Fr., 26.2.2016 Informatik Q2 M (Kr)

1. Klausur 13/I (Q2.2)

Dauer: 255+30 Minuten (8:10 bis 12:55 Uhr) *(insges. 255 Punkte)* Name: ___www.r-krell.de_ _

Hilfsmittel: Bereitgestellte Laptops mit Dia nur für Aufg. 1 a1) und 1 b1); Anhang abstrakte Datentypen

* *Achte auf sorgfältige Darstellung mit vollständigem, nachvollziehbarem Lösungsweg!* *
* *Kommentiere deine Programme!* *

1 *(84 Punkte)* UML- und ERD-Diagramme mit Dia: Ein Hotel verspricht gestressten Abiturient(inn)en Sonderpreise für 1-tägige Aufenthalte. Bei einer telefonischen Reservierung bucht ein Angestellter für einen Gast ein Zimmer jeweils für einen Tag (bzw. die Nacht ab dem genanntem Ankunftsdatum).

a) Softwareenginierung für eine Java-Projekt mit UML.

a1) [*24 P*] Zeichne mit Dia ein UML-Diagramm u.a. mit Klassen für ein Zimmer, einen Gast und einen Angestellten. Gast und Angestellter sind Personen, aber Personen ‚an sich' soll es nicht geben (können). Sinnvoll ist außerdem eine Klasse für eine Reservierung, in der festgehalten wird, welcher Angestellte an welchem Buchungstag (Anruftag) für welchen Gast und für welchen Ankunftstag welches Zimmer reserviert hat. Für die Tage soll eine eigene Klasse Datum verwendet werden, die Tag, Monat und Jahr als (private) Ganzzahl-Attribute enthält. Da es mehrere Zimmer, Gäste, Angestellte und Reservierungen gibt, sollen diese von/in einer zentralen Klasse „Hotel" verwaltet werden. Skizziere in jeder Klasse (außer der „Hotel"-Klasse) ein bis zwei wesentliche Attribute und notiere bei Reservierung einen Konstruktor und bei Datum die beiden in a2) erwähnten Methoden (mit Sichtbarkeit, Parametern und Rückgabewert). Zeichne Assoziationen, Aggregationen oder auch ggf. Kompositionen mit Multiplizitäten ein.

a2) [*9 P*] Schreibe nur die Klasse *Datum* mit privaten Attributen (wie in a1)) vollständig in Java, wobei das heutige Datum etwa mit *setze (26,2,2016)* gespeichert/gesetzt und mit *nenneDatum()* als String „26.2.2016" zurückgegeben können werden soll. Erzeuge dann außerhalb der Klasse Datum ein Datums-Objekt *beispiel* mit dem Inhalt 26.2.2016!

a3) [*9 P*] Erläutere die Begriffe Methode, Attribut, Objekt und Klasse allgemein sowie unter Bezug auf a2)!

a4) [*9 P*] Das Hotel hat viele Angestellte. Nenne zwei mögliche Datenstrukturen, in denen die Angestellten innerhalb der „Hotel"-Klasse verwaltet werden können – mit Vor- und Nachteilen von jedem der beiden Speichertypen.

b) Statt in Java können die Daten des Hotels auch in einer Datenbank verwaltet werden.

b1) [*18 P*] Zeichne mit Dia das ERD (Entity-Relationship-Diagramm bzw. -Modell): Angestellte sollen in einer Personal-Tabelle, Gäste in einer Gast-Tabelle gespeichert sein (jetzt gibt es keine gemeinsame Oberklasse oder Ober-Tabelle!). Ein(e) Angestellte(r) nimmt wieder eine Reservierung für den anrufenden Gast vor, um ab dem vereinbarten Ankunftsdatum für eine Nacht ein Zimmer des Hotels zu buchen. Jedes Zimmer gehört zu genau einer der vier Preiskategorien I = Suite, II = Junior Suite, III = Komfortzimmer und IV = Standardzimmer. Gib die Multiplizitäten in mc-Notation an. Als einzige Attribute sollen der Buchungstag und der reservierte Ankunftstag an die zugehörige Beziehung oder Entität gehängt werden.

b2) [*7 P*] Passend zu b1) sollen alle nötigen Tabellenköpfe (Tabellenname mit den wichtigsten bzw. typischen Attributen) notiert werden, wobei Primärschlüssel unterstrichen und Fremdschlüsseln ein Pfeil ↑ vorangestellt werden soll.

b3) Die Hotelleitung bietet ihren Mitarbeitern verschiedene Fortbildungsmaßnahmen an.

Angestellter				
Id	VName	Name	Funktion	..
15	Elena	Fox	Portier	..

Teilnahme	
↑ Mitarbeiter	↑Maßnahme
15	213

Fortbildung		
FNr	Thema	..
213	Servicequalität	..

Gezeichnet sind die Datenbank-Tabellen mit je einer Beispielzeile; tatsächlich enthalten die Tabellen aber jeweils viele Datensätze. Notiere die SQL-Abfragen zu:
(1) [*3 P*] Wie viele Angestellte gibt es in jeder Position? (Ausgabe etwa: 3 Portier, 10 Zimmermädchen, ..)
(2) [*5 P*] Nenne die Ids und die (Nach-)Namen aller Angestellten, die an der Fortbildung „Moderne Touristiktrends“ teilgenommen haben.

2 *(95 Punkte)* Die Gäste sollen im Hotel in einer Reihung (array) *Gast[] gäste = new Gast[1000]* gespeichert sein, wobei die Variable *anzahl* angibt, wie viele Gäste aktuell tatsächlich in der Reihung sind. Die Gäste sollen nach (reserviertem) Ankunftsdatum sortiert werden – wobei das Ankunftsdatum jetzt schon im Gast (und nicht erst aus einem Reservierungs-Objekt) abgefragt werden kann. Weil das Datum aus mehreren Komponenten zusammengesetzt ist, gibt es in der Klasse *Gast* schon die Methoden *getTag()*, *getMonat()* und *getJahr()*, die jeweils ein ganzzahliges Ergebnis liefern.

a) Zunächst soll die Gästereihung nur nach (Ankunfts-)Tagen sortiert werden. Dazu wird nebenstehendes Verfahren vorgeschlagen.

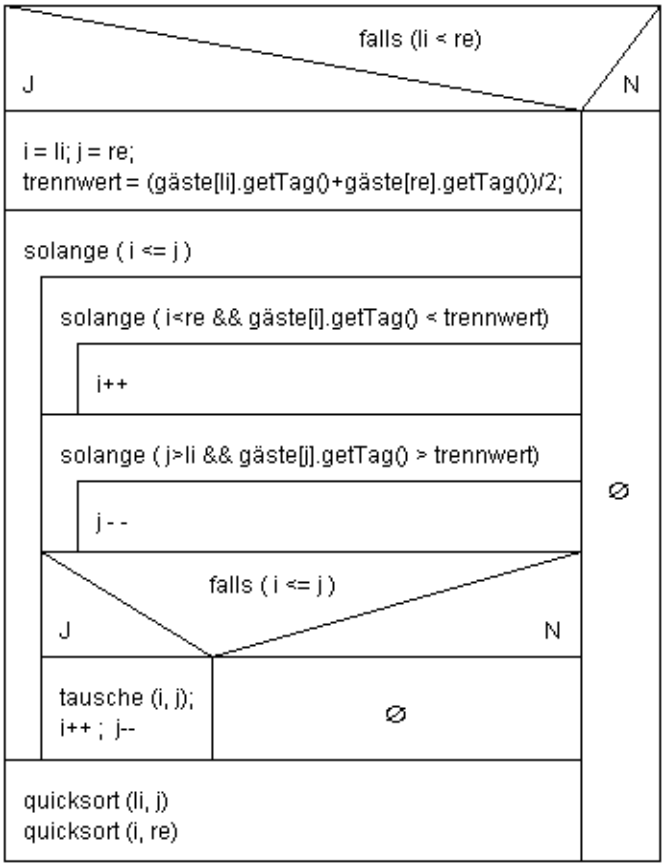
a1) [*17 P*] Führe das Verfahren von Hand durch jeweils für *anzahl=6* und die Tage 7 14 22 6 18 20.
Schreibe dazu den Inhalt der Reihung *gäste* nach jedem Tausch auf und markiere das Ende jedes Durchgangs mit einem Querstrich. An welche(s) Sortierverfahren erinnert der Vorschlag? Und sortiert er immer richtig? Woher kommen die Probleme?

a2) [*15 P*] Auch wenn *sort()* nicht wirklich brauchbar ist, soll sein Aufwand untersucht werden: Notiere mit kurzer Erläuterung die Zahl der Vergleiche und die Zahl der Umspeicherungen (im schlimmsten Fall) bei *anzahl=6* bzw. allgemein bei *anzahl=n* als Term und in O-Notation! Spart man wegen der geringeren Zahl der Durchgänge wesentlich gegenüber dem Sortieren durch Auswahl (selection sort)?

b) Jetzt soll die Sortierung der Gäste-Reihung mit einer Quicksort-Variante durchgeführt werden. Als Trennwert (Pivot-Element) wird aber nicht ein Wert aus der Reihung, sondern der Mittelwert zweier Elemente gewählt: *trennwert = (gäste[li].getTag()+gäste[re].getTag) / 2;*

```
public void sort()  //Vorschlag für Aufg. 2a
{
 for (int durchg = 0; durchg < anzahl/2; durchg++)
 {
  int i1 = durchg;
  int i2 = anzahl-1-durchg;
  for (int i=i1+1; i<i2; i++)
  {
   if (gäste[i].getTag() < gäste[i1].getTag())
   {
     i1 = i;
   }
   else if (gäste[i].getTag() > gäste[i2].getTag())
   {
     i2 = i;
   }
  }
  tausche (durchg, i1);
  tausche (anzahl-1-durchg, i2);
 }
}
```

quicksort (int li , int re)



b1) [*22 P*] Sortiere wieder 7 14 22 6 18 20 von Hand, jetzt mit *quicksort(0,5)*! Notiere außerdem, wie oft dabei Vergleiche der Art *gäste [k].getTag() ? trennwert* und wie viele *gäste*-Umspeicherungen stattgefunden haben (konkrete Zahlen für dieses Beispiel; keine allgemeinen Angaben gefragt).

b2) [*8 P*] Zeige, dass der hier gewählte Trennwert günstiger ist als z.B. *trennwert = gäste[li]*, wenn eine bereits sortierte Gästereihung nochmal sortiert wird. Nenne dazu begründet für beide Pivotelemente den Aufwand von Quicksort in O-Notation für die schon sortierte Reihung in Abhängigkeit von der Anzahl n der Gäste.

c) Die Sortierung nur nach Tagen wie in a) und b) liefert noch keine Sortierung nach dem ganzen Datum. Deshalb soll nach der Sortierung nach dem Tag eine Sortierung der Gäste mit Bubblesort nach dem Monat und danach eine Bubblesort-Sortierung nach dem Jahr erfolgen.

c1) [*17 P*] Schreibe (in Java oder als Struktogramm) die Sortierung nach Monat in der Bubblesort-Version, die unnötige Durchgänge vermeidet.

c2) [*7 P*] Begründe, ob nach dem dreimaligen Sortieren – erst nach Tag mit einem beliebigen (aber funktionierenden) Sortier-Verfahren, dann nach Monat und zum Schluss nach Jahr jeweils mit Bubblesort – tatsächlich die Gäste-Reihung nach Datum sortiert ist! Welche Eigenschaft von Bubblesort ist hier wichtig? Und könnte man die richtige Sortierung auch erreichen, wenn man die Gäste nur einmal mit Bubblesort sortiert, wobei sofort die vollständigen Daten als String-Objekte (in der Form "26.2.2016") mit *compareTo* verglichen werden? (Zusatz: In MySQL werden Daten stattdessen in der Form "2016-02-26" gespeichert. Nenne einen Grund, warum dies sinnvoll ist.)

d) [*9 P*] Mit einer 2-dim. Hilfs-Tabelle für einen Monat (Spalten = Tage von 0 bzw. 1 bis 31, Zeilen = Zimmernummern) könnte vielleicht noch schneller nach Tagen sortiert werden: jeder Gast kommt an die richtige Stelle in die Hilfstabelle; dann wird die Tabelle tage-(=spalten-) weise durchsucht und die Gäste aus allen belegten Zeilen nacheinander in die eindimensionale Reihung *gäste* zurück kopiert. Bei dem Verfahren werden nie zwei Gäste miteinander verglichen. Bestimme die Anzahl der Umspeicherungen bei *n* Gästen und bewerte dieses Verfahren im Vergleich zu den bekannten Sortierverfahren.

3 *(76 Punkte)* Der Hotelier will nun ein Online-Buchungssystem einführen, wo Reservierungswünsche per SMS, Web oder App automatisch in der Reihenfolge ihres Eingangs bearbeitet werden.

a) [*5 P*] Begründe kurz, warum die Wünsche am besten in einem Objekt vom Typ Queue (Schlange) statt in einem Stack (Keller) oder einer List (Liste) verwaltet werden.

b) [*8 P*] Die Reservierungswünsche werden in Java im Objekt *wünsche* vom Typ Queue verwaltet. Zuletzt kam von einem Gast der Wunsch *wunsch267* herein, der mit

*wünsche.enqueue(wunsch267); //**

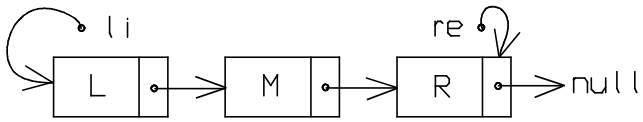
hinten an die Wünsche-Schlange angefügt wurde. Schreibe jetzt ähnlich wie //* alle Aufrufe für folgende Vorgänge:

Das Programm übernimmt den am längsten in der Schlange *wünsche* befindlichen Reservierungswunsch in eine eigene Variable *Object inArbeit* zur Bearbeitung und entfernt ihn aus der Schlange. Kurz später kommen online erst der *wunsch268* und dann der *wunsch269* herein, die gespeichert werden. Erst dann kann das Programm den nächsten Wunsch zur Bearbeitung entnehmen.

c) Die Schlange soll in Java dynamisch mit Knoten realisiert werden.

c1) [*6 P*] Schreibe in Java die Klasse Knoten (Node) mit Konstruktor laut nebenstehendem UML-Diagramm.

c2) Gezeigt sind drei Knoten L, M und R (links, mitte, rechts).



(1) [*3 P*] Untersuche und begründe, ob ein neuer Knoten besser links von L oder rechts von R angefügt wird, d.h. ob für *enqueue* das linke oder das rechte Ende der Schlange günstiger ist (Erklärung in Deutsch reicht).
(2) [*6 P*] Begründe ebenso, ob mit *dequeue* leichter der Knoten L oder der Knoten R entfernt werden kann (beim ursprünglichen Bild -- ohne den neuen Knoten aus (1))

c3) [*20 P*] Schreibe den Anfang der Klasse Queue in Java mit den Methoden *enqueue, dequeue* und *isEmpty* (also ohne *front* – Eigenschaften wie im Anhang!).

d) Die Schlange könnte in Java auch mit/in einer Reihung (array) *Object[] reihe = new Object[200]* realisiert werden. Das allererste Element in eine neue, leere Schlange soll in *reihe[0]* gespeichert werden, das nächste mit *enqueue* eingefügte Element kommt in *reihe[1]*, dann in *reihe[2]* usw., während nach *reihe[199]* wieder in *reihe[0]* eingefügt wird. Bei *dequeue* soll das jeweils älteste Element unbesehen entfernt werden, ohne dass innerhalb der *reihe* Verschiebungen stattfinden.

d1) [*13 P*] Schreibe den Anfang der neuen Klasse *Queue* mit allen Attributen (inkl. Startwerten) und nur den Methoden *dequeue* und *isEmpty*

d2) [*2 P*] Entscheide begründet, ob die 4 Standardmethoden aufwändiger werden würden, wenn das allererste Element in *reihe[199]* kommt, das nächste in *reihe[198]* usw., also die Schlange von rechts nach links wächst.

e) Gästen, die gegen Bezahlung am Plus-Programm teilnehmen, werden verschiedene Vorteile versprochen. Außer einer Flasche Sekt auf dem Zimmer sollen sie auch bei der Buchung bevorzugt werden.

e1) [*4 P*] Entweder könnte (1) anstelle der Schlange aus a) bis d) eine einzige Vorrang-Warteschlange (Priority-Queue) eingerichtet werden, die Plus-Kunden vorrangig behandelt. Oder es könnten (2) zwei Schlangen eingerichtet werden – eine Schlange für normale Gäste und eine Schlange für Plus-Kunden, wobei erst dann (wieder) Wünsche aus der normalen Schlange bearbeitet werden, wenn die Plus-Schlange abgearbeitet wurde bzw. gerade leer ist. Bewerte, welche Lösung dir besser erscheint.

e2) [*9 P*] Schreibe in Java die Methode *public Object nächster()*, die aus den beiden (in der umhüllenden Klasse vorhandenen) Schlangen *normal* und *plus* wie in e1)(2) den nächsten zu bearbeitenden Wunsch holt und *null* liefert, falls gerade beide Schlangen leer sind.