FACULTEIT
WETENSCHAPPEN

TESTED: ONE JUDGE TO RULE THEM ALL

Niko Strijbol

Studentennummer: 01404620

Promotoren: prof. dr. Peter Dawyndt, dr. ir. Bart Mesuere
Begeleiding: Charlotte Van Petegem

Masterproef ingediend tot het behalen van de academische graad van
Master of Science in de informatica

Academiejaar: 2019 – 2020

UNIVERSITEIT
GENT

INHOUDSOPGAVE

1 INLEIDING

Programmeren -> steeds belangrijker en nuttigere kennis om te hebben Goed programmeren -> vereist veel oefening Zeker in cursussen met meer mensen -> goede ondersteuning lesgever Automatiseren van beoordeling programmeeroefeningen -> Dodona

TODO: eventueel samenvoegen met volgende hoofdstuk?

Dit is de inleiding van mijn thesis. Hallo aan iedereen! Omdat we met een initiaal werken, is het aangewezen dat de eerste alinea redelijk wat tekst bevat.

Verder is dit aan de orde:

$$y = 5 + 6$$

Hallo ik ben kleinkapitaal!

Hallo, dit is tekst met code: `dit is dan de code`.

Zeekomkommers (Holothuroidea) vormen een groep van ongewervelde dieren die behoren tot de klasse van stekelhuidigen. De meeste soorten hebben een langwerpig en worstvormig lichaam dat zowel aan de voor- als achterzijde stomp eindigt. Er zijn ook vormen met een sliertige of een bolle lichaamsvorm.

2 DODONA

2.1 Wat is Dodona?

Intro over Dodona: korte geschiedenis, terminologie, hoe Dodona werkt (oefeningen, judges, enz.)
-> Over de judge wordt in het deel hierna meer verteld.

2.2 Evalueren van een oplossing

Zoals reeds vermeld worden de oplossingen van studenten geëvalueerd door een zelfgeschreven evaluatiescript, de *judge*. In wezen is dit een eenvoudig programma: het krijgt de configuratie via de standaardinvoerstroom (stdin) en schrijft de resultaten van de evaluatie naar de standaarduitvoerstroom (stdout). Zowel de invoer als de uitvoer van de judge zijn json. Het formaat van deze uitwisseling ligt vast in een json-schema, dat publiekelijk beschikbaar is. [1]

Een judge ondersteunt één programmeertaal. In de praktijk ondersteunt elke judge oplossingen in de programmeertaal waarin hij geschreven is, m.a.w. de Java-judge ondersteunt Java, de Python-judge Python, enz. Ook heeft elke judge een eigen manier waarop de testen voor een oplossing opgesteld moeten worden. Zo worden in de Java-judge jUnit-testen gebruikt, terwijl de Python-judge doctests en een eigen formaat ondersteunt.

In grote lijnen verloopt het evalueren van een oplossing van een student als volgt:

1. De student dient de oplossing in via de webinterface van Dodona.
2. Dodona start een Docker-image met de judge.
3. De judge wordt uitgevoerd, met als invoer de configuratie, zoals hierboven vermeld.
4. De judge evalueert de oefening aan de hand van de code van de student en de evaluatiecode opgesteld door de lesgever (ie. de jUnit-test, de doctests, ...).
5. De judge vertaalt het resultaat van deze evaluatie naar het Dodona-formaat en schrijft het uit.
6. Dodona vangt die uitvoer op, en toont het resultaat aan de student.

2.3 Probleemstelling

Het huidige systeem waarop de judges werken resulteert in twee nadelen. Bij het bespreken van de nadelen is het nuttig een voorbeeld in het achterhoofd te houden, teneinde de nadelen te kunnen concretiseren. Als voorbeeld gebruiken we de „Lotto”-oefening[2], met volgende opgave:

[1] Een tekstuele beschrijving is te vinden in de handleiding (Dodona-team 2020).
[2] Vrij naar een oefening van prof. Dawyndt.

De **lotto** is een vorm van loterij die voornamelijk bekend is vanwege de genummerde balletjes, waarvan er een aantal getrokken worden. Deelnemers mogen zelf hun eigen nummers aankruisen op een lottoformulier. Hoe groter het aantal overeenkomstige nummers tussen het formulier en de getrokken balletjes, hoe groter de geldprijs.

Opgave

Schrijf een functie `loterij` waarmee een lottotrekking kan gesimuleerd worden. De functie moet twee parameters `aantal` en `maximum` hebben. Aan de parameter `aantal` kan doorgegeven worden hoeveel balletjes a er moeten getrokken worden (standaardwaarde 6). Aan de parameter `maximum` kan doorgegeven worden uit hoeveel balletjes m er moet getrokken worden (standaardwaarde 42). Beide parameters kunnen ook weggelaten worden, waarbij dan de standaardwaarde gebruikt moet worden. De balletjes zijn daarbij dus genummerd van 1 tot en met m. Je mag ervan uitgaan dat $a \leq m$. De functie moet een string teruggeven die een strikt stijgende lijst van a natuurlijke getallen beschrijft, waarbij de getallen van elkaar gescheiden zijn door een spatie, een koppelteken (-) en nog een spatie. Voor elk getal n moet gelden dat $1 \leq n \leq m$.

Oplossingen voor deze opgave staan in codefragmenten 2.1 en 2.2, voor respectievelijk Python en Java.

Het belangrijkste nadeel aan de huidige werking is het bijkomende werk voor lesgevers indien zij hun oefeningen in meerdere talen willen aanbieden. De Lotto-oefening heeft een eenvoudige opgave en oplossing. Bovendien zijn de verschillen tussen de versie in Python en Java minimaal, zij het dat de Java-versie wat langer is. Deze oefening zou zonder problemen in nog vele andere programmeertalen geïmplementeerd kunnen worden.

Ook bij ingewikkeldere oefeningen die zich concentreren op algoritmen, waar de uiteindelijke taal van de implementatie niet relevant is. Een voorbeeld hiervan is het vak „Algoritmen en Datastructuren" dat gegeven wordt door prof. Fack aan de wiskunde[3]. Daar zijn de meeste opgaven vandaag al beschikbaar in Java en Python op Dodona, maar dan als afzonderlijke oefeningen.

Het evalueren van een oplossing voor de Lotto-oefening is minder eenvoudig, daar er met willekeurige getallen gewerkt wordt: het volstaat niet om de uitvoer gegenereerd door de oplossing te vergelijken met een op voorhand vastgelegde verwachte uitvoer. De geproduceerde uitvoer zal moeten gecontroleerd worden met code, specifiek gericht op deze oefening, die de verwachte vereisten van de oplossing controleert. Deze evaluatiecode moet momenteel voor elke programmeertaal en dus elke judge opnieuw geschreven worden. In de context van ons voorbeeld controleert deze code bijvoorbeeld of de gegeven getallen binnen het bereik liggen en of ze gesorteerd zijn.

Voor de lesgevers is het opnieuw opstellen van deze evaluatiecode veel en repetitief werk. Duur het twee minuten om deze code te schrijven en een vak heeft 30 oefeningen, dan duurt het een uur. In twee talen duurt al twee uur, tien talen vraagt al tien uur.

Het tweede nadeel aan de huidige werking, maar wel veel kleiner van belang, betreft het implementeren van de judges zelf. Hoewel de interface voor de judges eenvoudig is, blijkt in de praktijk dat het implementeren van een judge verre van eenvoudig is. Uiteraard is een deel van die complexiteit ingevolge het evalueren van de code, een taak die niet in alle talen eenvoudig is. Doch is

[3]De studiefiche is voor de geïnteresseerden beschikbaar op https://studiegids.ugent.be/2019/NL/studiefiches/C002794.pdf

```java
import java.util.HashSet;
import java.util.Set;
import java.util.concurrent.ThreadLocalRandom;
import java.util.stream.Collectors;

class Main {

    public static String loterij(int aantal, int maximum) {
        var r = ThreadLocalRandom.current();
        var result = new HashSet<Integer>();
        while (result.size() < aantal) {
            result.add(r.nextInt(1, maximum + 1));
        }
        return result.stream()
                .sorted()
                .map(Object::toString)
                .collect(Collectors.joining(" - "));
    }

    public static String loterij(int aantal) {
        return loterij(aantal, 42);
    }

    public static String loterij() {
        return loterij(6, 42);
    }
}
```

Codefragment 2.1: Voorbeeldoplossing in Java.

```python
from random import randint


def loterij(aantal=6, maximum=42):
    getallen = set()
    while len(getallen) < aantal:
        getallen.add(randint(1, maximum))

    return " - ".join(str(x) for x in sorted(getallen))
```

Codefragment 2.2: Voorbeeldoplossing in Python.

de complexiteit ten dele te wijten aan de noodzaak om in elke judge opnieuw de hele evaluatieprocedure te implementeren. Aan het deel van de code dat het resultaat van een evaluatie omzet naar het Dodona-formaat is niets taalspecifieks.

Het probleem hierboven beschreven laat zich samenvatten als volgende onderzoeksvraag, waarop deze thesis een antwoord wil bieden:

> Is het mogelijk om een judge zo te implementeren, dat de opgave en evaluatiecode van een oefening slechts eenmaal opgesteld dienen te worden, waarna de oefening beschikbaar is in alle talen die de judge ondersteunt?

2.4 Opbouw

Het volgende hoofdstuk van deze thesis handelt over het antwoord op bovenstaande vraag. Daarna volgt ter illustratie van het gebruik van de judge een gedetailleerde beschrijving van hoe een nieuwe taal moet toegevoegd worden aan de thesis. Dat hoofdstuk heeft ook ten doel te dienen als documentatie voor zij die de judge willen gebruiken. Tot slot wordt afgesloten met een hoofdstuk over beperkingen van de huidige implementaties, en waar er verbeteringen mogelijk zijn (het „toekomstige werk").

3 DE UNIVERSELE JUDGE

Het antwoord op de onderzoeksvraag uit het vorige hoofdstuk manifesteert zich als de *universele judge*. Deze judge voor het Dodona-platform kan dezelfde oefening in meerdere talen evalueren. Dit hoofdstuk licht de werking en implementatie van deze judge toe, beginnend met een algemeen overzicht, waarna elk onderdeel in meer detail besproken wordt.

TODO: terminologie uitleggen (oplossing, evaluatie, ...) Een groot deel hiervan zal waarschijnlijk uitgelegd zijn bij de werking van Dodona.

3.1 Overzicht

Figuur 3.1 toont de opbouw van de judge op schematische wijze. De judge kan worden opgedeeld in drie gebieden, naar hun functionaliteit:

1. Het evaluatieproces, dat de verkregen resultaten interpreteert en beoordeelt.
2. Het kernproces, dat zorgt voor de coördinatie tussen de andere processen, alsook de basistaken vervult. Dit proces is dan ook het start- en eindpunt van een evaluatie.
3. Het uitvoeringsproces, dat de code van de student uitvoert om zo resultaten te bekomen.

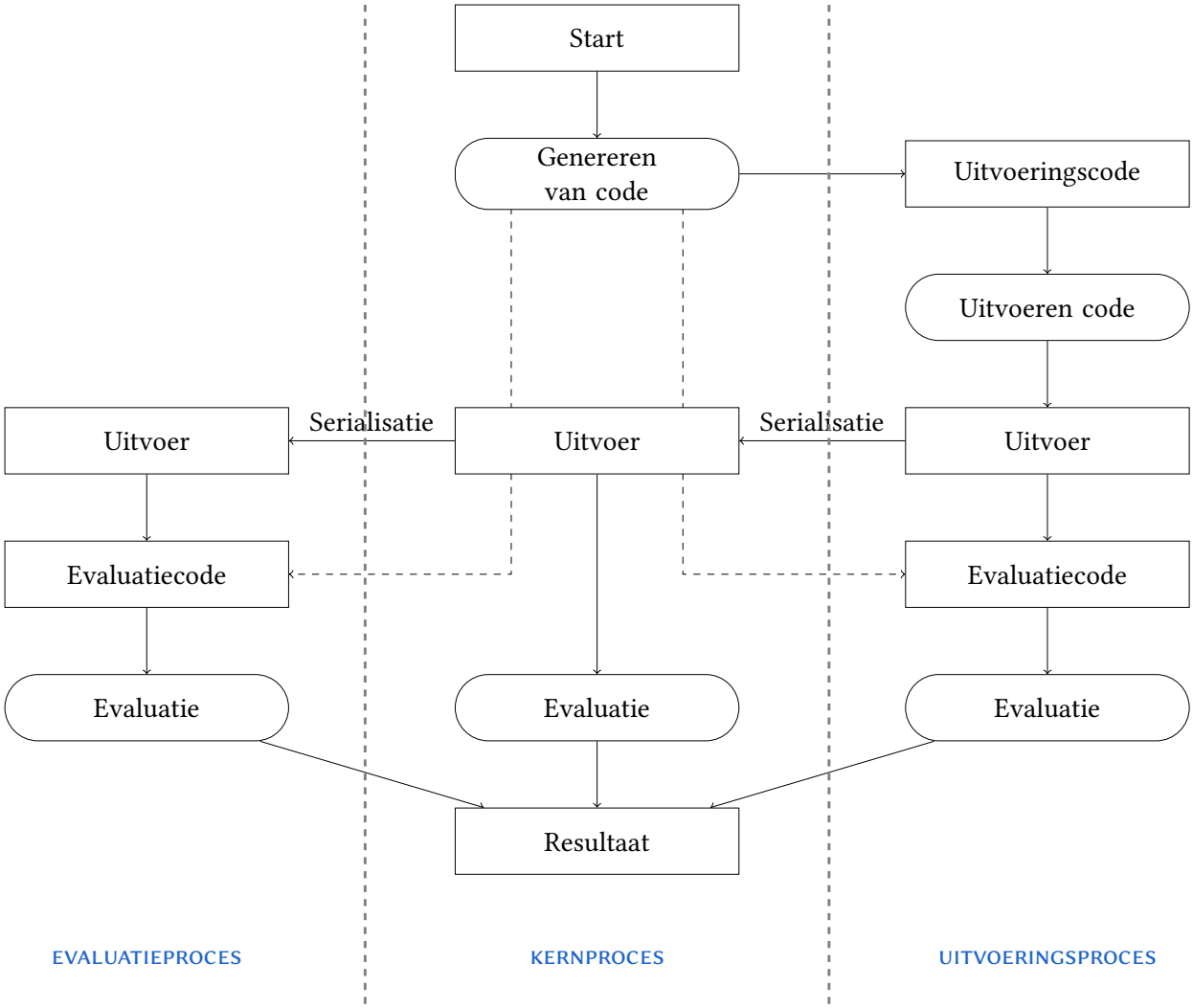
Daarnaast is het *testplan* ook van belang, dat de evaluatiecode definieert.

3.2 Beschrijving van een oefening

Uitleg over het testplan.

3.3 Uitvoeren van de oplossing

3.4 Evalueren van een oplossing



Figuur 3.1: Schematische voorstelling van de opbouw van de universele judge.

4 CASE-STUDY: TOEVOEGEN VAN EEN TAAL

Allerlei uitleg

5 BEPERKINGEN EN TOEKOMSTIG WERK

Wat kunnen we al en vooral wat niet? Waar kan nog aan gewerkt worden?

Korte samenvatting

5.1 Performance

-> Uitleg over eerste implementatie met jupyter kernels -> Uitleg over verschillende stadia van codegeneratie (alles apart -> zoveel mogelijk samen)

5.2 Functies

-> Dynamisch testplan -> Herhaalde uitvoeringen

BIBLIOGRAFIE

Dodona-team (23 oktober 2020). *Creating a new Judge*. Universiteit Gent. URL: https://dodona-edu.github.io/en/guides/creating-a-judge/.