TU Darmstadt — Fachbereich Informatik — Telekooperation / Rechnerbetriebsgruppe

Bachelor-Modulprüfung

Grundlagen der Informatik 1 – Sommer 2009

23. 09. 2009 – 08:00 - 10:00 Uhr

Hinweise:

- Als Schreibmittel ist nur ein schwarzer oder blauer Schreibstift erlaubt.
- Als Hilfsmittel können Sie unsere Folien, Übungen, sonstige Notizen, Bücher, Wörterbücher und sonstige Literatur benutzen.
- Bitte 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 unter Angabe der dazugehörigen Aufgabennummer.
- **Achtung: Aufgabe 10 richtet sich ausschließlich an Teilnehmer der Sonderregelung!**

Nachname	
Vorname	
Matrikelnr.	
Studiengang	
Semester	

Aufgabe	1	2	3	4	5	6	7	8	9	*10*	$\sum$
Erreichbar	18	11	6	10	15	7	15	12	6	*20*	100/*120*
Punkte											

1 OO Design (18 P.)

1.1 Klassendiagramm (3 P.)

1. Zeichnen Sie ein Klassendiagramm für eine Klassenhierarchie aus folgenden Klassen: Tree, FruitTree, MangoTree und PearTree[1] (2 P.).

   Hinweis: Markieren Sie ggf. abstrakte Klassen mit einem Unterstrich.

2. Erläutern Sie kurz, warum Sie das Diagramm so und nicht anders gezeichnet haben (1 P.).

Die Klasse Forest wird für den Rest der Aufgabe benötigt.

```
1  public class Forest {
2     private Tree[] trees;
3
4     public Forest(Tree[] trees) {
5        this.trees = trees;
6     }
7
8     public void workOnTrees(TreeWorker w) {
9      System.out.println(w + " enters forest");
10       for (Tree t: trees) {
11          t.letWorkerDoWork(w);
12       }
13    }
```

[1] fruit tree: *engl.* Obstbaum; pear: *engl.* Birne

```
14
15      public static void main(String[] args) {
16          MangoTree m = new MangoTree();
17          PearTree p = new PearTree();
18          Forest f = new Forest(new Tree[] {m,p});
19
20          Harvester h = new Harvester();
21          f.workOnTrees(h);
22
23              //A
24              //Woodman w = new Woodman();
25              //f.workOnTrees(w);
26
27              //B
28              //AdvancedWoodman a = new AdvancedWoodman();
29              //f.workOnTrees(a);
30
31      }
32  }
```

1.2 Harvester (3 P.)

Die Klassen *MangoTree*, *FruitTree* und *PearTree* sind zunächst leer, d.h. übernehmen die vollständige Implementierung ihrer Oberklassen. Die Klassen *Tree*, *TreeWorker* und *Harvester* sind wie folgt definiert.

```
1  public class Tree {
2    public String toString() { return "tree"; }
3     public void letWorkerDoWork(TreeWorker w) {
4        w.workOn(this);
5     }
6  }
```

```
1  public abstract class TreeWorker {
2    public String toString() { return "treeworker"; };
3     public abstract void workOn(Tree t);
4  }
```

```
1  public class Harvester extends TreeWorker {
2    public void workOn(Tree t) {
3      System.out.println("harvesting tree");
4    }
5    public void workOn(MangoTree t) {
6      System.out.println("harvesting mango tree");
7    }
8    public void workOn(PearTree t) {
9      System.out.println("harvesting pear tree");
10   }
11 }
```

Welche Ausgabe entsteht bei Ausführung der main-Methode der Klasse Forest? (3 P.)

1.3 Förster - Klassendiagramm (4 P.)

Die Aufgabe des Försters ist es, die Bäume in einem Wald zu zählen.

1. Zeichnen Sie ein Klassendiagramm der Klassen, *TreeWorker*, *Harvester* und *Woodman* (engl. Woodman = Förster), so dass der auskommentierte, mit A markierte Code in Forest kompiliert (3 P.).

2. Begründen Sie kurz die Wahl ihrer Oberklasse, bzw. wieso die Oberklasse *Object* ist, für *Woodman* (1 P.).

1.4 Förster - Implementierung (3 P.)

Vervollständigen Sie das Gerüst unten so, dass der folgende auskommentierte und mit A markierte Code in *Forest* den Förster Bäume zählen lässt:

```
1 Woodman w = new Woodman();
2 f.workOnTrees(w);
```

Der Förster soll dabei in diesem Beispiel folgende Ausgabe erzeugen:

```
tree #1
tree #2
```

```
1  public class Woodman
2
3
4
5
6
7
8
9
10
11
12
13
14    public Woodman() {
15      super();
16    }
17
18    //alias for System.out.println to save some typing
19    public static void p(String s) {
20      System.out.println(s);
21    }
22  }
```

1.5 Fortgeschrittener Förster (5 P.)

Ein fortgeschrittener Förster soll nun alle Baumarten getrennt zählen. Die Ausgabe im Beispiel sollte also sein:

```
mango tree #0 peartree #1
mango tree #1 peartree #1
```

Unten ist ein Teil der Klasse *AdvancedWoodman* angegeben. Die Implementierung der Methoden *workOnMangoTree* und *workOnPearTree* sind so ähnlich wie Ihre in der vorigen Aufgabe und daher nicht nochmal angegeben.

```
1 public class AdvancedWoodman extends Woodman {
2   public void workOnMangoTree(MangoTree t) {
3     // siehe oben / see above
4   }
5
6   public void workOnPearTree(PearTree t) {
7     // siehe oben / see above
8   }
9 }
```

1. Wie müssen Sie die Klassen MangoTree und PearTree ändern, damit die entsprechende Methode in *AdvancedWoodman* aufgerufen wird? Es reicht die Änderung exemplarisch für eine von beiden Klassen zu beschreiben (1 P.).

2. Wie müssen Sie die Klasse TreeWorker ändern, damit dieser Aufruf kompiliert (1 P.)?

3. Welche der existierenden Klassen müssen sonst noch geändert werden und warum (2 P.)?

   Beschreiben Sie die Änderungen im Vergleich zur Oberklasse bzw. bisherigen Version, etwa als

   `Klasse MangoTree, Konstruktor: Überschreiben mit System.out.println("neu");`

4. Ist das Design geeignet, wenn Sie annehmen müssen, dass in Zukunft viele neue Baumarten hinzukommen (1 P.)?

2 Testen mit JUnit (11 P.)

Bitte implementieren Sie die folgenden Testfälle für Methoden aus einer Bibliothek *StatLib*.
Wichtiger Hinweis: Alle Methoden der Klasse *StatLib* sind *Klassenmethoden* (`static`).

2.1 Setup-Methode (2 P.)

Um die Funktionen der Bibliothek sinnvoll verwenden zu können, ist *einmalig* die Klassenmethode *StatLib.initRand()* aufzurufen. Beachten Sie, dass der Aufruf insgesamt nur einmal erfolgen soll, *nicht* einmal pro durchgeführtem Test!
Geben Sie eine mit JUnit-Annotationen versehene Methode **public static void** init () an, die diese einmalige Initialisierung durchführt.

2.2 Normalfall (2 P.)

Geben Sie einen JUnit-Test **public void** averageTest() an, der überprüft, ob die Klassenmethode *double avg(int[] a)* korrekt arbeitet. Diese Methode berechnet den Durchschnittswert der Zahlen in einem Array. Verwenden Sie dazu *mindestens zwei* Testeingaben!

2.3 Behandlung von Exceptions (3 P.)

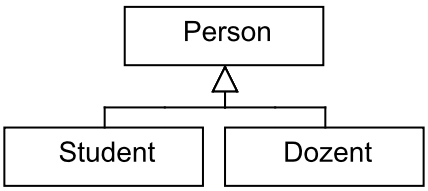
Geben Sie einen JUnit-Test **public void** avgNull() an, der überprüft, ob die Klassenmethode *double avg(int[] a)* bei Aufruf mit **null** eine *IllegalArgumentException* auslöst. Nur bei dem Auftreten dieser Exception soll der Test erfolgreich sein.

2.4 Umgang mit zeitintensiven Funktionen (4 P.)

Geben Sie einen JUnit-Test **public void** ackTest() an, der überprüft, ob die Klassenmethode *long ack(long n, long m)* korrekt arbeitet. Diese Methode berechnet die Ackermann-Funktion (siehe Übungsblatt 6), die sehr komplex zu berechnen ist. Für *ack(3, 4)* lautet das Ergebnis 125, für *ack(3, 10)* ist es 8189.
Um den Test zu bestehen, müssen beide Ergebnisse korrekt *innerhalb von 250 ms* berechnet worden sein. Bei Überschreitung dieser Zeit soll der Test automatisch fehlschlagen.

3 Generische Datentypen (6 P.)

In dieser Aufgabe wird die generische Datenstruktur `Vector<E>` genutzt. Dabei beschränken wir uns auf die Methode **void** add(E elem), die ein Element vom Typ E in den Vector einfügt. Zusätzlich betrachten wir eine Typhierarchie bestehend aus der Klasse `Person` mit den Unterklassen `Student` und `Dozent`.



1. Deklarieren und initialisieren Sie eine Variable `v` vom Typ `Vector`, so dass darin alle genannten Typen (Person, Student und Dozent) gespeichert werden können. Nutzen Sie einen generischen Typparameter! (2 P.)

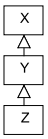
2. Geben Sie Java-Code an, um in den obigen Vector `v` jeweils ein neues Dozenten- und ein Studenten-Objekt einzufügen. Sie dürfen zur Vereinfachung von einem parameterlosen Konstruktor der Klassen Student und Dozent ausgehen (2 P.).

3. Ist die folgende Anweisung zulässig? Begründen Sie kurz Ihre Antwort (2 P.).

```
1 LinkedList<Object> w = new LinkedList<?>(20, 10);
```

4 Collections und Generizität (10 P.)

Wir betrachten eine Typhierarchie mit einer Klasse `X` mit einem Erben `Y`. Von `Y` ist wiederum `Z` abgeleitet; `Z` ist also sozusagen der "Enkel" von `X`.



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 `listC.add(x);` ist 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 genericsTest(List<? super Z> listA,
2                           List<? extends Y> listB,
3                           List<X> listC) {
4    X x = new X();
5    Y y = new Y();
6    Z z = new Z();
7
8    listC.add(x); // * OK
9
10
11   listA.add(x); // *
12
13
14   listA.add(z); // *
15
16
17   listB.add(x); // *
18
19
20   listB.add(y); // *
21
22
23   listC.add(y); // *
24
25
26   listC.add(z); // *
27
28
29   z = listA.get(0); // *
30
31
32   z = listB.get(0); // *
33
34
35   x = listC.get(0); // *
36
37
38   y = listB.get(0); // *
39
40
41 }
```

5 Zweiundvierzig (15 P.)

In dieser Aufgabe sollen Sie Zahlen zwischen 1 und 999.999 in korrekte englische Zahlwörter umwandeln. Das Zahlwort zur Zahl 42 ist *fortytwo*. Das Zahlwort für 876.543 ist *eighthundredseventysixthousandfivehundredfortythree*.
Ihre Klasse soll das unten angegebene Interface `INumToString` implementieren[2]. Dabei soll Ihre Implementierung der Methode `numToString` zu jeder Zahl im oben angegebenen Intervall einen entsprechenden String mit dem Zahlwort zurückgeben.
Hinweis: Auf Kommentare dürfen Sie *in der gesamten Aufgabe verzichten*.

```
1 public interface INumToString {
2   public abstract String numToString(int i);
3 }
```

5.1 Klassengerüst (3 P.)

Ergänzen Sie die Implementierung unten in den Zeilen 1-8, so dass der folgende Aufruf die Ausgabe two erzeugt. Greifen Sie dabei auf die untenstehenden Methoden zu:

```
1 INumToString n = new NumWord();
2 n.numToString(); // Ausgabe: two
```

Sie können zunächst davon ausgehen, dass die Methoden *String thousand(int), String hundred(int)* (auf der folgenden Seite) und *String ten(int)* bereits wie in dem Kommentar beschrieben funktionieren.
Hinweis: Sie brauchen nur den Code in den Zeilen 1-8 "passend" zu vervollständigen.

```
1  public class
2
3
4
5
6
7
8
9    /**
10    * Liefert das Zahlwort fuer Zahlen von 1 bis 99.
11    * Andere Zahlen koennen zu einer Exception fuehren.
12    * Beispiel: ten(23) liefert "twentythree"
13    */
14   private String ten(int n) { /* Code momentan irrelevant */ }
15   /**
16    * Liefert das Zahlwort fuer Zahlen von 1 bis 999.
17    * Andere Zahlen koennen zu einer Exception fuehren.
18    * Beispiel: hundred(123) liefert "onehundredtwentythree"
19    */
20   private String hundred(int n) { /* Code momentan irrelevant */ }
21
22   /**
23    * Liefert das Zahlwort fuer Zahlen von 1 bis 999.999.
24    * Andere Zahlen koennen zu einer Exception fuehren.
25    * Beispiel: thousand(5234) liefert
26    * "fivethousandonehundredtwentythree"
27    */
28   private String thousand(int n) { /* Code momentan irrelevant */ }
29 }
```

[2]Die Realisierung über ein Interface statt einer Klassenmethode–was ebenfalls denkbar gewesen wäre– erleichtert die Realisierung für verschiedene Ausgabesprachen, hier Englisch.

5.2 Repräsentation der Tausender (6 P.)

Geben Sie nun eine Implementierung der Funktion **private** String thousand(**int**) an. Nehmen Sie an, die Funktionen *hundred* und *ten* funktionieren bereits wie im Kommentar angegeben.
Hinweis: Sie können die in Java vorhandenen Operationen / und % (Modulo) verwenden (`14 / 3 = 4, 14 % 3 = 2`).

5.3 Zehner und Einer (6 P.)

Implementieren Sie nun die Funktion **private** String ten(**int**). Da die Zahlwörter für Zahlen kleiner als 20 und die Bezeichnungen der Zehner sehr unregelmäßig gebildet werden, können Sie die Konstanten aus der untenstehenden Klasse SmallNumWords verwenden.
Hinweis: Achten Sie darauf, an welcher Feldposition die einzelnen Werte stehen!

```
1  public final class SmallNumWords {
2    public final static String[] ten = new String[] {
3      "twenty", "thirty", "forty", "fifty",
4      "sixty", "seventy", "eighty", "ninety"
5    };
6
7    public final static String[] small = new String[] {
8      "one", "two", "three", "four", "five",
9      "six", "seven", "eight", "nine", "ten",
10     "eleven", "twelve", "thirteen",
11     "fourten", "fifteen", "sixteen",
12     "seventeen", "eighteen", "nineteen"
13   };
14 }
```

6 Bestimmung des besten DSL-Tarifs (7 P.)

Gegeben sei die folgende Datenstruktur für DSL-Tarife:

```
;; Struktur fuer DSL-Tarife
;; name: string - Name des Tarifs
;; preis: number - Preis pro Monat
(define-struct tarif (name preis))

;; Einige Tarife
;; "Tarif 1": 17.80 Euro pro Monat
(define tarif1 (make-tarif "Tarif 1" 17.80))
(define tarif2 (make-tarif "Tarif 2" 25.95))
(define tarif3 (make-tarif "Tarif 3" 15.95))
(define abzocke (make-tarif "Unvorsichtiger Kunde" 55.00))
(define tarife (list tarif1 tarif2 tarif3))
```

Implementieren Sie eine Funktion `bester-tarif`, die eine Liste von DSL-Tarifen konsumiert und den besten Tarif bestimmt. Dabei soll aus allen Tarifen derjenige mit dem *günstigsten Preis pro Monat* bestimmt werden und als Ergebnis der Name des Tarifs zurückgegeben werden. Sollte es mehr als einen Tarif mit identischen Kosten geben, können Sie einen beliebigen billigsten Tarif zurückgeben.
Vergessen Sie nicht die Angabe des Vertrages (1 P.); auf Beispiel, Zweck und Tests dürfen Sie verzichten!
Hinweis: Der "Standarddarif" (wenn kein anderer Tarif gewählt wurde) ist der Tarif `abzocke`. Das passiert, wenn man (DSL-)Verträge nicht genau liest...

7 Scheme: Gewichtete Summe (15 P.)

Implementieren Sie eine Prozedur *weighted-sum*, die eine Liste von Zahlen konsumiert und eine Zahl als Ergebnis produziert. Das Ergebnis soll dabei die *gewichtete Summe* der Eingabezahlen gemäß folgender Formel sein, wobei l_i für das Element an Position i der Liste steht (mit $1 \leq i \leq (length\ l)$):

$$\text{weighted} - \text{sum}(l) = \sum_{i=1}^{n} i * l_i$$

Die gewichtete Summe einer leeren Liste ist als 0 definiert.
Beispiel: {weighted−sum '(4 7 8)) $= 1 * 4 + 2 * 7 + 3 * 8 = 4 + 14 + 24 = 42$.
Hinweis: Mit der richtigen "Eingebung" ist das Problem sehr einfach und kompakt zu lösen. Generell können Hilfsprozeduren oder Akkumulatoren hilfreich sein.

7.1 Beispiele berechnen (3 P.)

Geben Sie zu jedem der folgenden Beispiel das korrekte Ergebnis an. Auf die Angabe von Zwischenschritten dürfen Sie verzichten.

- (weighted−sum '(13)) =
- (weighted−sum '(7 3)) =
- (weighted−sum (**list** 2 4 5)) =

7.2 Implementierung (12 P.)

Implementieren Sie nun die Prozedur *weighted-sum*. Vergessen Sie nicht die Angabe von *Vertrag (1 P.), Zweck (1 P.), Beispiel (0.5 P.)* und mindestens *zwei* Tests (0.5 P.)!
Hinweis: Für jede *lokale* Hilfsprozedur ist ebenfalls der Vertrag und Zweck anzugeben; bei *nicht-lokalen* Hilfsprozeduren zusätzlich auch ein Beispiel und mindestens zwei Tests.

8 Scheme: Lotto (12 P.)

In dieser Aufgabe werden die Gewinner mit sechs Richtigen im Lotto bestimmt.
Hinweis: Für die Aufgabe dürfen Sie die beiden folgenden Prozeduren nutzen:

```
1  ;; contained?: X (listof X) -> boolean
2  ;; Bestimmt, ob das Element mind. einmal in der Liste enthalten ist
3  ;; Beispiel: (contained? 5 '(1 3 5 5)) = true
4  (define (contained? value alon) ...)
5
6  ;; remove-duplicates: (listof X) -> (listof X)
7  ;; Erzeugt eine neue Liste aus dem Parameter, in der alle mehrfach
8  ;; enthaltenen Elemente nur einmal auftreten.
9  ;; Beispiel: (remove-duplicates '(1 1 1 2) = (list 1 2)
10 (define (remove-duplicates alox) ...)
```

8.1 Abgleich von Listen (2 P.)

Implementieren Sie ein Prädikat `subset?`, das zwei Listen von Zahlen konsumiert und genau dann *true* produziert, wenn alle Zahlen der ersten Liste in der zweiten Liste enthalten sind.
Hinweis: Auf die Angabe von Vertrag, Zweck, Beispiel und Tests dürfen Sie verzichten.

8.2 Validierung eines Lottotips (4 P.)

Implementieren Sie eine Prozedur *valid-tip?*, die eine Liste von Zahlen erhält und genau dann *true* liefert, wenn **alle** folgenden Bedingungen erfüllt sind:

- Die Eingabeliste besteht aus genau 6 Zahlen.
- Es kommen nur die Zahlen von 1 bis 49 vor.
- Keine Zahl kommt mehrfach vor.

Sie können *nicht* unterstellen, dass die Eingabeliste aufsteigend sortiert vorliegt!
Hinweis: Es kann sinnvoll sein, Hilfsprozeduren zu definieren. Bekannte Funktionen wie *length* dürfen benutzt werden.

8.3 Berechnung der Sieger (6 P.)

Implementieren Sie nun eine Prozedur `winner`, die eine Liste von Instanzen von `lotto` sowie eine Liste der sechs gezogenen Gewinnzahlen erhält. `winner` produziert eine Liste von *Namen* der Sieger mit sechs Richtigen.
Beachten Sie, dass weder die von den Spielern getippten Zahlen noch die gezogenen Gewinnzahlen sortiert vorliegen müssen!
Dazu wird die folgende Struktur *lotto* benutzt:

```
;; Repraesentiert einen Lottotipp
;; name: string - Der Name des Spielers
;; tip: (listof number) - Ein Lottotip (6 Zahlen)
(define-struct lotto (name tip))
```

Hinweis: Bitte geben Sie den Vertrag von `winner` an; auf Zweck, Beschreibung und Beispiele sowie Tests dürfen Sie verzichten.
Hinweis: Mit *map, foldl / foldr* und/oder *filter* können Sie einigen Schreibaufwand sparen.

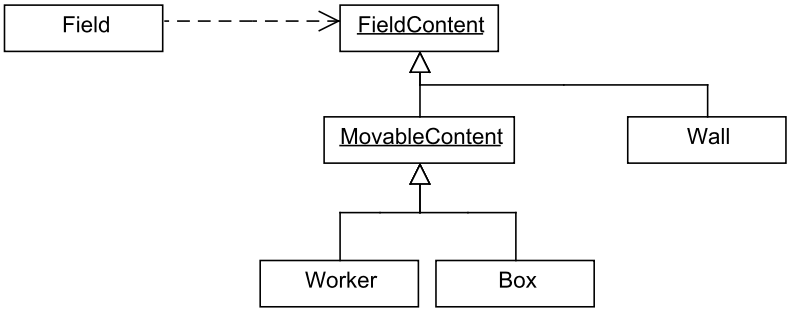
9 Scheme: Listentransformation (6 P.)

Implementieren Sie eine Funktion *transform-matching*, die zwei einstellige Funktionen f und g konsumiert und eine einstellige Ergebnisfunktion produziert. f und g konsumieren jeweils eine Zahl; f produziert ein Ergebnis beliebigen Typs, während g immer *boolean* produziert.
Als Ergebnis soll eine Funktion geliefert werden, die f auf diejenigen Elemente einer Liste anwendet, für die g *true* ergibt. Ihre Funktion soll den unten angegebenen Test erfolgreich passieren. Geben Sie den Vertrag (2P) und ein Beispiel (1P) an. Der Zweck ist bereits angegeben, auf weitere Testfälle können Sie verzichten.

```
;; Test
(check-expect ((transform-matching sqr odd?) '(1 3 6 4 5)) (list 1 9 25))
;; transform-matching wendet f auf alle Elemente einer
;;   Liste an, bei denen g true liefert.
```

10 Sokoban (20 P.) - Nur für Teilnehmer der Sonderregelung

Folgendes UML Diagramm zeigt die in einer Implementierung des Spieleklassikers Sokoban verwendete Modellierung.



Die Implementierung verwendet die klassischen Sokoban-Regeln. Als Erweiterung sollen Wurmlöcher eingefügt werden. Für diese gelten folgende Regeln:

- Wurmlöcher verbinden immer zwei Felder auf dem Spielfeld.
- Wird ein Gegenstand (eine Kiste oder der Arbeiter) auf einem Feld A platziert, das über ein Wurmloch mit dem Feld B verbunden ist, so wird der Gegenstand auf das Feld B platziert, wenn das Feld B frei ist.
- Ist das Feld B nicht frei, so bleibt der Gegenstand auf dem Feld A. Auch wenn später das Feld B frei werden sollte, bleibt der Gegenstand auf dem Feld A. Wurmlöcher treten also immer nur beim Verschieben eines Gegenstands auf ein Wurmloch-Feld in Aktion.

10.1 Re-Design (3 P.)

Ergänzen Sie das UML-Diagramm um die Klasse *WormHole*. Diese Klasse soll die Implementierung für ein per Wurmloch verbundenes Feld enthalten. Beachten Sie, dass ihre Erweiterung nicht dazu führen darf, dass existierende Implementierungen der anderen Klassen nicht mehr funktionieren.
Hinweis: Sie dürfen auch in dem obigen Diagramm zeichnen, solange Ihre Lösung eindeutig erkennbar ist.

10.2 Implementierung I (7 P.)

Die Klasse Field, FieldContent und MovableContent sind bisher folgendermaßen implementiert.

```
1  import java.util.Arrays;
2
3  public class Field {
4    private Field[] neighbors = new Field[4];
5    public FieldContent content;
6
7    public Field(Field[] nb) {
8      neighbors = Arrays.copyOf(nb, nb.length);
9    }
10
11   public Field getNeighbor(char direction) {
12     switch (direction) {
13       case 'U': return neighbors[0];
14       case 'R': return neighbors[1];
15       case 'D': return neighbors[2];
16       case 'L': return neighbors[3];
17     }
18     return null;
19   }
20 }
```

```
1  public abstract class FieldContent {
2     public Field placedOn;
3     public abstract void move(char direction) throws
        CannotMoveException;
4  }
```

```
1  public abstract class MovableContent 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 }
```

Ändern Sie die Implementierung der Klassen Field und MovableContent, indem Sie eine Methode *placeContent* in die Klasse Field einfügen und diese in MovableContent aufrufen, wenn ein Gegenstand auf ein Feld bewegt wird. Überlegen Sie sich geeignete Parameter und eine Implementierung, so dass das normale Sokoban-Spiel weiter funktioniert. Die Regelerweiterung soll nur durch Ändern der Implementierung dieser Methode in der Klasse `WormHole` realisiert werden.
Hinweis: Geben Sie für die Klasse *Field* nur die Methode *placeContent* mit Kommentar `// in Field.java` an.

10.3 Implementierung II (8 P.)

Geben Sie eine Implementierung der Klasse WormHole an. Wie bei der Klasse Field sollen auch bei WormHole alle Felder im Konstruktor gesetzt werden.

10.4 Implementierung von Worker und Wall (2 P.)

Im folgenden sind die Implementierungen von Worker und Wall angegeben.

```
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     super.move(direction);
7   }
8 }
```

```
1 public class Wall extends FieldContent {
2   public void move(char direction) throws CannotMoveException {
3     throw new CannotMoveException();
4   }
5 }
```

Welche Änderungen sind an der Implementierung von Worker und Wall notwendig, damit die neue Sokoban-Variante funktioniert? Beschreiben Sie die Änderungen in einem kurzen Text bzw. begründen Sie, warum an einer Klasse keine Änderungen notwendig sind.
Änderungen an Worker:

Änderungen an Wall: