TENTAMEN

Programmeren 1

vakcode: 213500
datum: 2 november 2009
tijd: 13:45–17:15 uur

Algemeen

- Bij dit tentamen mag gebruik worden gemaakt van het Niño en Hosch boek, de Programmeren 1 handleiding en een kopie van de collegesheets zonder aantekeningen.
- Dit tentamen bestaat uit 4 opgaven, waarvoor in totaal 100 + 5 bonus punten behaald kunnen worden. Het minimale aantal punten per opgave bedraagt 0. Het cijfer is het aantal punten gedeeld door 10, afgerond tot een geheel getal, plus 1 (en maximaal 10).
- Bij de opdrachten in dit tentamen hoeven *géén* pre- en postcondities te worden gegeven, tenzij *expliciet* anders vermeld. Neem wel commentaar op waar dat nuttig is voor het begrijpen van uw oplossing.

Opgave 1 (20 punten)

In een Java programma willen we adressen representeren. Een adres bestaat uit de *straat* (een combinatie van naam en huisnummer), de *postcode* en de *plaats*. In dit programma worden deze drie delen elk door een `String` gerepresenteerd. Een mogelijke oplossing is zo'n adres met een enkele `String` te representeren, waarin de drie sub-strings (straat, postcode en plaats) achter elkaar staan, gescheiden door een regelovergang, in Java weergegeven door het teken `'\n'`. Een vooorbeeld hiervan is de `String`

```
"Paal 8\n8881 HB\nTerschelling"
```

die een adres op deze manier representeert. Als dit adres geprint wordt ziet het er uit als

```
Paal 8
8881 HB
Terschelling
```

a. *(5 punten)* Bedenk twee *andere* manieren om een adres in Java te representeren, die fundamenteel van elkaar en van de hierboven geschetste oplossing verschillen. Bijvoorbeeld, het gebruik van een ander scheidingsteken is *niet* fundamenteel verschillend van de oplossing hierboven. Geef voor elk van deze twee manieren aan hoe het adres voorbeeld (`"Paal␣8..."`) *gerepresenteerd en aangemaakt* kan worden.

b. *(6 punten)* Geef voor elk van de drie oplossingen (dus óók de oplossing die hierboven is gegeven!) een klassemethode `getStraat` die, gegeven een adres in het bedachte type, het "straat"-deel als `String` waarde oplevert. Met het adres voorbeeld hierboven levert deze methode de `String` waarde `"Paal␣8"` op. *Hint*: In het geval van de oplossing hierboven heeft deze methode de volgende signatuur

```
public static String getStraat(String adres)
```

c. *(9 punten)* Geef voor elk van de drie oplossingen (dus óók de oplossing die hierboven is gegeven!) een methode `equals` die, gegeven twee parameters in het bedachte type, aangeeft of de twee hetzelfde adres representeren. Twee adressen zijn hetzelfde als hun straat, postcode en plaats dezelfde `String` *waarden* hebben (d.w.z., de inhoud is gelijk).

Hint: De volgende methoden zouden in deze opgave (vragen b.ẽn c.) van pas kunnen komen:

- `int indexOf(char c)`: levert de index van teken `c` binnen de `String` op, of -1 als `c` niet in de `String` zit;
- `int indexOf(char c, int vanaf)`: levert de index van het eerste teken `c` vanaf positie `vanaf` op, of -1 als `c` niet in de `String` zit;
- `String substring(int begin, int eind)`: levert het deel van de `String` van positie `begin` tot (en niet met) `eind` op.

Opgave 2 (30 punten)

De standaard Java-interface `Comparable` heeft de volgende definitie:

```
public interface Comparable<T> {
    /**
     * Compares this object with the specified object for order.
     * @param   o the Object to be compared.
     * @return  a negative integer, zero, or a positive integer as this object
     *    is less than, equal to, or greater than the specified object.
     */
    public int compareTo(T o);
}
```

`Comparable` wordt gebruikt om op een generieke manier lijsten te kunnen sorteren en doorzoeken.

a. (*4 punten*) Leg uit wat de term `T` in de definitie van `Comparable` voorstelt. Wat is het belangrijkste voordeel van het gebruik van de `Comparable` interface?

b. (*7 punten*) Schrijf een interface `Tijdstip` om tijdstippen op de dag te representeren, van `0:00` uur tot `23:59` uur. `Tijdstip` dient `Comparable<Tijdstip>` uit te breiden met de volgende elementen:

- Een constante `UUR_PER_DAG`, die het aantal uren in een etmaal weergeeft;
- Een constante `MIN_PER_UUR`, die het aantal minuten in een uur weergeeft;
- Een methode `getUur()`, die het uur oplevert (bijvoorbeeld `12` voor het tijdstip `12:01`);
- Een methode `getMinuut()`, die het aantal minuten (vanaf het hele uur) oplevert (bijvoorbeeld `1` voor het tijdstip `12:01`);
- Een methode `getTotaal()`, die het totaal aantal minuten vanaf het begin van de dag oplevert (bijvoorbeeld `721` voor het tijdstip `12:01`).
- Een methode `later (int aantal)`, die gegeven een aantal minuten (meegegeven als parameter) een nieuw `Tijdstip` oplevert dat zoveel minuten later ligt. Als dit nieuwe `Tijdstip` echter niet bestaat omdat dat later zou worden dan `23:59` moet de methode het resultaat `null` opleveren.

Voorzie deze methoden van *pre- en postcondities* die hun werking zo nauwkeurig mogelijk beschrijven.

c. (*7 punten*) Schrijf een klasse `Kloktijd` die `Tijdstip` implementeert. Geef de klasse `Kloktijd` twee constructoren `Kloktijd(int uur, int minuten)` en `Kloktijd(int totaalMinuten)` die uitgaan, respectievelijk van uur- en minuut-waarden en een gegeven totaal aantal minuten. Voorzie de klasse van *klasseinvarianten* en de constructoren van *pre- en postcondities* die passen bij de specificatie in de interface. Vergeet interface `Comparable` niet in deze vraag!

d. (*6 punten*) Schrijf een klasse `Periode` die een subklasse is van `Kloktijd` (`Kloktijd` *uibreidt*), maar daarnaast ook de *duur* van een periode (een positief getal in minuten) bijhoudt. Het `Tijdstip` zelf representeert de begintijd van de periode. De eindtijd van de periode, die dus een aantal minuten gelijk aan *duur* later ligt dan de begintijd, moet een geldig tijdstip zijn.

Geef de klasse *Periode* de volgende constructoren en methoden:

- Een constructor die een `Tijdstip` en een duur meekrijgt;
- Een methode `getBegin()` die het begintijdstip van de periode oplevert.
- Een methode `getEind()` die het eindtijdstip van de periode oplevert.
- Een methode `getDuur()` die de duur oplevert.

e. (*6 punten*) Overschrijf `compareTo()` in `Periode` zodat een periode volgens `compareTo` 'kleiner is dan de andere' alleen als deze periode vóór een andere ligt, d.w.z., de *eindtijd* van de eerste periode kleiner of gelijk is aan de *begintijd* van de andere (de periodes overlappen niet!). Als de periodes overlappen levert `compareTo()` de waarde `0` op. Verder, als een `Periode` met een `Tijdstip` vergeleken wordt, moet `compareTo` zich gedragen als de `compareTo()` methode van de superklasse van `Periode` (`Kloktijd` in dit geval).

Opgave 3 (25 punten)

In deze en de volgende opgave specificeren en implementeren we (een gedeelte van) een eenvoudige makelaarsapplicatie. Een makelaar heeft panden in de verkoop en hij kan zelf bij andere makelaars panden aankopen. Er zijn verschillende soorten panden, zoals, bijvoorbeeld, appartementen, villas, tussenwoningen en hoekwoningen. In de makelaarsapplicatie hebben al deze panden één abstracte superklasse, de klasse `Pand`. Een pand kan maar bij één makelaar tegelijk in de verkoop zijn. Als een makelaar een pand in de verkoop heeft, dan kunnen andere makelaars zich bij de makelaar aanmelden als geïnteresseerden voor dat pand. Een verkopende makelaar kan maar met één geïnteresseerde makelaar tegelijk in onderhandeling zijn. De volgorde van aanmelden bij de verkopende makelaar is hierbij belangrijk: wie-het-eerst-komt, die-het-eerst-maalt. Als de verkopende makelaar niet met de eerste geïnteresseerde makelaar tot overeenstemming kan komen, dan is de tweede makelaar aan de beurt. Een makelaar wordt gerepresenteerd door de klasse `Makelaar`.

Wanneer een pand in de verkoop komt, dan stellen de eigenaar en de verkopende makelaar samen een vraagprijs op; dit is het bedrag waarvoor ze het pand willen verkopen. Tijdens de onderhandeling doet de geïnteresseerde makelaar een tegenbod, waarvoor hij het pand wel zou willen kopen. Als de verkopende partij het tegenbod te laag vindt, maar wel verder wil onderhandelen kan hij de vraagprijs verlagen. Dit afdingen gaat net zolang door totdat de verkopende en kopende partij het over de prijs eens zijn. Maar het komt natuurlijk ook voor dat de beide partijen niet tot overeenstemming kunnen komen. Hieronder volgt allereerst de partiële definitie van de klasse `Makelaar`.

```
public class Makelaar {
  private String naam;        // naam van de Makelaar
  private List<Pand> tekoop; // panden in de verkoop
  public Makelaar(String naam) {
    this.naam = naam;
    this.tekoop = new ArrayList<Pand>();
  }
  public String getNaam() { return naam; }
  public List<Pand> getPanden() { return tekoop; }
  /**
   * Bepaalt de gemiddelde vraagprijs van de panden die deze
   * Makelaar in de verkoop heeft.
   * @require ! this.getPanden().isEmpty()
   * @ensure result == gemiddelde vraagprijs van de
   *                   panden in this.getPanden()
   */
  public double gemiddeldeVraagPrijs() {
    // Te programmeren
  }
  /**
   * Breng een bod van bedrag euros uit op een pand.
```

```
    * @require pand != null && bedrag >= 0.0
    * @ensure als !this.getPanden().contains(pand)
    *              && pand.getVerkoper() == this
    *              && pand.getLaatsteBod() == bedrag
    *         dan result
    *         anders ! result
    * @param pand het Pand-object waarop een bod wordt gedaan.
    * @param bedrag het bod dat wordt uitgebracht
    */
   public boolean doeBod(Pand pand, double bedrag) {
     // Te programmeren
   }
}
```

Zoals u kunt zien worden in de klasse `Makelaar` alleen de naam van de makelaar en de lijst van `Pand`-objecten die de makelaar in de verkoop heeft bijgehouden. Hieronder volgt de eveneens partiële definitie van de abstracte klasse `Pand`.

```
public abstract class Pand {
  private String adres;
  private String postcode;
  private Date tekoopPer;
  private double vraagprijs;
  private double laatsteBod;
  private Makelaar verkoper;
  private List<Makelaar> geinteresseerden;
  // Constructor
  protected Pand(String adres, String postcode, Date tekoopPer,
      double vraagprijs, Makelaar verkoper) {
    this.adres = adres;
    this.postcode = postcode;
    this.tekoopPer = tekoopPer;
    this.vraagprijs = vraagprijs;
    this.verkoper = verkoper;
    this.laatsteBod = 0.0;
    this.geinteresseerden = new ArrayList<Makelaar>();
  }
  // De gebruikelijke get-queries.
  public String getAdres() { return adres; }
  public String getPostcode() { return postcode; }
  public Date getTeKoopDatum() { return tekoopPer; }
  public double getVraagPrijs() { return vraagprijs; }
  public double getLaatsteBod() { return laatsteBod; }
  public Makelaar getVerkoper() { return verkoper; }
  public List<Makelaar> getGeinteresseerden() { return geinteresseerden; }
  public boolean isHuidigeKoper(Makelaar makelaar) {
  // Te implementeren
  }
  public void voegKoperToe(Makelaar makelaar) {
  // Te implementeren
  }
  public void verwijderKoper(Makelaar makelaar) {
  // Te implementeren
  }
  public boolean doeBod(Makelaar makelaar, double bod) {
  // Te implementeren
  }
```

De meeste instantievariabelen van `Pand`, de constructor en de `get`-methoden spreken voor zich. De

variabele `tekoopPer` bevat de datum (`Date` object van het `java.util` package) waarop het `Pand` in de verkoop is gegaan. De variabele `laatsteBod` bevat het laatste bod van de huidige (potentiële) koper. De variabele `geinteresseerden` bevat de lijst van geinteresseerde `Makelaar`-objecten; de `Makelaar` die als eerst aan de beurt is, staat vooraan in de lijst. Merk op dat de klasse `Pand` niet over de gebruikelijke `set`-methoden beschikt om de instantievariabelen te veranderen; de meeste instantievariabelen worden alleen bij de constructie van een `Pand`-object van een waarde voorzien.

a. *(4 punten)* Implementeer de methode `gemiddeldeVraagPrijs()` van de klasse `Makelaar`.

b. *(8 punten)* Maak de specificaties van de methoden `isHuidigeKoper()`, `voegKoperToe()` en `verwijderKoper()` van de klasse `Pand` af (d.w.z., geeft hun *pre- en postcondities*) en implementeer deze methoden.

c. *(5 punten)* Implementeer de methode `doeBod()` van de klasse `Makelaar`. Een makelaar kan alleen een bod doen op een pand als hij het niet zelf in de verkoop heeft. Het is niet voldoende om dit in de preconditie van `doeBod()` te eisen, d.w.z., deze conditie moet in deze methode getest worden. De methode levert `true` op als het gelukt is om een bod uit te brengen.

d. *(8 punten)* Maak de specificatie van `doeBod()` van de klasse `Pand` af (geeft *pre- en postcondities*)en implementeer de methode. Het doen van een bod op een huis is alleen toegestaan als het `Makelaar`-argument de huidige potentiële koper is. Het is niet voldoende om dit in de preconditie van `doeBod()` te eisen, d.w.z., deze conditie moet in deze methode getest worden. Verder moet een nieuw bod altijd hoger zijn dat het laatst gedane bod. De methode levert `true` op als het gelukt is om een bod uit te brengen.

Opgave 4 (25 + 5 bonus punten)

Een makelaar heeft vaak een aanzienlijk aantal panden in de verkoop. Om een goed overzicht te houden over de lijst met panden in de verkoop, moet een makelaar zijn tekoop-lijst op verschillende manieren kunnen sorteren. Daartoe wordt de volgende interface gedefiniëerd:

```
interface PandSorteerCriterium {
  public boolean kleinerDan(Pand p1, Pand p2);
}
```

a. *(5 punten)* Schrijf een klasse `PSC_TeKoopDatum` die `PandSorteerCriterium` implementeert en de methode `kleinerDan(Pand p1, Pand p2)` de waarde `true` laat opleveren als `p1` eerder in de verkoop is gegaan dan `p2`. Hiervoor kun je gebruik maken van de methode `before()` van de `Date` klasse. Deze methode heeft de volgende signatuur (van de Java 1.5 API):

```
// Tests if this date is before the specified date.
boolean before(Date when)
```

b. *(5 punten)* Schrijf een klasse `PSC_AantalKopers` die `PandSorteerCriterium` implementeert en `kleinerDan(Pand p1, Pand p2)` de waarde `true` laat opleveren als `p1` meer potentiële kopers heeft dan `p2`.

c. *(5 punten + 5 bonus punten)* De volgende methode `sorteerPanden` wordt aan de klasse `Makelaar` toegevoegd om de lijst met panden in de verkoop te sorteren volgens een sorteercriterium.

```
/**
 * Sorteert de lijst met Panden volgens een PandSorteerCriterium.
 * @require sc != null
 * @ensure getPanden() is gesorteerd volgens sc
 * @param sc het PandSorteerCriterium
 */
public void sorteerPanden(PandSorteerCriterium sc)
{
```

```
    List<Pand> pl = tekoop;
    int n = pl.size();
    // Dit algoritme is een variant op bubblesort
    for ( int i = 1; i < n; i++ ) {
      Pand p = pl.get(i);
      int j;
      for (j = i; j > 0 && sc.kleinerDan(p, pl.get(j-1)); j--)
        pl.set(j, pl.get(j-1));
      pl.set(j, p);
    }
  }
```

Geef de lusinvarianten van deze methode en leg uit waarom deze lusinvarianten samen met de lusvoorwaarden de postconditie van deze methode garanderen. De lusinvariant van binnenste lus telt als bonus punten.

d. *(10 punten)* Schets een klassendiagram met daarin de klassen `Pand`, `Makelaar`, `PandSorteerCriterium`, `PSC_TeKoopDatum` en `PSC_AantalKopers`. Hierin hoeft u niet de methoden van de klassen op te nemen.