TENTAMEN

Programmeren 1

vakcode: 192135000
datum: 7 november 2011
tijd: 13:45-17:15

Algemeen

- Bij dit tentamen mag gebruik worden gemaakt van het boek van Niño en Hosch, de Programmeren 1 handleiding en een kopie van de collegesheets zonder aantekeningen.
- Dit tentamen bestaat uit 4 opgaven, waarvoor in totaal 105 punten (100 punten + 5 bonus punten) behaald kunnen worden. Het minimale aantal punten per opgave bedraagt 0. Het cijfer wordt berekend als (aantal punten)/10, afgerond tot een geheel getal.
- 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 de oplossing.

Opgave 1 (20 punten)

In Java is het mogelijk declaraties te voorzien van het gewenste niveau van zichtbaarheid, zoals in de code hieronder:

```
package pl.a;

public class A {
    public int p;
    protected int q;
    int r;
    private int s;
}
```

```
package pl.a;

public class B extends A {
    public int v;
    protected int w;
    int x;
    private int y;
}
```

```
package pl.c;

import pl.a.*;

public class C extends A {

}
```

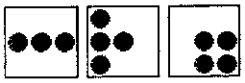
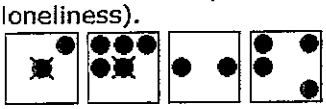
1. (*12 punten*) Neem de onderstaande tabel over en vul deze in met '+' als de variabele zichtbaar is vanuit een instantie van de betreffende klasse en met '–' als dit niet zo is.

	Zichtbaar in A	Zichtbaar in B	Zichtbaar in C
p			
q			
r			
s			
v			
w			
x			
y			

2. (*8 punten*) Leg voor iedere ingevulde '–' kort uit waarom de variabele daar niet zichtbaar is.

Opgave 2 (25 punten)

De volgende tekst is gekopieerd van `http://www.math.com/students/wonders/life/life.html`. De plaatjes illustreren wat met de cel in het midden gebeurt.

> Rules of the Game of Life
> Life is played on a grid of square cells--like a chess board but extending infinitely in every direction. A cell can be ***live*** or ***dead***. A live cell is shown by putting a marker on its square. A dead cell is shown by leaving the square empty. Each cell in the grid has a neighborhood consisting of the eight cells in every direction including diagonals.
>
> To apply one step of the rules, we count the number of live neighbors for each cell. What happens next depends on this number.
>
> - A dead cell with exactly three live neighbors becomes a live cell (birth).
>
> 
>
> - A live cell with two or three live neighbors stays alive (survival).
>
> 
>
> - In all other cases, a cell dies or remains dead (overcrowding or loneliness).
>
> 

We programmeren een stukje van dit spelletje. Ga uit van de volgende klasse:

```
public class Life {
    // de breedte van het bord (x-coordinaat gaat van 0 tot breedte-1)
    private int breedte;
    // de hoogte van het bord (y-coordinaat gaat van 0 tot hoogte-1)
    private int hoogte;
    // het bord, geinitialiseerd op correcte breedte en hoogte
    private boolean[][] bord;

    /** Berekent de volgende generatie van het spelbord.
     *  @ensure this.bord bevat de volgende generatie t.o.v. old.bord */
    public void generatie() {
        // Te programmeren
    }

    /** Berekent het aantal levende buren van een veld met gegeven x- en y-
     *  coordinaten. Velden aan de rand van het bord hebben minder buren! */
    public int aantalLevendeBuren(int x, int y) {
        // Te programmeren
    }

    /** Test of een veld met gegeven x- en y-coordinaten op het bord ligt. */
    public boolean isGeldigVeld(int x, int y) {
        // Te programmeren
    }
}
```

De variabele `bord` geeft voor elk veld (gegeven door een combinatie van geldige x- en y-coordinaten) aan of het levend is of niet.

1. *(5 punten)* Programmeer de methode `isGeldigVeld`
2. *(10 punten)* Programmeer de methode `aantalLevendeBuren`
3. *(10 punten)* Programmeer de methode `generatie`

Opgave 3 (15 + 5 bonus punten)

Een priemgetal is per definitie slechts deelbaar door 1 en zichzelf. Een eenvoudige, hoewel inefficiënte, manier om vast te stellen of een bepaald getal wel of géén priemgetal is, is te testen of het getal deelbaar is door één van de getallen die groter zijn dan 1 en kleiner dan of gelijk aan de wortel uit dat getal (naar beneden afgerond). In dit geval is het getal géén priemgetal.

1. *(7 punten)* De wortel van een willekeurig getal kunt u berekenen met de klassenmethode `(static) double sqrt(double n)` van de klasse `Math`. Programmeer een methode `boolean isPriem(int n)` die voor de parameter `n` (een positief getal) aangeeft of dit al dan niet een priemgetal is. Zorg dat er maar één `return`-opdracht in uw oplossing staat.
2. *(8 punten)* Geef pre- en postcondities voor de methode `isPriem` en een lusinvariant die de postconditie garandeert.
3. *(5 bonus punten)* Leg uit waarom de lusinvariant de correctheid van de methode garandeert.

Opgave 4 (40 punten)

Deze opgave bouwt voort op de volgende definities:

```
public interface Slot {
    /**
     * Test of het slot open is.
     */
    boolean isOpen();

    /**
     * Maakt het slot open als de sleutel past.
     * Levert true op als het slot voor aanroep dicht was,
     * en na aanroep open.
     */
    boolean maakOpen(Sleutel sleutel);

    /**
     * Doet het slot dicht.
     */
    void doeDicht();
}

public interface Sleutel { }
```

1. *(2 punten)* Definieer een interface `SleutelFabriek`, die `Slot` uitbreidt met een methode `maakSleutel()` die een `Sleutel` oplevert die het `Slot` open kan maken.
2. *(5 punten)* Geef, waar dit mogelijk is, postcondities voor de methoden van `Slot` en `SleutelFabriek` die het effect van deze methoden zo precies mogelijk beschrijven.

3. *(4 punten)* Schrijf een abstracte klasse `AbstractSlot`, die `Slot` implementeert door een `boolean` instantievariabele `open` te definiëren, die van waarde veranderd wordt in `maakOpen` en `doeDicht` en getest wordt in `isOpen`. De methode `maakOpen` in `AbstractSlot` gaat er van uit dat de gegeven `Sleutel` altijd past.

   Alle methodes van de klasse `AbstractSlot` worden geïmplementeerd (hebben een 'romp'), maar de klasse is als `abstract` gedefinieerd alleen om te voorkomen dat instanties (objecten) van deze klasse aangemaakt kunnen worden. Gebruik `AbstractSlot` in de hieronder gevraagde implementaties, waar dit van toepassing is.

4. *(6 punten)* Geef implementaties `CodeSleutel` van `Sleutel` en `CodeSlot` van `SleutelFabriek`, zodat beide `CodeSleutel` en `CodeSlot` klassen een constructor hebben waaraan een `int`-parameter wordt meegegeven, en `CodeSleutel` tevens een methode `int getCode()` heeft waarmee de code wordt opgevraagd. Een `CodeSlot`-instantie kan slechts worden geopend met een `CodeSleutel` met de gelijke code. Bijvoorbeeld, na het volgende programmastuk

```
SleutelFabriek codeSlot = new CodeSlot(666);
Sleutel codeSleutel = codeSlot.maakSleutel();
boolean fout = codeSlot.maakOpen(new CodeSleutel(999));
boolean goed = codeSlot.maakOpen(codeSleutel);
```

   hebben de variabelen `fout` de waarde `false`, `goed` de waarde `true`, en `codeSleutel.getCode()` de waarde `666`.

5. *(7 punten)* Geef een implementatie `OneShotSlot` van `SleutelFabriek` die alle door de methode `maakSleutel()` geconstrueerde sleutels in een lijst bijhoudt. De methode `maakOpen` slaagt alleen als de meegegeven sleutel in de lijst voorkomt; zo ja, dan wordt de sleutel meteen uit de lijst verwijderd. Het effect is dat elke sleutel het slot precies één keer kan openen. Bijvoorbeeld, na het volgende programmastuk

```
SleutelFabriek slot = new OneShotSlot();
Sleutel sleutel1 = slot.maakSleutel();
Sleutel sleutel2 = slot.maakSleutel();
boolean goed = slot.maakOpen(sleutel1);
slot.doeDicht();
boolean fout = slot.maakOpen(sleutel1);
boolean weerGoed = slot.maakOpen(sleutel2);
```

   hebben de variabelen `goed` de waarde `true`, `fout` de waarde `false`, en `weerGoed` de waarde `true`. Definieer een passende klasse voor de `Sleutel`-instantie die door `maakSleutel` wordt opgeleverd.

   Hint: Maak desgewenst gebruik van de `List<E>`-methoden `boolean add(E elem)` (voor het toevoegen van een element), `boolean contains (Object o)` (voor het testen of een element in de lijst zit) en `boolean remove (Object o)` (voor het verwijderen van een element).

6. *(7 punten)* Geef een implementatie `CombinatieSlot` van `Slot`, met daarin de volgende definities:

```
private List<Slot> sloten;

public void voegSlotToe(Slot slot) {
    sloten.add(slot);
}
```

   die een lijst van `Slot`-instanties bijhoudt. `CombinatieSlot` heeft een methode `voegSlotToe(Slot slot)` die het als parameter meegegeven slot aan `sloten` toevoegt. Een `Sleutel` past alleen op een `CombinatieSlot` als de sleutel op álle elementen van `sloten` past.

Probeert iemand een (gesloten) `CombinatieSlot` open te maken met een sleutel die niet op alle sloten past, dan dienen alle sloten dicht te gaan.

7. *(9 punten)* Teken een klassendiagram met daarin alle interfaces en (abstracte) klassen die in deze opgave voorkomen. Met `Slot` en `Sleutel` er bij *zijn er in totaal 9 interfaces en (abstracte) klassen.* Neem in het diagram geen afhankelijkheden of instantievariabelen op, maar wel overerving, associaties met multipliciteiten en *nieuwe* methoden, d.w.z., die geen implementatie of overschrijving zijn van een methode van een supertype (interface of klasse).