*Aufbewahrung bis: 2025-07-28*

## Unterweisung zu Verhaltensrichtlinien

| | |
|---|---|
| **Prüfung** | inf031, Objektorientierte Modellierung und Programmierung |
| **Datum** | 2020-07-28 |
| **Uhrzeit** | 08:00 |
| **Raum** | Hörsaal 1 |

- Bitte tragen Sie Ihren Mund-Nasen-Schutz, wenn Sie sich im Raum oder im Gebäude bewegen.
- Wenn Sie eine Frage haben, setzen Sie bitte Ihren Mund-Nasen-Schutz auf und geben einen kurzen mündlichen Hinweis. Eine Aufsichtsperson wird dann den Mund-Nasen-Schutz ebenfalls aufsetzen und sich Ihnen soweit es der Mindestabstand und die Bedingungen hier im Raum zulassen annähern, damit sie Rücksprache halten können.
- Wenn Sie die Toilette benutzen müssen, setzen Sie bitte Ihren Mund-Nasen-Schutz auf und geben ein Handzeichen. Eine Aufsichtsperson wird dann ggf. Personen, die mit Ihnen in der Reihe sitzen, bitten, ebenfalls ihren Mund-Nasen-Schutz aufzusetzen und Ihnen den Weg unter Einhaltung der Mindestabstände frei zu machen. Bei der Rückkehr an den Platz wird ebenso verfahren.
- Ein endgültiges Verlassen des Prüfungsraums ist erst nach Ende der Prüfung möglich.
- Nach Ende des Prüfungszeitraums werden Sie gebeten, den Mund-Nasen-Schutz aufzusetzen. Bitte räumen Sie dann zügig Ihre Sachen zusammen und folgen den Anweisungen zum Verlassen des Raumes. Ihre Klausurbögen, das aufgefüllte Formular mit Ihren Kontaktdaten sowie diese Unterweisung lassen Sie an Ihrem Platz liegen. Diese werden anschließend eingesammelt.
- Nach Verlassen des Raumes verlassen Sie bitte auch umgehend, aber ruhig und geordnet das Gebäude. Vermeiden Sie dabei Gruppenbildung auf den Fluren und an den Ausgängen.

Ich habe die Hinweise zur Kenntnis genommen.

---

**Name in Druckbuchstaben**

2020-07-28

---

**Datum** **Unterschrift**

*Aufbewahrung bis: 2020-08-18*

## Erhebung personenbezogener Daten

| | |
|---|---|
| **Prüfung** | inf031, Objektorientierte Modellierung und Programmierung |
| **Datum** | 2020-07-28 |
| **Uhrzeit** | 08:00 |
| **Raum** | Hörsaal 1 |

Die folgenden Daten werden im Rahmen des § 2h S. 4 Corona-VO erhoben und für drei Wochen aufbewahrt.
Bitte füllen Sie das Formular vollständig aus:

| | |
|---|---|
| Name | |
| Vorname | |
| Vollständige Anschrift | |
| Telefon | |

Den Datenschutzhinweis finden Sie im Stud.IP-Eintrag dieser Veranstaltung im Dateibereich.

Carl von Ossietzky
Universität
Oldenburg

Dr. C. Schönberg
Carl von Ossietzky Universität Oldenburg, Fakultät II, Department für Informatik

# Klausur – Gruppe B

## Objektorientierte Modellierung und Programmierung (inf031)

Sommersemester 2020

- Lassen Sie die Blätter in jedem Fall zusammengeheftet.
- Geben Sie auf dieser Seite Name, Vorname und Matrikelnummer an und unterschreiben Sie die Erklärung.
- Notieren Sie die Lösungen zu den Aufgaben direkt auf dem Aufgabenzettel oder auf der vorhergehenden oder nachfolgenden Rückseite.
  Wenn Sie Lösungen an anderer Stelle notieren, markieren Sie dies deutlich sowohl dort als auch auf der Seite der Aufgabe.
- Für die Bearbeitung der Klausur stehen Ihnen 120 Minuten zur Verfügung.
- Schalten Sie Mobiltelefone und ähnliche Geräte aus und verstauen Sie sie sicher.
  Während der Klausur sichtbare technische Hilfsmittel gelten als Täuschungsversuch.
- Benutzen Sie nur Stifte mit blauer oder schwarzer Tinte.
  Alles, was mit Bleistift oder in roter Tinte geschrieben ist, wird nicht gewertet.
- Als einziges Hilfsmittel ist ein doppelseitig *handbeschriebenes* DIN A4 Blatt zugelassen.

**Name, Vorname:** _____

**Matrikelnummer:** _____

**Erklärung:** Ich versichere, dass ich die Klausur selbstständig, ohne fremde Hilfe und ohne weitere Hilfsmittel bearbeitet habe.

_____
Unterschrift

**Bewertung (vom Prüfer auszufüllen):**

| Aufgabe: | 1 | 2 | 3 | 4 | 5 | 6 | $\sum$ |
|---|---|---|---|---|---|---|---|
| Punkte: | 18 | 20 | 20 | 14 | 16 | 12 | 100 |
| Erreicht: | | | | | | | |

**Aufgabe 1:** *Entwurf in Java* **(18 Punkte)**

Der bekannte Rennfahrer *Sebastian Stiefelmacher* beauftragt Sie, ein konzeptionell ausgeklügeltes System zur Berechnung der Fahrtgeschwindigkeit eines Gefährts zu entwerfen.

Dazu gibt es bereits einige Vorarbeiten. Die Klasse `ValueUnit` repräsentiert einen mit einer Einheit versehen Wert, z.B. 300 Meter. Der Wert wird als **`double`** angegeben, die Einheit wird über den generischen Parameter der Klasse angegeben. Eine Instanz der Einheit wird außerdem gespeichert, um Werte bei abgeleiteten Einheiten umrechnen zu können, z.B. 300 Meter werden zu 0.3 Kilometern.

Das Interface `Unit` ist die Schnittstelle für Einheiten, die einen Namen und eine Methode zur Umrechnung bereit stellt.

Die Klassen `Meter` und `Second` (dt. Sekunde) repräsentieren Basiseinheiten, hier ist keine Umrechnung nötig, so dass die Berechnungsmethode der Identitätsfunktion entspricht.

Die Klassen `Kilometer` und `Hour` (dt. Stunde) repräsentieren abgeleitete Einheiten, die jeweils die Basiseinheiten `Meter` bzw. `Second` erweitern. Ein Kilometer (km) entspricht 1000 Metern, eine Stunde (h) entspricht 3600 Sekunden.

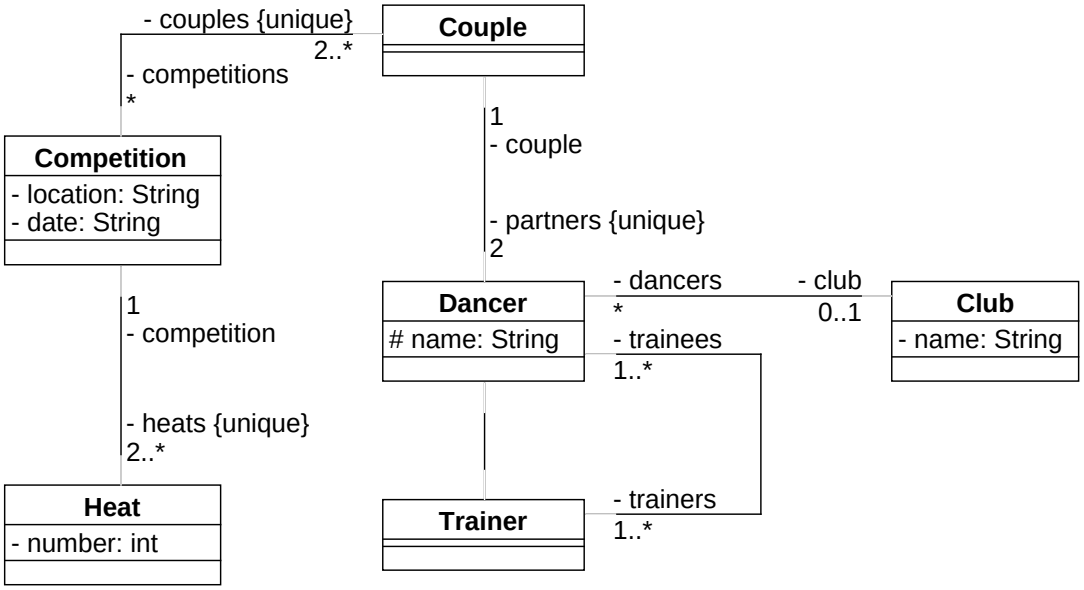
Ergänzen Sie die gegebenen Klassen an den markierten Stellen. Achten Sie dabei auf saubere Modellierung, insbesondere auf Konzepte wie Polymorphie, Kapselung, Lokalität und Wiederverwendung statt Duplizierung.

```
1  public class UnitCalculator {
2
3    public static void main(String[] args) {
4      ValueUnit<? extends Meter> distance =
5          new ValueUnit<Kilometer>(50, new Kilometer());
6      ValueUnit<? extends Second> duration = new ValueUnit<Hour>(1, new Hour());
7      System.out.println(distance.getValue()); // 50000.0
8      System.out.println(duration.getValue()); // 3600.0
9      System.out.println(distance.getValue() / duration.getValue()); // 13.89
10   }
11
12 }
13
14 public class ValueUnit<_____________________________________> {
15
16   protected double value;
17
18   protected ______ unit;
19
20   public ValueUnit(double value, ______ unit) {
21     this.value = value;
22     this.unit = unit;
23   }
24
25   public double getValue() {
26     return unit.calculate(value);
27   }
28
29 }
30
31 public interface Unit {
32
33   String getName();
34   double calculate(double value);
35
36 }
37
```

```
38 public class Meter ____________________________________ {
39
40   @Override
41   public String getName() {
42     return ”m”;
43   }
44
45   @Override
46   public double calculate(double value) {
47     return value;
48   }
49
50 }
51
52 public class Kilometer _____________________________ {
53
54   @Override
55   public String getName() {
56     ____________________________________________;
57   }
58
59   @Override
60   public double calculate(double value) {
61     return ____________________________________________;
62   }
63
64 }
65
66 public class Second ____________________________________ {
67
68   @Override
69   public String getName() {
70     return ”s”;
71   }
72
73   @Override
74   public double calculate(double value) {
75     return value;
76   }
77
78 }
79
80 public class Hour extends Second {
81
82   @Override
83   public String getName() {
84     return ”h”;
85   }
86
87   @Override
88   public double calculate(double value) {
89
90     return ____________________________________________;
91   }
92
93 }
```

**Aufgabe 2:** *Objektdiagramme* **(11 + 9 Punkte)**

Gegeben sei folgendes UML-Klassendiagramm:




Es beschreibt TÄNZER (engl. DANCER) und verschiedene verwandte Konzepte.

Ein VEREIN (engl. CLUB) besteht aus Tänzern.

Jeder Tänzer hat mindestens einen TRAINER, jeder Trainer trainiert mindestens einen Tänzer. Alle Trainer sind auch Tänzer.

Je zwei unterschiedliche Tänzer können sich zu einem PAAR (engl. COUPLE) zusammenschließen.

Ein TURNIER (engl. COMPETITION) kommt zustande, wenn mindestens zwei verschiedene Paare daran teilnehmen. Es können nur Paare an Turnieren teilnehmen, bei denen beide Partner dem gleichen Verein angehören. Letztere Bedingung ist im Klassendiagramm nicht ausgedrückt, muss aber bei der Umsetzung eingehalten werden.

Ein Turnier besteht aus mindestens zwei getrennten RUNDEN (engl. HEAT).

**a)** Vervollständigen Sie das oben angegebene Klassendiagramm: Zeichnen Sie an allen 12 Stellen die Pfeilenden ein. Verwenden Sie insbesondere auch Kompositionen oder Aggregationen, wo dies sinnvoll ist. Geben Sie die Navigierbarkeiten explizit an.

**b)** Zeichnen Sie ein Objektdiagramm, das zu dem obigen Klassendiagramm passt. Es muss genau eine Instanz eines Turniers (`Competition`) enthalten, und soll sonst die kleinst-mögliche Anzahl von Objekten enthalten, die durch das Klassendiagramm noch erlaubt ist.
Sie dürfen einzelne Buchstaben wie “A” oder “B” als Namen verwenden. Sie dürfen außerdem die Rollennamen der Links weglassen.

**Aufgabe 3:** *Input/Output* **(20 Punkte)**

Der bekannte Buchladen *Tarnish and Dots* will sein Geschäft um virtuelle Bücher erweitern. Dazu hat es einen etwas zwielichtigen Programmierer beauftragt, der eine E-Book Implementierung in Java bereits in Teilen umgesetzt hat. Leider musste er sehr spontan „verreisen", und konnte seine Arbeit nicht fertig stellen. Dies ist nun Ihre Aufgabe!

Die Klasse EBook verfügt über die notwendigen Typen und Attribute, um ein einfaches E-Book zu repräsentieren. Es hat einen TITEL (engl. TITLE), eine Liste von AUTOREN (engl. AUTHORS) und eine Reihe von KAPITELN (engl. CHAPTERS). Ein Kapitel hat ebenfalls einen Titel, und außerdem eine Reihe von ABSÄTZEN (engl. PARAGRAPHS) sowie eine Liste von VERWEISEN (engl. REFERENCES) auf andere Kapitel.

Auch eine Methode `save` zum Speichern des Buchs in einer Datei existiert. Leider fehlt noch das Gegenstück: Die Methode `load` ist nicht fertig implementiert.

Analysieren Sie den bestehende Code. Sie dürfen diesen Code *nicht* verändern. Vervollständigen Sie dann die Methode `load`, so dass sie das aktuelle E-Book mit den Daten aus der gegebenen Datei überschreibt. Dabei gibt es allerdings noch zwei Besonderheiten: Wenn in der Datei keine Kapitel gespeichert sind, soll eine `EmptyBookException` geworfen werden. Wenn für das Buch (nicht für einzelne Kapitel) kein Titel gespeichert wurde, soll eine `MissingTitleException` geworfen werden.

```
1  public class EBook {
2
3    private String title;
4    private List<String> authors = new ArrayList<>();
5    private List<Chapter> chapters = new ArrayList<>();
6
7    // Getters and Setters
8
9    public void save(File file) throws FileNotFoundException, IOException {
10     try (DataOutputStream out = new DataOutputStream(
11           new FileOutputStream(file))) {
12       if (title != null) {
13         out.writeBoolean(true);
14         out.writeUTF(title);
15       } else {
16         out.writeBoolean(false);
17       }
18       out.writeInt(authors.size());
19       for (String author : authors) {
20         out.writeUTF(author);
21       }
22       out.writeInt(chapters.size());
23       for (Chapter chapter : chapters) {
24         chapter.save(out);
25       }
26       for (Chapter chapter : chapters) {
27         out.writeInt(chapter.references.size());
28         for (Chapter reference : chapter.references) {
29           out.writeInt(chapters.indexOf(reference));
30         }
31       }
32     }
33   }
34
35   public void load(File file) throws FileNotFoundException, IOException,
         EmptyBookException, MissingTitleException {
36     try (DataInputStream in = new DataInputStream(
37           new FileInputStream(file))) {
38
```

```
    }
  }

  public class Chapter { // inner class
    private String title;
    private List<String> paragraphs = new ArrayList<>();
    private List<Chapter> references = new ArrayList<>();

    public Chapter() {
      super();
      chapters.add(this);
    }
```

```
100
101     // Getters and Setters
102
103     private void save(DataOutputStream out) throws IOException {
104       if (title != null) {
105         out.writeBoolean(true);
106         out.writeUTF(title);
107       } else {
108         out.writeBoolean(false);
109       }
110       out.writeInt(paragraphs.size());
111       for (String paragraph : paragraphs) {
112         out.writeUTF(paragraph);
113       }
114     }
115   }
116
117   public class MissingTitleException extends Exception {
118     private static final long serialVersionUID = 1L;
119   }
120
121   public class EmptyBookException extends Exception {
122     private static final long serialVersionUID = 1L;
123   }
124
125 }
```

**Aufgabe 4:** *Locks* **(4 + 10 Punkte)**

Der bekannte Pokerspieler *Johannes Bund* beauftragt Sie, ein einfaches Kartenspiel zu entwerfen.

Die grundlegenden Klassen sind bereits gegeben: `CardGame` erzeugt, startet und beendet ein Spiel.

`Card` repräsentiert eine einzelne Spielkarte.

Die Klasse `Game` verwaltet das Spiel: Das `deck` ist der verwendete Kartenstapel. `running` gibt an, ob das Spiel gerade läuft oder nicht. Es gibt mehrere `Player` (dt. Spieler (m/w/d)). Zur späteren Synchronisation wird außerdem ein `ReadWriteLock` bereitgestellt, das die Methoden `readLock(): Lock` und `writeLock(): Lock` definiert.

Neben Gettern für den Kartenstapel, für den Lauf-Zustand `running` und für das Lock gibt es Methoden zum Starten und Beenden des Spiels (`start()` und `stop()`).

Die Klasse `Player` implementiert das `Runnable`-Interface und hält einen Verweis auf das Spiel. Ein Spieler hat außerdem einen Namen und eine Liste von Karten auf der Hand (`hand`). Ein Spieler kann die oberste Karte vom Kartenstapel des Spiels anschauen (`look`) oder nehmen (`take`), und er kann einer seiner Karten dem Kartenstapel des Spiels hinzufügen (`give`). Die `interrupt`-Methode ist eine reine Komfort-Methode: sie ruft die `interrupt`-Methode des Threads auf, der den Spieler ausführt. Die `run`-Methode ruft, solange das Spiel läuft, zufällig eine der drei Methoden `look`, `take` oder `give` auf.

Zur Vereinfachung findet hier keine Prüfung statt, ob der Kartenstapel oder die Hand leer ist, bevor Karten entnommen werden.

**a)** Vervollständigen Sie die Methoden `start` und `stop` der Klasse `Game`: Sorgen Sie dafür, dass die gegebenen `Player` (pseudo-)parallel ihren Tätigkeiten nachkommen.
Sorgen Sie außerdem dafür, dass die Spieler beim Beenden des Spiels ihre Tätigkeit sofort einstellen und nicht noch etwas warten.

**b)** Vervollständigen Sie die Methoden `look`, `take` und `give` in `Player`: Sorgen Sie dafür, dass es bei (pseudo-)paralleler Ausführung nicht zu Konflikten kommen kann. Nutzen Sie dafür das `lock` der `Game`-Instanz!
Synchronisieren Sie die kritischen Abschnitte (der eingerückte Code in den drei Methoden) so, dass trotzdem möglichst viele Aktionen gleichzeitig ausgeführt werden können, sofern diese keine Konflikte erzeugen.

```
1  public class CardGame {
2
3    public static void main(String[] args) {
4      Game game = new Game();
5      game.start();
6      try {
7        Thread.sleep(5000);
8      } catch (InterruptedException e) { }
9      game.stop();
10   }
11
12 }
13
14 class Card {
15   public Card(String value) {
16     this.value = value;
17   }
18   private String value;
19   public String toString() {
20     return value;
21   }
22 }
```

```
23
24 class Game {
25   private static final int PLAYER_COUNT = 4;
26   private LinkedList<Card> deck = new LinkedList<>();
27   private boolean running;
28   private List<Player> players = new ArrayList<>();
29   private ReadWriteLock lock = new ReentrantReadWriteLock();
30   public synchronized boolean isRunning() {
31     return running;
32   }
33   public LinkedList<Card> getDeck() {
34     return deck;
35   }
36   public ReadWriteLock getLock() {
37     return lock;
38   }
39
40   public synchronized void start() {
41     deck.clear();
42     for (int i = 0; i < 1000; i++) {
43       deck.add(new Card("C" + Integer.toString(i)));
44     }
45     for (int i = 0; i < PLAYER_COUNT; i++) {
46       Player p = new Player("P" + Integer.toString(i + 1), this);
47       players.add(p);
48
49
50
51
52     }
53     running = true;
54   }
55
56   public synchronized void stop() {
57     running = false;
58
59
60
61
62
63
64   }
65 }
66 class Player implements Runnable {
67   private String name;
68   private Game game;
69   private LinkedList<Card> hand = new LinkedList<>();
70   public Player(String name, Game game) {
71     this.name = name;
72     this.game = game;
73   }
74
75   private void look() {
76
77
78
79
80
81       Card c = game.getDeck().getFirst();
82       System.out.println(name + ": Looking at " + c + ".");
83
```

```
84
85
86    }
87
88    private void take() {
89
90
91
92
93
94        Card c = game.getDeck().removeFirst();
95        hand.add(c);
96        System.out.println(name + ": Taking " + c + ".");
97
98
99
100   }
101
102   private void give() {
103
104
105
106
107
108       Card c = hand.removeFirst();
109       game.getDeck().addLast(c);
110       System.out.println(name + ": Giving " + c + ".");
111
112
113
114   }
115
116   public void interrupt() {
117     Thread.currentThread().interrupt();
118   }
119   @Override
120   public void run() {
121     while (game.isRunning()) {
122       double rand = Math.random();
123       if (rand < 0.5) {
124         look();
125       } else if (rand < 0.85 && !hand.isEmpty()) {
126         give();
127       } else {
128         take();
129       }
130       try {
131         Thread.sleep(Math.round(10 + 10 * Math.random()));
132       } catch (InterruptedException e) {
133         break;
134       }
135     }
136   }
137 }
```

**Aufgabe 5:** *Dynamische Programmierung* **(4 + 4 + 4 + 4 Punkte)**

Kreuzen Sie für jede Aussage entweder *Ja* oder *Nein* an. Korrekte Antworten werden positiv gewertet, falsche Antworten negativ, fehlende Antworten gehen nicht in die Bewertung ein. Sie können in einer Teilaufgabe *keine* negativen Punkte erreichen, auch nicht bei überwiegend falschen Antworten. Falsche Antworten führen innerhalb einer Teilaufgabe zwar zu Punktabzug, haben aber keinen Einfluss auf andere Teilaufgaben und können auch in der gleichen Teilaufgabe nicht zu einer negativen Punktezahl führen.

**Hinweis:** Versuchen Sie nicht zu raten. Damit verlieren Sie sonst ggf. Punkte aus richtigen Antworten.

**a)** Welche Aussagen gelten für Probleme, die per DYNAMISCHER PROGRAMMIERUNG gelöst werden können?

| Ja | Nein | |
|---|---|---|
| [ ] | [ ] | Es muss das Bellman'sche Optimalitätskriterium gelten. |
| [ ] | [ ] | Es muss ein Optimierungsproblem sein. |
| [ ] | [ ] | Überlappende Teilprobleme können zu einer effizienteren Lösung beitragen. |
| [ ] | [ ] | Es müssen sauber getrennte Kritische Abschnitte existieren. |

**b)** Welche Aussagen gelten allgemein für die DYNAMISCHE PROGRAMMIERUNG?

| Ja | Nein | |
|---|---|---|
| [ ] | [ ] | Sie kann auch NP-schwere Probleme effizient und optimal lösen. |
| [ ] | [ ] | Sie wandelt durch Memoization Laufzeitkomplexität in Speicherkomplexität um. |
| [ ] | [ ] | Sie verwendet Memoization, um Speicherplatz zu sparen. |
| [ ] | [ ] | Sie verwendet oft (auch) Rekursion zum Finden von Lösungen. |

**c)** Die folgende Methode `knapSack` soll eine Variante des Rucksack-Problems lösen: Es steht eine Reihe von Gegenständen mit den Werten `values` und den Gewichten `weights` bereit, die in einen Rucksack mit einem maximalen Pack-Gewicht `capacity` gepackt werden sollen. Der `index` gibt an, welcher Gegenstand gerade betrachtet wird (begonnen wird mit `index = values.length - 1`). Der `cache` hält die bereits berechneten Lösungen für verschiedene Rest-Kapazitäten:

```
1  public static int knapSack(int[] values, int[] weights, int index, int
       capacity, int[] cache) {
2    if (capacity < 0) { return Integer.MIN_VALUE; }
3    if (index < 0 || capacity == 0) { return 0; }
4    if (cache[capacity - 1] == 0) {
5      int option1 = values[index] + knapSack(values, weights, index,
           capacity - weights[index], cache);
6      int option2 = knapSack(values, weights, index - 1, capacity, cache);
7      cache[capacity - 1] = Math.max(option1, option2);
8    }
9    return cache[capacity - 1];
10 }
```

Welche Aussagen darüber sind wahr (ja) bzw. falsch (nein)?

| Ja | Nein | |
|---|---|---|
| [ ] | [ ] | Zeilen 2 und 3 können ohne das Ergebnis zu verändern zu `if (index < 0 || capacity <= 0)return 0;` zusammengefasst werden. |
| [ ] | [ ] | Als Ergebnis gibt die Methode den höchst-möglichen Wert zurück, der transportiert werden kann. |
| [ ] | [ ] | Ergebnisse für Rucksäcke mit geringerer Kapazität können hier für die Ergebnisse für Rucksäcke mit größerer Kapazität herangezogen werden. |
| [ ] | [ ] | Die Memoization kommt hier nicht zum Tragen, da die rekursiven Aufrufe nie auf bereits berechnete Teilergebnisse treffen. |

**d)** Betrachten Sie folgende Implementierung der FIBONACCI REIHE $fib(n) = \begin{cases} 0, \text{ falls } n = 0 \\ 1, \text{ falls } n = 1 \\ fib(n-1) + fib(n-2), \text{ sonst} \end{cases}$

```
1  public static int fib(int n) {
2    if (n <= 0) {
3      return 0;
4    }
5    int last = 0;
6    int current = 1;
7    for (int i = 0; i < n - 1; i++) {
8      int next = last + current;
9      last = current;
10     current = next;
11   }
12   return current;
13 }
```

Welche Aussagen darüber sind wahr (ja) bzw. falsch (nein)?

| Ja | Nein | |
|---|---|---|
| [ ] | [ ] | Der zweite Basisfall fehlt: Für $n = 1$ liefert die Methode ein falsches Ergebnis. |
| [ ] | [ ] | In Zeile 8 speichern `last` und `current` jeweils die Werte für $fib(i-2)$ und $fib(i-1)$. |
| [ ] | [ ] | Es handelt sich nicht um Dynamische Programmierung, da kein Array zur Speicherung der vorherigen Werte existiert. |
| [ ] | [ ] | Es kann sich nicht um eine korrekte Umsetzung der Fibonacci-Reihe handeln, da die Rekursion fehlt. |

**Aufgabe 6:** *SML* **(3 + 4 + 5 Punkte)**

Der bekannte Schulleiter *Canus Rumbledore* benötigt Ihre Hilfe bei der Verwaltung seiner Schülerdaten.

Gegeben ist eine Datenbank mit dem Schema `(Name, Durchschnittsnote)` in Standard ML:

```
(* schema: name, avgGrade *)
val db = [("Harry", 2.1), ("Hermine", 1.0), ("Ron", 3.3)];
```

```
val db = [("Harry",2.1),("Hermine",1.0),("Ron",3.3)] : (string * real) list
```

Definieren Sie folgende SML-Funktionen:

**a)** Eine Funktion `insert` zum Einfügen (an beliebiger Stelle) von neuen Daten in die Datenbank. Die neuen Daten sowie die vorhandene Datenbank werden der Funktion übergeben, die neue Datenbank wird zurückgegeben:

```
val insert = fn : 'a * 'a list -> 'a list
```

**b)** Eine Funktion `lookup` zum Nachschlagen der Durchschnittsnote einen Schülers mit gegebenem Namen. Der Name und die Datenbank werden der Funktion übergeben, die passende Note wird zurückgegeben:

```
val lookup = fn : ''a * (''a * real) list -> real
```

**c)** Eine Funktion `update` zum Ersetzen der Durchschnittsnote eines Schülers durch eine andere. Der Name, die neue Note sowie die Datenbank werden der Funktion übergeben, die neue Datenbank wird zurückgegeben:

```
val update = fn : ''a * 'b * (''a * 'b) list -> (''a * 'b) list
```