TENTAMEN

Programmeren 1

vakcode: 213500
datum: 15 augustus 2002
tijd: 13:30 – 17:00 uur

VOORBEELDUITWERKING

Algemeen

- Bij dit tentamen mag gebruik worden gemaakt van het boek van Niño/Hosch, en van de handleiding van Programmeren 1.
- Dit tentamen bestaat uit 5 opgaven, waarvoor in het totaal 100 punten behaald kunnen worden. Het minimaal aantal punten per opgave bedraagt 0 punten.

Opgave 1 **(15 punten)**

Beoordeling: max. 5 punten per deelopgave, -2 voor elk gemiste verschil.

a. Noem drie verschillen tussen een *primitief type* (of *basistype*) en een *referentietype*.

b. Noem drie verschillen tussen een *constructor* en een *methode*.

c. Noem drie verschillen tussen een *interface* en een *abstracte klasse*.

Antwoord op Opgave 1

Beoordeling: 5 punten per deelopgave, -2 voor elk gemiste verschil. Beoordeel alleen de goede antwoorden; reken dubbele (of teveel op elkaar lijkende) antwoorden voor 1.

a. Mogelijke antwoorden zijn:

- Er is een vast aantal primitieve typen, terwijl elke klassendefinitie resulteert in een nieuw referentietype;
- Primitieve typen zijn in de taaldefinitie vastgelegd, terwijl referentietypen door de programmeur gedefinieerd kunnen worden;
- Een waarde van een referentietype is een (referentie naar een) object;
- Van referentietypen kunnen *nieuwe* waarden worden gecreëerd;
- De vergelijkingsoperator (==) vergelijkt de primitieve waarden op hun inhoud maar referentiewaarden op hun identiteit;
- Referentietypen zijn doorgaans samengesteld uit meerdere waarden van andere typen; primitieve typen zijn enkelvoudig.

b. Mogelijke antwoorden zijn:

- Een constructor heeft altijd dezelfde naam als de klasse, een methode heeft een willekeurige naam;

- Een constructor heeft geen terugkeertype, een methode wel;
- Een constructor wordt aangeroepen m.b.v. `new`, een methode niet;
- Een constructoraanroep resulteert in een nieuw object, een methode wordt aangeroepen op een bestaand object.

c. Mogelijke antwoorden zijn:

- De declaratie van een interface begint met `interface`, van een klasse met `abstract class`.
- Een abstracte klasse gebruik je m.b.v. `extends`, een interface m.b.v. `implements`.
- Een abstracte klasse kan attributen en constructoren bevatten, een interface niet.
- Een abstracte klasse kan complete methodendefinities bevatten, een interface niet.

Opgave 2 (15 punten)

Gegeven zijn de volgende klassendefinities:

```
public class Plaats {
    private String plaatsnaam;
    private String land;
}

public class Adres extends Plaats {
    private String straat;
}
```

a. *(4 punten.)* Geef voor `Plaats` en `Adres` constructordefinities, waarin alle attributen van een (begin)waarde worden voorzien.

b. *(5 punten.)* Overschrijf in `Plaats` en `Adres` de methode `equals` van `Object` zodanig dat deze methode `true` oplevert als (en alleen als) alle attributen van de `Plaats`, resp. het `Adres`, inhoudelijk aan elkaar gelijk zijn.

c. *(6 punten.)* Geef klasseninvarianten voor de attributen van `Plaats` en `Adres`, en leg uit waarom deze nodig zijn. Geef tevens precondities voor de zojuist gedefinieerde constructoren en methoden, en leg uit waarom deze nodig zijn.

Antwoord op Opgave 2

a. *(4 punten.)* Het gaat met name om het gebruik van `super` in de constructor van `Adres`.

```
/** @require plaatsnaam != null && land != null */
public Plaats(String plaatsnaam, String land) {
    this.plaatsnaam = plaatsnaam;
    this.land = land;
}

/** @require straat != null && plaatsnaam != null && land != null */
public Adres(String straat, String plaatsnaam, String land) {
    super(plaatsnaam, land);
    this.straat = straat;
}
```

b. *(5 punten.)* Hier moet opnieuw `super` gebruikt worden. Daarnaast is het belangrijk dat het parametertype `Object` is (anders wordt `Object.equals` niet overschreven maar overladen!) en dat de `Strings` m.b.v. `equals` vergeleken worden.

`Plaats.equals`:

```
/** @require obj instanceof Plaats */
public boolean equals(Object obj) {
    Plaats plaats = (Plaats) obj;
    return this.plaatsnaam.equals(plaats.plaatsnaam) &&
        this.land.equals(plaats.land);
}
```

`Adres.equals`:

```
/** @require obj instanceof Adres */
public boolean equals(Object obj) {
    Adres adres = (Adres) obj;
    return super.equals(obj) && this.straat.equals(adres.straat);
}
```

c. *(6 punten.)* De invarianten zijn:

```
/** @invariant plaatsnaam != null */
private String plaatsnaam;
/** @invariant land != null */
private String land;

/** @invariant straat != null */
private String straat;
```

Voor de precondities zie boven. De redenen zijn:

- De invarianten zijn nodig omdat anders de `equals`-aanroepen op de `String`s een `NullPointerException` kunnen opleveren;
- De precondities van de constructoren zijn nodig vanwege de invarianten;
- De precondities van de `equals`-methoden zijn nodig omdat anders een `ClassCastException` gegenereerd wordt.

Opgave 3 (25 punten)

Gegeven zij de volgende interface:

```
package meteo;

public interface MooieDag {
    public Datum getDag();
    public boolean mooierDan(MooieDag ander);
}
```

en de volgende klassen:

```
package meteo;

public class Datum {
    private final int dag;   // dag van de maand;   1 <= dag <= 31
    private final int maand; // maand van het jaar; 1 <= maand <= 12
    private final int jaar;  // 1 <= jaar

    public Datum(int dag, int maand, int jaar) {
        this.dag = dag;
        this.maand = maand;
        this.jaar = jaar;
    }
}

public abstract class Meting implements MooieDag {
    protected int wind;     // windsterkte in Beaufort: 0 <= wind <= 12
    protected int temp;     // temperatuur in graden Celcius: -273 <= temp
    protected int neerslag; // neerslag in mm: 0 <= neerslag
    private Datum dag;      // datum van deze Meting; dag != null

    public Datum getDag() {
        return dag;
    }
}
```

a. *(2 punten.)* Is de declaratie van `Datum` nog correct als aan de attributen (naast `final`) ook de aanduiding `static` toegevoegd wordt? Waarom, of waarom niet?

b. *(2 punten.)* Is de aanduiding `abstract` in de klassendefinitie van `Meting` noodzakelijk? Waarom, of waarom niet?

c. *(2 punten.)* Wat betekent de aanduiding `protected` bij de attributen van `Meting`? Wat zou er anders zijn als deze aanduiding geheel weggelaten was?

d. *(7 punten; -3 als de beperking tot 1 `return`-statement niet in acht genomend wordt, -2 voor ontbrekende precondities.)* Definieer twee subklassen van `Meting`:

- Een klasse `MooieZeilDag`, waarin de methode `mooierDan` zo gedefinieerd wordt dat een dag mooier is dan een andere wanneer de eerste beter zeilweer kent. De kwaliteit van het zeilweer wordt voornamelijk bepaald door de windkracht, die van slecht naar goed gegeven is door:

$$12 - 11 - \cdots - 8 - 0 - 1 - \cdots - 6 - 7$$

  Bij gelijke windkracht telt de temperatuur: hoe warmer hoe beter.

- Een klasse `MooieStrandDag`, waarin de methode `mooierDan` zo gedefinieerd wordt dat een dag mooier is dan een andere wanneer de eerste beter strandweer kent. Als de temperatuur onder de 16 graden is of als het geregend heeft is het strandweer altijd slecht: `mooierDan` levert

dan altijd `false` op. In andere gevallen wordt de kwaliteit van het strandweer bepaald door de temperatuur: hoe warmer hoe beter. Als de temperatuur gelijk is telt de wind: hoe minder wind hoe beter.

Schrijf in beide gevallen `mooierDan` zo dat de methode maar één `return`-statement bevat. Voorzie uw methoden bovendien van precondities.

e. *(10 punten; zonder lusinvariant geen punten.)* Definieer een methode `DatumList mooiereDagen(MooieDagList van, MooieDag grens)`, die als resultaat precies alle data van de elementen in `van` oplevert die mooier zijn dan `grens`. Hierbij zijn `DatumList` en `MooieDagList` analoog aan `StudentList` in het boek (§ 12.2), en kennen dus in ieder geval methoden

- `int size()`
- `MooieDag get(int i)`, resp. `Datum get(int i)`
- `void append(MooieDag dag)`, resp. `append(Datum dag)`

Formeel luidt de postconditie van de gevraagde methode `mooiereDagen`:

```
@ensure res bestaat precies uit alle van.get(j).getDag()
        zodat 0 <= j < van.length en van.get(j).mooierDan(grens)
```

Voorzie de lus(sen) in uw oplossing van een lusinvariant, en beargumenteer dat daarmee wordt aangetoond dat de postconditie inderdaad geldt.

Antwoord op Opgave 3

a. *(2 punten; alleen toekennen als het hele antwoord goed is.)* Nee, dat kan niet. `static` wordt gebruikt voor een klassenattribuut; een constant klassenattribuut moet ter plekke een beginwaarde krijgen, dit kan niet in de constructor. Bovendien strookt dit niet met de intuïtie aangaande deze attributen, die per object moeten kunnen verschillen. *(*Alleen het laatste antwoord is ook goed.)

b. *(2 punten; alleen toekennen als het hele antwoord goed is.)* `abstract` betekent dat er methodendefinities in `Meting` ontbreken, zodat de klasse niet geïnstantieerd kan worden. In dit geval gaat het om de methode `mooierDan`, die blijkens de declaratie `extends MooieDag` nodig is maar niet in de klasse gedefinieerd is.

c. *(2 punten; alleen toekennen als het hele antwoord goed is.)* `protected` betekent dat het attribuut in een subklasse gebruikt kan worden. Als de aanduiding weggelaten wordt kan dit alleen als de subklasse in hetzelfde package staat als de klasse `Meting` (nl. `meteo`).

d. *(7 punten; -3 als de beperking tot 1 `return`-statement niet in acht genomend wordt, -2 voor ontbrekende precondities. Dubbele fouten enkel aanrekenen.)*

```
package meteo;

public class MooieZeilDag extends Meting {
    /** @require ander instanceof Meting */
    public boolean mooierDan(MooieDag ander) {
        Meting meting = (Meting) ander;
        boolean res;
        if (wind == meting.wind)
            res = (temp > meting.temp);
        else if (wind > 7)
            res = (wind < meting.wind);
        else res = (meting.wind > 7 || wind > meting.wind);
        return res;
    }
}
```

```
package meteo;

public class MooieStrandDag extends Meting {
    /** @require ander instanceof Meting */
    public boolean mooierDan(MooieDag ander) {
        Meting meting = (Meting) ander;
        boolean res;
        if (temp < 16 || neerslag > 0)
            res = false;
        else res = (temp > meting.temp ||
                    temp == meting.temp && wind < meting.wind);
        return res;
    }
}
```

e. *(10 punten; zonder lusinvariant geen punten.)* Het gebruik van de gesuggereerde `List`-klasse heeft het voordeel dat de lengte van het resultaat automatisch bijgehouden wordt. Hier een oplossing met het array-type:

```
    /**
     * @ensure res bestaat precies uit alle van.get(j).getDag()
     *         zodat 0 <= j < van.length en van.get(j).mooierDan(grens)
     */
    public DatumList mooiereDagen(MooieDagList van, MooieDag grens) {
        DatumList res = new DatumList();
        int i = 0; // 0 <= i <= van.size()
        while (i < van.size()) {
            /* res bestaat precies uit alle van.get(j).getDag()
             * zodat 0 <= j < i en van.get(j).mooierDan(grens)
             */
            if (van.get(i).mooierDan(grens)) {
                res.append(van.get(i).getDag());
                /* res bestaat precies uit alle van.get(j).getDag()
                 * zodat 0 <= j <= i en van.get(j).mooierDan(grens)
                 */
            }
            i++;
        }
        // i = van.size()

        return res;
    }
```

Opgave 4 (25 punten)

U plant een fietsvakantie, waarin u elke dag een etappe fietst en vervolgens in een hotel slaapt, dat u zoveel mogelijk van tevoren boekt. Deze vakantie wilt u modelleren, waarbij in ieder geval de volgende eigenschappen naar voren komen:

- Beginpunt van de vakantie (plaatsnaam en land) en af te leggen etappes;
- Per af te leggen etappe: afstand (in km), te bereiken eindpunt (plaatsnaam en land), en (indien geboekt) het hotel aldaar;
- Per geboekt hotel: naam van het hotel, kosten van de overnachting, adres (straatnaam, plaatsnaam en land).

Maak in uw model gebruik van concepten `Vakantie`, `Etappe` en `Hotel`, en de klassen `Plaats` en `Adres` uit Opgave 2.

a. *(10 punten.)* Geef uw model in de vorm van een klassendiagram, waarin u per klasse de bijbehorende eigenschappen weergeeft (incl. type). Voorzie uw diagram van multipliciteiten. Constructoren en methoden mag u uit het diagram weglaten.

In de *volgende* deelopgaven gaan we ervan uit dat elke eigenschap van elke klasse in uw model door middel van een `get`-methode op te vragen is — bijvoorbeeld, als de klasse `Hotel` een attribuut `naam` heeft dan is de waarde daarvan op te vragen met behulp van een methode `getNaam()`. Bovendien nemen we aan dat de etappes van een `Vakantie` worden bijgehouden in een attribuut van het type `List` (zie § 16.1 van het boek) — hetgeen wil zeggen dat in ieder geval de volgende methoden beschikbaar zijn:

- `public int size()` om de lengte van de lijst op te vragen;
- `public Object get(int i)` om de `i`-de etappe op te vragen (waarbij de eerste etappe nummer 0 heeft).

De in de volgende deelopgaven gevraagde methoden moeten in de klasse `Vakantie` komen te staan. U hoeft geen pre- en postcondities te geven, maar voorzie uw methoden van commentaar waar dat de duidelijkheid ten goede kan komen.

b. *(3 punten.)* Schrijf een methode `isRond`, die `true` oplevert als (en alleen als) de laatste etappe eindigt in dezelfde plaats (d.w.z., met dezelfde plaats- en landnaam) als waar de vakantie begon.

c. *(6 punten.)* Schrijf een methode `etappeZonderHotel`, die een etappe oplevert waarvoor nog geen hotel geboekt is, of `null` als voor alle etappes al een hotel geboekt is.

d. *(6 punten.)* Schrijf een methode `totaleAfstand`, die de som van de afstanden van alle etappes oplevert.

Antwoord op Opgave 4

a. *(10 punten.)* Zie Figuur 1.

b. *(3 punten; een* cast *is hier niet nodig.)* De gevraagde methode:

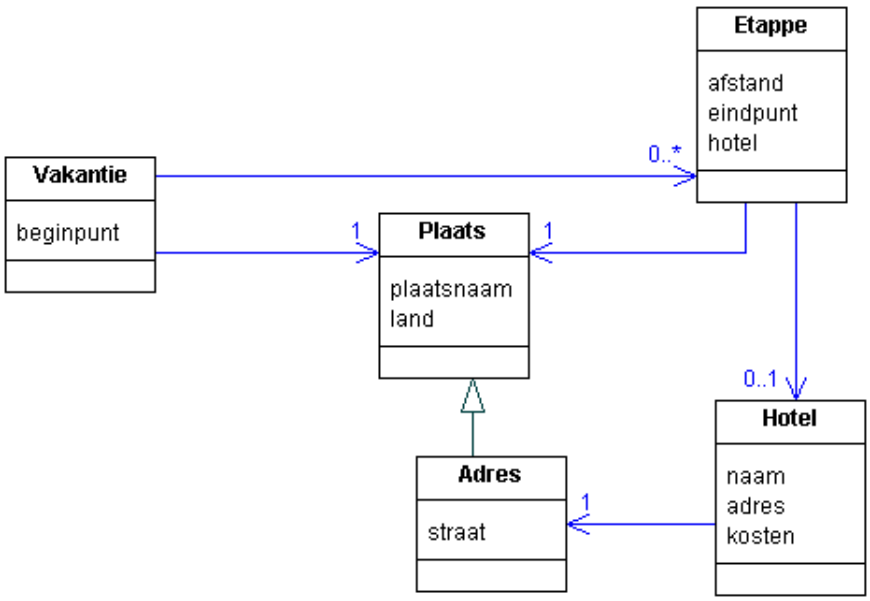
```
public boolean isRond() {
    return beginpunt.equals(etappes.get(etappes.size()-1));
}
```

c. *(6 punten; let ook op de* cast*.)* De gevraagde methode:

```
public Etappe etappeZonderHotel() {
    int i = 0;
    Etappe res = null;
    while (i < etappes.size() && res == null) {
        Etappe deze = (Etappe) etappes.get(i);
        if (deze.getHotel() == null)
            res = deze;
        i++;
    }
    return res;
}
```

d. *(6 punten; let ook op de* cast*.)* De gevraagde methode:

```
public double totaleAfstand() {
    double res = 0.0;
    int i = 0;
    while (i < etappes.size()) {
```



Figuur 1: Het gevraagde klassendiagram

```
            res += ((Etappe) etappes.get(i)).getAfstand();
            i++;
        }
        return res;
    }
```

Opgave 5 (20 punten)

Gegeven zijn de volgende klassendefinities: (De aanroep `Console.println("tekst")` heeft tot gevolg dat het woordje `tekst` op de standaarduitvoer wordt geprint.)

```
class A {
    void m1(A par) {
        Console.println("A");
    }

    void m2(A par) {
        Console.println("A");
    }

    void m3(A par) {
        this.m1(par);
    }

    void m4(B par) {
        par.m2(this);
    }
}

class B extends A {
    void m1(A par) {
        Console.println("B");
```

```
    }

    void m2(B par) {
        Console.println("B");
    }
}
```

Kruis in onderstaande tabel aan welke uitvoer bij de gegeven methodenaanroepen wordt geprint, ofwel dat de aanroep fout is (d.w.z., in een compilatie- of executiefout resulteert), gegeven de volgende variabelendeclaraties:

```
A aa = new A();
A ab = new B();
B bb = new B();
```

Antwoord op Opgave 5

Beoordeling: -2 per fout antwoord.

Aanroep	"A"	"B"	fout
aa.m1(ab)	X		
aa.m1(bb)	X		
ab.m1(ab)		X	
ab.m1(bb)		X	
bb.m1(ab)		X	
bb.m1(bb)		X	
aa.m2(ab)	X		
aa.m2(bb)	X		
ab.m2(ab)	X		
ab.m2(bb)	X		
bb.m2(ab)	X		
bb.m2(bb)		X	
aa.m3(ab)	X		
aa.m3(bb)	X		
ab.m3(bb)		X	
bb.m3(bb)		X	
aa.m4(ab)			X
aa.m4(bb)	X		
ab.m4(bb)	X		
bb.m4(bb)	X		