RHEINISCH-
WESTFÄLISCHE
TECHNISCHE
HOCHSCHULE
AACHEN

LEHR- UND FORSCHUNGSGEBIET INFORMATIK II
RWTH Aachen · D-52056 Aachen · GERMANY
http://www-i2.informatik.rwth-aachen.de/lufgi2
Prof. Dr. Jürgen Giesl

Diplom-Vorprüfung Lösungsvorschlag
Informatik I - Programmierung
6. 3. 2002

Vorname: _____

Nachname: _____

Matrikelnummer: _____

Studiengang (bitte ankreuzen):

○ Informatik Diplom ○ Informatik Lehramt ○ Sonstige: _____

- Schreiben Sie bitte auf jedes Blatt **Vorname**, **Name** und **Matrikelnummer**.
- Geben Sie Ihre Antworten bitte in lesbarer und verständlicher Form an. Schreiben Sie bitte nicht mit roten Stiften.
- Bitte beantworten Sie die Aufgaben auf den **Aufgabenblättern**. Benutzen Sie auch die Rückseiten der **zur jeweiligen Aufgabe gehörenden** Aufgabenblätter.
- Antworten auf anderen Blättern können nur berücksichtigt werden, wenn **Name, Matrikelnummer und Aufgabennummer** deutlich darauf erkennbar ist.
- Was nicht bewertet werden soll, kennzeichnen Sie bitte durch **Durchstreichen**.
- Werden Täuschungsversuche beobachtet, so wird die Klausur mit **nicht bestanden** bewertet.
- Geben Sie bitte am Ende der Klausur **alle Blätter zusammen mit den Aufgabenblättern ab**.

	Anzahl Punkte	Erreichte Punkte
Aufgabe 1	14	
Aufgabe 2	11	
Aufgabe 3	17	
Aufgabe 4	18	
Aufgabe 5	21	
Aufgabe 6	15	
Summe	96	
Note	-	

Vorname	Name	Matr.-Nr.

Aufgabe 1 (Programmanalyse, 12 + 2 Punkte**)**

(a) Geben Sie die Ausgabe des Programms für den Aufruf `java M` an. Schreiben Sie hierzu jeweils die ausgegebenen Zeichen hinter den Kommentar "`AUSGABE:`".

```
public interface Z {
  public void f (Z object);
}

public class X implements Z {
  public int wert1 = 1,
             wert2 = 2;

  public X() {
    wert1 = wert1 + wert2;
  }
  public void f(Z object) {
    object = this;
  }
}

public class Y extends X {
  public int wert2 = 3;

  public Y() {
    wert1 = wert1 + wert2;
  }
  public void f (Z object) {}
}

public class M {
  public static void main(String[] arguments) {
    X x = new X();
    Y y = new Y();
    System.out.println(x.wert1 + ", " + x.wert2);        // AUSGABE: 3, 2
    System.out.println(y.wert1 + ", " + y.wert2);        // AUSGABE: 6, 3
    x.f(y);
    System.out.println(y.wert1 + ", " + y.wert2);}       // AUSGABE: 6, 3
}
```

(b) In der Klasse `Y` wird die Methode `f` nun durch folgenden Code ersetzt. Finden und erklären Sie den Fehler in dieser Methode, der das Compilieren verhindert.

```
protected void f (Y object) {
  Z z = new Y();
  Y y = z;
}
```

Vorname	Name	Matr.-Nr.

`Y y = z;` ist falsch, weil `z` vom Typ `Z` ist und daher explizit konvertiert werden muss (d.h., es müsste `Y y = (Y) z;` heißen).

Vorname	Name	Matr.-Nr.

Aufgabe 2 (Verifikation, 10 + 1 Punkte)

Der Algorithmus P berechnet die Hälfte einer geraden natürlichen Zahl.

(a) Vervollständigen Sie die folgende Verifikation des Algorithmus P im Hoare-Kalkül, indem Sie die unterstrichenen Teile ergänzen. Hierbei dürfen zwei Zusicherungen nur dann direkt untereinander stehen, wenn die untere aus der oberen folgt. Hinter einer Programmanweisung darf nur eine Zusicherung stehen, wenn dies aus einer Regel des Hoare-Kalküls folgt.

Algorithmus: P
Eingabe: $\mathtt{x} \in \{\, 2 * \mathtt{n} \mid \mathtt{n} \in \mathbb{N} \,\} = \{0, 2, 4, ...\}$
Ausgabe: `res`
Vorbedingung: $\mathtt{x} \geq 0 \wedge 2|\mathtt{x}$ (wobei $2|\mathtt{x}$ für die Aussage "2 teilt x" steht)
Nachbedingung: $2 * \mathtt{res} = \mathtt{x}$

$\langle\, \mathtt{x} \geq 0 \wedge 2|\mathtt{x} \,\rangle$

$\langle\, \mathtt{x} \geq 0 \wedge 2|\mathtt{x} \wedge 0 = 0 \,\rangle$

```
z = 0;
```

$\langle\, \mathtt{x} \geq 0 \wedge 2|\mathtt{x} \wedge \mathtt{z} = 0 \,\rangle$

$\langle\, \mathtt{x} \geq 0 \wedge 2|\mathtt{x} \wedge \mathtt{z} = 0 \wedge 0 = 0 \,\rangle$

```
res = 0;
```

$\langle\, \mathtt{x} \geq 0 \wedge 2|\mathtt{x} \wedge \mathtt{z} = 0 \wedge \mathtt{res} = 0 \,\rangle$

$\langle\, \mathtt{x} \geq 0 \wedge 2|\mathtt{x} \wedge 2 * \mathtt{res} = \mathtt{z} \,\rangle$

```
while (z != x) {
```

$\langle\, \mathtt{x} \geq 0 \wedge 2|\mathtt{x} \wedge \mathtt{z} \neq \mathtt{x} \wedge 2 * \mathtt{res} = \mathtt{z} \,\rangle$

$\langle\, \mathtt{x} \geq 0 \wedge 2|\mathtt{x} \wedge 2 * (\mathtt{res} + 1) = \mathtt{z} + 2 \,\rangle$

```
  res = res + 1;
```

$\langle \mathtt{x} \geq 0 \wedge 2|\mathtt{x} \wedge 2 * \mathtt{res} = \mathtt{z} + 2 \,\rangle$

```
  z = z + 2;
```

$\langle\, \mathtt{x} \geq 0 \wedge 2|\mathtt{x} \wedge 2 * \mathtt{res} = \mathtt{z} \,\rangle$

```
}
```

$\langle\, \mathtt{x} \geq 0 \wedge 2|\mathtt{x} \wedge \mathtt{z} = \mathtt{x} \wedge 2 * \mathtt{res} = \mathtt{z} \,\rangle$

$\langle 2 * \mathtt{res} = \mathtt{x} \rangle$

Vorname	Name	Matr.-Nr.

(b) Geben Sie (ohne Beweis) eine Variante für die `while`-Schleife an, die belegt, dass die Schleife für $\mathtt{x} \geq 0 \wedge 2|\mathtt{x}$ terminiert.

Die Variante ist $x - z$. Da z in jedem Schleifendurchlauf um `2` erhöht wird, verkleinert sich der Wert von $x - z$ mit jedem Schleifendurchlauf.

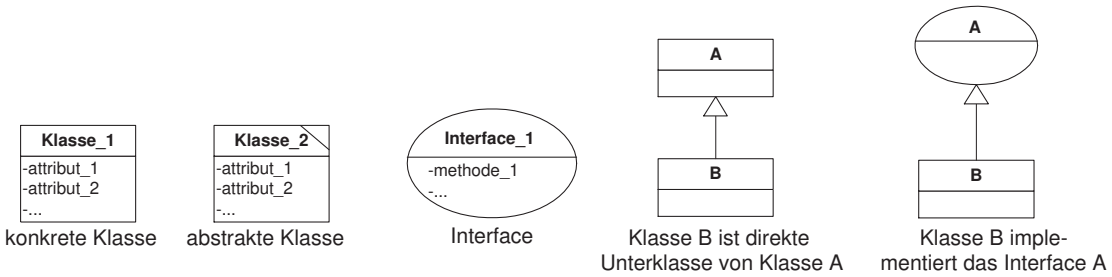
Vorname	Name	Matr.-Nr.

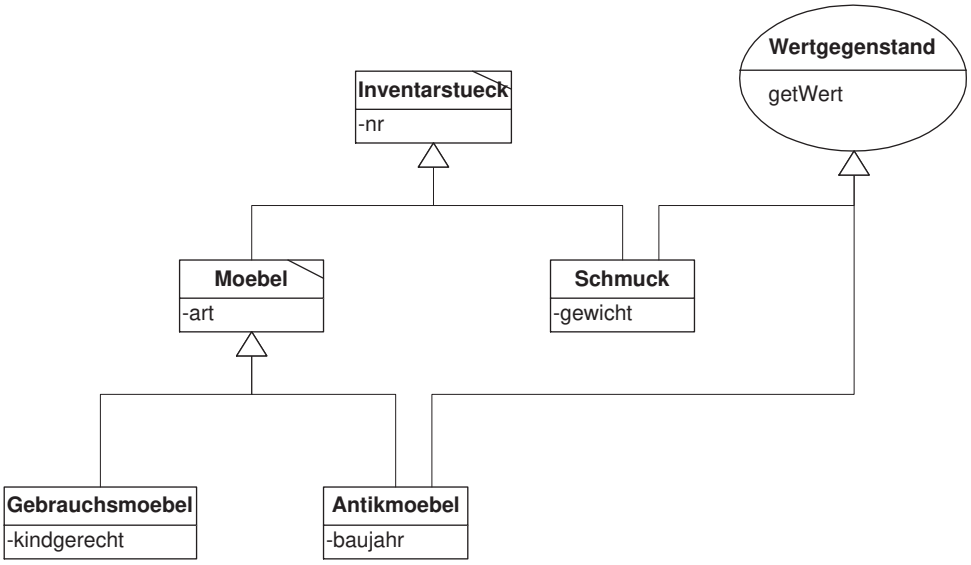
Aufgabe 3 (Datenstrukturen in Java, 6 + 5 + 6 Punkte)

Herr Müller möchte seinen Besitz inventarisieren. Dazu beschließt er, ein objektorientiertes Programm in Java zu entwickeln. Jedes seiner Besitzstücke soll eine Inventarnummer haben. Herr Müller besitzt Möbel- und Schmuckstücke. Jedes Möbelstück ist durch seine Art gekennzeichnet (z.B. “Sofa”, “Tisch”, etc.). Nur Schmuckstücke und antike Möbel sind Wertgegenstände, von denen man den Wert durch eine Methode berechnen können soll. Hierzu speichert man bei antiken Möbeln ihr Baujahr und bei Schmuckstücken ihr Gewicht (als ganze Zahl). Bei nicht antiken Möbeln ist es hingegen von Interesse, ob sie kindgerecht sind.

(a) Entwerfen Sie eine geeignete Klassenhierarchie, um den obigen Sachverhalt in Java zu realisieren. Achten Sie dabei darauf, dass gemeinsame Attribute und/oder Methoden in abstrakten Klassen bzw. Interfaces zusammengefasst werden. Verwenden Sie soweit wie möglich Vererbungstechniken.

Notieren Sie Ihren Entwurf grafisch und verwenden Sie dazu die folgende Notation: Für jede Klasse werden ihr Name und die Namen ihrer Attribute angegeben. (Methoden werden bei Klassen nicht mit aufgeführt.) Für jedes Interface werden sein Name und die Namen seiner Methoden angegeben.





Vorname	Name	Matr.-Nr.

(b) Implementieren Sie in Java die Klasse, die die Schmuckstücke realisiert, und die Klasse, die die antiken Möbel realisiert. Ihre Klassen sollen jeweils einen Konstruktor besitzen, der die Attributwerte auf die Parameter des Konstruktors setzt. Außerdem soll jede Ihrer Klassen alle Selektoren sowie eine Methode "`public void drucke()`" besitzen, die die Werte aller Attribute auf dem Bildschirm ausgibt. Es genügt, wenn Sie die Implementierungen der Konstruktoren, Selektoren und der `drucke`-Methode in einer der beiden zu implementierenden Klassen angeben. Gehen Sie davon aus, dass derartige Konstruktoren, Selektoren und `drucke`-Methoden auch in den Oberklassen vorhanden sind.

Außerdem sollen beide Klassen eine Methode besitzen, mit der man den Wert eines Objektes dieser Klasse berechnen kann. Hierzu wird die aktuelle Jahreszahl als Parameter übergeben. Der Wert eines antiken Möbelstücks ist sein hundertfaches Alter. Der Wert eines Schmuckstücks berechnet sich aus seinem Gewicht multipliziert mit der aktuellen Jahreszahl.

Verwenden Sie auch hier wieder soweit wie möglich Vererbungstechniken und Prinzipien der Datenkapselung.

```
public class Schmuck extends Inventarstueck implements Wertgegenstand {

    private int gewicht;

    public Schmuck (int nr, int gewicht) {
        super(nr);
        this.gewicht = gewicht;
    }

    public int getGewicht() {
        return gewicht;
    }

    public void setGewicht(int gewicht) {
        this.gewicht = gewicht;
    }

    public void drucke() {
        super.drucke();
        System.out.print(", " + gewicht);
    }

    public int getWert(int jahreszahl) {
        return gewicht * jahreszahl;
    }
}
```

```
public class Antikmoebel extends Moebel implements Wertgegenstand {

    private int baujahr;

    public Antikmoebel (int nr, String art, int baujahr) {
        super(nr,art);
        this.baujahr = baujahr;
    }

    public int getBaujahr() {
        return baujahr;
    }

    public void setBaujahr(int baujahr) {
        this.baujahr = baujahr;
    }

    public void drucke() {
        super.drucke();
        System.out.print(", " + baujahr);
    }

    public int getWert(int jahreszahl) {
        return 100 * (jahreszahl - baujahr);
    }
}
```

(c) Schreiben Sie eine Methode `teuersterWertgegenstand` in Java, der ein Array von Wertgegenständen und die aktuelle Jahreszahl übergeben wird. Die Methode soll daraus den Gegenstand mit dem höchsten Wert ermitteln und diesen Wertgegenstand zurückliefern.

```
public static Wertgegenstand teuersterWertgegenstand(Wertgegenstand[] w,
                                                     int jahreszahl)     {

    if (w == null || w.length == 0) return null;

    Wertgegenstand maximum = w[0];

    for (int i = 1; i < w.length; i++) {
        if (maximum == null || // kein Fehler wenn dies fehlt
            maximum.getWert(jahreszahl) < w[i].getWert(jahreszahl))
            maximum = w[i];
    }
    return maximum;
}
```

Vorname	Name	Matr.-Nr.

Aufgabe 4 (Programmierung in Java, 18 Punkte)

In dieser Aufgabe sollen Mengen objektorientiert in Java realisiert werden. Eine Menge soll dabei beliebige Objekte von Unterklassen des Interfaces `Vergleichbar` enthalten. Mengen können beliebig viele Objekte enthalten, sie enthalten jedoch kein Objekt doppelt. Hierzu sind das Interface `Vergleichbar` und die Klasse `Element` vorhanden:

```
public interface Vergleichbar {

  public boolean gleich (Vergleichbar zuvergleichen);
   /* Untersucht zwei Objekte auf Gleichheit */

  public String toString ();
   /* Überführt ein Objekt in einen String zur Ausgabe */
}

public class Element {
  private Vergleichbar wert;
  private Element next;

  public Element (Vergleichbar wert, Element next) {...}
    /* Erzeugt ein neues Element mit den Attributen wert und next */

  public Vergleichbar getWert () {...}
  public void setWert(Vergleichbar wert) {...}
  public Element getNext () {...}
  public void setNext(Element next) {...}
    /* Selektoren für die Attribute */
  }
```

Die Klasse `Menge` hat den folgenden Konstruktor und die folgenden öffentlich sichtbaren Methoden.

- `public Menge ()`
  `/* Erzeugt eine neue leere Menge. */`

- `public boolean enthalten (Vergleichbar wert)`
  `/* Untersucht, ob ein Objekt in der Menge vorkommt. */`

- `public void einfuegen (Vergleichbar wert)`
  `/* Fügt ein Objekt in die Menge ein, falls es noch nicht enthalten ist.*/`

- `public void vereinigungMit (Menge m)`
  `/* Vereinigt die aktuelle Menge mit der Menge m, ohne die Menge m zu verändern.*/`

Implementieren Sie die Klasse `Menge`. Beachten Sie dabei auch die Sichtbarkeiten der Attribute und Methoden, da die Daten gekapselt werden sollen. Die Methode `enthalten` muss dabei *rekursiv* (ohne Verwendung von Schleifen) realisiert werden. Hierbei dürfen Sie Hilfsmethoden definieren, die aber nach außen nicht sichtbar sein sollen. Achten Sie darauf, dass eine Menge keine Objekt doppelt enthält.

Vorname	Name	Matr.-Nr.


```
public class Menge {

    private Element kopf;

    public Menge () {
        kopf = null;
    }

    public boolean enthalten (Vergleichbar wert) {
        return enthalten (wert, kopf);
    }

    private static boolean enthalten (Vergleichbar wert, Element kopf) {
        if      (kopf == null)                return false;
        else if (kopf.getWert().gleich(wert)) return true;
        else                                  return enthalten(wert, kopf.getNext());
    }

    public void einfuegen (Vergleichbar wert) {
        if      (kopf == null)        kopf = new Element(wert,null);
        else if (! enthalten (wert))  kopf = new Element(wert,kopf);
    }

    public void vereinigungMit (Menge m) {
        if (m != null) {
           Element element = m.kopf;

           while (element != null) {
                 einfuegen (element.getWert());
                 element = element.getNext();
           }
        }
    }
}
```

Vorname	Name	Matr.-Nr.

Aufgabe 5 (Funktionale Programmierung in Haskell, 4 + 2 + 3 + 4 + 2 + 6 Punkte)

(a) Die Funktionen `f`, `g`, und `h` sind wie folgt definiert:

```
f x = 1 : x

g x = x ++ x

h x  = \y -> [x] : [y]
```

Geben Sie den allgemeinsten Typ von `f`, `g` und `h` an. Gehen Sie hierbei davon aus, dass `1` den Typ `Int` hat.

Der allgemeinste Typ von `f` ist `[Int] -> [Int]`, der allgemeinste Typ von `g` ist `[a] -> [a]` und der allgemeinste Typ von `h` ist `a -> [a] -> [[a]]`.

(b) Geben Sie für die folgenden Ausdrücke jeweils das Ergebnis der Auswertung an.

(i) `(\x -> \y -> x + y) ((\x -> x + 1) 1) 1`

(ii) `map (\x -> if x /= "Mond" then "Licht" else "Dunkel") ["Sonne", "Mond", "Stern"]`

(i) `3`

(ii) `["Licht", "Dunkel", "Licht"]`

(c) Schreiben Sie eine Funktion `expand :: a -> Int -> [a]`, die zu einem Objekt `x` und einer Zahl `n` eine Liste `[x,x,...,x]` der Länge `n` liefert. Beispielsweise berechnet `expand 'B' 3` das Ergebnis `['B','B','B']`.

```
expand :: a -> Int -> [a]
expand x 0     = []
expand x (n+1) = x : expand x n
```

(d) Um Listen auf kürzere Art und Weise zu repräsentieren, kann man wie folgt vorgehen:

Sei $\mathtt{L} = [\mathtt{e}_1, \mathtt{e}_2, ..., \mathtt{e}_n]$ eine Liste von Elementen vom Typ `a`. Jede nicht-leere Liste $\mathtt{K} = [\mathtt{e}_i, \mathtt{e}_{i+1}, ..., \mathtt{e}_j]$ mit $1 \leq i \leq j \leq n$ heißt dann eine *Teilliste* von `L`. Falls alle Elemente in `K` identisch sind (d.h. $\mathtt{e}_i = \mathtt{e}_{i+1} = \ldots = \mathtt{e}_j$) und die angrenzenden Elemente in `L` aber hiervon verschieden sind (d.h. $\mathtt{e}_{i-1} \neq \mathtt{e}_i$, falls $i > 2$, und $\mathtt{e}_j \neq \mathtt{e}_{j+1}$, falls $j < n - 1$), so nennt man `K` eine *geschlossene Teilliste* für das Element $\mathtt{e}_i$.

Eine Liste `L` kann nun wie folgt komprimiert werden: Jede geschlossene Teilliste der Länge `k` für das Element `x` wird durch ein Paar `(x, k)` repräsentiert.

Vorname	Name	Matr.-Nr.

Beispiel: Die Liste `L = ['B','B','B','A','A','C','C','C','C']` kann durch die komprimierte Liste `[('B', 3), ('A', 2), ('C', 4)]` repräsentiert werden.

Implementieren Sie in Haskell eine Funktion `decompress :: [(a,Int)] -> [a]`, die die oben beschriebene Komprimierung wieder rückgängig macht. Beispielsweise ergibt `decompress [('B', 3),('A', 2),('C', 4)]` das Ergebnis `['B','B','B','A','A','C','C','C','C']`.

```
decompress :: [(a,Int)] -> [a]
decompress []          = []
decompress ((x,n) : l) = expand x n ++ decompress l
```

(e) Definieren Sie in Haskell eine Datenstruktur zur Repräsentation der komprimierten Listen, welche der Beschreibung in Teil (d) entspricht. Hierbei dürfen Sie jedoch die in Haskell vordefinierten Listen *nicht* verwenden.

```
data CompList a = Nil | Cons a Int (CompList a) deriving Show
```

(f) Implementieren Sie in Haskell eine Funktion `compress`, die die oben beschriebene Komprimierung durchführt, d.h., eine beliebige Liste wird in ein Objekt der Datenstruktur aus Teilaufgabe (e) transformiert. Geben Sie auch die Typdeklaration von `compress` an. Sie dürfen hierbei davon ausgehen, dass der Vergleichsoperator `(==)` für alle Typen vordefiniert ist.
Hinweis: Implementieren Sie gegebenenfalls geeignete Hilfsfunktionen.

```
compress :: [a] -> CompList a
compress xs = build xs 0
  {- build xs n komprimiert die Liste, die aus xs entsteht, indem vorne
     noch n Kopien des ersten Elements angehaengt werden.              -}
  where build :: [a] -> Int -> (CompList a)
        build []  n = Nil
        build [x] n = Cons x (n+1) Nil
        build (x:y:xs) n | x == y    = build (y:xs) (n+1)
                         | otherwise = Cons x (n+1) (build (y:xs) 0)
```

Vorname	Name	Matr.-Nr.

Aufgabe 6 (Logische Programmierung in Prolog, 3 + 5 + 2 + 5 Punkte**)**

(a) Definieren Sie in Prolog ein dreistelliges Prädikat `del`, das ein Vorkommen eines Elements aus einer Liste löscht. Genauer bedeutet `del(L,X,K)`, dass die Liste `K` aus der Liste `L` entsteht, indem darin ein Vorkommen des Elements `X` gelöscht wird. Beispielsweise gilt `del([2,1,5,1],1,[2,5,1])`. Hierbei dürfen Sie keine in Prolog vordefinierten Prädikate verwenden.

```
del([X|L], X, L).
del([Y|L], X, [Y|K]) :- del(L, X, K).
```

(b) Definieren Sie in Prolog ein zweistelliges Prädikat `perm`, wobei `perm(L,K)` ausdrückt, dass die Liste `L` eine Permutation der Liste `K` ist.
Eine Liste ist dabei eine Permutation einer anderen Liste, wenn beide Listen dieselben Elemente (aber ggf. in unterschiedlicher Reihenfolge) enthalten. Beispielsweise hat die Liste `[1,2,2]` die Permutationen `[1,2,2]`, `[2,1,2]` und `[2,2,1]`. Es gilt also z.B. `perm([1,2,2], [2,1,2])`.

```
perm([], []).
perm(L, [X|K]) :- del(L, X, R), perm(R, K).
```

Vorname	Name	Matr.-Nr.

(c) Formulieren Sie (basierend auf dem zuvor definierten Prädikat) eine Anfrage, um alle Permutationen der Liste `[1,2,3]` zu berechnen, die mit `2` beginnen.

```
?- perm([1,2,3], L), L = [2|_].
```

(d) Betrachten Sie folgendes Prolog-Programm:

```
kante(a,b).

trans(X, Y) :- kante(X,Y).
trans(X, Y) :- trans(Z,Y), kante(X, Z).
```

Das Faktum `kante(a,b).` definiert dabei folgenden Graph:

Terminiert das Programm bei der Suche nach den ersten beiden Antworten auf die folgende Anfrage? Geben Sie ggf. die erste bzw. die ersten beiden Antworten an.

```
?- trans(a, U).
```

Die erste Antwort ist `U = b`. Bei der Suche nach der zweiten Antwort terminiert das Programm nicht.