TU Darmstadt — Fachbereich Informatik — Telekooperation / Rechnerbetriebsgruppe

Klausur

Grundlagen der Informatik 1 – WS 2008/09

30.03.2009 – 8:00 - 10:00 Uhr

Hinweise:

- Als Schreibmittel ist nur ein schwarzer oder blauer Schreibstift erlaubt.
- Füllen Sie das Deckblatt vollständig aus!
- Schreiben Sie auf jedes Aufgabenblatt Ihren Namen und Matrikelnummer.
- Schreiben Sie Ihre Lösung in die vorgesehenen Zwischenräume oder auf die Rückseite des jeweiligen Aufgabenblattes.

Nachname	
Vorname	
Matrikelnr.	
Studiengang	
Semester	

Aufgabe	1	2	3	4	5	6	7	8	9	$\sum$
Punkte										
Maximum	18	11	6	10	15	6	7	15	12	**100**

1 OO Design (18P)

1.1 Analyse (3P)

Zeichnen Sie ein UML-artiges Diagramm (wie in der Vorlesung verwendet) der Typhierachie der unten gezeigten Klassen aus einer Sokoban Implementierung.

- Bei abstrakten Klassen unterstreichen Sie den Klassennamen.
- **Methoden und Felder brauchen Sie nicht anzugeben.**

```java
public class Field {
  // Contents of the field or null if the field is empty
  public FieldContent content;
  public Field getNeighbor(char direction) {
    //...
  }
  // additional methods...
}
```

```java
public abstract class FieldContent {
  // Field on which this object is placed
  public Field placedOn;
  public abstract void move(char direction)
    throws CannotMoveException;
}
```

```java
public class Worker extends FieldContent {
  public void move(char direction) throws CannotMoveException {
    //...
  }
}
```

```java
public class Box extends FieldContent {
  public void move(char direction) throws CannotMoveException {
    //...
  }
}
```

1.2 Klasse Wall (2P)

Im Spiel Sokoban spielen neben dem Arbeiter (Worker) und Kisten (Box) auch Wände (Wall) eine große Rolle. Ergänzen Sie ihr Diagramm um eine Klasse `Wall`, so dass der Inhalt (gespeichert im Feld `content`) eines `Field` Objektes auch ein `Wall` Objekt sein kann, ohne Änderungen an der Implementierung von `Field` vorzunehmen. Kopieren Sie dazu das Diagramm von Teilaufgabe 1.1 und ergänzen Sie die Klasse `Wall`.

1.3 Implementierung der Klasse Wall (3P)

Die Methode `move` soll eine `CannotMoveException` werfen, wenn sie auf einem Objekt aufgerufen wird, dass nach den Sokoban-Regeln nicht in die angegebene Richtung ('U'p, 'L'eft, 'D'own, 'R'ight) bewegt werden darf. Diese Exception erbt *nicht* von `java.lang.RuntimeException`. Bei einer ungültigen Bewegung darf in der Methode keine Änderung am Spielfeld vorgenommen werden. Wände dürfen nach den Sokoban Regeln nie bewegt werden.
Geben Sie eine Java-Implementierung der Klasse `Wall` an, so dass alle anderen Klassen unverändert bleiben können. Auf die Angabe von `imports` können Sie verzichten. **Um Schreibaufwand zu sparen, dürfen Sie in allen Teilaufgaben von Aufgabe 1 auf Kommentare verzichten.**

1.4 Re-Design (4P)

Nehmen Sie an, dass die `move` Methoden in `Box` und `Worker` redundante Codeteile enthalten. Das soll durch ein geeignetes Re-Design behoben werden. Eine neue **abstrakte** Klasse `MoveableContent` soll in die Typhierarchie eingefügt werden. In diese Klasse sollen die Codeteile, die von `Box` und `Worker` verwendet werden, ausgelagert werden, so dass sie von `Worker` und `Box` wiederverwendet werden können. Zeichnen Sie ein **neues** Diagramm, das alle Klassen aus dem Diagramm oben enthält sowie die zusätzliche Klasse `MoveableContent`.

1.5 Implementierung mit neuem Design (6P)

Betrachten Sie nun die unten angegebene vollständige Java-Implementierung der Klassen `Worker` und `Box`.

```
1  public class Box extends FieldContent {
2    public void move(char direction) throws CannotMoveException {
3      if (placedOn.getNeighbor(direction).content == null) {
4        placedOn.content = null;
5        placedOn = placedOn.getNeighbor(direction);
6        placedOn.content = this;
7      } else {
8        throw new CannotMoveException();
9      }
10   }
11 }
```

```
1  public class Worker extends FieldContent {
2    public void move(char direction) throws CannotMoveException {
3      if (placedOn.getNeighbor(direction).content != null) {
4        placedOn.getNeighbor(direction).content.move(direction);
5      }
6
7      if (placedOn.getNeighbor(direction).content == null) {
8        placedOn.content = null;
9        placedOn = placedOn.getNeighbor(direction);
10       placedOn.content = this;
11     } else {
12       throw new CannotMoveException();
13     }
14   }
15 }
```

Geben Sie Java-Implementierungen für die Klassen `MoveableContent`, `Box` und `Worker` an, so dass keine Redundanz in der Implementierung der `move` Methode in den einzelnen Klassen mehr vorliegt.

2 JUnit (11 P)

Ihre Aufgabe ist es, einige Testfälle für eine Methode aus einer Bibliothek für Geometrie-Funktionen zu schreiben. Dabei betrachten wir nur die Funktion `triangleArea`, die den Flächeninhalt eines Dreiecks berechnet und dazu die Längen der einzelnen Seiten (a, b und c) *int*-Werte konsumiert.

2.1 Setup-Methode (2P)

Um die Funktionen der Bibliothek verwenden zu können, muss zunächst ein Objekt vom Typ `GeoLib` erzeugt werden. Hierzu können Sie den parameterlosen Standard-Konstruktor der Klasse `GeoLib` verwenden. Geben Sie eine entsprechend mit JUnit-Annotationen versehene Methode an, die für jeden Testfall eine neue Instanz von `GeoLib` in dem bereits deklarierten Attribut `geoLib` speichert.

2.2 Normalfall (2P)

Geben Sie einen JUnit-Test an, der überprüft, ob die Methode für ein einfaches Dreieck korrekt arbeitet. Verwenden Sie hierzu die Seitenlängen 3, 4 und 5, die ein Dreieck mit einem Flächeninhalt von 6 ergeben sollten. Sie können davon ausgehen, dass alle benötigten imports von JUnit zur Verfügung stehen.

2.3 Behandlung von Exceptions (3P)

Geben Sie einen JUnit-Test an, der überprüft, ob die Methode `triangleArea` für fehlerhafte Daten korrekt arbeitet. Laut Spezifikation soll die Methode eine `NoTriangleException` werfen, wenn die übergebenen Seiten kein Dreieck ergeben, beispielsweise wenn eine Seite die Länge 0 hat. Nur bei Auftreten einer Exception exakt der Klasse `NoTriangleException` soll der Testfall erfolgreich sein.

2.4 Weiterer Testfall (4P)

Überlegen Sie sich einen weiteren sinnvollen Testfall. Geben Sie zunächst eine kurze Begründung, warum Sie diesen Testfall wählen würden und wieso dieser Fall nicht von den beiden Testfällen oben abgedeckt wird (2P). Geben Sie dann eine JUnit Methode zu diesem Testfall an.

3 Generische Klassen (6P)

Betrachten Sie die folgende generische Java-Klasse für Paare.

```
1  public class Pair<X,Y> {
2    private X first;
3    private Y second;
4    public Pair(X a1, Y a2) {
5      first  = a1;
6      second = a2;
7    }
8    public X getFirst()  { return first; }
9    public Y getSecond() { return second; }
10 }
```

- Geben Sie einen Java-Code an, um eine Variable `intPair` zu *deklarieren*, die Paare von `Integer`-Objekten speichert (2P).

- Geben Sie Java-Code an, um in der Variablen `intPair` das Paar (3,4) zu speichern (2P).

- Akzeptiert der Java-Compiler den folgenden Code? Begründen Sie Ihre Antwort kurz (2P).

```
1  System.out.println(intPair instanceof Pair<Integer, Integer>);
```

4 Collections und Generics (10P)

Betrachten Sie folgenden Java-Code. Geben Sie hinter den mit * markierten Zeilen an, ob diese Zeile vom Compiler akzeptiert wird („OK“) oder ob sie nicht akzeptiert wird („Fehler“). Für die erste Zeile `A = 5;` is bereits beispielhaft „OK“ eingetragen, da diese Zeile keinen Compiler-Fehler verursacht.
Korrekt markierte Zeilen geben 1 Punkt, **falsch markierte Zeilen geben 1 Punkt Abzug**. *Wenn Sie sich nicht sicher sind, lassen Sie die Zeile frei*, dann erhalten sie zwar keine Punkte, aber auch keinen Abzug. Die Gesamtpunktzahl dieser Teilaufgabe kann nicht unter 0 sinken.

```
1  public void foo(List<Integer> listA,
2      List<? extends Object> listB, List<? super Integer> listC) {
3
4    Integer A = new Integer(1);
5    Object B = new Object();
6    Integer C;
7
8    A = 5; //* OK
9
10   listA.add(A); //*
11
12
13   listA.add(B); //*
14
15
16   listB.add(A); //*
17
18
19   listB.add(B); //*
20
21
22   listC.add(A); //*
23
24
25   listC.add(B); //*
26
27
28   C = listA.get(0); //*
29
30
31   C = listB.get(0); //*
32
33
34   C = listC.get(0); //*
35
36
37   B = listC.get(0); //*
38
39
40 }
```

5 GdI 1-Notenberechnung (15P)

Die Formalitäten zur Bestimmung der Noten in der GdI 1-Vorlesung sind kompliziert, daher möchte sich der Dozent von einem Java Programm unterstützen lassen. In den folgenden Teilaufgaben sollen Sie verschiedene Teile des Programms implementieren. **Auf die Kommentierung mit Javadoc dürfen Sie verzichten.**

5.1 Modellierung eines Bachelor Informatik Studenten (4P)

Der Typ `Student` ist wie unten angegeben definiert. Der Konstruktor der Klasse ist nicht angegeben. Sie können davon ausgehen, dass dieser wie erwartet funktioniert.

```
1  public abstract class Student {
2    private double mts, hws, ls, fes;
3    private int id;
4
5    // the student ID ("Matrikelnummer")
6    public int getStudentID() {
7      return id;
8    }
9
10   // points reached in the mid-term exam (Dec 1)
11   public double midTermExamScore() {
12     return mts;
13   }
14
15   // points reached in the homework assignments
16   public double homeworkScore() {
17     return hws;
18   }
19
20   // points reached in the project
21   public double labScore() {
22     return ls;
23   }
24
25   // points reached in the final exam (March 30)
26   public double finalExamScore() {
27     return fes;
28   }
29
30   // total number of final exam points including bonus points
31   public abstract double totalScore();
32
33   // Did this student reach the exam admission ("Studienleistung")?
34   public abstract boolean passedExamAdmission();
35 }
```

Implementieren Sie die nicht-abstrakte Klasse `BachelorInformatik`, die von `Student` erbt. *Auf den Konstruktor dürfen Sie verzichten.*

Für `BachelorInformatik` gilt diese Regelung: Sie sind zugelassen (d.h. `passedExamAdmission()` liefert `true`), wenn

- mindestens 50 Punkte in den Hausübungen erzielt wurden (→ `homeworkScore()`)
- *und* mindestens 50 Punkte in der Zwischenklausur erzielt wurden (→ `midTermExamScore()`)
- *und* mindestens insgesamt 110 Punkte in der Zwischenklausur und den Hausübungen erzielt wurden (→ `midTermExamScore(), homeworkScore()`).

Die Gesamtpunktzahl (→ `totalScore`) ergibt sich dann wie folgt:

- Die Gesamtpunktzahl ist die Summe aus der Punktzahl der Abschlussklausur, Praktikumspunktzahl und möglichen Bonuspunkten.
- Bonuspunkte sind die Punkte aus der Punktsumme von Hausübungen und Zwischenklausur, die über 130 hinausgehen. Das heißt bei 135 Punkten in Hausübung und Zwischenklausur bekommt der Student 5 Bonuspunkte.
- *Achtung*: Der Bonus wird nur hinzugerechnet, wenn in der Klausur mindestens 30 Punkte erzielt wurden.

5.2 Notenberechnung (6P)

Nehmen Sie an, dass für alle Studiengänge entsprechende Klassen angelegt wurden, die alle von `Student` erben. Schreiben Sie nun eine Methode `gradeStudents`, die eine Liste von Objekten vom Typ `Student` erhält und eine `HashMap` (→ T15.35f) zurückgibt, bei der jeder Eintrag aus der Matrikelnummer eines Studenten und seiner Note (vom Typ `double`) besteht. Die Note `Double.NaN` stehe dabei für „Prüfungsleistung nicht vollständig erbracht".
Für zugelassene Studenten wird die Note direkt aus deren Gesamtpunktzahl t (aus `totalScore()`) berechnet. Zur Vereinfachung gibt es nur ganzzahlige Noten (1.0, 2.0, 3.0, 4.0, 5.0). Die Formel für die Notenberechnung sei wie folgt:

$$grade(t) = \begin{cases} \textit{Double.NaN} & passedExamAdmission() == false \\ \textit{Double.NaN} & labScore() == 0 \text{ oder } finalExamScore() == 0 \\ 5.0 & t < 110 \text{ oder } labScore() < 50 \text{ oder} finalExamScore() < 50 \\ 4.0 - \lfloor \frac{t-110}{27} \rfloor & 110 \leq t < 191 \\ 1.0 & sonst \end{cases}$$

Dabei ist $\lfloor x \rfloor$ der ganzzahlige Anteil von x, d.h. $\lfloor 3.7 \rfloor = 3$. Verwenden Sie dafür die statische Methode `Math.floor(double)`.
Implementieren Sie nun die Methode `gradeStudents`. Auf die Angabe von `imports` und JavaDoc können Sie verzichten.

```
public HashMap<Integer, Double> gradeStudents(List<Student> l){
```

5.3 Filter (5P)

Implementieren Sie nun eine Methode `passedStudents`, die die `HashMap` der vorherigen Aufgabe übergeben bekommt. Als Ergebnis soll eine Liste von Strings geliefert werden, die aus der Matrikelnummer, einem Tab und der Note des jeweiligen Studierenden besteht:

`0123456\t4.0`

Diese Liste soll *nur* die Studierenden umfassen, die die Prüfung vollständig abgelegt haben, also nicht die Note *Double.NaN* haben. Auch hier können Sie auf Javadoc verzichten.
Hinweis: Sie können die Methode `Set<Integer> keySet()` zum Zugreifen auf die Schlüssel der `HashMap<Integer, String>` verwenden.

public List<String> getGradeList(HashMap<Integer, Double> s){

6 Komposition (6P)

Die *Komposition* zweier Funktionen f und g ist die Funktion $(f \circ g)$ mit

$$(f \circ g)(x) = f(g(x))$$

Implementieren Sie die *Scheme*-Funktion `compose`, die als Argumente zwei Funktionen f und g erhält und als Ergebnis die Funktion $(f \circ g)$ liefert (3P).
Die Funktionen f und g sollen dabei je eine Zahl konsumieren und eine Zahl zurückgeben. Geben Sie ebenfalls den Vertrag (2P) und ein Beispiel (1P) an. Der Zweck ist bereits angegeben, auf Testfälle können Sie verzichten.

```
;; purpose: Zu übergebenen Funktionen f und g
;;           liefert compose die Komposition (f o g)
```

7 Event Handler (7P)

Gegeben sei eine Funktion `display`, die ein übergebenes beliebiges Argument auf der Konsole ausgibt. Gehen Sie davon aus, dass für diese Funktion eine geeignete Funktion sowohl in Java als auch in Scheme implementiert ist.
Der folgende Java Code definiert einen GUI Button und weist ihm einen Event-Handler zu.

```
1 JButton myButton = new JButton();
2
3 myButton.addActionListener(
4  new ActionListener() {
5   public void actionPerformed(ActionEvent e) {
6      display(e);
7   }
8  }
9 );
```

- (1P) Java kennt keine Lambda Ausdrücke und verwendet stattdessen anonyme verschachtelte Klassen. Von welchem Typ ist die anonyme verschachtelte Klasse im Beispiel abgeleitet?

- (2P) Gegeben sei folgende Scheme Definition:

  ```
  1 (define-struct jbutton (action-listener))
  ```

  Geben Sie einen gültigen Scheme Ausdruck an, der eine Variable mit dem Namen `my-button` erzeugt, die eine Struktur vom Typ `jbutton` enthält. Den Wert des Parameters `action-listener` dürfen Sie beliebig wählen, z.B. `0`.

- (4P) Gehen Sie nun davon aus, dass das Argument `action-listener` eine Funktion mit einem Argument sein soll, die automatisch vom System aufgerufen wird, wenn auf den Button geklickt wird. Geben Sie einen gültigen Scheme Ausdruck an, der...
  - eine Variable mit dem Namen `my-button` erzeugt
  - die eine Struktur vom Typ `jbutton` enthält,
  - so dass bei einem Klick auf den Button das Verhalten äquivalent zum oben gegebenen Java Code ist.

8 Akkumulatoren: Roulette (15P)

Im Folgenden betrachten wir eine vereinfachte Form des Spiels Roulette, in der man nur auf eine der beiden Farben Rot oder Schwarz setzen kann. Setzt man auf die richtige Farbe, erhält man das doppelte seines Einsatzes zurück, ansonsten verliert man seinen Einsatz.
Herr Meier geht bei diesem Roulette nach folgendem Prinzip vor:

- Er setzt stets auf Rot.
- Er beginnt mit einem Einsatz von 1€.
- Verliert er, verdoppelt er den Einsatz.
- Gewinnt er, so beginnt er wieder mit einem Einsatz von 1€.

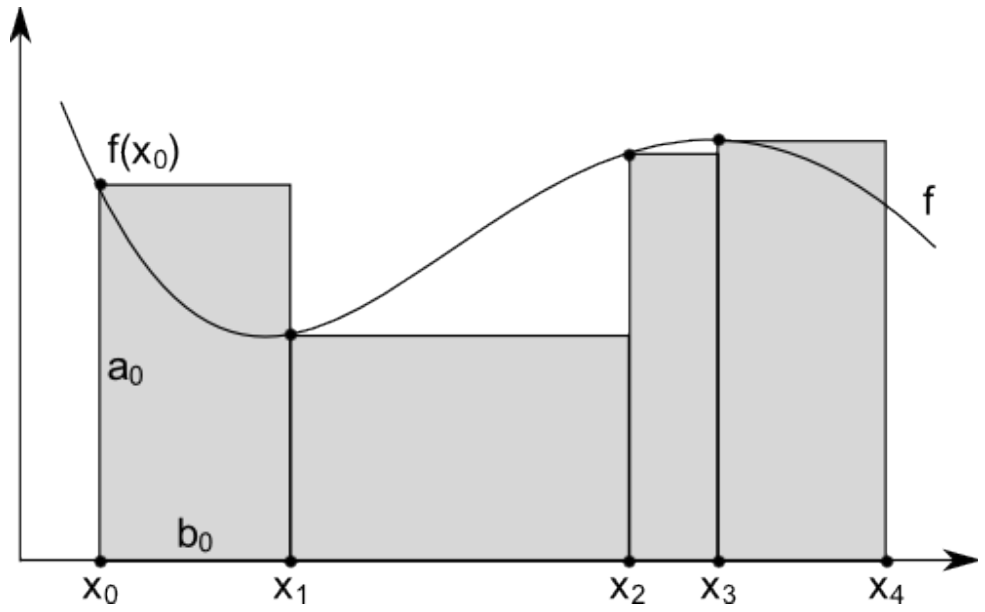
Zur Illustration folgender beispielhafter Spielablauf:

- Herr Meier beginnt das Spiel mit dem Einsatz von 1€.
- Die Kugel fällt auf Rot und Herr Meier bekommt seinen Einsatz zurück und seinen Einsatz nocheinmal als Gewinn. Der Gesamtgewinn ist somit 1€ und der nächste Einsatz immer noch 1€. Im Folgenden wird hierzu folgende Kurzschreibweise verwendet: Rot → Gewinn: 1€, Einsatz 1€.
- Schwarz → Gewinn: 1-1 = 0€, Einsatz 2€
- Rot → Gewinn: 0+2 = 2€, Einsatz 1€
- Schwarz → Gewinn: 2-1 = 1€, Einsatz 2€

Schreiben Sie eine Scheme Prozedur, die eine Liste von Symbolen bestehend aus `'red` und `'black` konsumiert und zurückgibt, wieviel Geld Herr Meier bei dieser Folge gewonnen bzw. verloren hätte. Verwenden Sie dazu Akkumulatoren. Verwenden Sie kein `local`! Geben Sie für jede verwendete Prozedur den *Vertrag* und die *Beschreibung* und einen *Test* an. Auf ein Beispiel können Sie verzichten.

9 Map und Fold (12P)

Das Integral einer Funktion f kann numerisch durch *Quadratur* angenähert werden. Dazu zerlegt man die Fläche unter der Funktion in Rechtecke. Die Breite der Rechtecke wird durch die Stützstellen $x_0 ... x_n$ (die x_i seien aufsteigend nummeriert) festgelegt, die Höhe durch den Funktionswert an der entsprechenden Stelle (siehe Abbildung). Das Integral wird durch die Summe der Flächeninhalte der Rechtecke angenähert.



Hinweis: Funktionen, die bereits in Übung oder Skript definiert wurden, dürfen zur Lösung der Aufgabe verwendet werden, insbesondere die Funktion `head`, die das letzte Element einer Liste entfernt.
Beispiel: `(head (list 1 2 3 4))` → `(list 1 2 3)`
Lösen Sie die folgenden Teilaufgaben **ohne Verwendung von Rekursion**!

9.1 Berechnung von a_i (2P)

Die Höhe a_i der bei Quadratur von f mit Stützstellen $x_0 ... x_n$ entstehenden Rechtecke ist

$$a_i = f(x_i), \text{ für } 0 \leq i \leq n$$

Somit ergeben die Höhen eine Folge $f(x_0), f(x_1), ..., f(x_n)$.
Implementieren Sie eine Funktion `a-list`, die eine Funktion `f` und eine Liste `x-list` von Stützstellen erhält und eine Liste mit allen a_i wie oben definiert zurückliefert. Verwenden Sie `map` (2P)! Auf Angabe von Zweck, Vertrag, Beispiel und Test können Sie verzichten.

9.2 Berechnung von b_i (5P)

Die Breite b_i der der bei Quadratur mit den Stützstellen $x_0 ... x_n$ entstehenden Rechtecke ist

$$b_i = x_{i+1} - x_i, \text{ für } 0 \leq i \leq n-1$$

Damit ergeben die Breiten eine Folge $(x_1 - x_0), (x_2 - x_1), ..., (x_n - x_{n-1})$.

Implementieren Sie eine Funktion `b-list`, die eine Liste von Stützstellen `x-list` erhält und eine Liste mit allen Werten b_i wie oben definiert zurückliefert. Verwenden Sie **keine** Rekursion! Auf Angabe von Zweck, Vertrag, Beispiel und Test können Sie verzichten.

9.3 Berechnung von Q (5P)

Das Integral wird bei der Quadratur von f mit den Stützstellen $x_0...x_n$ durch folgende Summe angenähert:

$$Q(f, (x_0...x_n)) = \sum_{i=0}^{n-1} a_i \cdot b_i$$

also durch die Summe $a_0 \cdot b_0 + a_1 \cdot b_1 + ... + a_{n-1} \cdot b_{n-1}$.

Implementieren Sie eine Funktion `Q`, die eine Funktion `f` und eine Liste von Stützstellen `x-list` erhält und Q wie oben definiert ausrechnet. Sie können die Funktionen `a-list` und `b-list` aus den vorigen Teilaufgaben verwenden. Verwenden Sie keine Rekursion sondern `map` und `foldl`! Auf Angabe von Zweck, Vertrag, Beispiel und Test können Sie verzichten.