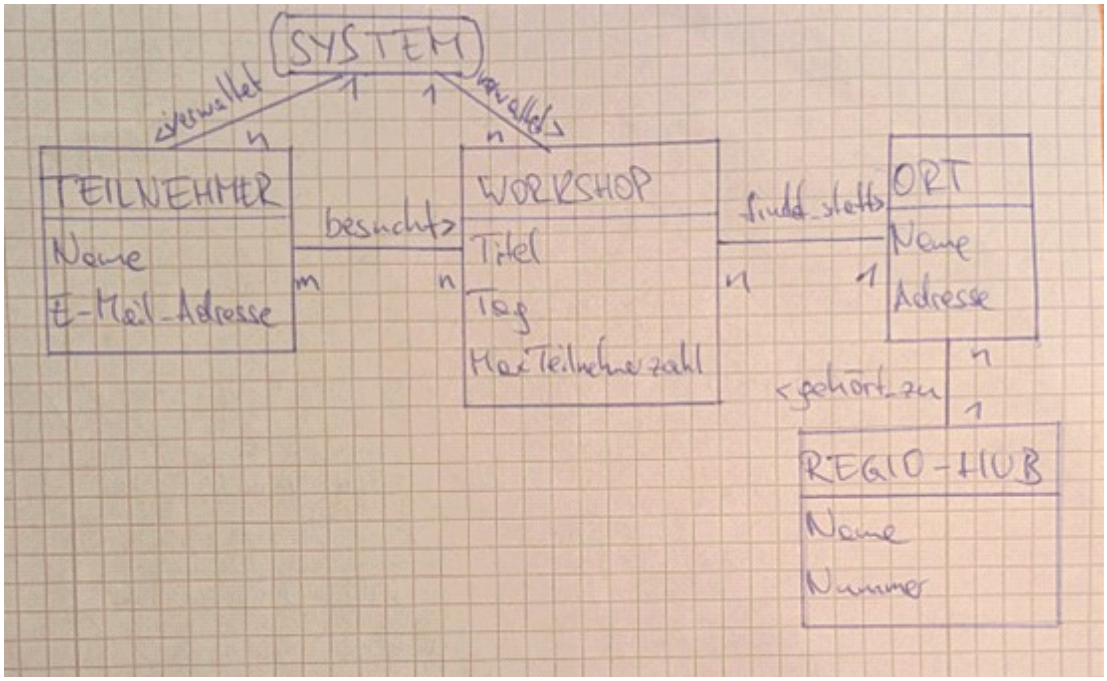
Informatik Abitur Bayern 2024/II - Beispiellösung

Autor:
Naumann
(4,5)

1 

8

2a Ein Feld wird mit einer vordefinierten Länge erzeugt, wohingegen eine Liste eine variable Länge hat. Da die Anzahl der Workshops wahrscheinlich im Vorhinein unbekannt ist (evtl. ist sie abhängig von der Anzahl der Anmeldungen) eignet sich hier eine Liste besser als ein Feld. 2

Die Verwendung eines Feldes, würde das sortierte Einfügen eines neuen Eintrags sehr ineffizient machen. Eine Liste ist wesentlich besser für das dynamische Einfügen eines neuen Elements geeignet.

b Die Klasse WORKSHOP ist verantwortlich für das Verwalten der Daten. Die Struktur wird in den Klassen LISTENELEMENT, KNOTEN, ABSCHLUSS und WORKSHOP definiert. Daraus ergeben sich folgende Vorteile: 3

Durch das modulare Design ist der Code besser wartbar.

Die Klassen, die die Struktur definieren, können für andere Anwendungszwecke wiederverwendet werden.

c In WORKSHOPLISTE: 8

```
void sortiertEinfügen(WORKSHOP workshopNeu) {
  anfang.sortiertEinfügen(workshopNeu);
}
```

In LISTENELEMENT:

```
abstract LISTENELEMENT sortiertEinfügen(WORKSHOP workshop);
```

In KNOTEN:

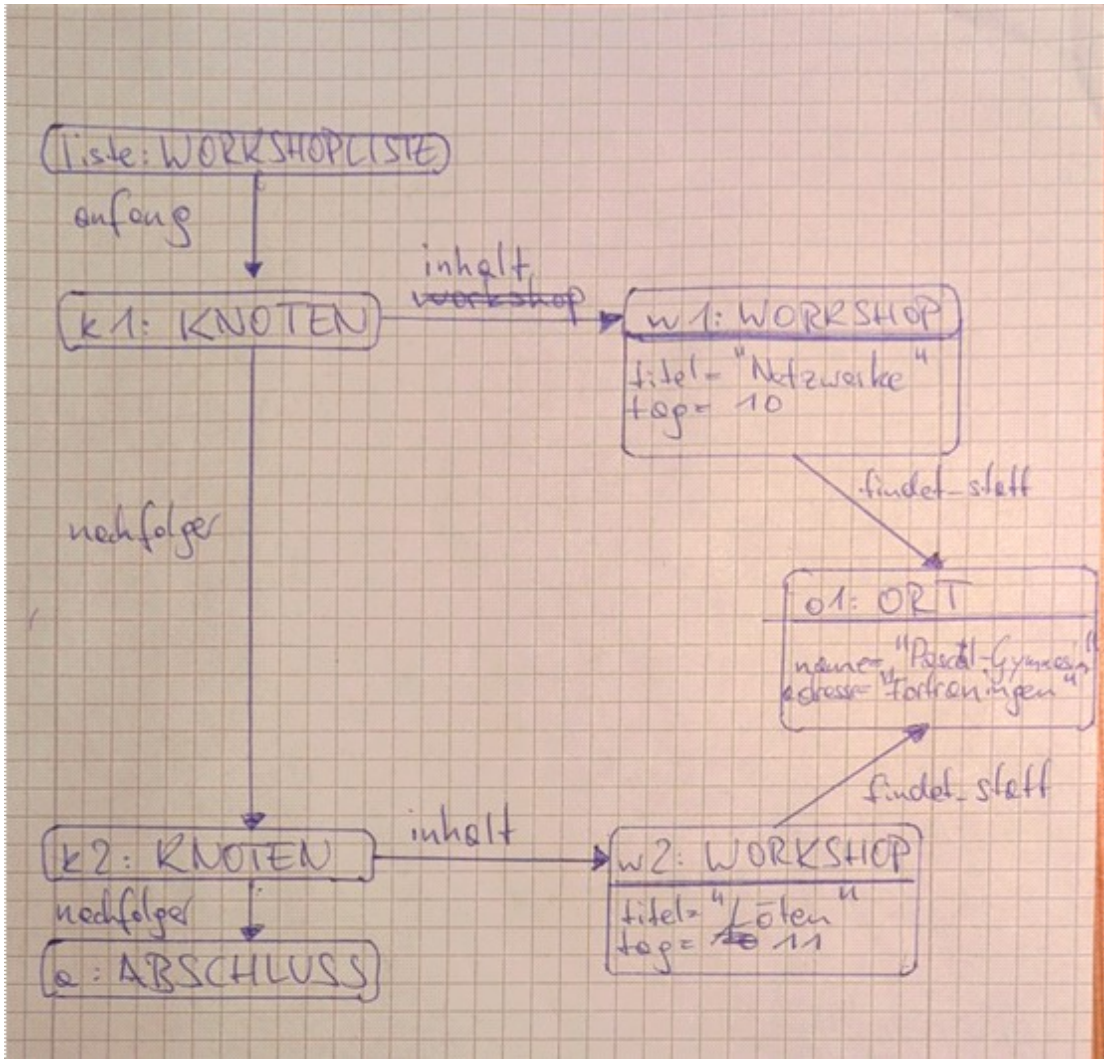
```
public KNOTEN(WORKSHOP inhalt, LISTENELEMENT nachfolger) {

  this.inhalt = inhalt;
```

```
    this.nachfolger = nachfolger;

  }
  LISTENELEMENT sortiertEinfügen(WORKSHOP einWorkshop) {

    if (inhalt.tagGeben() > einWorkshop.tagGeben()) {
      return new Knoten(einWorkshop, this);
    }
    else {
      nachfolger = nachfolger.sortiertEinfügen(einWorkshop);
      return this;
    }
  }
```

In ABSCHLUSS:

```
  LISTENELEMENT sortiertEinfügen(WORKSHOP workshop) {
    return new Knoten(workshop, this);
  }
```

d



5

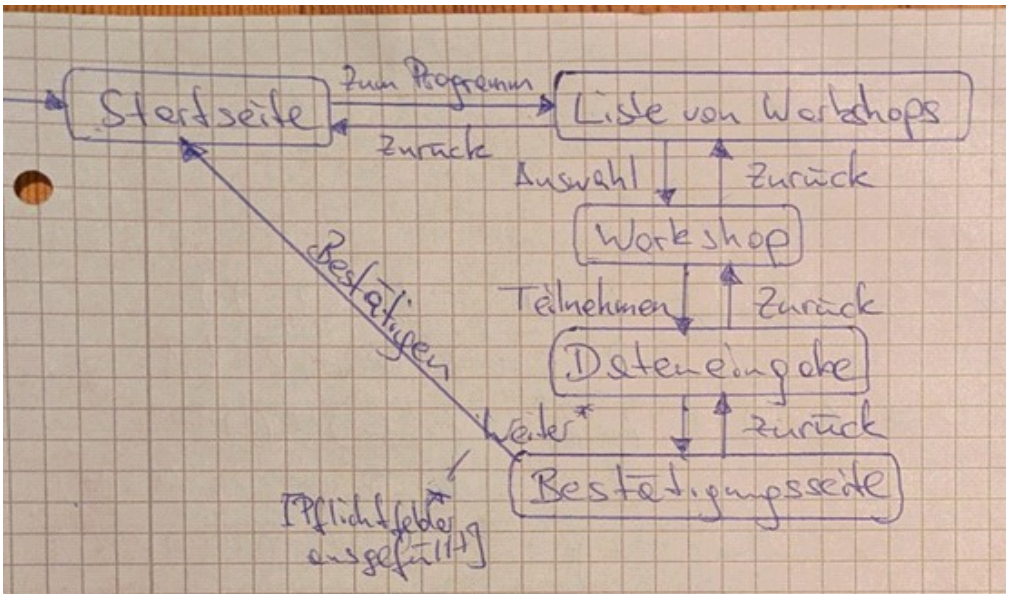
3a **Anforderungsanalyse**: Die Anforderungen an die Software werden ermittelt. 4

Entwurf: Das System wird mithilfe von Modellen modelliert. Hierfür eignen sich Klassen- und Objektdiagramme.

Implementierung: Der Entwurf wird in einer Programmiersprache praktisch ausprogrammiert.

Test: Zuerst erfolgt das Testen der einzelnen Module. Danach erfolgt die Durchführung eines Systemtests, bei dem das gesamte System getestet wird.

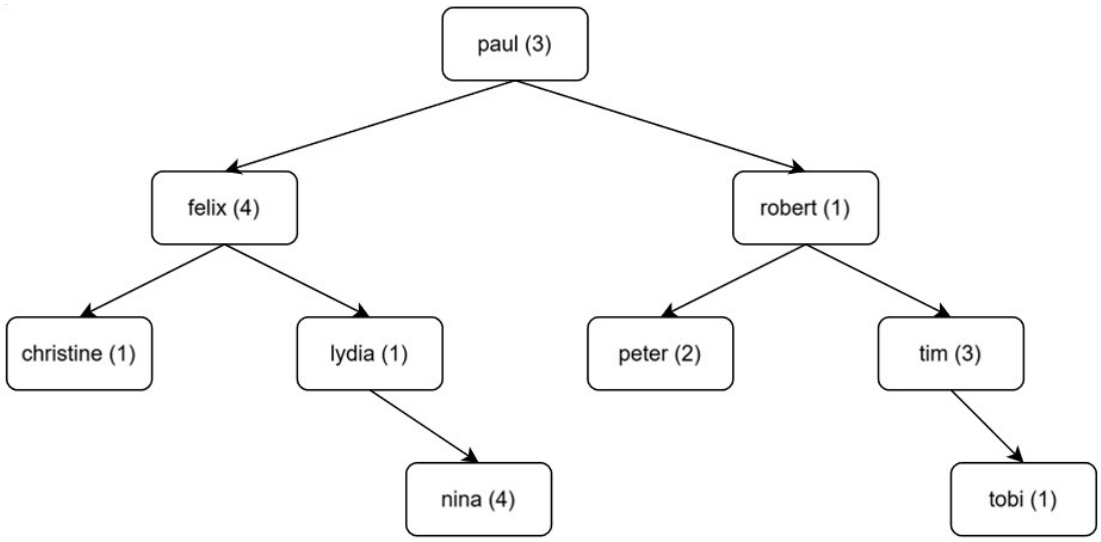
b



6

4a Damit der Binärbaum über möglichst wenige Ebenen verfügt, müssen die Elemente lexikographisch „aus der Mitte“ eingefügt werden, d.h. für jeden Teilbaum das mittlere Element der verbleibenden Elemente. 5

Damit ergibt sich zum Beispiel folgende Einfügereihenfolge:
paul (3) > felix (4) > robert (1) > christine (1) > lydia (1) > peter (2) > tim (3) > nina (4) > tobi (1)
Grafische Darstellung des Baumes:



b 3

Für Binärbäume mit k Elementen der Tiefe n gilt: $2^n - 1 \geq k$. Durch Einsetzen des gegebenen Wertes für k erhalten wir die folgende Ungleichung: $2^n - 1 \geq 987$. Umstellen nach n ergibt $n \geq \log_2(987 + 1) \approx 9{,}95$, also werden mindestens 10 Ebenen benötigt. Diese 10 Ebenen können maximal $2^{10} - 1 = 1023$ Elemente aufnehmen, es können also bestenfalls noch $1023 - 987 = 36$ weitere Elemente aufgenommen werden.

c Die Methode anzahlAnmeldungenErhöhen(teilnehmerEmail) wird in der Klasse REGIOTEILNEHMERBAUM implementiert. Hier wird der Aufruf lediglich an die Wurzel des Baumes 11

weitergegeben:

```
void anzahlAnmeldungenErhöhen(String teilnehmerEmail) {
                wurzel = wurzel.anzahlAnmeldungenErhöhen(teilnehmerEmail);
}
```

Dem Entwurfsmuster Kompositum folgend haben alle Elemente des Baumes einen gemeinsamen Typ, in diesem Fall realisiert durch die abstrakte Klasse BAUMELEMENT. Somit muss die neue Methode auch in definiert sein:

```
abstract BAUMELEMENT anzahlAnmeldungenErhöhen(String email);
```

Diese Methode muss nun in den entsprechenden Implementierungen der verschiedenen Ausprägungen der Klasse BAUMELEMENT konkretisiert werden. In der Klasse KNOTEN muss zudem der Konstruktor definiert werden:

```
KNOTEN(REGIOTEILNEHMER teilnehmer, BAUMELEMENT links, BAUMELEMENT rechts) {
       this.inhalt = teilnehmer;
       this.lnf = links;
       this.rnf = rechts;
}

BAUMELEMENT anzahlAnmeldungenErhöhen(String email) {
       //Vergleich mit E-Mail dieses Knotens
       int vgl = inhalt.EmailGeben().vergleichenMit(email);
       //vgl == 0 à E-Mail stimmt überein à Anzahl erhöhen
       if (vgl == 0) {
          inhalt.anzahlAnmeldungenSetzen(inhalt.anzahlAnmeldungenGeben()
+1);
       }
       // Wenn vgl != 0, rekursiv an linken oder rechten Nachbarn
else if (vgl > 0) {
          lnf = lnf.anzahlAnmeldungenErhöhen(email);
       } else {
          rnf = rnf.anzahlAnmeldungenErhöhen(email);
       }
       // Rückgabe: Selbstreferenz
       return this;
}
```

In der Klasse ABSCHLUSS (oder BLATT) muss nun noch die Logik für den Fall, dass ein Element neu einsortiert werden soll, implementiert werden. Fälle, wo die betreffende E-Mail bereits im Baum vorhanden sind, werden vorher abgefangen und können hier nicht ankommen.

```
BAUMELEMENT anzahlAnmeldungenErhöhen(String email) {
    return new KNOTEN(new REGIOTEILNEHMER(email, 1), this, new
ABSCHLUSS());
}
```

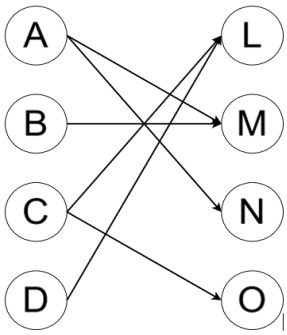
d Die Methode *m2(zahl)*, die aus der Methode *m1(zahl)* aufgerufen wird, traversiert den Baum rekursiv und gibt diejenigen E-Mail-Adressen aus, für die mindestens *zahl* Anmeldungen verbucht sind. Für den Baum aus 4a) sind dies: 4

felix@abc.de; nina@lnet.de; paul@abc.de; tim@lnet.de

Insbesondere erfolgt die Ausgabe stets beginnend mit dem linken Nachbarknoten, daher ist sie für den sortierten Binärbaum alphabetisch sortiert.

•

5a Es ergibt sich der nebenstehende Graph. Der Graph ist gerichtet und ungewichtet/unbewertet. Die durch den Graphen dargestellte Sachsituation ist eine Abbildung zwischen zwei disjunkten Mengen, konkret von Teilnehmenden auf Workshopplätze, innerhalb dieser Mengen gibt es keine Kanten. Dadurch fallen der obere linke und der untere rechte Teil weg. Zudem ist der Graph gerichtet mit einer vorgegebenen Abbildungsrichtung, nämlich Teilnehmende > Workshopplätze, damit fällt auch der linke untere Bereich aus. (Ein solcher Graph heißt *bipartit* mit den Partitionsklassen Teilnehmende und Workshopplätze.) 4

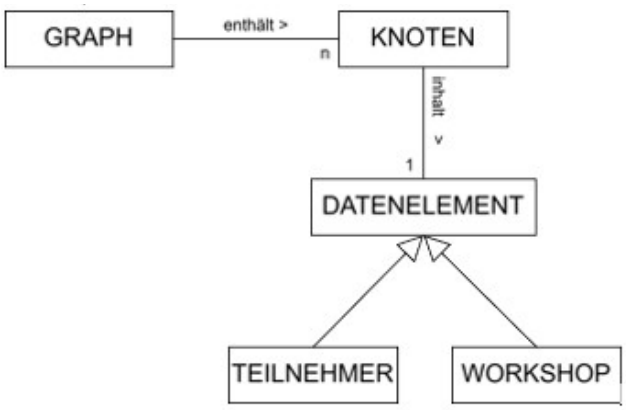


b Eine mögliche Zuordnung (und auch die einzige, bei der alle Teilnehmenden einen Workshopplatz erhalten) ist: 2

A → N B → M C → O D → L

Die Strategie hierzu ist, zuerst die Teilnehmenden zu „befriedigen“, die nur einen Workshop gewählt haben (B und D), anschließend bleibt für A und C jeweils nur noch eine Möglichkeit übrig.

c Es müssen zwei neue Klassen TEILNEHMER und WORKSHOP eingeführt werden, die von der bestehenden Klasse DATENELEMENT erben. 2



d Der Aufruf von *zuordnen(0)* iteriert über die erste Zeile der Adjazenzmatrix. Es wird direkt der erste Eintrag ungleich *falsch* bei M zunächst vorgemerkt und direkt zugeordnet, da in *zuordnung* noch keine Einträge existieren. Damit sind die jeweiligen Belegungen: 8

- vorgemerkt[5] = **wahr**
- zuordnung[5] = **0**
- Rückgabe: **wahr**

Nun wird *zuordnen(1)* aufgerufen. Der erste Eintrag ungleich *falsch* ist wiederum bei M, hier ist nun allerdings bereits eine Zuordnung eingetragen (*zuordnung[5] = 0*), daher wird der sonst-Teil der bedingten Anweisung ausgeführt. Es wird der aktuelle Eintrag neu zugeordnet (Aufruf in der Bedingung: *zuordnen(zuordnung[i])*). Dies gelingt, da Teilnehmerin A sich für zwei Workshops gemeldet hatte, und somit sind die folgenden Belegungen nach Abschluss der Methode gegeben:

- vorgemerkt[5] = **wahr**

- vorgemerkt[6] = **wahr**
- zuordnung[5] = **1**
- zuordnung[6] = **0**
- Rückgabe: **wahr**

e Die Methode *allesZuordnen()* muss *zuordnen(index)* für alle Indices (also alle Knoten) aufrufen. Eine mögliche Implementierung wäre wie folgt: 5

```
void allesZuordnen() {
	for (int index = 0; index < anzahlKnoten; index++) {
		// Einträge in vorgemerkt zurücksetzen
		for (int i=0; i < anzahlKnoten; i++) {
			vorgemerkt[i] = false;
		}
		zuordnen(index);
	}
}
```

80