenig elegant, sie sind auch *ineffizient*, denn das Laufzeitsystem muß während der Ausführung des Programmes überprüfen, ob denn der Programmierer mit seinen Typ-Casts-Hilfestellungen auch richtig gelegen hat, und diese Überprüfung kostet Zeit und kann, wenn der Überprüfung negativ ausfällt, zu einer Ausnahmesituation führen.

Aufgabe 3: [Vererbung & Überschreiben]

Nachdem das Auswerten der booleschen Ausdrücke der vorangegangenen Aufgabe so einfach war, soll das gleiche auf Mengen-Ausdrücken (mit Index S für *Set*) versucht werden. Stellen Sie sich vor, wir besäßen folgende Mengenoperationen:

$$expr_S ::= Const \quad | \quad expr_S \cup expr_S \quad | \quad expr_S \cap expr_S \tag{2}$$

Mengen-Konstanten sind natürlich was anderes als boolesche Konstanten, aber abgesehen davon ist die Grammatik der für boolesche Ausdrücke aus Gleichung 1 doch sehr ähnlich. Von dieser Ähnlichkeit wollen wir profitieren.

Die Gemeinsamkeit beider Arten von Ausdrücken ist ihre Signatur. Um die Aufgabe angehen zu können, brauchen wir einen Datentyp für Mengen, der Mengenkonstanten darzustellen erlaubt sowie die Vereinigung und den Schnitt von Mengen. Um den Aufwand in Grenzen zu halten, stelle ich eine mögliche Lösung auf der Netzseite des Kurses zur Verfügung.[6]

Wie Sie sich vielleicht bereits gedacht haben, wollen wir Mengenausdrücke und Boolesche Ausdrücke vereinheitlichen (also `Expr_D` statt `B_Expr_D` und `S_Expr_D`) und ebenso ihre Auswertung. Es soll also möglich sein, folgendes zu schreiben:[7]

```
Expr_D   b_expr  = new Und(new Const(new Boolean(true)),
                           new Const(new Boolean(false)));
Expr_D set_expr  = new Und(new Const(set2), new Const(set1));
```

[5] Na gut, paramametrische Polymorphie ist 20 Jahre alt, die meisten OO-Konzepte ebenso, aber immerhin.

[6] Es ist nur ein Vorschlag. Sie ist ein wenig anders programmiert als die bisherigen Listen-Beispiele und verwendet geschachtelte Klassen.

[7] Wie erwähnt, die Erstellung von Mengen wie `set1` oder `set2` werde ich zur Verfügung stellen.

Das Auswerten soll natürlich ebenfalls vereinheitlicht werden, das bedeutet, wir haben für die verschiedenen Arten von Expressions nur *eine* Methode `accept`, die beide Visitoren und vielleicht später noch weitere als Argument akzeptiert:

```
System.out.println("Wert = " +   b_expr.accept(new BoolEval_V()));
System.out.println("Wert = " + set_expr.accept(new  SetEval_V()));
```

Das Zusammenfassen soll, wie wir es bereits ein paarmal kennengelernt haben, durch Angabe eine gemeinsamen abstrakten *Oberklasse*

```
public abstract class Eval_D implements ExpressionVisitor
```

geschehen, wobei die konkreten Evaluatoren Unterklassen bilden[8]

$$\begin{tikzcd} & \texttt{Eval\_D} & \\ \texttt{BoolEval\_V} \arrow[ur] & & \texttt{SetEval\_V} \arrow[ul] \end{tikzcd}$$

und das allgemeine Interface die folgende Form annimmt

```
public interface ExpressionVisitor {
  Object forConst(Object o);
  Object forOder (Expr_D e1, Expr_D e2);
  Object forUnd  (Expr_D e1, Expr_D e2);
  };
```

und damit die Gemeinsamkeiten beider Signaturen aus Gleichung 1 und 2 korrekt und so abstrakt wie möglich einfängt.

Soweit die Aufgabe. Wer will, kann sich noch überlegen, was zu ändern ist, will man die booleschen Ausdrücke um $\neg expr_b$ erweitern, Mengen aber nicht um Komplementbildung.

[8] Die Oberklasse ist nicht mit einem `V` geschmückt, da sie abstrakt und selber keine Visitoreninstanzen besitzt, aber die Notation ist Geschmackssache.