Universität Regensburg

PD Dr. J. Reischer 29.7.2016

Klausur "ADP" SS 2016

Nachname, Vorname	
Abschluss (BA, MA, FKN etc.)	
Matrikelnummer, Semester	
Versuch (1/2/3)	

Bitte füllen Sie zuerst den Kopf des Angabenblattes aus!

Die Klausur dauert 90 Minuten.

Es sind maximal 50 Punkte zu erreichen.

Es sind keine Hilfsmittel zugelassen.

Die Klausur besteht aus 8 Seiten.

Bitte beantworten Sie alle Fragen direkt auf das Angabenblatt.
Nutzen Sie ggf. die Rückseite und kennzeichnen Sie dies entsprechend.
Eigene Schmierblätter sind nicht erlaubt.

Bei mehreren oder mehrdeutigen Lösungen wird die schlechtere Lösung gewertet. Streichen Sie daher ungültige Lösungen eindeutig durch.

Verwenden Sie nur Java, C# oder Pseudokode für Programmieraufgaben.

Viel Erfolg!

Aufgabe 1: *Konzeptionelle Fragen* (8 Punkte)

Bitte fassen Sie sich bei der Beantwortung der Fragen kurz. Überflüssige Ausführungen und Begründungen werden negativ gewertet.

Aufgabe 1.1: (2 Punkte)

Was unterscheidet abstrakte von nicht-abstrakten Klassen? Wozu können abstrakte Klassen verwendet werden?

Aufgabe 1.2: (2 Punkte)

Können auch einzelne Member einer Klasse abstrakt sein? Wenn ja, welche? Wenn nein, warum nicht?

Aufgabe 1.3: (2 Punkte)

Kann man bei abstrakten Klassen Instanzkonstruktoren definieren und wenn ja/nein, warum (nicht)?

Aufgabe 1.4: (2 Punkte)

Könnte von einer abstrakten Klasse eine weitere abstrakte Klasse abgeleitet werden? Warum (nicht)?

Aufgabe 2: *Iteration Synthese* (8 Punkte)

Schreiben Sie eine Funktion **bool** `CompareMatrices(`**int**`[][] M1, `**int**`[][] M2)`, die zwei ausgefranste 2D-Integer-Arrays (Matrizen) *beliebiger Größe* vergleicht und nur bei *gleicher Größe und gleichem Inhalt* **true** zurückliefert. Beachten Sie *alle* möglichen Fälle für die Parameter `M1` und `M2`, die *jeden erlaubten Wert* annehmen können.

Beispiel: Für `M1 = {{1,2,3},{4,5},{6}}` und `M2 = {{1,2,3},{4,5},{6}}` liefert die Funktion **true**, für `M1 = {{6,5,4},{3,2},{1}}` und `M2 = {{1,2,3},{4,5},{6}}` ergibt sich **false**. Achtung: Es gibt weitere zu beachtende Fälle!

Aufgabe 3: *Objektorientierte Programmierung* (20 Punkte)

Aufgabe 3.1:

Definieren Sie eine Klassenhierarchie aus den unten beschriebenen Klassen für die Familie Bundy. Entscheiden Sie selbst, welche Member welche Modifikationen (Eigenschaften) besitzen. Für nicht-sinnvolle Modifikatoren gibt es Punktabzug.

1) abstrakte Oberklasse `Bundy`: (8 Punkte)
    a. Attribut `SurName` für den gemeinsamen Nachnamen `"Bundy"` aller Bundy-Mitglieder, Attribut `PreName` für den Vornamen und Attribut `Gender` für das Geschlecht jedes einzelnen Bundys;
    b. Instanzkonstruktor, der als Parameter den Vornamen und das Geschlecht erhält und die entsprechenden Attribute initialisiert;
    c. Methode `ToText()`, die Vorname plus Leerzeichen plus Nachname zurückliefert und in den Unterklassen überschrieben wird (z. B. `"Al Bundy"`).
2) zwei Unterklassen `MaleBundy` und `FemaleBundy`, die beide von `Bundy` abgeleitet und strukturell identisch sind (**`true`** = männlich): (8 Punkte)
    a. jeweils Instanzkonstruktor pro Klasse, der jeweils den Vornamen als Parameter erhält und jeweils den Instanzkonstruktor der Oberklasse `Bundy` benutzt, um zugleich Vorname und Geschlecht zu initialisieren (man beachte die Konstruktorparameter der Oberklasse);
    b. jeweils überschriebene Methode `ToText()`, die Vorname plus Leerzeichen plus Nachname mit *vorangestellter* Anrede `"Mr"` (`MaleBundy`) bzw. `"Mrs"` (`FemaleBundy`) zurückgibt und jeweils hierzu *die passende Methode der Oberklasse* benutzt (z. B. `"Mr Al Bundy"` bzw. `"Mrs Peg Bundy"`).

Aufgabe 3.2: Schreiben Sie eine Methode `Bundy[] CreateBundies()`, die vier Instanzen für die Mitglieder der Familie `Bundy` erzeugt und als Array entsprechender Größe zurückgibt. Die männlichen Familienmitglieder sind Al und Bud Bundy, die weiblichen Peg und Kelly Bundy. (4 Punkte)

Aufgabe 4: *Rekursion Analyse* (7 Punkte)

Gegeben sind folgende rekursive Funktionen F1 und F2:

```
// Rekursive Funktion F1
int F1(int N)
{
  if (N % 2 != 0) return F1(N-1);
  else return F2(N);
}

// Rekursive Funktion F2
int F2(int N)
{
  N = N / 2 - 1;
  if (N <= 0) return N;
  else return F1(N);
}
```

4.1: Welche Ergebnisse liefert die Funktion F1 für die Aufrufe N = 0 bis 5 zurück? Tragen Sie die Werte für F1 in unten stehende Tabelle ein (2 Versuche, Fehlversuche bitte deutlich durchstreichen). (3 Punkte)

N	0	1	2	3	4	5
F1(N)						
F1(N)						

4.2: Welchen Komplexitätsgrad bezüglich des Laufzeitverhaltens O(?) weist die Funktion F1 auf (Begründung!)? (3 Punkte)

4.3: Welche Formen der Rekursion treten in F1 auf? Nennen Sie zwei. (1 Punkt)

Aufgabe 5: *Rekursion Synthese* (7 Punkte)

Gegeben ist folgende Definition eines Baumknotens:

```
class TreeNode
{
  private int Cont = 0;
  private TreeNode Left = null;
  private TreeNode Right = null;

  public TreeNode(int _Cont, TreeNode _Left, TreeNode _Right)
  { Cont = _Cont; Left = _Left; Right = _Right; }
  public void SwapTree() { … }                    // Ihre Aufgabe!
}
```

Implementieren Sie obige Methode **void** `SwapTree()`, die *rekursiv* für jeden Knoten den linken und rechten Unterbaum vertauscht, so dass der komplette Baum von links nach rechts gewendet wird (s. Beispiel u.). Beachten Sie, dass der Baum beliebig groß sein kann und nicht unbedingt balanciert ist.

Beispiel: