Toets Objectoriëntatie (IPI005)

15 april 2010, van 13:30 tot 15:30, HG00.307

- Deze toets is gesloten boek. Je mag geen literatuur, PC, mobiel, PDA etc. gebruiken.
- Het geheel bestaat uit twee opgaven met verschillend gewicht: opdracht 2 weegt driemaal zo zwaar als opdracht 1. De onderdelen tellen niet gelijk mee voor het cijfer. De waardering per onderdeel staat vermeld in de kantlijn.
- Ga niet overhaast te werk en lees de opgaven nauwkeurig. Begin met de opgaven die je het eenvoudigst lijken. Denk eerst goed na. Hou je antwoorden kort en controleer ze voor je het werk inlevert. Schroom niet vragen te stellen over eventuele onduidelijkheden in de opgaven.

Succes!

1 Goed of fout en kleine opgaven **totaal 10 punten**

Zijn de volgende uitspraken goed of fout? Zonder goede motivatie krijg je geen punten.

2 punten **1.a)** Interfaces in Java zijn in feite geheel overbodig. Het zijn in feite niet meer dan helemaal abstracte klassen.

2 punten **1.b)** Onderstaande implementatie van de methode `equals` van een klasse `C` is een goede manier om gelijkheid in Java te implementeren.

```
public boolean equals(C c)
{   return this == c; }
```

2 punten **1.c)** Het ultieme doel van UML is een visiuele software ontwikkelde methode te maken die een specificatie oplevert waaruit de benodigde Java code automatisch gegenereerd kan worden.

2 punten **1.d)** De workflows in de activity diagrams uit UML zijn een visuele weergave van (deel)activiteiten uit de use cases. Deze workflows zijn altijd eindig en bevatten dus geen cycles.

Als laatste onderdeel van deze opgave een JUnit codefragment. Gegeven de volgende twee methoden uit een Java klasse.

```
/**
 * @return a && b && c
 */
public static boolean and3 (boolean a, boolean b, boolean c)
{   return toInt(a)+toInt(b)+toInt(c)==3; }
/**
 * @param boolean to convert
 * @return 1 for true, 0 for false
 */
public static int toInt(boolean b)
```

```
{   return b ? 1 : 0; }
/**
 * @param int to convert
 * @return 1 for true, 0 for false
 */
public static boolean toBool(int i)
{   return i % 2 == 1; }
```

De JUnit test die hiervoor gegenereerd wordt is:

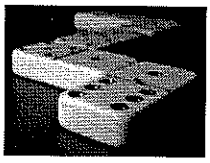
```
@Test
public void testAnd3()
{
    System.out.println("and3");
    boolean a = false;
    boolean b = false;
    boolean c = false;
    boolean expResult = false;
    boolean result = Main.and3(a, b, c);
    assertEquals(expResult, result);
    fail("The test case is a prototype.");
}
```

2 punten

1.e) Geef een versie van de methode `testAnd3` die test of `and3(a,b,c)` een *correcte* implementatie is van `a && b && c` door alle gevallen systematisch te controleren.

2 Domino

totaal 30 punten



Deze opdracht lijkt enigszins op opdracht 4 van het practicum. Dit keer nu ga je het spel *Domino* dat implementeren. We nemen aan dat iedereen het spel kent, maar mocht je niet precies op de hoogte zijn van de regels volgt hier een beknopte uitleg. Het spel wordt met speciale dominostenen gespeeld. Deze stenen zijn langwerpig van vorm en verdeeld in twee helften. In de standaard variant van het spel (waarvan we in deze opgave uit gaan) hebben de helften ieder nul tot zes ogen. Dit levert 28 mogelijke combinaties van 0 tot 6 op die in het spel alle éénmaal voorkomen. De bedoeling is een kant met een bepaald aantal ogen tegen één van de kanten van de reeds gelegde stenen met hetzelfde aantal ogen te leggen. Tijdens het spel ontstaat op die manier een *slang* van gelegde stenen, die steeds aan de uiteinden zal worden verlengd. Wie als eerste al zijn stenen kwijt is, heeft gewonnen. Heeft de speler die aan de beurt is geen passende steen dan moet hij/zij een nieuwe steen van de hoop nemen en gaat de beurt over naar de volgende speler. Heeft hij/zij wel een passende steen dan *mag* die aan de slang worden toegevoegd. De speler kan er ook voor kiezen om in plaats hiervan een nieuwe steen van de hoop nemen.

Uiteraard hoef je niet het hele spel te implementeren, maar geven we een groot deel van deze implementatie voor en dien je zelf de ontbrekende programmafragmenten toe te voegen

De klasse Steen

De klasse `Steen` waarmee de dominostenen gepresenteerd worden definiëren we rechttoe rechtaan. Zoals je kunt zien implementeert deze klasse het `Comparable` interface.

```
public class Steen implements Comparable
{
```

```
    public final static int MinOgen = 0, MaxOgen = 6;

    private int o1, o2;

    Steen (int o1, int o2)
    {   this.o1 = o1; this.o2 = o2; }

    public int compareTo(Object o)
    { ... }

    public boolean pastRechts (Steen s, boolean gedraaid)
    {   return gedraaid ? o2 == s.o2 : o2 == s.o1; }

    public boolean pastLinks (Steen s, boolean gedraaid)
    {   return gedraaid ? o1 == s.o1 : o1 == s.o2; }

    public void draai ()
    {   int t = o1; o1 = o2; o2 = t; }
}
```

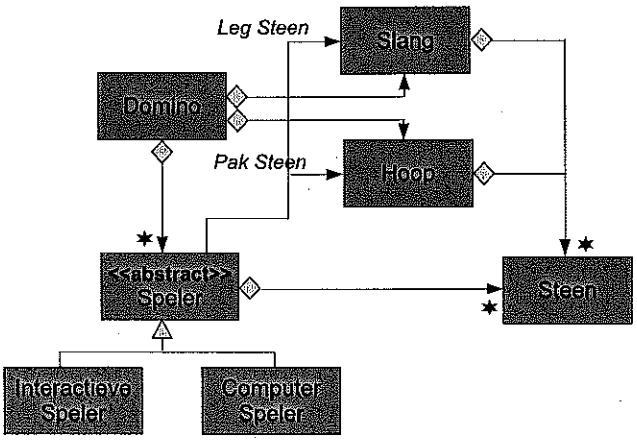
Met de methodes `pastRechts` en `pastLinks` kan worden nagegaan op 2 stenen passen, oftewel het overeenkomstige aantal ogen gelijk is. De boolean `gedraaid` geeft aan of de meegegeven steen `s` daarvoor eerst gedraaid dient te worden. Voorbeeld: voor de stenen s_1 = `[0,3]` en s_2 = `[5,3]` geldt dat s_1`.pastRechts(`s_2`,true)` de waarde **`true`** oplevert.

Zoals je kunt zien implementeert de klasse `Steen` het `Comparable` interface. Dit interface bevat één abstracte methode `compareTo` waarmee objecten vergeleken kunnen worden. De aanroep `a.compareTo(b)` dient een negatieve waarde op te leveren als `a` < `b`, 'n positieve waarde als `a` > `b` en 0 indien `a` en `b` gelijk zijn.

4 punten

2.a) Implementeer `compareTo` voor de bovenstaande klasse, waarbij we een steen s_1 kleiner dan s_2 beschouwen als het kleinste aantal ogen van s_1 kleiner is dan het kleinste aantal ogen van s_2, óf, indien deze gelijk zijn, het grootste aantal ogen van s_1 kleiner is dan het grootste aantal ogen van s_2.

Onderstaand diagram geeft welke klassen voor het dominospel gebruikt worden en wat hun onderlinge relaties zijn. Wat nog ontbreekt is de klasse `dominoGI` waarmee we interactie met de gebruiker regelen. Deze klasse komt later aan de orde.



In het vervolg zal voor het opslaan van reeksen van stenen steeds gebruik worden gemaakt van flexibele rijen: `ArrayList<Steen>` in Java. Omdat je mogelijk niet (meer) weet welke

methodes er voor dit type beschikbaar zijn volgt hier een kort overzicht. Met `size` kun je de grootte (het aantal opgeslagen elementen bepalen). De methode `get(i)` levert je het i^{de} element. Met `add(st)` kun je een steen `st` achteraan toevoegen. Invoegen op positie `i` kan met `add(i, st)`. De elementen die al aanwezig zijn worden vanaf positie `i` allemaal één plek opgeschoven. Verwijderen van het i^{de} element kan met `remove(i)`. Als er nog elementen na positie `i` aanwezig zijn worden deze alle één plek terug geschoven.

De klassen Hoop en Slang

Met deze `ArrayLists` worden resp. de *hoop* van (de nog niet in het spel zijnde) dominostenen en de *slang* van de reeds gelegde stenen gerepresenteerd. Van de klasse `Hoop` geven we de methodes zonder bodies.

```
public class Hoop
{
    public static final int TotaalAantalStenen = 28;
    private ArrayList<Steen> stenen;

    /**
     * De constructor creeert een hoop van 28 verschillende stenen
     */
    public Hoop ()

    /**
     * Husselt de stenen door elkaar
     */
    public void hussel ()

    /**
     * Haalt de bovenste steen van de hoop en levert deze als resultaat op
     */
    public Steen pak ()
}
```

Evenals de klasse `Hoop` wordt in `Slang` een `ArrayList` van stenen gebruikt om bij te houden welke stenen er gelegd zijn.

```
public class Slang
{
    public ArrayList<Steen> stenen;

    /**
     * De constructor creeert een slang die uit 1 steen bestaat.
     */
    public Slang (Steen s)
    {
        stenen = new ArrayList<Steen> (0);
        stenen.add(s);
    }

    /**
     * Test of Steen s links (vooraan) kan worden aangesloten
     */
    public boolean pastLinks (Steen s)
    {
        return stenen.get(0).pastLinks(s, false) ||
               stenen.get(0).pastLinks(s, true);
```

```
    }

    /**
     * Test of Steen s rechts (achteraan) kan worden aangesloten
     */
    public boolean pastRechts (Steen s)
    {
        return stenen.get(stenen.size()-1).pastRechts(s, false) ||
               stenen.get(stenen.size()-1).pastRechts(s, true);
    }

    /**
     * Voegt Steen s links aan de slang toe.
     * Indien nodig wordt s eerst gedraaid
     */
    public void legLinks (Steen s)
    { }

    /**
     * Voegt Steen s rechts aan de slang toe.
     * Indien nodig wordt s eerst gedraaid
     */
    public void legRechts (Steen s)
}
```

4 punten **2.b)** Implementeer de methode `legLinks`.

De (abstracte) klasse Speler

In het uiteindelijke spel zijn er twee soorten spelers: mens en computer. Om deze spelers op een uniforme manier te kunnen aanspreken maken we gebruik van een abstracte klasse, genaamd `Speler`. Deze bevat naast een aantal attributen en niet abstracte methodes één *abstracte methode* `speelBeurt` die de volgende beurt van de betreffende speler bepaalt. Gaat het hierbij om de menselijke speler dan vraagt het programma aan de gebruiker of er een steen van de hoop gepakt moet worden óf dat er een van de reeds getrokken stenen gespeeld gaat worden. Bij de computer speler bepaalt de computer de volgende zet.

Ook nu weer wordt een `ArrayList` gebruikt om stenen op te slaan.

```
public abstract class Speler
{
    private ArrayList<Steen> stenen;
    private String naam;

    public Speler (String naam)
    {
        this.naam = naam;
        stenen = new ArrayList<Steen> (0);
    }

    public String geefNaam () { return naam;}

    public void pakVanHoop (Hoop h) { ... }

    public boolean isKlaar () { return stenen.size() == 0;}

    public boolean steenPast (int sn, Slang sl, boolean rechts) { ... }
```

```
public void legSteen (int sn, Slang sl, boolean rechts) { ... }

public void startSpel ()
{   Collections.sort(stenen); }

abstract void speelBeurt (Slang sl, Hoop h);
```

4 punten **2.c)** Met de methode `pakVanHoop` pakt de speler een steen van hoop `h` en voegt die aan zijn eigen verzameling toe. Implementeer deze methode.

4 punten **2.d)** Met de methode `steenPast` wordt bepaald of de `sn`^{de} steen (uit de rij van `stenen`) aan de slang `s` kan worden toegevoegd. De boolean parameter `rechts` geeft aan, aan welke zijde van de slang dit gebeurt (**`true`** betekent rechts; **`false`** links). Implementeer deze methode.

Met de methode `legSteen` voegt de speler zijn `sn`^{de} steen aan de slang toe. Deze hoeft niet geïmplementeerd te worden.

De computer als speler

De klasse `ComputerSpeler` vormt een uitbreiding van de `Speler`-klasse. Hierin wordt de abstracte `speelBeurt` als volgt uitgewerkt: Als de compuer een passende steen heeft dan zal deze altijd neergelegd worden. Anders wordt er een steen van de hoop genomen.

5 punten **2.e)** Implementeer deze klasse in z'n geheel.

De klasse Domino

De klasse `Domino` beheert het spel. Hierin wordt o.a. bepaald welke spelers er mee doen, wie de beurt heeft, of het spel uit is en welke beginstenen iedere speler heeft.

```
public class Domino
{
    private Speler spelers [];
    private Slang slang;
    private Hoop hoop;

    private int volgendeSpeler;

    public Domino (Speler spelers [])
    { ... }

    public boolean spelVoorbij ()
    { return spelers[volgendeSpeler].isKlaar(); }

    public void speel ()
    { ... }
}
```

In de constructor (waarvan we de body verder niet zullen specificeren) worden de initieële hoop en slang aangemaakt en worden de spelers van stenen voorzien waarmee ze het spel moeten gaan spelen. De spelers die aan het spel meedoen worden opgeslagen in de rij `spelers` en de (index van de) speler die aan de beurt is wordt bijgehouden in `volgendeSpeler`.

4 punten

2.f) Implementeer de methode `speel` die de aan de beurt zijnde speler één beurt laat spelen, tenminste als het spel niet is afgelopen. Bovendien gaat de beurt over naar de eerstvolgende volgende speler in de rij. Houd er rekening mee dat als je aan het einde van de spelers-rij bent gekomen je weer vooraan moet beginnen.

De mens als speler

Om de gebruiker van het programma zelf aan het spel te laten deelnemen introduceren we 2 klassen: `DominoGI` en `InteractieveSpeler`. De eerste klasse `DominoGI` regelt de in- en uitvoer van het programma en de tweede klasse `InteractieveSpeler` zorgt er voor dat de invoer van de gebruiker op de juiste manier wordt verwerkt. Aangezien de gebruiker bij het spelen uit een aantal mogelijkheden moet kunnen kiezen spreken we af dat bij iedere beurt de gebruiker middels een geheel getal aangeeft welke zet hij wil doen. Voor de gebruiker zijn de beschikbare stenen " normaal" genummerd vanaf 1 tot en met het totaal aantal stenen. Door het invoeren van een positief getal i wordt de i^{de} steen *rechts* aan de slang toegevoegd. Met een negatief getal $-i$ wordt de i^{de} steen links aan de slang aangesloten. Met 0 tenslotte geeft de gebruiker aan een steen van de hoop te willen nemen.

We geven eerst de klasse `InteractieveSpeler` die de abstracte klasse `Speler` uitbreidt (door de methode `speelBeurt` te implementeren) en waarin duidelijk is te zien hoe de door de gebruiker ingevoerde beurt (opgeslagen in de attribuut `volgendeBeurt`) wordt verwerkt.

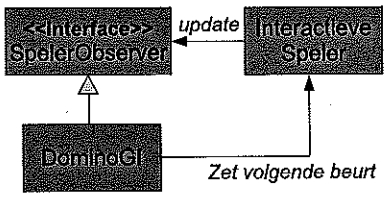
```
public class InteractieveSpeler extends Speler
{
    int volgendeBeurt;

    public InteractieveSpeler (String naam)
    {   super(naam); }

    public void speelBeurt(Slang sl, Hoop h)
    {
        if (volgendeBeurt == 0)
            pakVanHoop (h);
        else if (volgendeBeurt > 0)
            legSteen (sl, volgendeBeurt, true);
        else
            legSteen (sl, -volgendeBeurt, false);
    }

    public void zetVolgendeBeurt (int beurt)
    {   volgendeBeurt = beurt; }
}
```

De klasse `DominoGI` ziet er als volgt uit. De constructor wordt niet nader gespecificeerd en hetzelfde geldt voor de methode `leesVolgendeBeurt` waarmee de gebruikersinvoer wordt ingelezen. Zoals je ziet hebben we ervoor gekozen aan deze klasse de methode `speelDomino` toe te voegen waarmee iteratief de volgende zet in het spel wordt gespeeld totdat het spel is afgelopen. De manier waarop informatie over het verloop van het spel aan de gebruiker wordt gegeven laten we in het midden.

```
public class DominoGI
{
    private Domino spel;
    private static final int AantalSpelers = 2;

    public DominoGI ()
```

```
    { ... }

    public void speelDomino ()
    {
        while (! spel.spelVoorbij())
        {
            spel.speel();
        }
    }

    private int leesVolgendeBeurt ()
    { ... }
}
```

Wat rest is de koppeling tussen de `DominoGI` en de `InteractieveSpeler` klasse. Om ervoor te zorgen dat de klasse `InteractieveSpeler` geen weet heeft van de manier waarop de gebruikersinvoer wordt verkregen maken we bij deze koppeling gebruik van een *interface* genaamd `SpelerObserver`. Zodra de `InteractieveSpeler` nieuwe invoer verwacht stelt hij het `SpelerObserver`-object hiervan op de hoogte. Het volgende diagram geeft de relatie tussen de twee klassen en het interface weer.



Het `SpelerObserver` interface is als volgt gedefinieerd:

```
public interface SpelerObserver
{
    public void update (InteractieveSpeler speler);
}
```

4 punten **2.g)** Breid de klasse `InteractieveSpeler` uit met de methode `registreer` waarmee een `SpelerObserver` zich bij de interactieve speler kan registreren zodanig dat deze observer op de hoogte kan worden gesteld van veranderingen van het bord. Hiervoor heeft klasse `InteractieveSpeler` een extra attribuut nodig en moet de update methode op een geschikt moment worden aangeroepen. Geef deze aanpassingen ook duidelijk aan in `InteractieveSpeler`.

5 punten **2.h)** Breid de klasse `DominoGI` zodanig uit dat deze het bovenstaande interface implementeert.