PROGRAMMEREN 2

TOPIC 10: OGP - VARIA EN UML

Download PDF-versie

UCLL HOGESCHOOL

OVERZICHT

- Method overloading
- Operator overloading
- Iterators en generators
- OO-analyse en ontwerp met UML
- OO-analyse en UML inoefenen

Van de eerste drie topics moet je geen oefeningen maken voor het portfolio.

METHOD OVERLOADING

OVERLOADING

Functies en methodes hebben een **signatuur**, bestaande uit:

- naam
- aantal parameters, hun types en volgorde
- (in sommige talen) namen van parameters, return type

Method overloading houdt in dat een methode in een klasse meerdere keren gedefinieerd kan worden. Elke variant moet wel een unieke signatuur heben. Aangezien de naam al hetzelfde is, moet het verschil zitten in de rest van de signatuur.

Function overloading bestaat ook en volgt hetzelfde principe, maar dan met functies buiten klassen.

Python ondersteunt method/function overloading niet helemaal. Op de volgende slide is er een voorbeeld van Java, waar er wel volwaardige ondersteuning is.

METHOD OVERLOADING - VOORBEELD IN JAVA

Java

```java
public class Circle {
  private Point center;
  private double radius;

  // Constructor 1: punt en straal
  public Circle(Point center, double radius) {
    this.center = center;
    this.radius = radius;
  }

  // Constructor 2: twee punten
  public Circle(Point point1, Point point2) {
    double centerX = (point1.getX() + point2.getX()) / 2;
    double centerY = (point1.getY() + point2.getY()) / 2;
    double radius = Math.sqrt(
        Math.pow(
            point2.getX() - point1.getX(), 2) +
            Math.pow(point2.getY() - point1.getY(), 2))
        / 2;

    this.center = new Point(centerX, centerY);
    this.radius = radius;
  }
}
```

DETAILS

De cruciale lijnen zijn de twee signatures van de constructors:

```java
public Circle(Point center, double radius) { ... }

public Circle(Point point1, Point point2) { ... }
```

- De namen van de methodes zijn hetzelfde
- Toch is de signatuur verschillend: de types van de parameters zijn anders

Wanneer je een `Circle` creëert in Java, zal het programma automatisch kunnen afleiden welke constructor in welke situatie gebruikt moet worden, door de types van de parameters te controleren.

AANDACHTSPUNTEN

Verwar method **overloading** niet met method **overriding**. Bij overriding overschrijft een klasse een methode van hun *superklasse*. Overloading heeft niets te maken met superklassen of overerving.

Method overloading is iets extra dat soms van pas komt, maar is geen kernleerstof in de meeste vakken.

OPERATOR OVERLOADING

PROBLEEMSTELLING

In Python kan een `+` meerdere dingen doen:

- Twee `int`s optellen
- Een `int` met een `float` optellen
- Twee `string`s samenvoegen (concateneren)
- ...

Als we zelf nieuwe klassen maken, willen we soms ook gebruik maken van `+` en andere operatoren (`-`, `*`, `/`, `==`, `<=`, ...) om met onze klassen te werken.

OPERATOR OVERLOADING

Operator overloading houdt in dat je operators `+`, `-`, `*`, `/`, `==`, `<=`, ... een nieuwe definitie geeft voor bepaalde klassen.

In Python is elke operator gelinkt aan een speciale methode:

- `__add__()` voor `+`
- `__mul__()` voor `*`
- `__eq__()` voor `==`
- ...

Volledige lijst: handboek hoofdstuk 21 (of zoek online op)

Als je deze methode overschrijft in een bepaalde klasse, 'overload' je de operator. Je kan dan het overeenkomstige symbool gebruiken.

VOORBEELD: COMPLEXE GETALLEN

Een complex getal $a + bi$ bestaat uit twee delen:

- a is het reëel deel
- b is het imaginair deel

De code hieronder definieert een klasse `Complex` om complexe getallen voor te stellen. Ze bevat ook een aantal testen met operators. Momenteel falen de testen.

Python

```python
class Complex:
  def __init__(self, a, b):
    self.a = a
    self.b = b

x = Complex(1, 2)
y = Complex(3, 4)

assert x + y == Complex(4, 6)
assert x * y == Complex(-5, 10)
```

VOORBEELD

We moeten 3 operators overloaden:

- `+` om twee complexe getallen op te tellen
- `==` om twee complexe getallen te vergelijken (denk aan call by reference bij klassen!)
- `*` om twee complexe getallen te vermenigvuldigen

Optellen is mogelijk door `__add__()` te overloaden:

Python

```
class Complex:
  # ...
  def __add__(self, other):
    return Complex(self.a + other.a, self.b + other.b)
```

OPDRACHT

Kopieer de code van de vorige twee slides. Breidt ze dan verder uit:

- Overload `==` zodat de eerste test slaagt. Twee complexe getallen zijn gelijk als hun a en b gelijk zijn
- Overload `*` zodat de tweede test slaagt. De formule is

$$(a+bi)(c+di) = (ac-bd) + (ad+bc)i$$

AANDACHTSPUNTEN

Verwar operator overloading niet met method/function overloading.

Operator overloading is iets extra dat soms van pas komt, maar is geen kernleerstof in de meeste vakken.

ITERATOREN EN GENERATOREN

CONTEXT

In Programmeren 1 heb je een aantal datastructuren gezien waarover je kan itereren: tuples, lijsten, dictionaries en strings. Deze structuren/objecten zijn itereerbaar (Engels: *iterable*).

Van een itereerbaar object kan je een iterator opvragen door middel van `iter()`:

```python
fruit = "banana"
fruit_iterator = iter(fruit)
```

Nadien kan je telkens het **volgende** element opvragen uit de iterator met `next()`:

```python
print(next(fruit_iterator))  # b
print(next(fruit_iterator))  # a
```

> **i** Een `for x in my_list`-loop gebruikt achter de schermen `iter(my_list)` en `next(my_list_iterator)`.

ZELF ITERATORS MAKEN

Elke klasse kan een iterator worden. De klasse moet 2 methodes implementeren om als iterator beschouwd te worden: `__iter__()` en `__next()__`.

Hoe je dit exact schrijft, kan je terug vinden in Hoofdstuk 23 van het handboek.

GENERATOREN

Generatoren zijn **functies** die itereerbare objecten nabootsen. Ze gebruiken daarvoor het gereserveerde woord `yield`.

Generatoren zijn simpeler te schrijven dat iteratoren. Doordat het functies zijn, kunnen ze ook gecombineerd worden met andere functies (wordt niet verder behandeld).

Hoe je een generator exact schrijft, kan je terug vinden in Hoofdstuk 23 van het handboek.

Iteratoren en generatoren zijn niche-concepten. Ze zijn in bepaalde domeinen en problemen zeer nuttig, maar in de context van een middelbare school zullen ze zelden gebruikt worden. Niet elke taal ondersteunt ze even goed.

OO-ANALYSE EN ONTWERP MET UML

OO-ANALYSE

Als je een probleem krijgt voorgelegd, maak je daarvan een object-oriented-analyse, waarin je de te definiëren klassen bepaalt, met hun attributen en methodes.

Het resultaat van zo'n analyse is een beschrijving van de oplossing. Heel vaak wordt dit genoteerd in UML (Unified Modeling Language). UML kan het volgende beschrijven:

- Klassen met hun attributen en methodes
- Verbanden tussen klassen
- Principes van overerving: interfaces, abstracte klassen, hiërarchieën van klassen
- Nog veel meer - wordt niet verder behandeld in dit vak

Een diagram met klassen, verbanden en overerving noemt een **klassediagram**.
Het is één van de meest gebruikte diagrammen in UML.

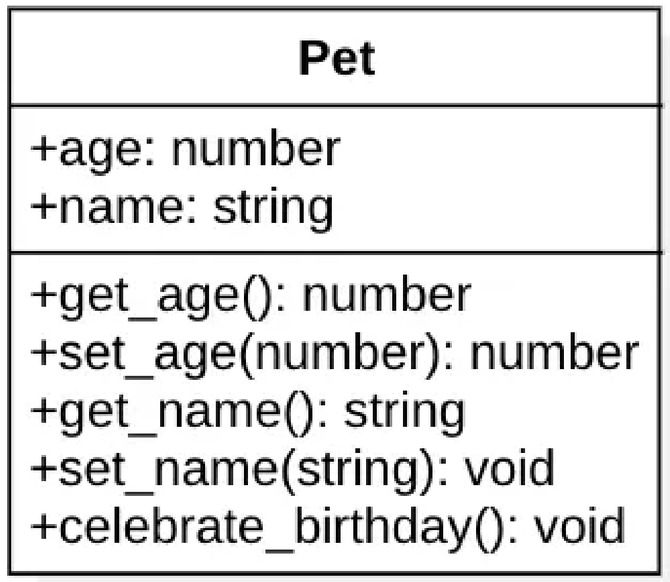
UML-DIAGRAMMEN TEKENEN

Er bestaan verschillende tools om UML-diagrammen te ontwerpen:

- StarUML: redelijk goed, maar moet lokaal geïnstalleerd worden
- draw.io: online software om diagrammen te tekenen. Ondersteunt onder andere UML. Iets minder gebruiksvriendelijk

UML: KLASSEN

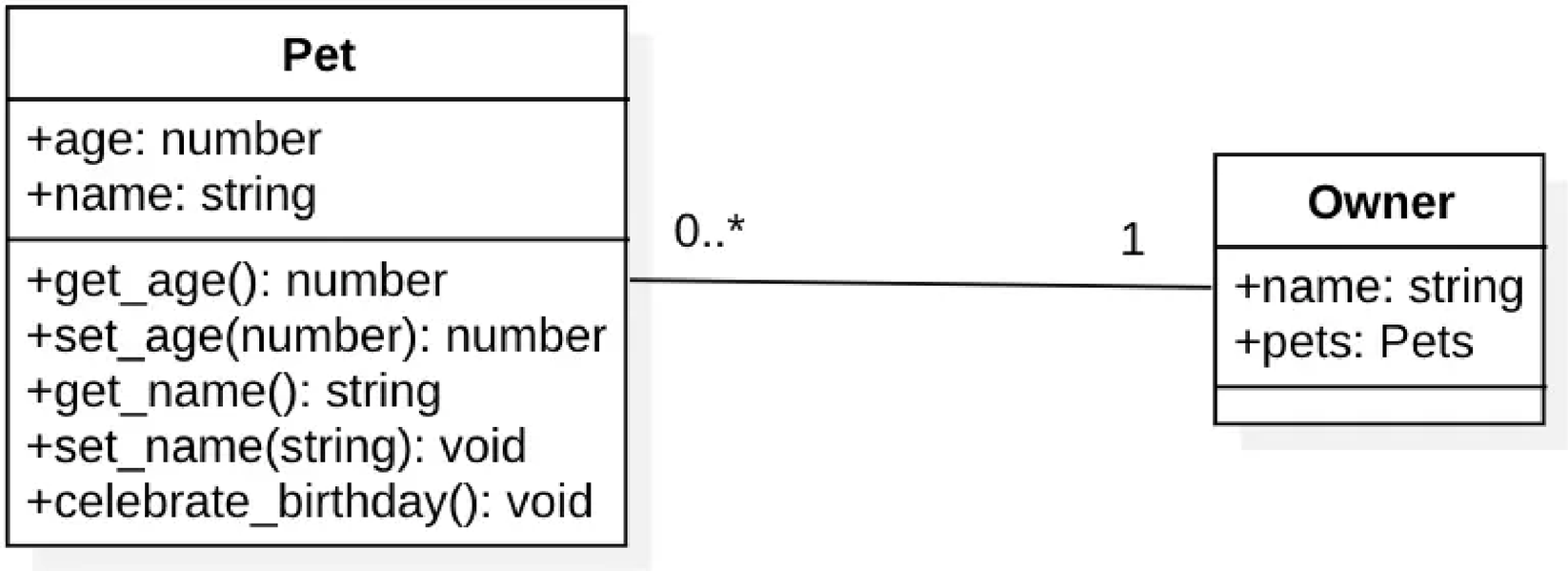
Een klasse in UML ziet er als volgt uit:



- Naam van klasse bovenaan
- Namen van attributen met type
- Namen van methodes met types voor parameters tussen haakjes, en return type achteraan. `void` wordt gebruikt om aan te geven dat een functie niets teruggeeft

UML: ASSOCIATIE

Een associatie (verband) tussen twee klassen noteer je met een lijn:

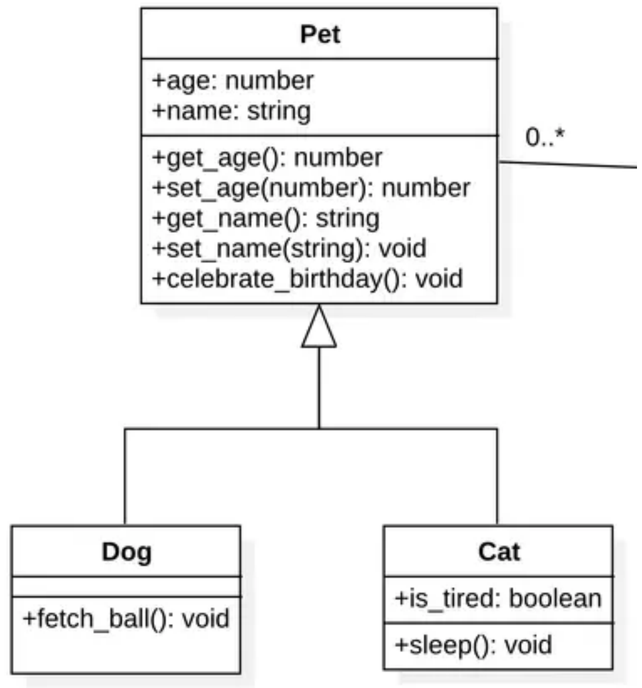


De cijfers bij de lijn geven de aantallen aan per collectie. Dit noemt men de **kardinaliteit**. In de figuur betekenen de kardinaliteiten:

- Elke `Owner` heeft tussen de 0 en oneindig aantal `Pet`s
- Elke `Pet` heeft exact 1 `Owner`

UML: OVERERVING

Overerving tussen twee of meer klassen noteer je met een pijl:

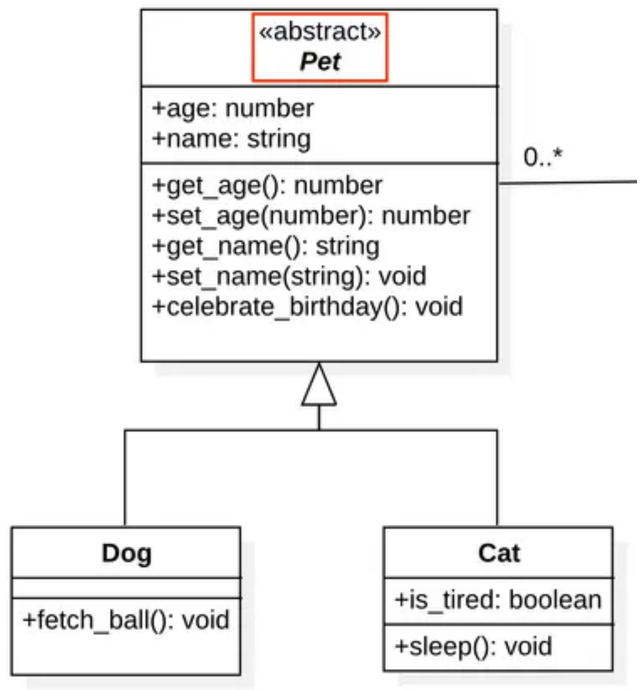


De pijl wijst naar de superklasse. In de figuur erven `Dog` en `Cat` beiden van `Pet`.

Merk op dat de attributen en methodes van `Pet` **niet** herhaald worden in de subklassen. Ze worden impliciet overgenomen.

UML: ABSTRACTE KLASSE

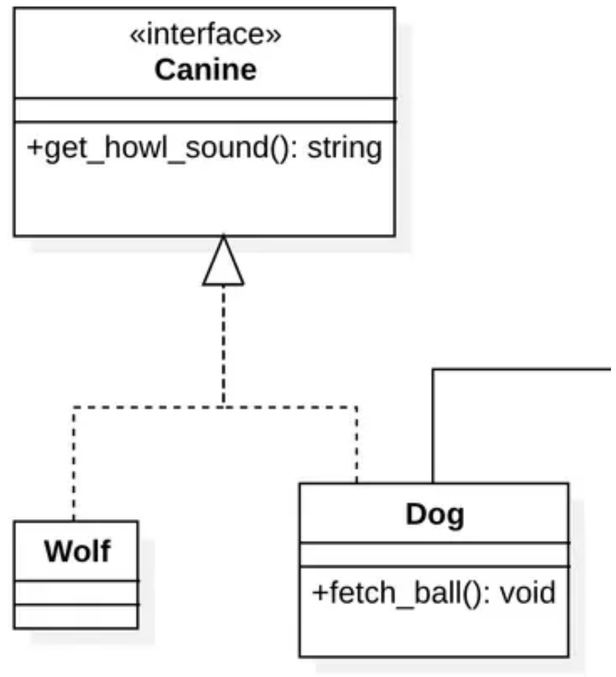
In UML kan een klasse getransformeerd worden naar een abstracte klasse als volgt:



De naam wordt *schuin* geschreven. Vaak wordt ook `<<abstract>>` ervoor genoteerd. Dit schema geeft aan dat van `Pet` zelf geen objecten gemaakt kunnen worden, enkel van de subklassen ervan.

UML: INTERFACE

Een interface in UML ziet er als volgt uit:



Een interface lijkt op een klasse, maar heeft `<<interface>>` ervoor staan.

Merk op dat `Dog` en `Wolf` de methode van `Canine` niet opnieuw vermelden. Ze nemen de methode impliciet over.

UML: ZICHTBAARHEID

Elk attribuut en methode heeft een bepaalde **zichtbaarheid**. In dit vak beperken we ons tot:

- **public**: overal zichtbaar. Aangegeven met `+`
- **private**: enkel zichtbaar binnen de klasse, nooit erbuiten. Aangegeven met `-`

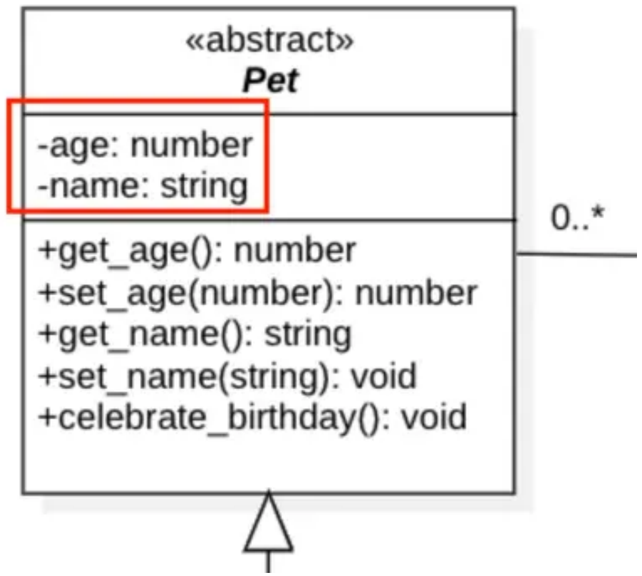
De algemeen gangbare regel is om attributen altijd private te maken.
Methodes mogen private of public zijn, dit hangt af van de vereisten.
Om attributen beschikbaar te maken voor de rest van het programma te maken, gebruik je getters en setters.

Afspraken voor dit vak:

- Je **moet** deze regel volgen wanneer je een OO-analyse uitvoert
- Het is aangeraden - maar niet verplicht - deze regel te volgen in andere OOP-opdrachten

UML: ZICHTBAARHEID TOEGEPAST

We zetten de attributen van `Pet` naar private. Er waren al publieke getters en setters toegevoegd in een eerdere stap, dus we moeten er geen bijmaken.



TIPS VOOR STARUML

- Om attributen en methodes toe te voegen aan een klasse:
  1. Dubbelklik op de klasse
  2. Gebruik het eerste icoon rechts om een attribuut toe te voegen, of het tweede icoon om een methode toe te voegen
- Om een abstracte klasse of interface te tekenen:
  1. Teken een gewone klasse
  2. Klik op de klasse
  3. Pas rechtsonderaan op het scherm de property **stereotype** aan naar **abstract** of **interface**
  4. Bij abstracte klassen moet je ook het vakje **isAbstract** aanvinken
- De pijl om interfaces te verbinden met klassen, vind je onder Classes (Advanced) > Realization
- Je kan de zichtbaarheid van een attribuut of methode aanpassen door erop te klikken en dan de property **visibility** aan te passen (icoon links).

OO-ANALYSE EN UML

INOEFENEN

PROBLEEMSTELLING

Op een kleuterschool wil een juf bijhouden wanneer de kinderen verjaren, op welke datum de kinderen ingeschreven zijn, en in welke kleuterklas ze zitten. Bouw een programma om de juf hierbij te helpen.

1. **Analyseer** de vereisten
2. **Ontwerp** een oplossing met behulp van UML
3. **Implementeer** de oplossing in Python

ANALYSE

Tijdens de analyse probeer je het probleem op te splitsen in aparte stukken (decompositie). Indien er volgens jou informatie ontbreek, probeer je hier een antwoord op te vinden, of je maakt bepaalde assumpties (die je duidelijk opschrijft). Het resultaat is een set van klasse, attributen, etc. waar je mee aan de slag kan.

Tips voor analyses:

- Zoek de zelfstandige naamwoorden in de probleemstelling. Let op gebruik van enkelvoud en meervoud, dit kan aangeven of iets een lijst gaat worden in de implementatie
- Schat in welke begrippen belangrijk zijn voor de oplossing van het probleem (abstractie). Veralgemeen/hernoem begrippen indien nodig (veralgemening)
- Bepaal hun rol: (abstracte) klasse, attribuut, methode, interface

Opdracht: pas dit toe op de probleemstelling van de juf.

ANALYSE: OPLOSSING

- Zelfstandige naamwoorden: kleuterschool, juf, kinderen, datum, kleuterklas
- Assumpties:
  - meerdere leerkrachten kunnen het systeem gebruiken
  - Een klas bestaat uit een leerjaar (cijfer) en een letter of andere code (string)
- Kernbegrippen: school, leerkracht, leerling, datum (dag, maand, jaar), klas
- Rollen:
  - Klassen: school, leerkracht, leerling, datum, klas
  - Attributen: dag, maand, jaar, klasjaar, klasnummer
  - Methodes: vraag verjaardag, vraag inschrijfdatum, vraag kleuterklas
  - Geen interfaces of abstracte klassen

ONTWERP

Nu we de benodigde informatie hebben, kan je het klassediagram tekenen. Gebruik StarUML om het te maken. Bekijk de vorige slides voor meer informatie omtrent het gebruik van UML en StarUML.

Een goed getekend klassediagram kan je geven aan eender welke programmeur die UML kent. Die zou dan in staat moeten zijn om de code te schrijven.

IMPLEMENTEER

Implementeer het klassediagram in Python. Je mag alles in één bestand schrijven, of je mag het opsplitsen en met `import` werken.

> **i** Om een attribuut of methode als private te markeren, laat je de naam ervan starten met `_`. Bijvoorbeeld: `_age`. Dit is een conventie specifiek voor Python.

Test je code uit: voldoet de oplossing aan de vereisten?

Opmerking: je hoeft geen uitgebreide user interface (UI) te programmeren of een hele reeks `input()` te schrijven om te testen. Creëer een simpele testsituatie bestaande uit een aantal objecten, voer een aantal methodes uit, en toon het resultaat op het scherm.

UML-OPDRACHT VOOR PORTFOLIO

DOEL

Op de volgende slides vind je een probleemomschrijving. Pas dezelfde algemene strategie toe als bij de vorige sectie:

1. Analyser het probleem: bepaal de vereisten van de oplossing, en splits de omschrijving op in aparte concepten die later klassen/attributen/methodes kunnen worden
2. Ontwerp een oplossing: maak een UML klassediagram op basis van de analyse. Maak gebruik van OO-concepten die we in de vorige lessen hebben gezien.
3. Implementeer de oplossing in Python. Hou rekening met de afspraken die gemaakt zijn in de vorige lessen omtrent objectgericht programmeren in Python.
4. Test het programma uit: creëer een aantal objecten en roep methodes op. Het resultaat moet voldoen aan de verwachtingen beschreven in de opgave.

Het maken van deze opdracht is een belangrijk deel van je portfolio. Zorg dat je opschrijft hoe je het probleem aanpakt. De analyse, eventuele assumpties, het klassediagram en de code moeten allemaal ingediend worden.

OPGAVE (GEBASEERD OP EXAMEN PROGRAMMEREN 2, JANUARI 2019)

Voor de finale van een muziekconcours viool moeten de kandidaten elk een sonate en een concerto spelen. Kijkers kunnen hun stem uitbrengen op de kandidaat van hun voorkeur. Een “gewone” kijker mag maar één keer stemmen, op één kandidaat. Een “expert” kijker mag twee keer stemmen, op één of twee verschillende kandidaten. In de stand worden kandidaten gerangschikt in overeenstemming met het aantal stemmen dat zij behaalden.

Het moet mogelijk zijn kandidaten in te schrijven voor de finale. Er moet een lijst kunnen getoond worden van de (al) ingeschreven kandidaten. In die lijst moet van elke kandidaat behalve zijn naam, ook de naam van de sonate en van het concerto dat hij/zij uitvoert, getoond worden. Ook de puntenstand moet op aanvraag op het scherm getoond worden en moet de namen van de kandidaten omvatten, evenals die van de gespeelde muziekstukken, en het aantal behaalde stemmen, en gerangschikt zijn in volgorde van afnemende hoeveelheid stemmen.

EXTRA INFORMATIE

Bij de evaluatie wordt rekening gehouden met:

- Ontwerp:
  - ontwerp en samenhang van klassen, en hun inhoud, volgens objectgerichte principes
  - duidelijkheid en functionaliteit van je ontwerp
- Implementatie:
  - correct en functioneel gebruik van controle- en gegevensstructuren, en eventueel recursie, in Python
  - implementatie van het OO-ontwerp in Python
- Gebruik en eventuele extra's
  - Correctheid en duidelijkheid van je voorbeeldcode waarmee je aantoont dat je oplossing voldoet aan de vereisten
  - Kwaliteit van de informatie (analyse, assumpties, commentaar in code, ...) die je toevoegt aan je oefening