FACULTEIT
WETENSCHAPPEN

TESTed: one judge to rule them all

Een universele judge voor het beoordelen van software in een educative context

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

Dankwoord

Dank aan iedereen!

1 Educational software testing

De evoluties op technologisch vlak hebben ervoor gezorgd dat onze maatschappij de laatste decennia in hoge mate gedigitaliseerd is, een proces dat nog steeds aan de gang is. Bovendien kan, door de snelheid waarmee deze veranderingen vaak optreden, eerder gesproken worden van een revolutie dan een evolutie: de veranderingen zijn vaak ingrijpend en veranderen fundamentele aspecten van de sectoren waarin de digitalisering plaatsvindt. Dit gaat over het ontstaan van nieuwe sectoren, zoals de deeleconomie, maar ook ingrijpende veranderingen bij bestaande sectoren, zoals de opkomst van *ride sharing* in de taxisector. Ook de impact op maatschappelijk vlak, zoals de sociale media in de politiek, mag niet vergeten worden [12].

Ook op educatief vlak heeft de digitalisering een grote impact. Enerzijds biedt digitalisering nieuwe mogelijkheden aan voor onderwijsdoeleinden, zoals het lesgeven op afstand, het online aanbieden van leermateriaal en het online indienen en verbeteren van opdrachten.

Anderzijds biedt het ook uitdagingen: om studenten voor te bereiden op de steeds digitalere maatschappij is een basis van digitale geletterdheid nodig. Net door de snelle evolutie op technologisch vlak volstaat het niet om studenten te leren werken met de technologie van vandaag; een grondige kennis van de onderliggende werking van de technologie is onontbeerlijk.

Een belangrijk aspect hierin is het concept van *computationeel denken*. Dat het aanleren van digitale vaardigheden nodig is, bewijst ook de opname van dat computationeel denken in de eindtermen, bijvoorbeeld in het katholieke basisonderwijs [33] of in het secundair onderwijs [11].

1.1 Computationeel denken

Op de vraag wat computationeel denken nu precies betekent lopen de antwoorden uiteen. Het Departement Onderwijs en Vorming van de Vlaams Overheid [4] definieert de term als volgt:

> *Computationeel denken verwijst dus naar het menselijke vermogen om complexe problemen op te lossen en daarbij computers als hulpmiddel te zien. Met andere woorden, computationeel denken is het proces waarbij aspecten van informaticawetenschappen herkend worden in de ons omringende wereld, en waarbij de methodes en technieken uit de informaticawetenschappen toegepast worden om problemen uit de fysische en virtuele wereld te begrijpen en op te lossen.*

Computationeel denken is dus ruimer dan programmeren, maar programmeren vormt wel een uitstekende manier om het computationeel denken aan te leren en te oefenen. Bovendien is programmeren op zich ook een nuttige vaardigheid om studenten aan te leren.

1.2 Programmeeroefeningen

Het aanleren van programmeren is niet eenvoudig en wordt door veel studenten als moeilijk ervaren [22]. Het maken van oefeningen kan daarbij helpen, indachtig het spreekwoord „oefening baart kunst". Studenten veel oefeningen laten maken, resulteert wel in twee uitdagingen voor de lesgevers:

1. Lesgevers moeten geschikte oefeningen opstellen, die rekening houden met welke programmeerconcepten studenten al kennen, tijdslimieten, moeilijkheidsgraden, enz. Het opstellen van deze oefeningen vraagt veel tijd.
2. De oplossingen voor deze oefeningen moeten voorzien worden van kwalitatieve feedback. Bij het aanleren van programmeren is feedback een belangrijk aspect om de programmeervaardigheden van de studenten te verbeteren [18].

Deze thesis focust op de eerste uitdaging, al wordt ook ingespeeld op de tweede uitdaging, onder andere in paragraaf 2.7. De uitdaging wordt beschouwd binnen de context van Dodona, een online leerplatform voor automatische feedback op ingediende oplossingen voor programmeeroefeningen.

1.3 Het leerplatform Dodona

Sinds 2011 wordt aan de onderzoeksgroep Computationele Biologie van de Universiteit Gent gewerkt met programmeeroefeningen die in een online systeem ingediend en beoordeeld worden. Oorspronkelijk werd hiervoor gebruik gemaakt van de *Sphere Online Judge* (SPOJ) [20]. Op basis van ervaringen met SPOJ ontwikkelde de onderzoeksgroep een eigen leerplatform, Dodona, dat in september 2016 beschikbaar werd. Het doel van Dodona is eenvoudig: lesgevers bijstaan om hun studenten niet alleen zo goed mogelijk te leren programmeren, maar dit ook op een zo efficiënt mogelijke manier te doen.

Het online leerplatform Dodona kan opgedeeld worden in verschillende onderdelen:

1. Het **leerplatform** zelf is een webapplicatie, die verantwoordelijk is om alle modules samen te laten werken en die ook de webinterface aanbiedt die de studenten en lesgevers gebruiken. Het is via deze interface dat lesgevers oefeningen beschikbaar maken en dat studenten hun oplossingen indienen. Het platform zelf is programmeertaalonafhankelijk geschreven.
2. De **judges** binnen Dodona zijn verantwoordelijk voor het beoordelen van ingediende oplossingen. Dit onderdeel wordt uitgebreid besproken in paragraaf 1.4.
3. De **oefeningen** worden niet in Dodona zelf aangemaakt, maar worden door de lesgever aangeleverd via een git-repository. De oefeningen bevatten de beschrijving van de opgave en de testen die uitgevoerd worden tijdens het automatisch beoordelen van een oplossing.

Met Dodona kunnen lesgevers een leertraject opstellen door een reeks oefeningen te selecteren. Studenten die dit leertraject volgen, zien onmiddellijk hun voortgang binnen het traject. Bij het indienen van hun oplossingen ontvangen de studenten ook onmiddellijk feedback over hun oplossing: deze feedback bevat niet alleen de correctheid van de oplossing, maar kan ook andere aspecten belichten, zoals de kwaliteit (bv. de programmeerstijl), het resultaat van het uitvoeren van linters, en de performantie van de oplossing.

1.4 Beoordelen van oplossingen

In Dodona wordt elke ingediende oplossing beoordeeld door een evaluatieprogramma, de *judge*. In wezen is dit een eenvoudig programma: via de standaardinvoerstroom (`stdin`) krijgt het programma een configuratie binnen van Dodona. Deze configuratie bevat onder andere de programmeertaal van de ingediende oplossing, de locatie van de oefeningenbestanden (die opgesteld zijn door de lesgever), de locatie van de ingediende oplossing zelf en configuratieopties, zoals geheugen- en tijdslimieten. Het resultaat van de beoordeling wordt uitgeschreven naar de standaarduitvoerstroom (`stdout`). Zowel de invoer als de uitvoer van de judge zijn JSON, waarvan het formaat vastgelegd is in JSON Schema.[1]

Concreet wordt elke beoordeling uitgevoerd in een Docker-container. Deze Docker-container wordt gemaakt op basis van een Docker-image die bij de judge hoort, en alle dependencies bevat die de judge in kwestie nodig heeft. Bij het uitvoeren van de beoordeling zal Dodona een *bind mount*[2] voorzien, zodat de code van de judge zelf, de code van de oefening en de code van de ingediende oplossing beschikbaar zijn in de container. Via de configuratie geeft Dodona aan de judge aan waar deze bestanden zich bevinden.

Samenvattend bestaat interface tussen de judge en Dodona uit drie onderdelen:

1. De judge zal uitgevoerd worden in een Docker-container, dus een Docker-image met alle dependencies moet voorzien worden. Deze Docker-image moet ook de judge opstarten.
2. De judge stelt de invoer van een beoordeling ter beschikking voor de judge. Bestanden worden via een bind mount aan de Docker-container gekoppeld. De paden naar deze bestanden binnen de container en andere informatie (zoals programmeertaal van de oplossing of natuurlijke taal van de gebruiker) worden via de configuratie aan de judge gegeven (via de standaardinvoerstroom).
3. De judge moet het resultaat van zijn beoordeling uitschrijven naar de standaarduitvoerstroom, in een vastgelegd formaat.

Buiten deze interface legt Dodona geen vereisten op aan de werking van judge. Door deze vrijheid lopen de manieren waarop de bestaande judges geïmplementeerd zijn uiteen. Sommige judges beoordelen oplossingen in dezelfde programmeertaal als de taal waarin ze geschreven zijn. Zo is de judge voor Python-oplossingen geschreven in Python en de judge voor Java-oplossingen in Java. Bij andere judges is dat niet het geval: de judges voor Bash en Prolog zijn bijvoorbeeld ook in Python geschreven. Daarnaast heeft elke judge een eigen manier waarop de testen voor een beoordeling van een oplossing opgesteld moeten worden. Zo worden in de Java-judge jUnit-testen gebruikt, terwijl de Python-judge doctests (en een eigen formaat) ondersteunt.

De beoordeling van een oplossing van een student verloopt als volgt:

1. De student dient de oplossing in via de webinterface van Dodona.
2. Dodona start een Docker-container voor de judge.
3. Dodona voorziet de container van de bestanden van de judge, de oefening en de ingediende oplossing.
4. De judge wordt uitgevoerd met de configuratie als op de standaardinvoerstroom.

[1] Dit schema en een tekstuele beschrijving ervan is te vinden in de handleiding op https://dodona-edu.github.io/en/guides/creating-a-judge/.

[2] Informatie over deze term is te vinden op https://docs.docker.com/storage/bind-mounts/

5. De judge beoordeelt de oplossing aan de beoordelingsmethodes opgesteld door de lesgever (d.w.z. de jUnit-test, de doctests, ...). Judges kunnen ook bijkomende taken uitvoeren, zoals linting, beoordeling van de performantie of *grading* van de code van de oplossing.
6. De judge vertaalt zijn beoordeling naar het Dodona-formaat en schrijft het resultaat naar de standaarduitvoerstroom.
7. Dodona slaat dat resultaat op in de databank.
8. Op de webinterface krijgt de student het resultaat te zien als feedback op de ingediende oplossing.

1.5 Probleemstelling

De manier waarop de huidige judges werken resulteert in twee belangrijke nadelen. Bij het bespreken hiervan is het nuttig een voorbeeld in het achterhoofd te houden, teneinde de nadelen te kunnen concretiseren. Als voorbeeld gebruiken we de „Lotto"-oefening[3], waarvan de opgave gegeven is in codefragment 1.1. Oplossingen voor deze oefening staan in codefragmenten 1.2 en 1.3, voor respectievelijk Python en Java.

Opstellen van oefeningen

Het eerste en belangrijkste nadeel aan de werking van de huidige judges heeft betrekking op de lesgevers en komt voor als zij een oefening willen aanbieden in meerdere programmeertalen. Enerzijds is dit een zware werklast: de oefening, en vooral de code voor de beoordeling, moet voor elke judge opnieuw geschreven worden. Voor de Python-judge zullen doctests nodig zijn, terwijl de Java-judge jUnit-testen vereist. Anderzijds lijdt dit ook tot verschillende versies van dezelfde oefening, wat het onderhouden van de oefeningen moelijker maakt. Als er bijvoorbeeld een fout sluipt in de beoordelingscode, zal de lesgever er aan moeten denken om de fout te verhelpen in alle varianten van de oefening. Bovendien geeft elke nieuwe versie van de oefening een nieuwe mogelijkheid voor het introduceren van fouten.

Kijkt men naar de Lotto-oefening, dan valt op dat het gaat om een eenvoudige opgave en een eenvoudige oplossing. Bovendien zijn de verschillen tussen oplossingen in verschillende programmeertalen niet zo groot. In de voorbeeldoplossingen in Python en Java zijn de verschillen minimaal, zij het dat de Java-oplossing wat langer is. De Lotto-oefening zou zonder problemen in nog vele andere programmeertalen opgelost kunnen worden. Eenvoudige programmeeroefeningen, zoals de Lotto-oefening, zijn voornamelijk nuttig in twee gevallen: studenten die voor het eerst leren programmeren en studenten die een nieuwe programmeertaal leren. In het eerste geval is de eigenlijke programmeertaal minder relevant: het zijn vooral de concepten die belangrijk zijn. In het tweede geval is de programmeertaal wel van belang, maar moeten soortgelijke oefeningen gemaakt worden voor elke programmeertaal die aangeleerd moet worden. In beide gevallen is het dus een meerwaarde om de oefening in meerdere programmeertalen aan te bieden.

Eenzelfde constatatie kan gemaakt worden bij meer complexe oefeningen die zich concentreren op algoritmen: ook daar zijn de concepten belangrijker dan in welke programmeertaal een algoritme

[3] Vrij naar een oefening van prof. Dawyndt. De originele oefening is beschikbaar op https://dodona.ugent.be/nl/exercises/2025591548/

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` (`int`) kan doorgegeven worden hoeveel balletjes a er moeten getrokken worden (standaardwaarde 6). Aan de parameter `maximum` (`int`) 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 $1 \leq a \leq m$. De functie moet een string (`str`) teruggeven die een strikt stijgende lijst (`list`) van a natuurlijke getallen (`int`) beschrijft, waarbij de getallen van elkaar gescheiden zijn door een spatie, een koppelteken (-) en nog een spatie. Voor elk getal n moet gelden dat $1 \leq n \leq m$.

Voorbeeld

```
> loterij()
'2 - 17 - 22 - 27 - 35 - 40'
> loterij(aantal=8)
'5 - 13 - 15 - 31 - 34 - 36 - 39 - 40'
> loterij(aantal=4, maximum=38)
'16 - 20 - 35 - 37'
```

Codefragment 1.1: De opgave van de voorbeeldoefening, Lotto.

```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 1.2: Oplossing in Python voor de voorbeeldoefening Lotto.

```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 1.3: Oplossing in Java voor de voorbeeldoefening Lotto.

uiteindelijk geïmplementeerd wordt. Een voorbeeld hiervan is het vak „Algoritmen en Datastructuren", dat gegeven wordt door prof. Fack binnen de opleiding wiskunde[4]. Daar zijn de meeste opgaven op Dodona vandaag al beschikbaar in de programmeertalen Java en Python, maar dan als afzonderlijke en onafhankelijke oefeningen.

Een ander aspect is de beoordeling van een oefening. Voor de Lotto-oefening is de beoordeling niet triviaal, door het gebruik van niet-deterministische functies. Het volstaat voor dit soort oefeningen niet om de uitvoer geproduceerd 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 de Lotto-oefening controleert deze code bijvoorbeeld of de gegeven getallen binnen het bereik liggen en of ze gesorteerd zijn. Toch is deze evaluatiecode niet inherent programmeertaalafhankelijk: controleren of een lijst gesorteerd is, heeft weinig te maken met de programmeertaal van de oplossing.

Implementeren van judges

Een tweede nadeel aan de huidige werking zijn de judges zelf: voor elke programmeertaal die men wil aanbieden in Dodona moet een nieuwe judge ontwikkeld worden. Ook hier is er dubbel werk: dezelfde concepten en features, die eigenlijk programmeertaalonafhankelijk zijn, moeten in elke judge opnieuw geïmplementeerd worden. Hierbij denken we aan bijvoorbeeld de logica om te bepalen wanneer een beoordeling positief of negatief moet zijn.

Onderzoeksvraag

Het eerste nadeel wordt beschouwd als het belangrijkste nadeel en de focus van deze thesis. Het nadeel valt te formuleren als de onderzoeksvraag waarop deze thesis een antwoord wil bieden:

> Is het mogelijk om een judge zo te implementeren dat de opgave en beoordelingsmethoden van een oefening slechts eenmaal opgesteld dienen te worden, waarna de oefening beschikbaar is in alle programmeertalen die de judge ondersteunt? Hierbij is het wenselijk dat eens een oefening opgesteld is, deze niet meer gewijzigd moet worden wanneer talen toegevoegd worden aan de judge.

Als bijzaak is het ook interessant om na te gaan of het antwoord op de onderzoeksvraag een voordeel kan bieden voor het implementeren van judges zelf.

De aandachtige lezer zal opmerken dat de opgave voor de Lotto-oefening (codefragment 1.1) programmeertaalspecifieke en taalspecifieke elementen bevat. Zo zijn de voorbeelden in Python en zijn de namen van functies en argumenten in het Nederlands. Het ondersteunen van opgaves met programmeertaalonafhankelijke voorbeelden en vertalingen wordt voor deze thesis expliciet als *out-of-scope* gezien en zal niet behandeld worden, zij het in paragraaf 5.3.3.

[4]De studiefiche is beschikbaar op https://studiegids.ugent.be/2019/NL/studiefiches/C002794.pdf

1.6 Opbouw van de thesis

Hoofdstuk 2 handelt over het antwoord op bovenstaande onderzoeksvraag, waar een prototype van een dergelijke judge wordt voorgesteld. Daarna volgt ter illustratie een gedetailleerde beschrijving van hoe een oefening opgesteld moet worden voor dit prototype. Nadien volgt een beschrijving van hoe een nieuwe programmeertaal moet toegevoegd worden aan het prototype. Daar deze twee hoofdstukken voornamelijk ten doel hebben zij die met het prototype (zoals lesgevers) moeten werken te informeren, nemen deze hoofdstukken de vorm aan van meer traditionele softwarehandleidingen. Tot slot volgt met een hoofdstuk over beperkingen van de huidig prototype, en waar er verbeteringen mogelijk zijn (het „toekomstige werk").

2 TESTed

In het kader van deze masterproef werd een prototype geïmplementeerd van een judge voor Dodona. Het doel hiervan is een antwoord te bieden aan de onderzoeksvraag uit het vorige hoofdstuk en de beperkingen van deze aanpak in kaart te brengen. Deze judge heeft de naam *TESTed* gekregen. Bij TESTed is een oefening programmeertaalonafhankelijk en kunnen oplossingen in verschillende programmeertalen beoordeeld worden aan de hand van een en dezelfde specificatie. Dit hoofdstuk begint met het ontwerp en de algemene werking van de judge toe te lichten, waarna elk onderdeel in meer detail besproken wordt.

2.1 Overzicht

2.1.1 Architecturaal ontwerp

Figuur 2.1 toont het architecturaal ontwerp van TESTed. De twee stippellijnen geven programmeertaalbarrières aan, en verdelen TESTed in drie logisch omgevingen:

1. TESTed zelf is geschreven in Python: in het middelste deel staat de programmeertaal dus vast. Dit onderdeel is verantwoordelijk voor de regie van de beoordeling op basis van het testplan.
2. De ingediende oplossing wordt uitgevoerd in de *uitvoeringsomgeving*, waar de programmeertaal overeenkomt met de programmeertaal van de oplossing.
3. Tot slot is er nog de *evaluatieomgeving*, waar door de lesgever geschreven evaluatiecode wordt uitgevoerd. Deze moet niet in dezelfde programmeertaal als de oplossing of TESTed geschreven zijn.

2.1.2 Stappenplan van een beoordeling

De rest van het hoofdstuk bespreekt alle onderdelen van en stappen die gebeuren bij een beoordeling van een ingediende oplossing in detail. In figuur 2.2 zijn deze stappen gegeven als een flowchart, en een uitgeschreven versie volgt:

1. De Docker-container voor TESTed wordt gestart. Dodona stelt de invoer ter beschikking aan de container: het testplan komt uit de oefening, terwijl de ingediende oplossing en de configuratie uit Dodona komen.
2. Als eerste stap wordt gecontroleerd dat het testplan de programmeertaal van de ingediende oplossing ondersteunt. De programmeertaal van de oplossing wordt gegeven via de configuratie uit Dodona. Merk op dat de ingediende oplossing zelf hierbij niet nodig is: deze controle zou idealiter gebeuren bij het importeren van de oefening in Dodona, zodat Dodona weet in welke programmeertalen een bepaalde oefening aangeboden kan worden (zie

hoofdstuk 5). Als het testplan bijvoorbeeld programmeertaalspecifieke code bevat die enkel in Java geschreven is, zal een oplossing in Python niet beoordeeld kunnen worden. Bevat het testplan bijvoorbeeld een functie die een verzameling moet teruggeven, dan zullen talen als Bash niet in aanmerking komen.

3. Het testplan (details in paragraaf 2.2.1) bestaat uit verschillende contexten. Elke context is een onafhankelijke uitvoering van de ingediende oplossing en kan verschillende aspecten van die uitvoering beoordelen. Voor elk van die contexten wordt in deze stap de testcode gegenereerd. Deze stap is de overgang naar de *uitvoeringsomgeving*.

4. De testcode wordt optioneel gecompileerd. Dit kan op twee manieren gebeuren (details in paragraaf 2.3.1):

   a) Batchcompilatie: hierbij wordt de testcode van alle contexten verzameld en gecompileerd tot één uitvoerbaar bestand (executable). Dit heeft als voordeel dat er slechts een keer een compilatie nodig is, wat voor een betere performantie zorgt. Bij deze manier resulteert de compilatiestap in één uitvoerbaar bestand.

   b) Contextcompilatie: hierbij wordt de testcode voor elke context afzonderlijk gecompileerd tot een uitvoerbaar bestand. Bij deze manier worden er n uitvoerbare bestanden geproduceerd tijdens de compilatiestap.

   In talen die geen compilatie nodig hebben of ondersteunen, wordt deze stap overgeslagen.

5. Nu kan het uitvoeren van de beoordeling zelf beginnen: de gegenereerde code wordt uitgevoerd (nog steeds in de uitvoeringsomgeving). Elke context uit het testplan wordt in een afzonderlijk subproces uitgevoerd, teneinde het delen van informatie tegen te gaan.

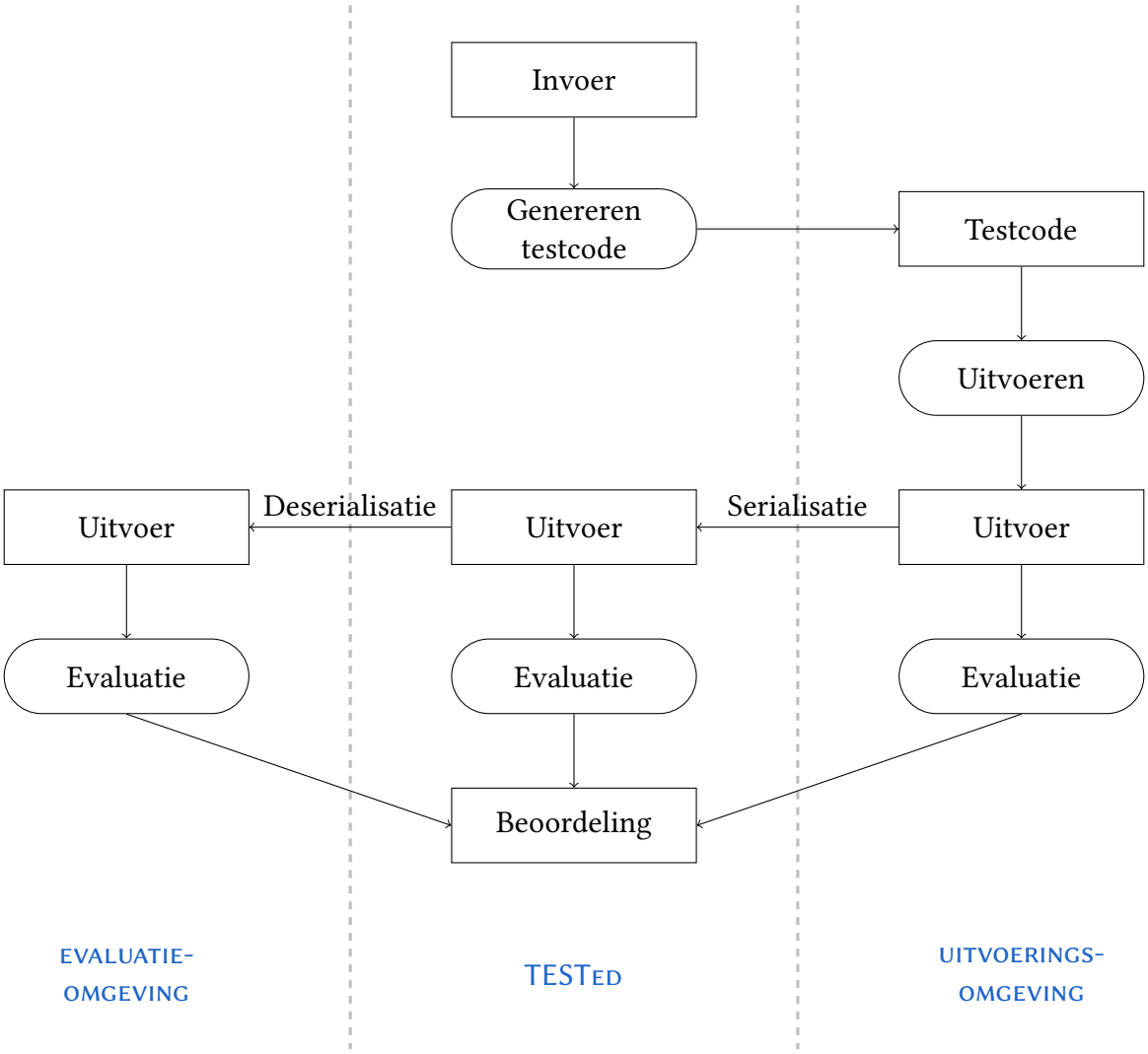
6. De uitvoering van de executable in de vorige stap produceert resultaten (voor elke context), zoals de standaarduitvoerstroom, de standaardfoutstroom, returnwaardes, exceptions of exitcodes. Deze bundel resultaten wordt nu geëvalueerd op juistheid. Hiervoor zijn drie mogelijke manieren:

   a) Programmeertaalspecifieke evaluatie (afgekort tot SE in de flowchart). De code voor de evaluatie is opgenomen in de executable en wordt onmiddellijk uitgevoerd in hetzelfde proces. Via deze mogelijkheid kunnen taalspecifieke aspecten gecontroleerd worden. Daar de evaluatie in hetzelfde proces gebeurt, blijft dit in de uitvoeringsomgeving.

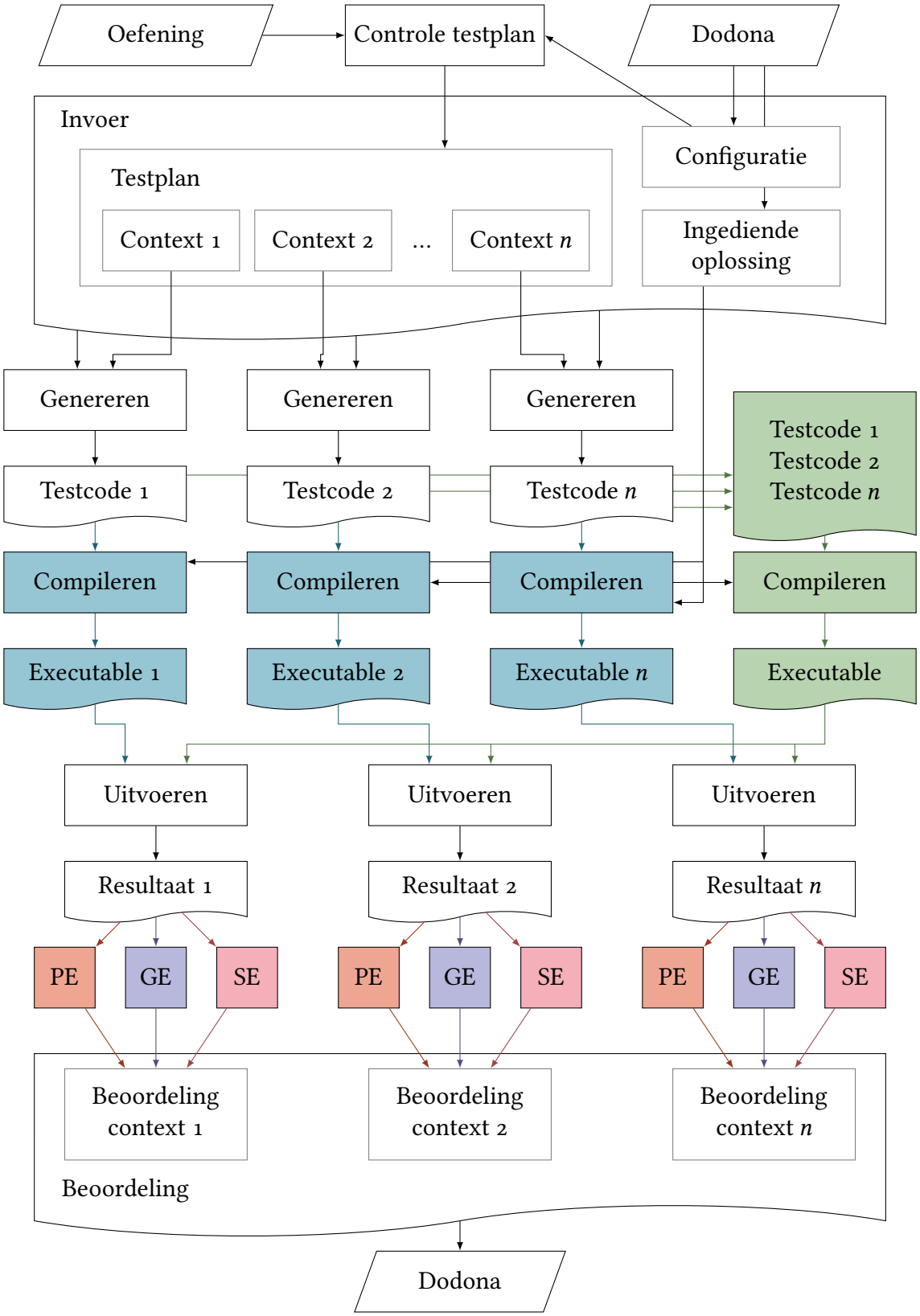
   b) Geprogrammeerde evaluatie (afgekort tot PE in de flowchart). Hierbij is er evaluatiecode geschreven die los staat van de oplossing, waardoor deze evaluatiecode ook in een andere programmeertaal geschreven kan zijn. De code ter uitvoering van de geprogrammeerde evaluatiecode wordt gegenereerd en dan uitgevoerd. Het doel van deze modus is om complexe evaluaties toe te laten op een programmeertaalonafhankelijke manier. Deze stap vindt plaats in de evaluatieomgeving.

   c) Generieke evaluatie. Hierbij evalueert TESTed zelf het resultaat. Deze modus is bedoeld voor gestandaardiseerde evaluaties, zoals het vergelijken van geproduceerde uitvoer en verwachte uitvoer. Hier gebeurt de evaluatie binnen TESTed zelf.

7. Tot slot verzamelt TESTed alle evaluatieresultaten en stuurt ze gebundeld door naar Dodona, waarna ze getoond worden aan de gebruiker.



Figuur 2.1: Schematische voorstelling van het architecturale ontwerp van de TESTed.



Figuur 2.2: Flowchart van een beoordeling door TESTed. In het schema worden kleuren gebruikt als er een keuze gemaakt moet worden voor een volgende stap. Er kan steeds slechts één mogelijkheid gekozen worden. De afkortingen PE, GE en SE staan respectievelijk voor geprogrammeerde evaluatie, generieke evaluatie en (programmeertaal)specifieke evaluatie.

2.2 Beschrijven van een oefening

De beoordeling van een ingediende oplossing van een oefening begint bij de invoer die TESTed krijgt. Centraal in deze invoer is een *testplan*, een specificatie die op een programmeertaalonafhankelijke manier beschrijft hoe een oplossing voor een oefening beoordeeld moet worden. Het vervangt de taalspecifieke testen van de bestaande judges (ie. de jUnit-tests of de doctests in respectievelijk Java en Python). Het testplan *sensu lato* wordt opgedeeld in verschillende onderdelen, die hierna besproken worden.

2.2.1 Het testplan

Het testplan *sensu stricto* beschrijft de structuur van de beoordeling van een ingediende oplossing voor een oefening. Deze structuur lijkt qua opbouw sterk op de structuur van de feedback zoals gebruikt door Dodona. Dat de structuur van de oplossing in Dodona en van het testplan op elkaar lijken, heeft als voordeel dat er geen mentale afbeelding moet gemaakt worden tussen de structuur van het testplan en dat van Dodona. Concreet is de structuur een hiërarchie met volgende elementen:

Plan Het top-level object van het testplan. Dit object bevat twee belangrijke objecten: de tabbladen en de configuratie. Deze configuratie is de plaats om opties aan TESTed mee te geven.

Tab Een testplan bestaat uit verschillende *tab*s of tabbladen. Deze komen overeen met de tabbladen in de gebruikersinterface van Dodona. Een tabblad kan een naam hebben, die zichtbaar is voor de gebruikers.

Context Elk tabblad bestaat uit een of meerdere *context*en. Een context is een onafhankelijke uitvoering van een evaluatie. De nadruk ligt op de „onafhankelijkheid", zoals al vermeld. Elke context wordt in een nieuw proces en in een eigen map (directory) uitgevoerd, zodat de kans op het delen van informatie klein is. Hierbij willen we vooral onbedoeld delen van informatie (zoals statische variabelen of het overschrijven van bestanden) vermijden. De gemotiveerde student zal nog steeds informatie kunnen delen tussen de uitvoeringen, door bv. in een andere locatie een bestand aan te maken en later te lezen.

Testcase Een context bestaat uit een of meerdere *testcase*s of testgevallen. Een testgeval bestaat uit invoer en een aantal tests. De testgevallen kunnen onderverdeeld worden in twee soorten:

- **Main testcase** of hoofdtestgeval. Van deze soort is er maximaal één per context (geen hoofdtestgeval is ook mogelijk). Dit testgeval heeft als doel het uitvoeren van de mainfunctie (of de code zelf als het gaat om een scripttaal zoals Bash of Python). Als invoer voor dit testgeval kunnen enkel de standaardinvoerstroom en de programmaargumenten meegegeven worden. De exitcode van een uitvoering kan ook enkel in het hoofdtestgeval gecontroleerd worden.
- **Normal testcase** of normaal testgeval. Hiervan kunnen er nul of meer zijn per context. Deze testgevallen dienen om andere aspecten van de ingediende oplossing te testen, nadat de code van de gebruiker met success ingeladen is. De invoer is dan ook uitgebreider: het kan gaan om het standaardinvoerkanaal, functieoproepen en variabeletoekenningen. Een functieoproep of variabeletoekenning is verplicht (zonder functieoproep of toekenning aan een variabele is er geen code om te testen).

Het hoofdtestgeval wordt altijd als eerste uitgevoerd. Dit is verplicht omdat bepaalde programmeertalen (zoals Python en andere scripttalen) de code onmiddellijk uitvoeren bij het inladen. Om te vermijden dat de volgorde van de testgevallen zou verschillen tussen de programmeertalen, wordt het hoofdtestgeval altijd eerst uitgevoerd.

Test De beoordeling van een testgeval bestaat uit meerdere *test*s, die elk één aspect van het testgeval controleren. Met aspect bedoelen we de standaarduitvoerstroom, de standaardfoutstroom, opgevangen uitzonderingen (*exceptions*), de teruggegeven waarden van een functieoproep (returnwaarden) of de inhoud van een bestand. De exitcode is ook mogelijk, maar enkel in het hoofdtestgeval. Het beoordelen van de verschillende aspecten wordt in meer detail beschreven in paragraaf 2.4

Bij de keuze voor een formaat voor het testplan (JSON, XML, ...), zijn vooraf enkele vereisten geformuleerd waaraan het gekozen formaat moet voldoen. Het moet:

- leesbaar zijn voor mensen,
- geschreven kunnen worden met minimale inspanning, met andere woorden de syntaxis dient eenvoudig te zijn, en
- programmeertaalonafhankelijk zijn.

Uiteindelijk is gekozen om het testplan op te stellen in JSON. Niet alleen voldoet JSON aan de vooropgestelde voorwaarden, het wordt ook door veel talen ondersteund.

Toch zijn er ook enkele nadelen aan het gebruik van JSON. Zo is JSON geen beknopte of compacte taal om met de hand te schrijven. Een oplossing hiervoor gebruikt de eigenschap dat veel talen JSON kunnen produceren: andere programma's kunnen desgewenst het testplan in het json-formaat genereren, waardoor het niet met de hand geschreven moet worden. Hiervoor denken we aan een DSL (*domain specific language*), maar dit valt buiten de thesis en wordt verder besproken in hoofdstuk 5.

Een tweede nadeel is dat JSON geen programmeertaal is. Terwijl dit de implementatie van de judge bij het interpreteren van het testplan weliswaar eenvoudiger maakt, is het tevens beperkend: beslissen of een testgeval moet uitgevoerd worden op basis van het resultaat van een vorig testgeval is bijvoorbeeld niet mogelijk. Ook deze beperking wordt uitgebreider besproken in hoofdstuk 5.

Tot slot bevat codefragment 2.1 een testplan met één context voor de voorbeeldoefening Lotto uit hoofdstuk 1.

2.2.2 Dataserialisatie

Bij de beschrijving van het testplan wordt gewag gemaakt van returnwaarden en variabeletoekenningen. Aangezien het testplan programmeertaalonafhankelijk is, moet er dus een manier zijn om data uit de verschillende programmeertalen voor te stellen en te vertalen: het *serialisatieformaat*.

```json
{
  "tabs": [
    {
      "name": "Feedback",
      "contexts": [
        {
          "input": {
            "function": {
              "type": "function",
              "name": "loterij",
              "arguments": [
                { "type": "integer", "data": 6 },
                { "type": "integer", "data": 15 }
              ]
            }
          },
          "output": {
            "result": {
              "value": {
                "type": "text",
                "data": "1 - 6 - 7 - 11 - 13 - 14"
              },
              "evaluator": {
                "type": "custom",
                "language": "python",
                "path": "./evaluator.py",
                "arguments": [
                  { "type": "integer", "data": 6 },
                  { "type": "integer", "data": 15 }
                ]
              }
            }
          }
        }
      ]
    }
  ]
}
```

Codefragment 2.1: Een ingekorte versie van het testplan voor de voorbeeldoefening Lotto. Het testplan bevat maar één context.

Keuze van het formaat

Zoals bij het testplan, werd voor de voorstelling van waarden ook een keuze voor een bepaald formaat gemaakt. Daarvoor werden opnieuw enkele voorwaarden vooropgesteld, waaraan het serialisatieformaat moet voldoen. Het formaat moet:

- door mensen geschreven kunnen worden (*human writable*),
- onderdeel van het testplan kunnen zijn,
- in meerdere programmeertalen bruikbaar zijn, en
- de basisgegevenstypes ondersteunen die we willen aanbieden in het programmeertaalonafhankelijke deel van het testplan. Deze gegevenstypes zijn:
  - Primitieven: gehele getallen, reële getallen, Boolese waarden en tekenreeksen.
  - Collecties: rijen (eindige, geordende reeks; `list` of `array`), verzamelingen (eindige, ongeordende reeks zonder herhaling; `set`) en afbeeldingen (elk element wordt afgebeeld op een ander element; `map`, `dict` of `object`).

Een voor de hand liggende oplossing is om ook hiervoor JSON te gebruiken, en zelf in JSON een structuur op te stellen voor de waarden. In tegenstelling tot de situatie bij het testplan bestaan er al een resem aan dataserialisatieformaten, waardoor het de moeite loont om na te gaan of er geen bestaand formaat voldoet aan de vereisten. Hiervoor is gestart van een overzicht op Wikipedia [10]. Uiteindelijk is niet gekozen voor een bestaand formaat, maar voor de JSON-oplossing. De redenen hiervoor zijn samen te vatten als:

- Het gaat om een binair formaat. Binaire formaten zijn uitgesloten op basis van de eerste twee voorwaarden die we opgesteld hebben: mensen kunnen het niet schrijven zonder hulp van bijkomende tools en het is moeilijk in te bedden in een JSON-bestand (zonder gebruik te maken van encoderingen zoals base64). Bovendien zijn binaire formaten moeilijker te implementeren in sommige talen.
- Het formaat ondersteunt niet alle gewenste types. Sommige formaten hebben ondersteuning voor complexere datatypes, maar niet voor alle complexere datatypes die wij nodig hebben. Uiteraard kunnen de eigen types samengesteld worden uit basistypes, maar dan biedt de ondersteuning voor de complexere types weinig voordeel, aangezien er toch een eigen dataschema voor die complexere types opgesteld zal moeten worden.
- Sommige formaten zijn omslachtig in gebruik. Vaak ondersteunen dit soort formaten meer dan wat wij nodig hebben.
- Het formaat is niet eenvoudig te implementeren in een programmeertaal waarvoor geen ondersteuning is. Sommige dezer formaten ondersteunen weliswaar veel talen, maar we willen niet dat het serialisatieformaat een beperkende factor wordt in welke talen door de judge ondersteund worden. Het mag niet de bedoeling zijn dat het implementeren van het serialisatieformaat het meeste tijd in beslag neemt.

Een lijst van de overwogen formaten met een korte beschrijving:

Apache Avro Een volledig „systeem voor dataserialisatie”. De specificatie van het formaat gebeurt in JSON (vergelijkbaar met JSON Schema), terwijl de eigenlijke data binair geëncodeerd wordt. Heeft uitbreidbare types, met veel ingebouwde types [2].

Apache Parquet Minder relevant, dit is een bestandsformaat voor Hadoop [3].

ASN.1 Staat voor *Abstract Syntax Notation One*, een formaat uit de telecommunicatie. De hoofdstandaard beschrijft enkel de notatie voor een dataformaat. Andere standaarden beschrijven dan de serialisatie, bijvoorbeeld een binair formaat, JSON of XML. De meerdere serialisatievormen zijn in theorie aantrekkelijk: elke taal moet er slechts een ondersteunen, terwijl de judge ze allemaal kan ondersteunen. In de praktijk blijkt echter dat voor veel talen er slechts één serialisatieformaat is, en dat dit vaak het binaire formaat is [15].

Bencode Schema gebruikt in BitTorrent. Het is gedeeltelijk binair, gedeeltelijk in text [8].

Binn Binair dataformaat [30].

BSON Een binaire variant op JSON, geschreven voor en door MongoDB [7].

CBOR Een lichtjes op JSON gebaseerd formaat, ook binair. Heeft een goede standaard, ondersteunt redelijk wat talen [6].

FlatBuffers Lijkt op ProtocolBuffers, allebei geschreven door Google, maar verschilt wat in implementatie van ProtocolBuffers. De encodering is binair [25].

Fast Infoset Is eigenlijk een manier om XML binair te encoderen (te beschouwen als een soort compressie voor xml), waardoor het minder geschikt voor ons gebruik wordt [16].

Ion Een superset van JSON, ontwikkeld door Amazon. Het heeft zowel een tekstuele als binaire voorstelling. Naast de gebruikelijke JSON-types, bevat het enkele uitbreidingen. [1].

MessagePack Nog een binair formaat dat lichtjes op JSON gebaseerd is. Lijkt qua types sterk op JSON. Heeft implementaties in veel talen [13].

OGDL Afkorting voor *Ordered Graph Data Language*. Daar het om een serialisatieformaat voor grafen gaat, is het niet nuttig voor ons doel [24].

OPC Unified Architecture Een protocol voor intermachinecommunicatie. Complex: de specificatie bevat 14 documenten, met ongeveer 1250 pagina's [26].

OpenDLL Afkorting voor de *Open Data Description Language*. Een tekstueel formaat, bedoeld om arbitraire data voor te stellen. Wordt niet ondersteund in veel programmeertalen, in vergelijking met bijvoorbeeld JSON [21].

ProtocolBuffers Lijkt zoals vermeld sterk op FlatBuffers, maar heeft nog extra stappen nodig bij het encoderen en decoderen, wat het minder geschikt maakt [28].

Smile Nog een binaire variant van JSON [17].

SOAP Afkorting voor *Simple Object Access Protocol*. Niet bedoeld als formaat voor dataserialisatie, maar voor communicatie tussen systemen over een netwerk [23].

SDXF Binair formaat voor data-uitwisseling. Weinig talen ondersteunen dit formaat [34].

Thrift Lijkt sterk op ProtocolBuffers, maar geschreven door Facebook [31].

UBJSON Nog een binaire variant van JSON [32].

Geen enkel overwogen formaat heeft grote voordelen tegenover een eigen structuur in JSON. Daarenboven hebben veel talen het nadeel dat ze geen JSON zijn, waardoor we een nieuwe taal moeten inbedden in het bestaande JSON-testplan. Dit nadeel, gekoppeld met het ontbreken van voordelen, heeft geleid tot de keuze voor JSON.

```json
{
  "type": "sequence",
  "data": [
    { "type": "integer", "data": 5 },
    { "type": "integer", "data": 15 }
  ]
}
```

Codefragment 2.2: Een lijst bestaande uit twee getallen, geëncodeerd in het serialisatieformaat.

Dataschema

Json is slechts een formaat en geeft geen semantische betekenis aan JSON-elementen. Hiervoor stellen we een dataschema op, dat uit twee onderdelen bestaat:

- Het encoderen van waarden.
- Het beschrijven van de gegevenstypes van deze waarden.

Elke waarde wordt in het serialisatieformaat voorgesteld als een object met twee elementen: de geëncodeerde waarde en het bijhorende gegevenstype. Een concreet voorbeeld is codefragment 2.2.

Het encoderen van waarden slaat op het voorstellen van waarden als JSON-waarden. Json heeft slechts een beperkt aantal gegevenstypes, dus worden alle waarden voorgesteld als een van deze types. Zo worden bijvoorbeeld zowel `arrays` en `sets` voorgesteld als een JSON-lijst.

Het verschil tussen beiden wordt dan duidelijk gemaakt door het bijhorende gegevenstype. Er is dus nood aan een systeem om aan te geven wat het gegevenstype van een waarde is.

Codefragment 2.3 bevat het onder andere de structuur van een waarde in het serialisatieformaat, in een vereenvoudigde versie van JSON Schema. Hierbij staat `<types>` voor een van de gegevenstypes die hierna besproken werden.

Gegevenstypes

Het systeem om de gegevenstypes aan te duiden vervult meerdere functies. Het wordt gebruikt om:

- het gegevenstype van concrete data aan te duiden (beschrijvende modus). Dit gaat om de serialisatie van waarden uit de uitvoeringsomgeving naar TESTed, zoals het geval is bij returnwaarden van functies.
- te beschrijven welk gegevenstype verwacht wordt (voorschrijvende modus). Een voorbeeld hiervan is het aangeven van het gegevenstype van een variabele.
- zelf code te schrijven (letterlijke modus). Dit gaat om serialisatie vanuit het testplan zelf naar de uitvoeringsomgeving. Een voorbeeld hiervan is het opnemen van functieargumenten in het testplan: deze argumenten worden tijdens de serialisatie omgezet naar echte code.

```
{
  "Value": {
    "type": "object",
    "properties": {
      "data": {"type": ["number", "string", "boolean", "object", "array", "null"]},
      "type": {"type": "string", "enum": ["<types>"]}
    }
  },
  "FunctionCall": {
    "type": "object",
    "properties": {
      "type": {
        "type": "string",
        "enum": ["function", "namespace", "constructor", "property"]
      },
      "namespace": {"type": "string"},
      "name": {"type": "string"},
      "arguments": {
        "type": "array",
        "items": {"type": "#Expression"}
      }
    }
  },
  "Identifier": {"type": "string"},
  "Expression": {
    "anyOf": [
      {"type": "#Identifier"},
      {"$ref": "#Value"},
      {"$ref": "#FunctionCall"}
    ]
  },
  "Statement": {
    "type": "object",
    "properties": {
      "name": {"type": "string"},
      "expression": {"type": "#Expression"},
      "type": "<datatype>"
    }
  }
}
```

Codefragment 2.3: Het schema voor waarden, expressies en statements, in een vereenvoudigde versie van JSON Schema.

Bij het ontwerp van het systeem voor de gegevenstypes zorgen deze verschillende functies soms voor tegenstrijdige belangen: voor het beschrijven van een waarde moet het systeem zo eenvoudig mogelijk zijn. Een waarde met bijhorend gegevenstype `union[string, int]` is niet bijster nuttig: een waarde kan nooit tegelijk een `string` en een `int` zijn. Aan de andere kant zijn dit soort complexe gegevenstypes wel nuttig bij het aangeven van het verwachte gegevenstype van bijvoorbeeld een variabele. Daarnaast moet ook rekening gehouden worden met het feit dat deze gegevenstypes in veel programmeertalen implementeerbaar moeten zijn. Een gegevenstype als `union[string, int]` is eenvoudig te implementeren in Python, maar dat is niet het geval in bijvoorbeeld Java of C. Ook heeft elke programmeertaal een eigen niveau van details bij gegevenstypes. Python heeft bijvoorbeeld enkel `integer` voor gehele getallen, terwijl C beschikt over `int`, `unsigned`, `long`, enz. Daarenboven heeft het schrijven van code bijkomende vereisten: als functieargument zijn waarden alleen niet voldoende, ook andere variabelen moeten gerefereerd kunnen worden en het resultaat van andere functieoproepen moeten ook als argument gebruikt kunnen worden.

Om deze redenen zijn de gegevenstypes opgedeeld in drie categorieën:

1. De basistypes. Deze gegevenstypes zijn bruikbaar in alle modi. De lijst van basistypes omvat:

   `integer` Gehele getallen, zowel positief als negatief.

   `rational` Rationale getallen. Het gaat hier om `floats`, die ook vaak gebruikt worden als benadering van gehele getallen.

   `text` Een tekenreeks of string (alle vormen).

   `char` Een enkel teken.

   `boolean` Een Boolese waarde (of boolean).

   `sequence` Een wiskundige rij, wat wil zeggen dat de volgorde belangrijk is en dat dubbele elementen toegelaten zijn.

   `set` Een wiskundige verzameling, wat wil zeggen dat de volgorde niet belangrijk is en dat dubbele elementen niet toegelaten zijn.

   `map` Een wiskundige afbeelding: elk element wordt afgebeeld op een ander element. In Java is dit bijvoorbeeld een `Map`, in Python een `dict` en in Javascript een `object`.

   `nothing` Geeft aan dat er geen waarde is, ook wel `null`, `None` of `nil` genoemd.

   Een lijst van de implementaties in de verschillende programmeertalen is tabel 2.1. Elke implementatie van een programmeertaal moet een keuze maken wat de standaardimplementatie van deze types is. Zo implementeert de Java-implementatie het gegevenstype `sequence` als een `List<>`, niet als een `array`. Een implementatie in een programmeertaal kan ook aangeven dat een bepaald type niet ondersteund wordt, waardoor testplannen met dat type niet zullen werken.

2. De uitgebreide types: dit zijn een hele reeks bijkomende types. Deze gegevenstypes staan toe om meer details over de types te serialiseren en in het testplan op te nemen. Een voorbeeld is de lijst van types in tabel 2.2, die voor een reeks gegevenstypes voor gehele getallen de concrete types in verschillende programmeertalen geeft. Het grote verschil is dat deze uitgebreide types standaard vertaald worden naar een van de basistypes. Voor talen die bijvoorbeeld geen `tuple` uit Python ondersteunen, zal het type omgezet worden naar `list`. Er is ook de mogelijk dat implementaties voor programmeertalen expliciet een bepaald type

Tabel 2.1: Implementaties van de basistypes in de verschillende programmeertalen.

Type	Python	Java	Haskell
`integer`	`int`	`long`	`Integer`
`rational`	`float`	`double`	`Double`
`text`	`str`	`String`	`String`
`char`	`str`	`char`	`Char`
`boolean`	`bool`	`boolean`	`Boolean`
`sequence`	`list`	`List<>`	`List`
`set`	`set`	`Set<>`	`-`
`map`	`dict`	`Map<>`	`-`
`nothing`	`None`	`null`	`Nothing`

Tabel 2.2: Voorbeeld van de implementatie van types voor gehele getallen, met als basistype `integer`.

	`int8`	`uint8`	`int16`	`uint16`	`int32`	`uint32`	`int64`	`uint64`
Python	`int`	`int`	`int`	`int`	`int`	`int`	`int`	`int`
Java	`byte`	`short`	`short`	`int`	`int`	`long`	`long`	`-`
C[1]	`int8_t`	`uint8_t`	`int16_t`	`uint16_t`	`int32_t`	`uint32_t`	`int64_t`	`uint64_t`
Haskell	`Int8`	`Word8`	`Int16`	`Word16`	`Int32`	`Word32`	`Int64`	`Word64`

[1] Uiteraard met de gebruikelijke aliassen van `short`, `unsigned`, ...

niet ondersteunen. Zo zal de Java-implementatie geen `uint64` (een unsigned 64-bit integer) ondersteunen, omdat er geen equivalent bestaat in de taal[1].

3. Voorschrijvende types. Gegevenstypes in deze categorie kunnen enkel gebruikt worden bij het aangeven welk gegevenstype verwacht wordt, niet bij de eigenlijke encodering van waarden. In de praktijk gaat het om het type van variabelen. In deze categorie zouden gegevenstypes als `union[str, int]` komen. Er is echter expliciet gekozen om dit soort types niet te ondersteunen, door de moeilijkheid om dit te implementeren in statisch getypeerde talen, zoals Java of C. Twee types die wel ondersteund worden in deze modus zijn:

   any Het `any`-type geeft aan dat het type van een variabele onbekend is. Merk op dat dit in sommige talen tot moeilijkheden zal leiden: zo zal dit in C-code als `long` beschouwd worden (want C heeft geen equivalent van een `any`-type).

   custom Een eigen type, waarbij de naam van het type gegeven wordt. Dit is nuttig om bijvoorbeeld variabelen aan te maken met als gegevenstype een eigen klasse, zoals een klasse die de student moest implementeren.

2.2.3 Expressions en statements

Een ander onderdeel van het testplan verdient ook speciale aandacht: toekennen van waarden aan variabelen (*assignments*) en functieoproepen.

In heel wat oefeningen, en zeker bij objectgerichte en imperatieve programmeertalen, is het toekennen van een waarde aan een variabele, om deze later te gebruiken, onmisbaar. Bijvoorbeeld zou een opgave kunnen bestaan uit het implementeren van een klasse. Bij de evaluatie dient dan een

[1] Dit is slechts ter illustratie: in de implementatie van TESTed wordt `BigInteger` gebruikt.

instantie van die klasse aangemaakt te worden, waarna er methoden kunnen aangeroepen worden, zoals hieronder geïllustreerd in een fictief voorbeeld.

```
var variabele = new DoorDeStudentGemaakteKlasse();
assert variabele.testfunctie1() == 15;
assert variabele.testfunctie2() == "Vijftienduizend";
assert variabele.eigenschap == 42;
```

Om deze reden is het testplan uitgebreid met ondersteuning voor statements en expressies. Toch moet meteen opgemerkt worden dan deze ondersteuning beperkt is tot wat er nodig is om het scenario van hiervoor te kunnen uitvoeren; het is zeker niet de bedoeling om een volledige eigen programmeertaal te ontwerpen.

In codefragment 2.3 staat onder andere het formaat van een expressie en een functieoproep in een vereenvoudigde versie van JSON Schema. Een expressie is een dezer drie dingen:

1. Een waarde, zoals hiervoor besproken in subparagraaf *Dataschema* van paragraaf 2.2.2.
2. Een `identifier`, voorgesteld als een string.
3. Een functieoproep, die bestaat uit:

   `type` Het soort functie. Kan een van deze waarden zijn:

   `function` Een *top-level* functie. Afhankelijk van de programmeertaal zal deze functie toch omgezet worden naar een `namespace`-functie. Zo worden dit soort functies in Java omgezet naar statische functies.

   `namespace` Een methode (functie van een object) of een functie in een namespace. De invulling hiervan is gedeeltelijk programmeertaalafhankelijk: in Java gaat het om methodes, terwijl het in Haskell om functies van een module gaat. Bij dit soort functies moet de `namespace` gegeven worden.

   `constructor` Deze soort functie heeft dezelfde semantiek als een top-level functie, met dien verstande dat het om een constructor gaat. In Java zal bijvoorbeeld het keyword `new` vanzelf toegevoegd worden. De functienaam doet dienst als naam van de klasse.

   `property` De property van een instantie wordt gelezen. Deze soort functie heeft dezelfde semantiek van een namespace-functie, maar heeft geen argumenten.

   `namespace` De namespace van de functie.

   `name` De naam van de functie.

   `arguments` De argumenten van de functie. Dit is een lijst van expressies.

De ondersteuning voor statements in het testplan beperkt zich tot variabeletoekenningen of *assignments*. Er is expliciet voor gekozen om expressies geen statements te maken. De reden hiervoor is dat dit de implementatie ingewikkelder zou maken, zonder noemenswaardig voordeel. Een assignment kent een naam toe aan het resultaat van een expression. Codefragment 2.3 toont ook de vereenvoudigde JSON Schema van een statement (en dus van een assignment, daar er maar één soort statement bestaat). Hier staat `<datatype>` voor een van de gegevenstypes die hiervoor besproken zijn.

De `name` is de naam die aan de variabele gegeven zal worden. Het veldje `expression` moet een expressie zijn, zoals reeds besproken. Ook moet het gegevenstype van de variabele gegeven worden. Hiervoor kunnen types het het serialisatieformaat gebruikt worden, inclusief de types uit de letterlijke modus.

Een gecombineerd voorbeeld staat hieronder. Hier wordt de string `'Dodona'` toegekend aan een variabele met naam `name`.

```
{
  "name": "name",
  "expression": {
    "type": "text",
    "value": "Dodona"
  },
  "type": "text"
}
```

Tot slot is het nog het vermelden waard dat waarden van de gegevenstypes `sequence` en `map` als elementen geen andere waarden hebben, maar expressies. Dit niet het geval in de beschrijvende modus van de gegevenstypes, bijvoorbeeld bij het aangeven wat de verwachte returnwaarde van een functie is. Het testplan biedt namelijk geen ondersteuning voor het serialiseren van identifiers en functieoproepen, enkel waarden. Dit betekent dat constructies zoals deze mogelijk zijn in het testplan:

```
var variabele1 = functie1();
var variabele2 = [variabele1, 15, functie2()];
var object = new Test();

assert object.process(variabele2, [functie1()]) == 85;
```

2.2.4 Controle ondersteuning voor programmeertalen

In het stappenplan uit paragraaf 2.1 is al vermeld dat vóór een beoordeling start, een controle plaatsvindt om zeker te zijn dat het testplan uitgevoerd kan worden in de programmeertaal van de ingediende oplossing. Bij de controle worden volgende zaken nagekeken:

- Controle of de programmeertaal de nodige gegevenstypes ondersteunt. Dit gaat van de basistypes (zoals `sequence`) tot de geavanceerde types (zoals `tuple`). Voor elke programmeertaal binnen TESTed wordt bijgehouden welke types ondersteund worden en welke niet. Bevat een testplan bijvoorbeeld waarden met als type `set` (verzamelingen), dan kunnen enkel programmeertalen die verzamelingen ondersteunen gebruikt worden. Dat zijn bijvoorbeeld Python en Java, maar geen Bash.
- Controle of de programmeertaal over de nodige taalconstructies beschikt, zoals `exceptions` of `objects`. Bij deze controle wordt ook gecontroleerd of dat de programmeertaal optionele functieargumenten of functieargumenten met heterogene gegevenstypes nodig heeft. Een

voorbeeld van een functie met argumenten met heterogene gegevenstypes komt bijvoorbeeld uit de ISBN-oefening (deze oefening wordt besproken in hoofdstuk 4):

```
>>> is_isbn("9789027439642")
True
>>> is_isbn(9789027439642)
False
```

In talen als Python en Java kan deze functie geïmplementeerd worden, maar in talens als C is dat veel moeilijker.

2.3 Oplossingen uitvoeren

De eerste stap die wordt uitgevoerd bij de beoordeling van een ingediende oplossing is het genereren van de testcode, die de ingediende oplossing zal beoordelen.

2.3.1 Testcode genereren

Het genereren van de testcode gebeurt met een sjabloonsysteem genaamd Mako [5]. Dit soort systemen wordt traditioneel gebruikt bij webapplicaties (zoals Ruby on Rails met ERB, Phoenix met EEX, Laravel met Blade, enz.) om bijvoorbeeld html-pagina's te genereren. In ons geval zijn de sjablonen verantwoordelijk voor de vertaling van programmeertaalonafhankelijke specificaties in het testplan naar concrete testcode in de programmeertaal van de ingediende oplossing. Hierbij denken we aan de functieoproepen, assignments, enz. Ook zijn de sjablonen verantwoordelijk voor het genereren van de code die de oplossing van de student zal oproepen en evalueren.

Sjablonen

TESTed heeft een aantal standaardsjablonen nodig, waaraan vastgelegde parameters meegegeven worden en die een vaste functie moeten uitvoeren. Deze verplichte sjablonen zijn:

`assignment` Vertaalt een toekenningsopdracht uit het testplan naar code.

`context` Een sjabloon dat code genereert om een context te beoordelen. Deze code moet uitvoerbaar zijn (dat wil zeggen een main-functie bevatten of een script zijn).

`selector` Een sjabloon dat code genereert om een bepaalde context uit te voeren. Om performantieredenen (hierover later meer) wordt de code van alle contexten soms uit een keer gegenereerd en gecompileerd. Aan de hand van een parameter (de naam van de context), wordt bij het uitvoeren van deze selectiecode de testcode voor de juiste context gekozen. Dit sjabloon is enkel nodig indien batchcompilatie ondersteund wordt en de programmeertaal dit nodig heeft (bijvoorbeeld niet nodig in Python, maar wel in Java).

`evaluator_executor` Een sjabloon dat code genereert om een geprogrammeerde evaluatie te starten.

`function` Vertaalt een functie-oproep naar testcode.

Daarnaast moet het encoderen naar het serialisatieformaat ook geïmplementeerd worden in elke programmeertaal. Veel programmeertalen hebben dus nog enkele bijkomende bestanden met code. In alle bestaande configuraties van programmeertalen is dit geïmplementeerd als een module of een klasse met naam `Value`. Dit wordt geïllustreerd in hoofdstuk 3, dat het toevoegen van een nieuwe programmeertaal aan TESTed volledig uitwerkt.

Testcode compileren

TESTed ondersteunt twee modi waarin de code gecompileerd kan worden (bij programmeertalen die geen compilatie ondersteunen wordt deze stap overgeslagen):

Batchcompilatie In deze modus wordt de code voor alle contexten in een keer gecompileerd. Dit wordt gedaan om performantieredenen. In talen die resulteren in een uitvoerbaar bestand (zoals Haskell, C/C++), resulteert deze modus in één uitvoerbaar bestand voor alle contexten. Bij het uitvoeren wordt dan aan de hand van een parameter de juiste context uitgevoerd (met het `selector`-sjabloon van hierboven).

Contextcompilatie Hierbij wordt elke context afzonderlijk gecompileerd.

Dit wordt getoond in figuur 2.2 uit paragraaf 2.1 door twee kleuren te gebruiken: de stappen die enkel gebeuren bij batchcompilatie zijn in het groen, terwijl stappen die enkel bij contextcompilatie gebeuren in het blauw staan. Stappen die altijd gebeuren staan in de flowchart in het zwart.

Dit gedrag is configureerbaar in het testplan, maar standaard wordt de batchcompilatie gebruikt. Als er een compilatiefout optreed bij de compilatie in batchcompilatie, wordt valt TESTed terug op contextcompilatie. Deze terugval is handig voor programmeertalen waar de compilatie veel fouten ontdekt (vaak de meer statische programmeertalen). Een voorbeeldscenario is als volgt: stel een oefening waarbij de student twee functies moet implementeren. De student implementeert de eerste functie en dient een oplossing in om al feedback te krijgen. Bij programmeertalen als Java of Haskell zal dit niet lukken: daar alle contexten in één keer gecompileerd worden, zal de ontbrekende tweede functie ervoor zorgen dat de volledige compilatie faalt. In individuele modus is dit geen probleem: de contexten die de eerste functie testen zullen compileren en kunnen uitgevoerd worden. De individuele modus brengt wel een niet te verwaarlozen kost qua uitvoeringstijd met zich mee (zie ook hoofdstuk 5).

Codefragmenten 2.4 en 2.5 bevatten de testcode gegenereerd voor een context uit de voorbeeldoefening Lotto (het gaat om dezelfde context uit het voorbeeld van het testplan in codefragment 2.1), in respectievelijk Python en Java. Daarnaast bevat Z de code voor de `selector` in Java. Hiervan is geen versie in Python, daar Python selector nodig heeft in batchcompilatie (in Python kunnen meerdere onafhankelijke bestanden tegelijk gecompileerd worden). De selector bevat twee contexten om de werking duidelijk te maken.

2.3.2 Testcode uitvoeren

Vervolgens wordt de (gecompileerde) testcode voor elke context uit het testplan afzonderlijk uitgevoerd en worden de resultaten (het gedrag en de neveneffecten) verzameld. Het uitvoeren zelf gebeurt op de normale manier waarop code voor de programmeertaal uitgevoerd wordt: via de commandoregel. Deze aanpak heeft als voordeel dat er geen verschil is tussen hoe TESTed de ingediende code uitvoert en hoe de student zijn code zelf uitvoert op zijn eigen computer. Dit voorkomt dat er subtiele verschillen in de resultaten sluipen.

```python
import values
import sys

value_file = open(r"Ogdcj0QzN_values.txt", "w")
exception_file = open(r"Ogdcj0QzN_exceptions.txt", "w")

def write_delimiter(delimiter):
    value_file.write(delimiter)
    exception_file.write(delimiter)

def send(value):
    values.send_value(value_file, value)

def send_exception(exception):
    values.send_exception(exception_file, exception)

def e_evaluate_text_main(value):
    send_exception(value)

def v_evaluate_text_0(value):
    send(value)

def e_evaluate_text_0(value):
    send_exception(value)

# Import code.
try:
    from submission import *
except Exception as e:
    raise e

# Context 0-0.
try:
    v_evaluate_text_0(    loterij(6, 15)    )
except Exception as e:
    e_evaluate_text_0(e)

sys.stderr.write("--Ogdcj0QzN-- SEP")
sys.stdout.write("--Ogdcj0QzN-- SEP")
write_delimiter("--Ogdcj0QzN-- SEP")

value_file.close()
exception_file.close()
```

Codefragment 2.4: Gegenereerde testcode in Python voor de eerste context uit het testplan van de voorbeeldoefening Lotto.

```java
public class Context_0_0 {

    private final PrintWriter valueWriter;
    private final PrintWriter exceptionWriter;

    public Context_0_0() throws Exception {
        this.valueWriter = new PrintWriter("TDm75Wrze_values.txt");
        this.exceptionWriter = new PrintWriter("TDm75Wrze_exceptions.txt");
    }

    private void send(Object value) throws Exception {
        Values.send(valueWriter, value);
    }

    private void sendE(Exception exception) throws Exception {
        Values.sendException(exceptionWriter, exception);
    }

    private void vEvaluate0(Object value) throws Exception {
        send(value);
    }

    private void eEvaluate0(Exception value) throws Exception {
        sendE(value);
    }

    void execute() throws Exception {
        try {
            this.vEvaluate0(Main.loterij(6, 15));
        } catch (Exception e) {
            this.eEvaluate0(e);
        }
        System.err.print("--TDm75Wrze-- SEP");
        System.out.print("--TDm75Wrze-- SEP");
        valueWriter.write("--TDm75Wrze-- SEP");
        exceptionWriter.write("--TDm75Wrze-- SEP");
    }

    public static void main(String[] a) throws Exception {
        (new Context_0_1()).execute();
    }
}
```

Codefragment 2.5: Gegenereerde testcode in Java voor de eerste context uit het testplan van de voorbeeldoefening Lotto. Enkele hulpfuncties en imports zijn verwijderd om de code korter te maken.

```java
class Selector {
    public static void main(String[] a) throws Exception {
        var name = a[0];
        if ("Context_0_0".equals(name)) {
            Context_0_0.main(new String[]{});
        }
        if ("Context_0_1".equals(name)) {
            Context_0_1.main(new String[]{});
        }
    }
}
```

Codefragment 2.6: Gegenereerde selectiecode in Java voor twee contexten uit het testplan van de voorbeeldoefening Lotto.

Codefragment 2.7 illustreert dit met een voorbeeld voor een ingediende oplossing in de programmeertaal Python. Deze mapstructuur stelt de toestand van de werkmap van TESTed voor na het uitvoeren van de code. In de map `common` zit alle testcode en de gecompileerde bestanden voor alle contexten. Voor elke context worden de gecompileerde bestanden gekopieerd naar een andere map, bv. `context_0_1`, wat de map is voor context `1` van tabblad `0` van het testplan.

2.3.3 Beoordelen van gedrag

Het uitvoeren van de testcode genereert resultaten (gedrag en neveneffecten) die door TESTed beoordeeld moeten worden. Er zijn verschillende soorten gedragingen en neveneffecten die interessant zijn. Elke soort gedrag of neveneffect wordt een *uitvoerkanaal* genoemd. TESTed verzamelt volgende uitvoerkanalen:

- De standaarduitvoerstroom. Dit wordt verzameld als tekstuele uitvoer.
- De standaardfoutstroom. Ook dit wordt als tekst verzameld.
- Fatale uitzonderingen. Hiermee bedoelen we uitzonderingen die tot aan de testcode geraken. Een uitzondering die afgehandeld wordt door de ingediende oplossing wordt niet verzameld. De uitzonderingen worden verzameld in een bestand.
- Returnwaarden. Deze waarden worden geëncodeerd en ook verzameld in een bestand.
- Exitcode. Het gaat om de exitcode van de testcode voor een context. Daar de code per context wordt uitgevoerd, wordt de exitcode ook verzameld per context (en niet per testcase, zoals de andere uitvoerkanalen).
- Bestanden. Tijdens het beoordelen van de verzamelde resultaten is het mogelijk de door de ingediende oplossing gemaakte bestanden te bekijken.

De standaarduitvoer- en standaardfoutstroom worden rechtstreeks opgevangen door TESTed. De andere uitvoerkanalen (uitzonderingen en returnwaarden) worden naar een bestand geschreven. De reden dat deze niet naar een andere *file descriptor* geschreven worden is eenvoudig: niet alle talen (zoals Java) ondersteunen het openen van bijkomende file descriptors.

```
workdir                            //Werkmap van TESTed
├── common                         //Gemeenschappelijke code
│   ├── context_0_0.py             //Broncode voor context 0 van tabblad 0
│   ├── context_0_0.pyc            //Gecompileerde code voor context 0 van tabblad 0
│   ├── context_0_1.py
│   ├── context_0_1.pyc
│   ...
│   ├── context_0_49.py
│   ├── context_0_49.pyc
│   ...
│   ├── context_1_49.py            //Broncode voor context 49 van tabblad 1
│   ├── context_1_49.pyc
│   ├── submission.py              //Ingediende oplossing
│   ├── submission.pyc
│   ├── values.py                  //Values-module
│   └── values.pyc
├── context_0_0                    //Map voor context 0-0
│   ├── FaLd6WGRN_exceptions.txt   //Uitzonderingskanaal
│   ├── FaLd6WGRN_values.txt       //Kanaal voor returnwaarden
│   ├── context_0_0.pyc            //Code voor context 0-0
│   ├── submission.pyc
│   └── values.pyc
...
└── evaluators                     //Aangepaste evaluatoren
    ├── Aao9H18Ve_buzzchecker      //Map voor elke context
    │   ├── buzzchecker.py         //Code aangepaste evaluator
    │   ├── evaluation_utils.py
    │   ├── evaluator_executor.py
    │   └── values.py
    ├── B5WViK0zQ_buzzchecker
    │   ├── buzzchecker.py
    │   ├── evaluation_utils.py
    │   ├── evaluator_executor.py
    │   └── values.py
    ...
```

Codefragment 2.7: Mapstructuur na het uitvoeren van de testcode van een oplossing in Python. `context_0_0` staat voor de eerste context van het eerste tabblad.

```
{"data":"1 - 3 - 6 - 8 - 10 - 15","type":"text"}--gL9koJNv3-- SEP
```

Codefragment 2.8: Voorbeeld van het uitvoerkanaal voor returnwaarden na het uitvoeren van de eerste context uit de voorbeeldoefening Lotto.

Alle uitvoerkanalen (met uitzondering van de exitcode en de bestanden) worden per testcase verzameld. Aangezien de uitvoerkanalen pas verzameld worden na het uitvoeren van de context, moet er een manier zijn om de uitvoer van de verschillende testgevallen te onderscheiden. De testcode is hier verantwoordelijk voor, en schrijft een *separator* naar alle uitvoerkanalen tussen elk testgeval, zoals te zien is in codefragment 2.8.

Tijdens het genereren van de code krijgen de sjablonen een reeks willekeurige tekens mee, de *secret*. Deze secret wordt gebruikt voor verschillende dingen, zoals:

- De separator. Door het gebruik van de willekeurige tekens is de kans dat de separator overeenkomt met een echte waarde praktisch onbestaand.
- Bestandsnamen. De testcode is verantwoordelijk voor het openen van de bestanden voor de uitvoerkanalen die naar een bestand geschreven worden. Bij het openen zal de testcode de secret in de bestandsnaam gebruiken. Dit is om het per abuis overschrijven van deze bestanden door de ingediende oplossing tegen te gaan.

2.4 Oplossingen beoordelen

Na het uitvoeren van de testcode voor elke context heeft TESTed alle relevante uitvoer gemeten en verzameld. Deze uitvoer moet vervolgens beoordeeld worden om na te gaan in hoeverre deze uitvoer voldoet aan de verwachte uitvoer. Dit kan op drie manieren:

1. Generieke evaluatie: de uitvoer wordt beoordeeld door TESTed zelf.
2. Geprogrammeerde evaluatie: de uitvoer wordt beoordeeld door programmacode geschreven door degene die de oefening opgesteld heeft, in een aparte omgeving (de evaluatieomgeving).
3. Programmeertaalspecifieke evaluatie: de uitvoer wordt onmiddellijk na het uitvoeren van de testcode beoordeeld in het hetzelfde proces.

2.4.1 Generieke evaluatie

Voor eenvoudige beoordelingen (bijvoorbeeld tussen twee waarden) volstaat de generieke evaluatie binnen TESTed. Het is mogelijk om de verwachte resultaten in het testplan op te nemen. TESTed zal deze resultaten uit het testplan dan vergelijken met de resultaten geproduceerd door het uitvoeren van de testcode. Als *proof of concept* zijn drie eenvoudige evaluatiemethoden ingebouwd in TESTed, die hieronder besproken worden.

Tekstevaluatie

Deze evaluator vergelijkt de verkregen uitvoer van een uitvoerkanaal (standaarduitvoer, standaardfout, …) met de verwachte uitvoer uit het testplan. Deze evaluator biedt enkele opties om het gedrag aan te passen:

`ignoreWhitespace` Witruimte voor en na het resultaat wordt genegeerd. Dit gebeurt op de volledige tekst, niet regel per regel.

`caseInsensitive` Er wordt geen rekening gehouden met het verschil tussen hoofdletters en kleine letters.

`tryFloatingPoint` De tekst zal geïnterpreteerd worden als een zwevendekommagetal (*floating point*). Bij het vergelijken met de verwachte waarde zal de functie `math.isclose()`[2] uit de standaardbibliotheek van Python gebruikt worden. Deze functie controleert of twee zwevendekommagetallen „dicht bij elkaar" liggen. De standaardfoutmarges van Python worden gebruikt. Een punt voor de toekomst is het configureerbaar maken van deze foutmarges.

`applyRounding` Of zwevendekommagetallen afgrond moeten worden tijdens het vergelijken. Indien wel wordt het aantal cijfers genomen van de optie `roundTo`. Na de afronding worden ze ook vergeleken met de functie `math.isclose()`. Deze afronding is enkel van toepassing op het vergelijken, niet op de uitvoer.

`roundTo` Het aantal cijfers na de komma. Enkel nuttig als `applyRounding` waar is.

Deze configuratieopties worden op het niveau van de testen meegegeven. Dit laat toe om voor elke test (zelfs binnen eenzelfde testgeval) andere opties mee te geven. Een nadeel is wel dat dezelfde opties mogelijk veel herhaald moeten worden, bijvoorbeeld als een bepaalde oefening een optie voor elke test wil instellen. Echter wordt er verwacht dat dit soort zaken opgelost kunnen worden door een DSL of door het testplan te genereren.

Dit is de standaardevaluatievorm in het testplan als niets anders gegeven wordt. Codefragment 2.9 toont een fragment uit een testplan: de uitvoerspecificatie van een testgeval waarbij de tekstevaluatie gebruikt wordt.

Bestandsevaluatie

In deze evaluatievorm worden twee bestanden vergeleken met elkaar. Hiervoor bevat het testplan enerzijds een pad naar een bestand die met de oefening gegeven wordt met de verwachte inhoud en anderzijds de naam (of pad) van de locatie waar het verwachte bestand zich moet bevinden. De bestandsevaluatie ondersteunt enkel tekstuele bestanden, geen binaire bestanden. Het vergelijken van de bestanden gebeurt op één dezer manieren:

`exact` Beide bestanden moet exact hetzelfde zijn, inclusief regeleindes.

`lines` Elke regel wordt vergeleken met overeenkomstige regel in het andere bestand. De evaluatie van de lijnen is exact, maar zonder de regeleindes. Dit betekent dat de witruimte bijvoorbeeld ook moet overeenkomen.

`values` Elke regel in het bestand wordt afzonderlijk vergeleken met de tekstevaluatie. Indien deze modus gebruikt wordt, kunnen ook alle opties van de tekstevaluatie meegegeven worden.

[2] Documentatie is hier te vinden: https://docs.python.org/3/library/math.html#math.isclose

```
{
  "output": {
    "stdout": {
      "type": "text",
      "data": "3.14",
      "evaluator": {
        "type": "builtin",
        "name": "text",
        "options": {
          "ignoreWhitespace": true,
          "tryFloatingPoint": true,
          "applyRounding": true,
          "roundTo": 2
        }
      }
    }
  }
}
```

Codefragment 2.9: Fragment uit een testplan dat de uitvoerspecificatie van de standaarduitvoerstroom voor een testgeval toont, waarbij de tekstevaluatie gebruikt wordt.

Een voorbeeld van hoe dit eruitziet is codefragment 2.10. In dit fragment wordt de modus `values` gebruikt, en worden de opties van de tekstevaluatie ook meegegeven. Het bestand met de verwachte inhoud heeft als naam `bestand-uit-de-oefening.txt` gekregen, terwijl de ingediende oplossing een bestand moet schrijven naar `waar-het-verwachte-bestand-komt.txt`. Beide paden zijn relatief, maar ten opzichte van andere mappen: het bestand met verwachte inhoud is relatief tegenover de map van de oefening, terwijl het pad waar de ingediende oplossing naar moet schrijven relatief is ten opzichte van de werkmap van de context waarin de oplossing wordt uitgevoerd (zie codefragment 2.7 voor een overzicht van de structuur).

Waarde-evaluatie

Voor uitvoerkanalen zoals de returnwaarden moet meer dan alleen tekst met elkaar vergeleken kunnen worden. Staat er in het testplan welke waarde verwacht wordt (geëncodeerd in het serialisatieformaat), dan kan TESTed dit vergelijken met de eigenlijke waarde die geproduceerd werd door de ingediende oplossing.

Het vergelijken van een waarde bestaat uit twee stappen:

1. Het gegevenstype wordt vergeleken, waarbij beide waarden (de verwachte waarde uit het testplan en de geproduceerde waarde uit de ingediende oplossing) hetzelfde type moeten hebben. Hierbij wordt rekening gehouden met de vertalingen tussen de verschillende programmeertalen, waarbij twee gevallen onderscheiden kunnen worden:

    a) Specifieert het testplan een basistype, dan zullen alle types die tot dit basistype herleid kunnen worden als hetzelfde beschouwd worden. Is de verwachte waarde bijvoorbeeld `sequence`, zullen ook `arrays` uit Java en `tuples` uit Python goedgekeurd worden.

```json
{
  "output": {
    "file": {
      "expected_path": "./bestand-uit-de-oefening.txt",
      "actual_path": "./waar-het-verwachte-bestand-komt.txt",
      "evaluator": {
        "type": "builtin",
        "name": "file",
        "options": {
          "mode": "values",
          "ignoreWhitespace": true,
          "tryFloatingPoint": true,
          "applyRounding": true,
          "roundTo": 2
        }
      }
    }
  }
}
```

Codefragment 2.10: Fragment uit een testplan dat de uitvoerspecificatie van een bestand voor een testgeval toont, waarbij de bestandsevaluatie gebruikt wordt.

   b) Specifieert het testplan een uitgebreid type, dan zal het uitgebreid type gebruikt worden voor talen die dat type ondersteunen, terwijl voor andere talen het basistype gebruikt zal worden. Stel dat het testplan bijvoorbeeld een waarde met als gegevenstype `tuple` heeft. In Python en Haskell (twee talen die dat gegevenstype ondersteunen) zullen enkel `tuples` goedgekeurd worden. Voor andere talen, zoals Java, worden alle gegevenstypes goedgekeurd die herleidbaar zijn tot het basistype. Concreet zullen dus `Lists` en `arrays` goedgekeurd worden. Merk op dat momenteel bij collecties (`sequences`, `sets` en `maps`) enkel het type van de collectie gecontroleerd wordt.

2. De twee waarden worden vergeleken op inhoud (indien de vergelijking van de gegevenstypes uit de vorige stap positief is). Hierbij maakt TESTed gebruik van de ingebouwde vergelijking van Python om twee waarden te evalueren. Dit betekent dat de regels voor *value comparisons* uit Python[3] gevolgd worden. Eén uitzondering is zwevendekommagetallen, waarvoor opnieuw `math.isclose()` gebruikt wordt in plaats van $=$.

Bij deze evaluatievorm zijn geen configuratieopties. Een voorbeeld van het gebruik binnen een testplan is codefragment 2.11. Hier wordt als returnwaarde een verzameling met drie elementen (5, 10 en 15) verwacht.

2.4.2 Geprogrammeerde evaluatie

Bij oefeningen met niet-deterministische resultaten, zoals de voorbeeldoefening Lotto, kunnen de verwachte waarden niet in het testplan komen. Ook andere oefeningen waar geen directe ver-

[3] Zie https://docs.python.org/3/reference/expressions.html?highlight=comparison#value-comparisons

```json
{
  "output": {
    "return": {
      "value": {
        "type": "set",
        "data": [
          { "type": "integer", "data": 5 },
          { "type": "integer", "data": 10 },
          { "type": "integer", "data": 15 }
        ]
      },
      "evaluator": {
        "type": "builtin",
        "name": "value"
      }
    }
  }
}
```

Codefragment 2.11: Fragment uit een testplan dat de uitvoerspecificatie van de returnwaarde voor een testgeval toont, waarbij de waarde-evaluatie gebruikt wordt.

gelijking kan gemaakt worden, zoals het uitlijnen van sequenties (*sequence alignment*) uit de bioinformatica, volstaat een vergelijking met een verwachte waarde uit het testplan niet.

Toch is deze evaluatie niet programmeertaalafhankelijk: de logica om een sequentie uit te lijnen is dezelfde ongeacht de programmeertaal waarin dit gebeurt. Voor dergelijke scenario's is geprogrammeerde evaluatie een oplossing: hierbij wordt code geschreven om de evaluatie te doen, maar deze evaluatiecode staat los van de ingediende oplossing en moet ook niet in dezelfde programmeertaal geschreven zijn. Binnen TESTed wordt dit mogelijk gemaakt door geproduceerde waarden uit de ingediende oplossing te serialiseren bij het uitvoeren van de testcode, en terug te deserialiseren bij het uitvoeren van de evaluatiecode.

Deze evaluatiecode kan geschreven worden in een programmeertaal naar keuze, al moet de programmeertaal wel ondersteund worden door TESTed. De implementatie volgt in alle programmeertalen hetzelfde stramien, maar de implementatiedetails kunnen verschillen. In Python bestaat de evaluatiecode uit een module (een `.py`-bestand) met een functie die voldoet aan de definitie, zoals gegeven in codefragment 2.12. TESTed stelt ook een module `evaluation_utils` ter beschikking. De functie van hierboven moet dan één oproep doen naar de functie `evaluated()`. Deze module is redelijk eenvoudig, zoals te zien in codefragment 2.13

In de Java-implementatie is de situatie gelijkaardig: het gaat om het implementeren van een abstracte klasse. Deze abstracte klasse biedt ook de functionaliteit aan van de module `evaluation_utils` bij Python. De te implementeren klasse en haar ouderklasse staan in codefragmenten 2.14 en 2.15.

Een geprogrammeerde evaluatie wordt gebruikt in de voorbeeldoefening Lotto. Het gebruik in het testplan wordt getoond in codefragment 2.16, waar de evaluatiecode voor de aangepaste evaluatie in Python geschreven is. Er worden ook argumenten meegegeven aan deze code. De evaluatiecode zelf is gegeven in codefragment 2.17.

```python
def evaluate_text(expected, actual, arguments):
    """
    :param expected: The expected value from the testplan.
    :param actual: The actual value produced by the student's code.
    :param arguments: Arguments from the testplan.
    """
    pass
```

Codefragment 2.12: De definitie van de functie die aanwezig moet zijn in de evaluatiecode voor een geprogrammeerde evaluatie geschreven in Python.

```python
import values
import sys

from typing import List, Optional


def evaluated(result: bool,
              readable_expected: Optional[str] = None,
              readable_actual: Optional[str] = None,
              messages: Optional[List[str]] = None):
    """
    Report the result of an evaluation to the judge. This method should only
    be called once per evaluation. Calling this multiple times will result in
    undefined behaviour.

    :param result: The result of the evaluation (True or False).

    :param readable_expected: A string version of the channel value. Optional;
                              if not given, the judge will produce one on a
                              best-efforts basis.
    :param readable_actual: A string version of the actual value. Optional; if
                            not given, the judge will produce one on a best-
                            efforts basis.
    :param messages: Optional list of messages to be shown to the student.
    """
    if messages is None:
        messages = []

    values.send_evaluated(sys.stdout,
                          result, readable_expected, readable_actual, messages)
```

Codefragment 2.13: De implementatie van de module `evaluation_utils`

```java
import java.io.IOException;
import java.util.List;

abstract class AbstractCustomEvaluator extends AbstractEvaluator {
    abstract void evaluate(Object expected,
                           Object actual,
                           List<Object> arguments) throws IOException;
}
```

Codefragment 2.14: De implementatie van de klasse `AbstractCustomEvaluator`.

2.4.3 Programmeertaalspecifieke evaluatie

In sommige scenario's moeten programmeertaalspecifieke concepten beoordeeld worden. Een mogelijkheid is deze oefeningen niet aanbieden in TESTed, maar in de programmeertaalspecifieke judges. Toch zijn er nog voordelen om ook deze oefeningen in TESTed aan te bieden:

- Het bijkomende werk om meer programmeertalen te ondersteunen beperkt zich tot een minimum.
- Het werk om een nieuwe programmeertaal toe te voegen aan TESTed is kleiner dan een volledig nieuwe judge te implementeren.

Het is desalniettemin het vermelden waard dat het niet zeker is of deze evaluatiemethode (en dit scenario meer algemeen) veel zal voorkomen. Oefeningen die programmeertaalspecifieke aspecten moeten beoordelen zijn, net door hun programmeertaalspecifieke aard, moeilijker aan te bieden in meerdere programmeertalen. Een oefening in de programmeertaal C die bijvoorbeeld beoordeelt op juist gebruik van pointers zal weinig nut hebben in Python.

In gebruik lijkt de programmeertaalspecifieke evaluatie sterk op de geprogrammeerde evaluatie, met dat verschil dat het testplan niet evaluatiecode in één programmeertaal bevat, maar evaluatiecode in alle programmeertalen waarin de oefening aangeboden wordt, zoals geïllustreerd in codefragment 2.18. Als de programmeertaalspecifieke evaluatie gebruikt wordt en er wordt geen evaluatiecode voor een bepaalde programmeertaal, zal de oefening niet opgelost kunnen worden in die programmeertaal.

Ook de implementatie lijkt op de geprogrammeerde evaluatie, zij het dat de te implementeren functie afwijkt. In Python wordt dit codefragment 2.19, in Java codefragment 2.20. Om het resultaat van de evaluatie aan de judge te geven, wordt dezelfde `evaluated`-functie als bij de aangepaste evaluator gebruikt (zie codefragmenten 2.13 en 2.15). Het gebruik in het testplan is codefragment 2.18.

2.5 Performantie

Zoals eerder vermeld (paragraaf 2.3.2), wordt de testcode voor elke context afzonderlijk uitgevoerd. Dat de contexten strikt onafhankelijk van elkaar uitgevoerd worden, werd reeds in het begin als een doel vooropgesteld. Dit geeft wel enkele uitdagingen op het vlak van performantie. Het belang van performante judges in Dodona is niet te verwaarlozen, in die zin dat Dodona een interactief platform is, waar studenten verwachten dat de feedback op hun ingediende oplossing onmiddellijk

```java
import java.io.Closeable;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Collection;
import java.util.List;

abstract class AbstractEvaluator implements Closeable {

    protected final PrintWriter writer;

    public AbstractEvaluator() {
        this.writer = new PrintWriter(System.out);
    }

    @Override
    public void close() throws IOException {
        this.writer.close();
    }

    /**
     * Report the result of an evaluation to the judge. This method should only
     * be called once, otherwise things will break.
     *
     * @param result           The result of the evaluation.
     * @param readableExpected Optional string version of the expected value.
     * @param readableActual   Optional string version of the actual value.
     * @param messages         Optional list of messages to pass to the student.
     */
    protected void evaluated(boolean result, String readableExpected,
                             String readableActual,
                             Collection<String> messages) throws IOException {
        Values.evaluated(writer,
                result, readableExpected, readableActual, messages);
    }

    protected void evaluated(boolean result, String readableExpected,
                             String readableActual) throws IOException {
        Values.evaluated(writer,
                result, readableExpected, readableActual, List.of());
    }
}
```

Codefragment 2.15: De implementatie van de klasse `AbstractEvaluator`.

```
{
  "output": {
    "result": {
      "value": {
        "type": "text",
        "data": "1 - 6 - 7 - 11 - 13 - 14"
      },
      "evaluator": {
        "type": "custom",
        "language": "python",
        "path": "./evaluator.py",
        "arguments": [
          { "type": "integer", "data": 6 },
          { "type": "integer", "data": 15 }
        ]
      }
    }
  }
}
```

Codefragment 2.16: Fragment uit het testplan van de voorbeeldoefening Lotto, waar een geprogrammeerde evaluatie gebruikt wordt.

beschikbaar is. Deze paragraaf beschrijft de evolutie van de implementatie van TESTed vanuit het perspectief van de performantie.

2.5.1 Jupyter-kernels

Het eerste prototype van TESTed gebruikte Jupyter-kernels voor het uitvoeren van de testcode. Jupyter-kernels zijn de achterliggende technologie van Jupyter Notebooks [19]. De werking van een Jupyter-kernel kan als volgt samengevat worden: een Jupyter-kernel is een lokaal proces, dat code kan uitvoeren en de resultaten van die uitvoer teruggeeft. Zo kan men naar de Python-kernel de expressie `5 + 9` sturen, waarop het antwoord `14` zal zijn. Een andere manier om een Jupyter-kernel te bekijken is als een programmeertaalonafhankelijk protocol bovenop een REPL (een *read-eval-print loop*). Deze keuze voor Jupyter-kernels als uitvoering was gebaseerd op volgende argumenten:

- Hergebruik van bestaande kernels. Hierdoor is het niet nodig om voor elke programmeertaal veel tijd te besteden aan de implementatie of de configuratie: aangezien het protocol voor Jupyter-kernels programmeertaalonafhankelijk is, kunnen alle bestaande kernels gebruikt worden.
- De functionaliteit aangeboden door een Jupyter-kernel is de functionaliteit die nodig is voor TESTed: het uitvoeren van fragmenten code en het resultaat van die uitvoering verzamelen.
- Eerder werk [27], dat gebruik maakt van Jupyter-kernels voor een gelijkaardig doel, rapporteert geen problemen met het gebruik van Jupyter-kernels.

```python
import re
import evaluation_utils

def listing(numbers):
    if len(numbers) == 1:
        return str(numbers[0])
    else:
        return f"{', '.join(str(x) for x in numbers[:-1])} en {numbers[-1]}"

def valid_lottery_numbers(number_str, count=6, maximum=42):
    if not isinstance(number_str, str):
        return False, "lottogetallen moet een string zijn"

    if not re.match("^(-?(0 -?[1-9][0-9]*) - )*(0 -?[1-9][0-9]*)$", number_str):
        return False, "lottogetallen worden niet in correct formaat teruggegeven"

    nrs = [int(x) for x in number_str.split(" - ")]
    if len(nrs) ≠ count:
        return False, f"verwachtte {count} in plaats van {len(nrs)} lottogetallen"

    if wrong := [number for number in nrs if number > maximum]:
        return (False, "volgende lottogetallen zijn groter dan de maximale waarde "
                       f"{maximum}: {listing(wrong)}")

    if wrong := [number for number in nrs if number < 1]:
        return (False, "volgende lottogetallen zijn kleiner dan de minimale "
                       f"waarde 1: {listing(wrong)}")

    duplicates = {number for number in nrs if nrs.count(number) > 1}
    if wrong := sorted(duplicates):
        return (False, "volgende lottogetallen komen meer dan één keer "
                       f"voor: {listing(wrong)}")

    if list(sorted(nrs)) ≠ nrs:
        return False, "lottogetallen worden niet in stijgende volgorde opgelijst"

    return True, None

def evaluate(expected, actual, arguments):
    count = arguments[0]
    maximum = arguments[1]
    valid, message = valid_lottery_numbers(actual, count, maximum)
    messages = ["Fout: " + message] if message else []
    # We geven geen verwachte waarde mee; TESTed neemt de waarde uit het testplan.
    evaluation_utils.evaluated(valid, None, actual, messages=messages)
```

Codefragment 2.17: De evaluatiecode voor de geprogrammeerde evaluatie van de voorbeeldoefening Lotto.

```
{
  "output": {
    "result": {
      "value": {
        "type": "text",
        "data": "1 - 6 - 7 - 11 - 13 - 14"
      },
      "evaluator": {
        "type": "specific",
        "evaluators": {
          "python":  "./evaluator.py",
          "haskell": "./evaluator.hs",
          "java": "./Evaluator.java"
        }
      }
    }
  }
}
```

Codefragment 2.18: Fragment uit een testplan waar een programmeertaalspecifieke evaluatie gebruikt wordt.

```
def evaluate_text(actual):
    """
    :param actual: The actual value produced by the student's code.
    """
    pass
```

Codefragment 2.19: De definitie van de functie die aanwezig moet zijn in de evaluatiecode voor een programmeertaalspecifieke evaluatie geschreven in Python.

```
import java.io.IOException;
import java.util.List;

abstract class AbstractSpecificEvaluator extends AbstractEvaluator {
    abstract void evaluate(Object actual) throws IOException;
}
```

Codefragment 2.20: De implementatie van de klasse `AbstractSpecificEvaluator`.

Omdat contexten per definitie onafhankelijk van elkaar zijn, moet de gebruikte Jupyter-kernel gestopt en opnieuw gestart worden tussen elke context. Dit brengt een onaanvaardbare performantiekost met zich mee, daar de meeste oefeningen niet computationeel intensief zijn. Het beoordelen van eenvoudige oefeningen, zoals de Lotto-oefening, duurde als snel meerdere minuten.

Enkele ideeën om de performantiekost te verkleinen waren:

- Het gebruiken van een *pool* kernels (een verzameling kernels die klaar staan voor gebruik). De werkwijze is als volgt:

  1. Bij de start van het beoordelen van een ingediende oplossing worden meerdere kernels gestart.
  2. Bij elke context wordt een kernel uit de pool gehaald om de testcode uit te voeren.
  3. Na het uitvoeren wordt de kernel op een andere draad (*thread*) opnieuw opgestart en terug aan de pool toegevoegd.

  Het idee achter deze werkwijze is dat door op een andere draad de kernels te herstarten, er altijd een kernel klaarstaat om de testcode uit te voeren, en er dus niet gewacht moet worden op het opnieuw opstarten van die kernels. In de praktijk bleek echter dat zelfs met een twintigtal kernels in de pool, het uitvoeren van de testcode van eenvoudige oefeningen dermate snel gaat in vergelijking met het opnieuw opstarten van de kernels, de kernels nooit op tijd herstart zijn.

- De kernels niet opnieuw opstarten, maar de interne toestand opnieuw instellen. In bepaalde kernels, zoals de Python-kernel, is dit mogelijk. De Python-kernel (IPython) heeft bijvoorbeeld een magisch commando `%reset`, dat de toestand van de kernel opnieuw instelt. Het probleem is er maar weinig kernels zijn die een gelijkaardig commando hebben.

Op dit punt is besloten dat de overhead van de Jupyter-kernels niet naar tevredenheid kon opgelost worden. Bovendien is een bijkomend nadeel van het gebruik van Jupyter-kernels ondervonden: de kernels voor andere programmeertalen dan Python zijn van gevarieerde kwaliteit. Zo schrijven de Java-kernel en de Julia-kernel bij het opstarten altijd een boodschap naar de standaarduitvoerstroom, wat in bepaalde gevallen problemen gaf bij het verzamelen van de resultaten van de uitvoer van de testcode.

2.5.2 Sjablonen

De keuze viel om verder te gaan met een systeem van sjablonen (zie ook paragraaf 2.3.1). Hierbij wordt de code gegenereerd en vervolgens uitgevoerd via de commandoregel.

Voordelen ten opzichte van de Jupyter-kernels zijn:

- Het is sneller om twee onafhankelijke programma's uit te voeren op de commandolijn dan een Jupyter-kernel tweemaal te starten.
- Het laat meer vrijheid toe in hoe de resultaten (gedrag en neveneffecten) van het uitvoeren van de testcode verzameld worden.

- Het implementeren en configureren van een programmeertaal in TESTed met de basisfunctionaliteit is minder werk dan dan het implementeren van een nieuwe Jupyter-kernel. Optionele functionaliteit, zoals linting, kan ervoor zorgen dat meer tijd en werk nodig is bij het toevoegen van een programmeertaal. Het systeem met de sjablonen hanteert echter het designprincipe dat optionele functionaliteit geen bijkomend werk mag betekenen voor programmeertalen die er geen gebruik van maken.
- Er is geen verschil tussen hoe TESTed de ingediende oplossing uitvoert en hoe de student diezelfde oplossing uitvoert. Bij de Jupyter-kernels is dat soms wel het geval: bij de R-kernel zitten subtiele verschillen tussen het uitvoeren in de kernel en het uitvoeren op de commandoregel.
- Herbruikbaar in die zin dat het genereren van testcode in een programmeertaal op basis van de programmeertaalonafhankelijke specificatie in het testplan sowieso nodig is.
- De gegenereerde testcode bestaat uit normale codebestanden (bv. `.py`- of `.java`-bestanden), wat het toevoegen van een programmeertaal aan TESTed eenvoudiger te debuggen maakt: alle gegenereerde testcode is beschikbaar voor inspectie als een bestand.

Elke medaille heeft ook een keerzijde:

- Het gebruik van sjablonen zorgt ervoor dat het uitvoeren van de testcode minder dynamisch kan zijn. Daar de testcode eerst gegenereerd moet worden, moet vóór het uitvoeren bepaald worden welke testcode zal uitgevoerd worden.
- Er moet meer zelf geïmplementeerd worden: het ecosysteem van Jupyter is redelijk groot, en er bestaan kernels voor veel talen (of in elk geval een begin van een kernel, waarop verder zou kunnen gewerkt worden)[4].
- Het uitvoeren op de commandoregel is bij veel programmeertalen trager indien er geen reset moet gebeuren tussen de verschillende uitvoeringen. Bij het uitvoeren op de commandoregel is er bij veel programmeertalen de performantiekost van het opstarten van de interpreter (zoals Python) of virtuele machine (zoals Java). Bij een kernel is de opstartkost weliswaar groter, maar deze wordt maar één keer opgestart als er geen reset nodig is.

Bij een eerste iteratie van het systeem met sjablonen bestond enkel de contextcompilatie (zie opnieuw paragraaf 2.3.1 voor een beschrijving van deze term). Dit heeft een grote performantiekost bij alle programmeertalen, maar in het bijzonder bij talen zoals Java. Daar moest voor elke context eerst een compilatie plaatsvinden, waarna de gecompileerde testcode werd uitgevoerd. Bij een testplan met bijvoorbeeld 50 contexten vertaalt dit zich in 50 keer de `javac`-compiler uitvoeren en ook 50 keer het uitvoeren van de code zelf met `java`.

In een tweede iteratie werd een „partiële compilatie” geïmplementeerd. Het idee hier is dat er veel testcode is die hetzelfde blijft voor elke context (denk aan de ingediende oplossing en hulpbestanden van TESTed). In de partiële compilatie wordt de gemeenschappelijke testcode eerst gecompileerd (bij programmeertalen die dat ondersteunen). Bij het beoordelen van een context werd dan enkel de testcode specifiek voor te beoordelen context gecompileerd.

In een derde iteratie werd de partiële compilatie uitgebreid naar batchcompilatie. Hoewel dit een grote winst voor de uitvoeringstijd betekende, heeft deze modus wel een ander groot nadeel (zoals reeds vermeld in paragraaf 2.3.1): bij statische talen zorgen compilatiefouten in één context ervoor dat geen enkele context beoordeeld kan worden, daar de compilatiefout ervoor zorgt dat er geen code gegenereerd wordt. Dit wordt best geïllustreerd met het reeds vermelde scenario van een

[4]Zie deze pagina voor een lijst van kernels: https://github.com/jupyter/jupyter/wiki/Jupyter-kernels.

oefening waarbij de student twee functies dient te implementeren. In batchcompilatie in bijvoorbeeld Java kan de student niet de oplossing voor de eerste functie indienen en laten beoordelen: omdat de tweede functie ontbreekt zullen compilatiefouten optreden.

De laatste en huidige iteratie bevat de mogelijkheid om contextcompilatie en batchcompilatie te combineren, waarbij TESTed terugvalt op de contextcompilatie indien er iets misgaat bij de batchcompilatie. Een eerdere versie van deze iteratie TESTed bevatte ook de mogelijkheid om contexten in parallel uit te voeren, wat mogelijk is omdat ze onafhankelijk zijn. Uiteindelijk is ervoor gekozen dit niet te doen, door meerdere redenen:

- De parallelle uitvoeren is vooral een theoretisch voordeel bij tijdsmetingen. In werkelijkheid, bij een uitvoering in het Dodona-platform, zijn er praktisch altijd meerdere gebruikers tezelfdertijd actief. Zij dienen ook oefeningen tegelijkertijd in, waardoor er meerdere beoordelingen tegelijk uitgevoerd worden op de server. Er is dus weinig snelheidswinst te halen door parallellisatie binnen de judge.
- De parallelle uitvoering zorgt voor niet-triviale problemen bij het implementeren van andere functies. Zo is het onduidelijk wat het gewenste gedrag is voor tijds- en geheugenlimieten, zoals hoe er betrouwbare metingen kunnen gebeuren die niet teveel afhangen van de gebruikslast van de server op het moment van uitvoering.
- Geen parallelle uitvoering is nuttig met het oog op de toekomst, meer specifiek op een mogelijke uitbreiding van het testplan waarbij *dependencies* tussen contexten kunnen aangeduid worden, bijvoorbeeld context 25 enkel uitvoeren indien contexten 1 en 5 geslaagd zijn. Meer hierover in paragraaf 5.2.2.

Een laatste performantieverbetering werd behaald op het vlak van geprogrammeerde evaluatie (zie paragraaf 5.3.1 voor verdere verbeteringen op dit vlak). Zoals beschreven in paragraaf 2.4.2, wordt normaliter voor elke geprogrammeerde evaluatie de evaluatiecode opnieuw gecompileerd en uitgevoerd in een nieuw subproces. Voor evaluatiecode die geschreven is in Python worden deze stappen overgeslagen. Er is speciale ondersteuning in TESTed ingebouwd om geprogrammeerde evaluaties waarvan de evaluatiecode in Python geschreven is, rechtstreeks in TESTed zelf uit te voeren.

Tabel 2.3 enkele tijdsmetingen van de verschillende implementaties van het systeem met sjablonen, voor de programmeertalen Python en Java. In deze tabel is de tijd gemeten in seconden om een (juiste) ingediende oplossing voor twee oefeningen te beoordelen: de voorbeeldoefening Lotto (met geprogrammeerde evaluatie) en een eenvoudigere oefening Echo (met enkel generieke evaluatie). Bij die laatste bestaat de opgave uit het implementeren van een functie die haar parameters naar de standaarduitvoerstroom schrijft. Elke beoordeling is uitgevoerd met contextcompilatie, partiële compilatie en batchcompilatie. De Lotto-oefening is ook gemeten met de optimalisatie voor geprogrammeerde evaluatie (de evaluatiecode voor deze oefening is geschreven in Python). Dit is aangeduid met de naam „Evaluatie” in de kolom „Compilatie”. Deze meting is niet uitgevoerd in parallelle modus, omdat ondersteuning voor deze modus reeds uit TESTed verwijderd was (hierboven wordt besproken waarom). Deze meting is niet uitgevoerd op de Echo-oefening, daar die oefening geen geprogrammeerde evaluatie heeft. Elke andere combinatie van oefening en compilatiemanier is ook uitgevoerd met en zonder parallelle modus. Deze tijdsmetingen zijn uitgevoerd op een standaardcomputer (Windows 10, Intel i7-8550U, Python 3.8.1, 64-bit), niet op de Dodona-server en niet in een Docker-container. Die laatste twee factoren kunnen ervoor zorgen dat uitvoeringstijden op Dodona sterk verschillen van deze metingen.

Zoals verwacht levert batchcompilatie het meeste tijdswinst op bij Java: daar heeft de compilatiestap ook een veel groter aandeel in de uitvoeringstijd. Bij Python is de compilatie veel sneller

		Python (s)		Java (s)	
Oefening	Compilatie	1 thread	4 threads	1 thread	4 threads
Lotto	Context	13	9	48	36
	Partieel	9	6	46	25
	Batch	8	5	14	10
	Evaluatie	5	-	10	-
Echo	Context	6	3	48	28
	Partieel	5	3	46	26
	Batch	5	3	8	5

Tabel 2.3: Tijdsmetingen voor de oefeningen Lotto en Echo, voor de programmeertalen Python en Java, in contextcompilatie, partiële compilatie en batchcompilatie (bij Lotto is deze laatste uitgevoerd met en zonder optimalisatie voor geprogrammeerde evaluaties).

en ook veel minder belangrijk (zelfs optioneel, de stap wordt enkel uitgevoerd om bepaalde syntaxisfouten vroeg op te vangen).

2.6 Bijkomende taken

Sommige judges doen meer dan enkel het oordelen over de juistheid van de ingediende oplossing. Zo heeft de Python-judge ondersteuning voor *linting*. Dit betekent dat de code van de ingediende oplossing geanalyseerd (maar niet uitgevoerd) wordt om zo allerlei mogelijke problemen op te sporen, zoals stijlfouten, mogelijke bugs, verdachte constructies, enz.

TESTed voorziet hier ondersteuning voor middels een optionele functie in de configuratie van een programmeertaal (zie hoofdstuk 3 voor het configureren van een taal in detail). Deze functie krijgt als argumenten onder andere de configuratie, het testplan en het pad naar de code van de ingediende oplossing mee. Als resultaat geeft een de functie een lijst van berichten en annotaties, die naar Dodona gestuurd worden. Ter illustratie dat deze aanpak werkt, is linting geïmplementeerd voor Python. Ook geeft dit veel vrijheid: de implementatie in Python gebruikt een bestand voor het configureren van de linter, en het is mogelijk om de naam van een eigen configuratiebestand (uit de oefeningenrepository) mee te geven via de programmeertaalspecifieke configuratie van TESTed. Tot slot nog opmerken dat het formaat van de annotaties wordt voorgeschreven door Dodona, maar dit is voldoende generiek voor praktisch alle scenario's. Ze ondersteunen onder andere:

- Het aangeven van een gedetailleerde plaats in de ingediende oplossing, zoals de regel en kolom. Het aanduiden van meerdere na elkaar of meerdere kolommen is ook mogelijk.
- De boodschap is vrij tekstveld.
- De ernst van de boodschap, met ondersteuning voor een veelvoorkomende indeling: informatiebericht, waarschuwing en foutbericht.

Een mogelijke uitbreiding is bijvoorbeeld het toevoegen van identificatienummers aan de berichten. Bij veel linters heeft elke boodschap een unieke code. Aan de hand van die code zou Dodona de uitleg over de boodschap kunnen tonen (maar dit vereist wel dat Dodona de uitleg voor specifieke linters heeft). Een alternatief zou zijn dat TESTed deze zelf toevoegt. De gemakkelijkste oplossing is echter de annotatie uitbreiding met ondersteuning voor een URL: bij veel linters kan met de identificatiecode van het bericht een URL geconstrueerd worden waarop meer informatie staat. Deze URL zou dan aan de studenten getoond kunnen worden.

2.7 Robuustheid

Een belangrijk aspect bij educative software testing is de feedback als het verkeerd loopt. De feedback bij een verkeerde oplossing is in veel gevallen zelfs belangrijker dan de feedback bij een juiste oplossing: het is namelijk de bedoeling dat als studenten een verkeerde oplossing indienen, de feedback die Dodona geeft ze op weg kan helpen om hun oplossing te verbeteren. De feedback die Dodona toont is afkomstig uit de judges: het is de taak van TESTed om kwalitatief hoogstaande feedback te voorzien. Met kwalitatief hoogstaand wordt bedoeld dat de feedback nuttige informatie bevat, maar ook geen verkeerde of misleidende informatie bevat.

2.7.1 In de praktijk

In het vak *Scriptingtalen* van de bachelor informatica krijgen de studenten elke week een reeks oefeningen om op te lossen (op Dodona). Een van die oefeningen is de ISBN-oefening. Om een idee te krijgen van hoe TESTed werkt in de praktijk is besloten om een van de weken de ISBN-oefening te vervangen door een oefening met dezelfde opgave, maar die beoordeeld wordt door TESTed in plaats van de Python-judge.

De algemene bevindingen die uit de praktijktest naar boven kwamen waren:

- Geen ondersteuning voor de Python Tutor. Dit is al vermeld, en wordt verder besproken in paragraaf 5.3.6.
- De evaluatie duurt ongeveer dubbel zo lang als met de Python-judge. Aan de ene kant zou de optimalisatie van de geprogrammeerde evaluatie (zie paragraaf 2.5) dit moeten verbeteren (maar wel in dit specifieke geval, moest de evaluatiecode in een andere programmeertaal dan Python geschreven zijn, zou het geen verschil maken). Aan de andere kant ligt een tragere uitvoer in vergelijking met de Python-judge binnen de verwachtingen: TESTed voert elke context uit in een afzonderlijk subproces. De Python-judge doet dit niet. Qua uitvoer worden wel contexten gebruikt, maar de interne werking van de Python-judge kan het best vergeleken worden met een oefeningen waar alle testen in dezelfde context opgenomen zijn.
- Er is geen verwerking van stacktraces. Deze functie is opgenomen als mogelijke uitbreiding als onderdeel van paragraaf 5.3.6.
- Er is geen linting. Naar aanleiding van de praktijktest is hier ondersteuning voor toegevoegd, zie paragraaf 2.6.
- Geen ondersteuning voor vertalingen op het vlak van natuurlijke talen. Deze functie wordt besproken als uitbreiding in paragraaf 5.3.2.

Verder heeft professor Dawyndt heel heleboel fouten en scenario's uitgeprobeerd. Deze worden hieronder besproken.

2.7.2 Soorten fouten

TESTed moet robuust zijn tegen allerlei vormen van fouten in de ingediende oplossingen. Hieronder volgt een lijst van (categorieën) van fouten waarvoor TESTed nuttige feedback geeft. TESTed is zo opgebouwd dat er altijd iets van feedback komt, ook in onvoorziene omstandigheden, maar voor de soorten fouten op onderstaande lijst is expliciet gecontroleerd wat de kwaliteit van de feedback is.

Compilatiefouten De uitvoer van de compiler wordt altijd getoond aan de studenten, dus op dat vlak wordt juist afgehandeld. Er bestaat ook de mogelijkheid om de compilatie-uitvoer te verwerken en bijvoorbeeld om te zetten naar annotaties, die dan in de code worden getoond. Hierbij wordt vooral gedacht aan foutboodschappen als „syntaxisfout op regel 5, kolom 2". De stacktraces bij deze uitvoer bevatten wel referenties naar code die TESTed gegenereerd heeft. Deze verwijzingen naar interne code wegfilteren is opgenomen als uitbreiding als onderdeel van paragraaf 5.3.6. Bij een batchcompilatie wordt de uitvoer in een nieuw tabblad getoond, terwijl bij contextcompilatie de uitvoer bij de relevant context staat.

Uitvoeringsfouten Hier gaat het om crashes tijdens de uitvoering, zoals delen door nul. Ook hier wordt nog geen verdere verwerking van de foutboodschap gedaan (ook onderdeel van paragraaf 5.3.6). Een bijkomende opmerking is dat sommige fouten tijdens de uitvoering nu als uitzondering opgevangen worden.

Tijdslimieten TESTed heeft ondersteuning voor tijdslimieten in de judge, dit laat toe om meer uitvoer te tonen dan als de tijdslimiet aan Dodona wordt overgelaten. Momenteel werkt de implementatie ervan als volgt: TESTed houdt bij hoe lang de beoordeling al duurt, en gebruikt de resterende tijd als een limiet op de uitvoering van een context. De eerste context heeft dan als limiet de volledige toegestane tijd (minus een percentage dat voorbehouden wordt voor TESTed zelf). De laatste context zal de kleinste limiet krijgen (de tijd die nog overschiet). Scenario's waar dit voorkomt zijn bijvoorbeeld oplossingen die te traag werken, maar ook oplossingen met bijvoorbeeld oneindige lussen. Als uitbreiding opgenomen in paragraaf 5.3.6 wordt beschreven dat er ook een tijdslimiet per context mogelijk zou kunnen zijn (zodat de eerste context niet de volledige beschikbare tijd krijgt, maar slechts een deel ervan). De hoofdreden om zelf de tijdslimiet te implementeren in TESTed is omdat zo de overige, niet-uitgevoerde testen ook getoond kunnen worden. Momenteel is dat met een boodschap die uitlegt dat ze niet zijn uitgevoerd, maar op termijn is het de bedoeling in Dodona een bijkomende status toe te voegen.

Te grote uitvoer Bij te grote uitvoer zal TESTed deze limiteren. Hierbij gaat het om scenario's zoals een oneindige lus die blijft schrijven naar stdout. TESTed beperkt zowel het aantal regels (maximaal 20) en de regellengte (maximaal 150) tekens. Dit is enkel bij tekstuele uitvoer: het inkorten van grote verzamelingen (bijvoorbeeld een lijst van duizend elementen) is opgenomen als uitbreiding in paragraaf 5.3.6.

Te veel uitvoer Hierbij wordt gedacht aan zaken zoals uitvoer op stdout of stderr terwijl die niet verwacht wordt. Standaard wordt een oefening als fout beschouwd indien er te veel uitvoer is, maar de auteur van de oefening kan kiezen om het teveel aan uitvoer te negeren (bijvoorbeeld als de studenten debugberichten schrijven naar stdout is dit niet in elke oefening een probleem).

Vroegtijdig stoppen van uitvoering Hier gaat het om code van de studenten die bijvoorbeeld `exit(-5)` bevat. Dit zal fout gerekend worden door TESTed.

3 Case-study: configureren van een programmeertaal in TESTed

Allerlei uitleg

4 Case-study: nieuwe oefening

In dit hoofdstuk behandelen we het toevoegen van drie oefeningen in handleidingsstijl. Elke oefening gebruikt een andere evaluatiemethode uit paragraaf 2.4. We gaan er van uit dat de map voor de repo voor deze oefeningencollectie al bestaat. Deze moet voldoen aan de mappenstructuur voor oefeningen, opgelegd door Dodona[1].

4.1 ISBN

Bij de eerste oefening gebruiken we enkel ingebouwde evaluators.

4.1.1 Voorbereiding

We beginnen met het maken van een nieuwe map `isbn` in onze map voor de oefeningencollectie. Vervolgens maken we in deze map een configuratiebestand `config.json` voor onze oefening, met deze inhoud:

```
{
  "description": {
    "difficulty": 2.0,
    "names": {
      "en": "ISBN",
      "nl": "ISBN"
    }
  },
  "access": "public",
  "programming_language": "python",
  "evaluation": {
    "plan_name": "plan.json"
  }
}
```

In dit bestand doen we drie belangrijke dingen:

1. We geven onze oefening een naam.
2. We duiden aan dat het om een Python-oefening gaat (zie hoofdstuk 5 voor waarom we dit doen).
3. We zeggen dat ons testplan als naam `plan.json` heeft.

[1]Hier beschikbaar: https://dodona-edu.github.io/en/references/exercise-directory-structure/

4.1.2 Opgave

In deze stap stellen we de opgave op. Om dit deel kort te houden, gaan we er van uit dat de opgave al bestaat. Maak een map `description` in onze oefeningenmap, en kopieer de opgavebestanden naar deze map. Uiteindelijk moet deze map er zo uitzien:

```
isbn
└── description
    ├── description.en.html
    ├── description.nl.html
    └── media
        └── ISBN.gif
```

Om toch een idee te krijgen van waarover de oefening gaat, is hieronder een samenvatting van de opgave:

- Schrijf een functie `is_isbn` waaraan een string *c* (`str`) moet doorgegeven worden. De functie moet een Booleaanse waarde (`bool`) teruggeven, die aangeeft of *c* een geldige ISBN-code is. De functie heeft ook nog een optionele tweede parameter `isbn13` waaraan een Booleaanse waarde (`bool`) kan doorgegeven worden die aangeeft of het om een ISBN-10 code (`False`) of om een ISBN-13 code (`True`, standaardwaarde) moet gaan.
- Schrijf een functie `are_isbn` waaraan een lijst (`list`) met $n \in \mathbb{N}$ codes moet doorgegeven worden. De functie moet voor alle codes uit de gegeven lijst aangegeven of ze geldige ISBN-codes voorstellen. De functie heeft ook nog een tweede optionele parameter `isbn13` waaraan een Booleaanse waarde (`bool`) kan doorgegeven worden die aangeeft of het om ISBN-10 codes (`False`) of om ISBN-13 codes (`True`) moet gaan.
  Als er niet expliciet een waarde wordt doorgegeven aan de parameter `isbn13`, dan moet het type van elke code uit de lijst bepaald worden op basis van de lengte van die code. Als een code geen string (`str`) is, dan wordt die *a priori* als ongeldig bestempeld. Voor codes van lengte 13 moet getest worden of het geldige ISBN-13 codes zijn, en voor codes van lengte 10 of het geldige ISBN-10 codes zijn. Codes met afwijkende lengtes (geen 10 en geen 13) worden ook *a priori* als ongeldige ISBN-codes bestempeld.
  De functie moet een nieuwe lijst (`list`) met *n* Booleaanse waarden (`bool`) teruggeven, die aangeven of de code op de corresponderende positie in de gegeven lijst een geldige ISBN-code is.

4.2 Testplan

Nu komen we aan het belangrijkste deel van het opstellen van een oefening: het testplan. Om te beginnen kunnen we even nadenken over hoe de structuur er zal uitzien. We hebben twee functies die getest moeten worden. Laten we de in- en uitvoer voor deze functies X_i en X_o noemen. Aangezien elke functieoproep onafhankelijk is, kiezen we ervoor om ze in aparte contexten van het testplan te steken. Verder kunnen we al zeggen dat elke context juist één testgeval zal hebben: de functieoproep. Elk testgeval zal ook maar één test hebben, waar we de returnwaarde controleren.

Alle andere uitvoerkanalen mogen de standaardwaarde behouden (wat betekent dat er bv. geen exceptions mogen zijn, een lege standaardfoutstroom, de standaarduitvoerstroom wordt genegeerd, enz.). We verdelen de contexten bovendien in twee tabbladen, een voor elke functies. De structuur van het testplan ziet er dus als volgt uit:

- Tabblad voor `is_isbn`
  - Context voor invoer X_{i1} en uitvoer X_{o1}
  - Context voor invoer X_{i2} en uitvoer X_{o2}
  - Context voor invoer X_{i3} en uitvoer X_{o3}
  - Context voor invoer X_{i4} en uitvoer X_{o4}
  - ...
- Tabblad voor `are_isbn`
  - Context voor invoer X_{i1} en uitvoer X_{o1}
  - Context voor invoer X_{i2} en uitvoer X_{o2}
  - Context voor invoer X_{i3} en uitvoer X_{o3}
  - Context voor invoer X_{i4} en uitvoer X_{o4}
  - ...

In plaats van dit testplan manueel te schrijven, kiezen we ervoor om een Python-scriptje te schrijven, dat dit testplan voor ons genereert. Om te beginnen:

- Kopieer `solution.py` naar de map `solution`. Dit is de voorbeeldoplossing.
- Kopieer `values.py` naar de map `preparation`. Deze module encodeert waarden in het serialisatieformaat. In de toekomst is dit misschien niet meer nodig, moest dit als Python-package gepubliceerd worden.

Maak nu een bestand in de map `preparation` genaamd `generator.py`. Hieronder volgt het script dat het testplan genereert. Het is rijkelijk voorzien van commentaar, zodat alles duidelijk is.

```
import random
import string
import json

# De implementatie van het serialisatieformaat.
from isbn.preparation import values
# De voorbeeldoplossing.
from isbn.solution import solution

# Met een vaste seed krijgen we deterministische resultaten.
random.seed(123456789)

# We halen de naam van het testplan op uit de configuratie van de oefening.
with open("../config.json", "r") as config_file:
    config = json.load(config_file)
testplan_name = config["evaluation"].get("plan_name", "plan.json")
```

```python

# Of alle testgevallen in dezelfde context moeten plaatsvinden of niet.
# Aangezien ze onafhankelijk zijn, doen we dit niet.
# Meerdere contexten hebben echter wel een performantiekost, waardoor we de
# optie toch voorzien.
ONE_CONTEXT = False


def check_digit10(code):
    """Bereken het controlecijfer voor een ISBN van lengte 10."""
    check = sum((i + 1) * int(code[i]) for i in range(9)) % 11
    # Zet het controlecijfer om naar een string.
    return 'X' if check == 10 else str(check)


def check_digit13(code):
    """Bereken het controlecijfer voor een ISBN van lengte 13."""
    check = sum((3 if i % 2 else 1) * int(code[i]) for i in range(12))
    # Zet het controlecijfer om naar een string.
    return str((10 - check) % 10)


def random_characters(length, alphabet):
    """Genereer een aantal willekeurige tekens uit een alfabet."""
    return ''.join(random.choice(alphabet) for _ in range(length))


def generate_code():
    """Genereer een ISBN-code, met zowel lengte 10 als 13."""
    length = random.choice([10, 13])
    code = random_characters(length - 1, string.digits)
    if length == 10:
        if random.random() < 0.5:
            code += check_digit10(code)
        else:
            code += random.choice(string.digits + 'X')
    else:
        if random.random() < 0.5:
            code += check_digit13(code)
        else:
            code += random.choice(string.digits)
    return code


def generate_is_isbn():
    """
    Genereer de contexten voor de "is_isbn"-functie.
    :return: De gegenereerde contexten.
    """
```

```
    contexts = []

    # Genereer eerst de argumenten voor de functie "is_isbn".
    # We beginnen met wat vaste combinaties.
    args = [
        ('9789027439642', False),
        ('9789027439642', True),
        ('9789027439642', None),
        ('080442957X', None),
        ('080442957X', False),
        (9789027439642, None),
    ]
    # Voor de rest vullen we aan met willekeurige argumenten.
    while len(args) < 50:
        code = generate_code()
        args.append((code, random.choice([None, True, False])))

    # Genereer de eigenlijke contexten.
    for code, isbn13 in args:
        # Eerst doen we de functieoproep. We geven zeker de ISBN mee.
        function_arguments = [
            values.encode(code)
        ]
        # Indien nodig geven we ook het tweede argument mee.
        if isbn13 is not None:
            function_arguments.append(values.encode(isbn13))

        # Bereken het resultaat met de gegeven argumenten.
        result = solution.is_isbn(code, isbn13 if isbn13 is not None else True)

        # Ons testgeval bevat de functieoproep als invoer, en de berekende waarde
        # als verwachte uitvoer.
        testcase = {
            "input":  {
                "expression": {
                    "type":      "function",
                    "name":      "is_isbn",
                    "arguments": function_arguments
                }
            },
            "output": {
                "result": {
                    "value": (values.encode(result))
                }
            }
        }

        # Steek het testgeval in een context.
        context = {
```

```
            "testcases": [testcase]
        }
        contexts.append(context)

    return contexts


def generate_are_isbn():
    """
    Genereer de contexten voor de "are_isbn"-functie.
    :return: De gegenereerde contexten.
    """

    # Vaste invoerargumenten die we zeker in het testplan willen.
    codes = [
        '0012345678', '0012345679', '9971502100', '080442957X', 5, True,
        'The Practice of Computing Using Python', '9789027439642', '5486948320146'
    ]
    codes2 = ['012345678' + str(digit) for digit in range(10)]
    args = [
        (codes, None),
        (codes, True),
        (codes, False),
        (codes2, None),
        (codes2, True),
        (codes2, False),
    ]
    # Vul opnieuw de rest aan tot we aan 50 zitten.
    while len(args) < 50:
        codes = [generate_code() for _ in range(random.randint(4, 10))]
        args.append((codes, random.choice([None, True, False])))

    # Genereer de eigenlijke contexten.
    contexts = []
    for index, (codes, isbn13) in enumerate(args):
        index += 1
        # Deze keer pakken we het iets anders aan: de lijst van ISBN's kennen we
        # eerst toe aan een variabele, en geven het niet rechtstreeks mee als
        # argument. Maak het testgeval voor de assignment.
        assignment_testcase = {
            "input": {
                "statement": {
                    "name": f"codes{index:02d}",
                    "expression": values.encode(codes),
                    "type": "sequence"
                }
            }
        }

```

```
        # Stel de functieargumenten op. We geven opnieuw sowieso de variabele die
        # we eerst hebben aangemaakt mee als argument.
        function_arguments = [
            f"codes{index:02d}"
        ]

        # Voeg het tweede argument toe indien nodig.
        if isbn13 is not None:
            function_arguments.append(values.encode(isbn13))

        # Bereken het resultaat.
        result = solution.are_isbn(codes, isbn13 if isbn13 is not None else None)

        # Maak het normale testgeval. We hebben opnieuw als invoer de functie en als
        # uitvoer de verwachte waarde.
        testcase = {
            "input":  {
                "expression": {
                    "type":      "function",
                    "name":      "are_isbn",
                    "arguments": function_arguments
                }
            },
            "output": {
                "result": {
                    "value": (values.encode(result))
                }
            }
        }

        # Voeg beide testcases toe aan een context.
        # Elke testcase heeft hoogstens één functieoproep of assignment, maar niet
        # beide. Daarom hebben we nu twee testgevallen. Het eerste testgeval voor
        # de assignment heeft enkel de standaardtests, wat wil zeggen dat er bv.
        # geen uitvoer op stderr mag zijn.
        context = {
            "testcases": [assignment_testcase, testcase]
        }

        contexts.append(context)
    return contexts


def flatten_contexts(contexts):
    """Voeg alle testgevallen samen tot 1 context."""
    testcases = [context["normal"] for context in contexts]
    flat = [item for sublist in testcases for item in sublist]
    new_context = {
        "testcases": flat
```

```
    }
    return new_context


# Creëer de contexten.
tab_1_contexts = generate_is_isbn()
tab_2_contexts = generate_are_isbn()

# Creëer het testplan met de twee tabbladen.
plan = {
    "tabs": [
        {
            "name": "is_isbn",
            "contexts": tab_1_contexts
        },
        {
            "name": "are_isbn",
            "contexts": tab_2_contexts
        }
    ]
}

# Indien we alles in één context willen, doen we dat.
if ONE_CONTEXT:
    new_tab1_context = flatten_contexts(plan["tabs"][0]["contexts"])
    plan["tabs"][0]["contexts"] = [new_tab1_context]
    new_tab2_context = flatten_contexts(plan["tabs"][1]["contexts"])
    plan["tabs"][1]["contexts"] = [new_tab2_context]

# Schrijf het testplan.
with open(f"../evaluation/{testplan_name}", 'w') as fp:
    json.dump(plan, fp, indent=2)
```

4.2.1 Afsluiting

Op dit moment is de oefening klaar. Ter controle: de complete mapstructuur van de uiteindelijke oefening wordt nog eens getoond in codefragment 4.1. Nu rest nog enkel de oefening importeren in Dodona. Is de oefening toegevoegd aan een bestaande repo, dan volstaat het om op de knop „Alle oefeningen opnieuw verwerken” te drukken. Gaat het om een nieuwe oefeningenrepo, dan moet deze toegevoegd worden[2].

[2]De handleiding hiervoor staat op https://dodona-edu.github.io/en/guides/new-exercise-repo/

```
isbn
├── config.json
├── description
│   ├── description.en.html
│   ├── description.nl.html
│   └── media
│       └── ISBN.gif
├── evaluation
│   └── plan.json      # Gegenereerd door generator.py
├── preparation
│   ├── generator.py
│   └── values.py
└── solution
    └── solution.py
```

Codefragment 4.1: Finale mapstructuur van de oefening ISBN.

5 Beperkingen en toekomstig werk

De doelstelling van deze thesis is het formuleren van een antwoord op de vraag of programmeertaalonafhankelijke oefeningen mogelijk zijn (zie paragraaf 1.5). Als antwoord hierop is een prototype, TESTed, ontwikkelt. Op basis van het prototype blijkt dat het antwoord op de onderzoeksvraag positief is. Eigen aan een prototype is dat het nog niet *production ready* is. Dit hoofdstuk handelt over ontbrekende functionaliteit of beperkingen van TESTed, en reikt mogelijke ideeën voor een oplossing aan.

5.1 Dodona-platform

Geen enkele bestaande judge in het Dodona-platform ondersteunt meerdere programmeertalen. Dit heeft gevolgen voor hoe Dodona omgaat met programmeertalen, zoals hieronder duidelijk wordt.

5.1.1 Programmeertaalkeuze in het Dodona-platform

Een eerste probleem is dat binnen Dodona de programmeertaal gekoppeld is aan een oefening, niet aan een oplossing. Dit impliceert dat elke oplossing van een oefening in dezelfde programmeertaal geschreven is. Een aanname die niet meer geldt bij TESTed. Een uitbreiding van Dodona bestaat er dus in om ondersteuning toe te voegen voor het kiezen van een programmeertaal per oplossing. Dit moet ook in de gebruikersinterface mogelijk zijn, zodat studenten kunnen kiezen in welke programmeertaal ze een oefening oplossen.

Als *workaround* zijn er twee suboptimale opties:

- Dezelfde oefening in meerdere programmeertalen aanbieden, door bijvoorbeeld met symlinks dezelfde opgave, testplan en andere bestanden te gebruiken, maar met een ander configbestand. Op deze manier zullen er in Dodona meerdere oefeningen zijn. Deze aanpak heeft nadelen. Zo zijn de oefeningen effectief andere oefeningen vanuit het perspectief van Dodona. Dient een student bijvoorbeeld een oplossing in voor één van de oefeningen, worden de overige oefeningen niet automatisch als ingediend gemarkeerd. Ook is de manier waarop de opgave, het testplan, enzovoort gedeeld worden (de symlinks) niet echt handig.
- TESTed zelf de programmeertaal laten afleiden op basis van de ingediende oplossing. Hiervoor zijn verschillende implementaties mogelijk, zoals een heuristiek die op basis van de ingediende broncode de programmeertaal voorspelt of de student die programmeertaal expliciet laten aanduiden. Er is voorlopig gekozen om de tweede manier te implementeren: TESTed ondersteunt een speciale *shebang*[1] die aangeeft in welke programmeertaal de ingediende oplossing beoordeeld moet worden. De eerste regel van de ingediende oplossing kan er als volgt uitzien:

[1] Zie https://en.wikipedia.org/wiki/Shebang_(Unix).

```
#!tested [programmeertaal]
```

Hierbij wordt `[programmeertaal]` vervangen door de naam van de programmeertaal, zoals `java` of `python`. Dit is een voorlopige oplossing die toelaat om oefeningen in meerdere programmeertalen aan te bieden zonder dat Dodona moet aangepast worden. Een nadeel aan deze oplossing is dat de student niet kan zien in welke programmeertalen ingediend kan worden; dit moet door de lesgever aangeduid worden.

5.1.2 Detectie ondersteunde programmeertalen

Een ander gevolg van de koppeling van de programmeertaal aan een oefening binnen Dodona is dat Dodona momenteel altijd weet voor welke programmeertaal een oplossing werd ingediend: de programmeertaal van de oefening. Het is echter niet voldoende dat de programmeertaal van de oplossing gekend is bij het indienen, het is ook nuttig te weten in welke programmeertalen een oefening gemaakt kan worden. Dit om enerzijds dingen als syntaxiskleuring te kunnen toepassen, maar ook om ervoor te zorgen dat een student geen nutteloos werk verricht en een oplossing maakt in een niet-ondersteunde programmeertaal.

Zoals besproken in paragraaf 2.2.4, controleert TESTed of de programmeertaal van de ingediende oplossing ondersteund wordt door het testplan van de oefening. Die controle komt echter te laat, namelijk bij het beoordelen van een oplossing. Aan de andere kant is de controle onafhankelijk van de oplossing: enkel het testplan is nodig. Een beter ogenblik om de ondersteunde programmeertalen van een oefening te bepalen is bij het importeren van de oefening in het Dodona-platform. Bij het importeren gebeurt al een verwerking van de oefening door Dodona, om bijvoorbeeld de configuratiebestanden te lezen en de oefening in de databank op te slaan. Deze verwerking zou kunnen uitgebreid worden om ook de ondersteunde programmeertalen te bepalen. Merk op dat deze controle opnieuw moet uitgevoerd worden elke keer dat een nieuwe programmeertaal aan TESTed toegevoegd wordt.

De volgende vraag die zich stelt is hoe dit praktisch moet aangepakt worden. Het bepalen van de ondersteunde programmeertalen is iets specifieks voor TESTed, terwijl judge-specifieke code zoveel mogelijk buiten Dodona wordt gehouden. Een mogelijke oplossing is de verwerking van de oefeningen uit te breiden, zodat de judge ook een verwerking kan doen van de oefening. Deze aanpak heeft als voordeel dat het een generieke oplossing is: alle judges krijgen de mogelijkheid tot het verwerken van de oefening.

Een andere variant van deze oplossing wordt overwogen door het Dodona-team[2]. Het voorstel is om oefeningen en judges te combineren in Dodona tot één Docker-container. Interessant binnen TESTed is het toevoegen van een `prepare`-stap aan de judge (momenteel heeft een judge slechts één stap, namelijk `run`, het beoordelen). Deze voorbereidende stap resulteert dan in een Docker-container op maat van de oefening. Het is in deze voorbereidende stap dat het detecteren van de ondersteunde programmeertalen zou plaatsvinden.

Een andere oplossing voor dit probleem is de ondersteunde programmeertalen manueel opgeven in de configuratiebestanden van de oefening. Hier is dan weer het voordeel dat de verwerkingsstap van Dodona niet uitgebreider en trager moet worden. Aan de andere kant mag de auteur van de oefening dan niet vergeten om de programmeertalen in de configuratie op te nemen. Eventueel kan dit gemakkelijker gemaakt worden door een script te voorzien dat de ondersteunde programmeertalen detecteert en het resultaat in de configuratiebestanden schrijft. In de twee eerdere

[2] Zie https://github.com/dodona-edu/dodona/issues/1754

voorstellen wordt telkens gesproken over een verwerkingsstap, waarin het detecteren plaatsvindt. Bij deze laatste oplossing verschuift de verantwoordelijkheid voor het uitvoeren van deze stap van Dodona naar de auteur van de oefening.

5.1.3 Beperken ondersteunde programmeertalen

De vorige paragraaf heeft het steeds over het automatisch detecteren van de ondersteunde programmeertalen. Om twee redenen kan het echter nuttig zijn om deze automatisch detectie te overschrijven.

Ten eerste is de automatische detectie niet altijd voldoende streng, wat ertoe leidt dat TESTed een programmeertaal als ondersteund beschouwd, terwijl de oefening niet oplosbaar is in die programmeertaal. Een concreet voorbeeld hiervan is de detectie van functieargumenten met heterogene gegevenstypes (zie paragraaf 2.2.4 voor een voorbeeld). De automatische detectie houdt enkel rekening met *literal* functieargumenten. De gegevenstypes van functieoproepen en identifiers worden niet gecontroleerd, aangezien deze informatie niet eenvoudig af te leiden is uit het testplan. In dit eerste geval is het dus nuttig dat de auteur van de oefening op het niveau van een oefening een *blacklist* of zwarte lijst van programmeertalen kan meegeven: programmeertalen waarin de oefening niet gemaakt kan worden. Een belangrijk detail is dat als er nieuwe programmeertalen bijkomen in TESTed, de oefening standaard wel ondersteund zou worden in die nieuwe programmeertaal.

De andere grote reden is dat lesgevers de programmeertalen waarin een oefening opgelost kan worden willen beperken. Dit kan bijvoorbeeld zijn omdat de lesgever slechts programmeertalen wil toelaten die hij machtig is, of dat de oefening gebruikt wordt in een vak waar één bepaalde programmeertaal onderwezen wordt. Binnen Dodona stelt een lesgever binnen een vak reeksen oefeningen op. Die reeksen kunnen bestaan uit eigen, zelfgeschreven oefeningen, maar kunnen ook bestaan uit oefeningen die beschikbaar zijn in het Dodona-platform. De auteur van de oefening en lesgever zijn dus niet altijd dezelfde persoon. Om die reden is het nuttig als de lesgevers op het niveau van het vak, de reeks of de oefening een *whitelist* of witte lijst van programmeertalen kunnen opgeven: oefeningen in dat vak of die reeksen zullen enkel in de toegelaten programmeertalen kunnen opgelost worden. Bij deze beperking worden nieuwe programmeertalen niet automatisch ondersteund. Het zou ook nuttig zijn ook hier met een zwarte lijst te kunnen werken, waarbij nieuwe talen wel automatisch ondersteund worden.

5.2 Testplan

De gekozen aanpak voor het testplan heeft bepaalde beperkingen (zie paragraaf 2.2.1 voor de gemaakte keuze). Deze paragraaf bespreekt drie grote beperkingen en mogelijke oplossingen.

5.2.1 Handgeschreven testplannen zijn omslachtig

In TESTed is gekozen om het testplan in JSON op te stellen (zie paragraaf 2.2.1). Bovendien is een designkeuze geweest om zoveel mogelijk informatie expliciet te maken. Dit heeft als voornaamste voordeel dat de implementatie in TESTed eenvoudiger blijft. Anderzijds zorgt dit wel voor redelijk wat herhaalde informatie in het testplan.

De combinatie van JSON met de herhaalde informatie zorgt ervoor dat een testplan vaak lang is. Te lang om nog met de hand te schrijven. Een oplossing hiervoor, die al gebruikt wordt, is om

het testplan niet met de hand te schrijven, maar te laten generen door bijvoorbeeld een Python-script.

Een andere oplossing is het toevoegen van een bijkomende stap voor het testplan: het testplan wordt dan in een ander formaat geschreven, en bijvoorbeeld tijdens het importeren in Dodona omgezet naar het JSON-bestand. Voor dat andere formaat kan een DSL (*domain-specific language*) nuttig zijn. Dit is een programmeertaal die specifiek ontworpen is voor een bepaald onderwerp of doel. Hier zou ze specifiek ontworpen zijn om een testplan te schrijven.

Een bijkomend voordeel van JSON als formaat is dat het een neutraal formaat is. Verschillende soorten oefeningen hebben andere noden op het vlak van schrijven van het testplan. Het testplan voor een oefening die enkel stdin en stdout gebruikt zal er helemaal anders uitzien dan een testplan voor een oefening waar functieoproepen gebruikt worden. Het zou bijvoorbeeld kunnen dat voor verschillende soorten oefeningen een andere DSL ontworpen wordt.

Zo zijn er bijvoorbeeld meerdere standaarden voor het schrijven van programmeeroefeningen. Op het moment van schrijven lijken er wel nog geen wijdverspreide standaarden te zijn. Een programma dat oefening die in deze formaten beschreven zijn omzet naar het eigen formaat van TESTed kan een nuttige uitbreiding zijn. Bij wijze van voorbeeld worden twee standaarden besproken, telkens met een beschrijving van dezelfde oefening. Bij deze voorbeeldoefening moet invoer gelezen worden en dan als uitvoer moet aangegeven worden of de invoer al dan niet een palindroom is.

Een eerste standaard is PEXIL (*Programming Exercises Interoperability Language*) [29]. Dit is een op XML-gebaseerd formaat, gepubliceerd in 2011. Ontwikkeling van deze standaard lijkt gestopt te zijn. We hebben geen online oefeningen gevonden die in dit formaat beschreven zijn en de de nodige resources voor het zelf schrijven van een oefeningen (zoals het XML Schema) zijn niet meer beschikbaar online. De beschrijving van de voorbeeldoefening is codefragment 5.1.

Een tweede standaard is PEML (*Programming Exercise Markup Language*) [9]. PEML is een tekstueel formaat (een soort DSL) dat er veelbelovend uitziet om met de hand te schrijven. Op het moment van schrijven van deze thesis is PEML nog in volle ontwikkeling. Aangezien een van de *design goals* van PEML is dat de omzetting van PEML naar JSON eenvoudig zal zijn, lijkt dit een interessant formaat. De beschrijving van de voorbeeldoefening staat in codefragment 5.2 en is overgenomen uit de documentatie van PEML.

5.2.2 Dynamische scheduling van testen

In TESTed is gekozen om het testplan in JSON op te stellen (zie paragraaf 2.2.1). Een nadeel van deze aanpak is dat de *scheduling* (het plannen van welke testcode wanneer uitgevoerd worden) statisch gebeurt. Een analogie is bijvoorbeeld een gecompileerde taal zoals C/C++ en een dynamische taal zoals Python: TESTed vervult in deze analogie de rol van compiler. Door dit statische testplan is het niet mogelijk om in het algemeen op basis van het resultaat van een vorig testgeval het verloop van de volgende testgevallen te bepalen.

Het statisch testplan *an sich* is echter voor specifieke gevallen geen probleem. Het testplan kan uitgebreid kunnen worden met verschillende mogelijkheden. Zo is reeds voorzien in het testplan om een testgeval als essentieel aan te duiden: faalt het testgeval, dan zullen volgende testgevallen niet meer uitgevoerd worden. Dit zou uitgebreid kunnen worden zodat het ook op het niveau van contexten werkt.

```xml
<specification line_terminator="\n" id="input">
    <line>
        <data type="text" value="racecar"/>
    </line>
    <line>
        <data type="text" value="Flintstone"/>
    </line>
</specification>
```

```xml
<specification line_terminator="\n" id="output">
    <line>
        <data type="text">
            "racecar" is a palindrome."
        </data>
    </line>
    <line>
        <data type="text">
            "Flintstone" is not a palindrome.
        </data>
    </line>
</specification>
```

Codefragment 5.1: Beschrijving van de oefening „palindroom” (gesplitst in invoer en uitvoer) in PEXIL.

```
exercise_id: edu.vt.cs.cs1114.palindromes

title: Palindromes (A Simple PEML Example)

topics: Strings, loops, conditions
prerequisites: variables, assignment, boolean operators

instructions: ----------
Write a program that reads a single string (in the form of one line of text) from
its standard input, and determines whether the string is a _palindrome_. A
palindrome is a string that reads the same way backward as it does forward, such as
"racecar" or "madam". Your program does not need to prompt for its input, and should
only generate one line of output, in the following format:

<pre>"racecar" is a palindrome.</pre>

or

<pre>"Flintstone" is not a palindrome.</pre>
----------

assets.test_format: stdin-stdout

[assets.tests]
stdin: racecar
stdout: "racecar" is a palindrome.

stdin: Flintstone
stdout: "Flintstone" is not a palindrome.

stdin: url(some/local/input.txt)
stdout: url(some/local/output.txt)

stdin: url(http://example.com/some/local/generator/input)
stdout: url(http://example.com/some/local/generator/output)
```

Codefragment 5.2: Beschrijving van de oefening „palindroom” in PEML. Overgenomen uit [9].

Dit systeem kan verder uitgebreid worden door aan elke context een unieke identificatiecode toe te kennen. Daarna kan voor elke context een preconditie gegeven worden: welke contexten moet geslaagd zijn om de huidige context uit te voeren. Zijn niet alle contexten uit de preconditie geslaagd, dan wordt de context niet uitgevoerd. Men kan bijvoorbeeld aangeven dat context 15 enkel uitgevoerd kan worden indien contexten 1, 7 en 14 ook succesvol waren.

5.2.3 Herhaalde uitvoeringen van testgevallen

Een andere, maar aan het dynamisch uitvoeren verwante, beperking is het ontbreken van de mogelijkheid om contexten of testgevallen meerdere keren uit te voeren. Dit herhaald uitvoeren is nuttig bij niet-deterministische oefeningen. Een oefening waarbij waarden bijvoorbeeld uit een normale verdeling getrokken moeten worden, zal een duizendtal keren uitgevoerd moeten worden vooraleer met voldoende zekerheid kan geconstateerd worden of er effectief een normale verdeling gebruikt wordt of niet.

Dit is ook op te lossen met een uitbreiding van het testplan: per testgeval of per context zou kunnen aangeduid worden hoeveel keer deze moet uitgevoerd worden. Om de performantie te verbeteren, zouden ook stopvoorwaarden kunnen meegegeven worden. Een aspect dat dit complexer maakt is dat de evaluatoren (zowel de ingebouwde evaluatie binnen TESTed als de geprogrammeerde en programmeertaalspecifieke evaluatie) aangepast zullen moeten worden om niet enkel met één resultaat, maar met een reeks resultaten te werken.

5.3 TESTed

In deze paragraaf komen de beperkingen aan bod die ofwel enkel betrekking hebben op TESTed zelf, ofwel betrekking hebben op meerdere onderdelen, bijvoorbeeld zowel TESTed als het testplan.

5.3.1 Geprogrammeerde evaluatie is traag

Het uitvoeren van een geprogrammeerde evaluatie (zie paragraaf 2.4.2) zorgt voor een niet te verwaarlozen kost op het vlak van performantie (zie tabel 2.3 voor enkele tijdsmetingen). De reden hiervoor is eenvoudig te verklaren: zoals de contexten wordt elke geprogrammeerde evaluatie in een afzonderlijk subproces uitgevoerd (afhankelijk van de programmeertaal ook met eigen compilatiestap). Zoals vermeld in paragraaf 2.5 is TESTed oorspronkelijk gestart met het uitvoeren van code in Jupyter-kernels. De grootste reden dat we daar vanaf gestapt zijn, is dat de kost voor het opnieuw opstarten van een kernel te groot is, en het heropstarten noodzakelijk is om de onafhankelijkheid van de contexten te garanderen. Bij n contexten moet de kernel dus n keer opnieuw gestart worden.

Bij een geprogrammeerde evaluatie wordt geen code van de student uitgevoerd: het opnieuw starten van de kernel is dus niet nodig. Hierdoor wordt de opstartkost van de kernel verspreid over alle contexten die van die programmeerde evaluatie gebruikmaken. Wordt er n keer andere evaluatiecode in een andere programmeertaal gebruikt, zal dit uiteraard geen voordeel opleveren, omdat er dan nog steeds n keer een kernel gestart wordt. In de meeste gevallen wordt echter dezelfde evaluatiecode gebruikt voor alle testgevallen. Het *worst case scenario* is dan weliswaar trager, maar in de meeste gevallen zal de geprogrammeerde evaluatie sneller zijn.

Hierop is één uitzondering: een geprogrammeerde evaluatie waarbij de programmeertaal van de evaluatiecode in Python geschreven is. Voor Python is er speciale ondersteuning (vermits TESTed zelf ook in Python geschreven is, wordt deze evaluatie ook rechtsreeks in TESTed gedaan zonder subproces): hiervoor is het gebruik van Jupyter-kernels niet nuttig.

5.3.2 Ondersteuning voor natuurlijke talen

Sommige judges voor specifieke programmeertalen in Dodona hebben ondersteuning voor het vertalen op het gebied van natuurlijke talen, zoals de vertaling van bijvoorbeeld namen van functies of variabelen. De vertaling van de beschrijving van de opgave is dan weer geïmplementeerd in Dodona zelf door een tweede bestandsextensie. Zo zullen `description.nl.md` en `description.en.md` gebruikt worden voor respectievelijk Nederlands en Engels.

Een eenvoudige oplossing voor het vertalen in de code is iets gelijkaardigs doen met het testplan: `testplan.nl.json` en `testplan.en.json` voor respectievelijk Nederlands en Engels. Deze aanpak heeft wel als nadeel dat veel dingen in het testplan onnodig zullen gedupliceerd moeten worden; een waarde `5` is niet taalafhankelijk.

Een idee dat dit zou voorkomen is om binnen eenzelfde testplan ondersteuning te bieden voor vertalingen, bijvoorbeeld door telkens meerdere talen te aanvaarden. Momenteel ziet een functienaam er bijvoorbeeld als volgt uit:

```
{
  "name": "is_isbn"
}
```

Ondersteuning voor vertaling kan dan deze vorm aannemen:

```
{
  "name": {
    "en": "are_isbn",
    "nl": "zijn_isbn"
  }
}
```

Als een functie binnen een testplan veel wordt opgeroepen, zorgt dit nog steeds voor veel herhaling. Dit kan opgelost worden door een DSL, maar dat neemt niet weg dat het testplan onnodig lang zal worden. Een mogelijke oplossing hiervoor is om een soort vertaalwoordenboek mee te geven in het testplan, waar met sleutelwoorden gewerkt wordt. Een voorbeeld illustreert dit:

```
{
  "translations": {
    "are_isbn": {
      "en": "are_isbn",
      "nl": "zijn_isbn"
    }
```

```
  }
}
```

De functieoproep gebruikt dan een sleutelwoord uit het vertaalwoordenboek, waarna TESTed bij het uitvoeren de juiste naam gebruikt:

```
{
  "name": "$are_isbn"
}
```

Ook de foutboodschappen die TESTed zelf genereert zijn bijvoorbeeld nog niet vertaald: de boodschappen met studenten als doelpubliek zijn in het Nederlands, terwijl de boodschappen voor cursusbeheerders in het Engels zijn. Dit is echter eenvoudig op te lossen: TESTed krijgt de natuurlijke taal mee van Dodona in de configuratie, en Python heeft meerdere internationalisatie-API's[3].

5.3.3 Programmeertaalonafhankelijke opgaven

In veel oefeningen bevat de opgave een stuk code ter illustratie van de opgave. Dit is expliciet buiten het bestek van deze thesis gehouden, maar deze voorbeelden zijn idealiter in de programmeertaal waarin de student wenst in te dienen.

Een idee is hier de codevoorbeelden uit de opgave ook in het formaat van het testplan op te stellen. TESTed bevat namelijk alles om het testplan om te zetten naar concrete code.

Concreet gaat het om volgende zaken:

1. De programmeertaalkeuze in het Dodona-platform moet uitgebreid worden zodat de opgave ook programmeertaalafhankelijk is en dus kan gewijzigd worden als de student een andere programmeertaal kiest.
2. De uitleg bij een opgave (bijvoorbeeld omkadering of bijkomende informatie) is vaak programmeertaalonafhankelijk, terwijl de meer technische onderdelen programmeertaalafhankelijk zijn. Voorbeelden van dat laatste zijn datastructuren (`list`/`array`) of codevoorbeelden. Hiervoor zijn ook meerdere mogelijkheden: zo zou alles binnen eenzelfde bestand kunnen, of kan er één bestand per programmeertaal zijn, waarbij de gemeenschappelijke delen in een apart bestand komen.

5.3.4 Beperkte expressies in het testplan

De expressies in het testplan zijn opzettelijk eenvoudig gehouden (zie paragraaf 2.2.3). Dit levert wel beperkingen op: zo is het niet mogelijk om bijvoorbeeld een functieoproep als `test(5 + 5)` in het testplan te schrijven. Een idee zou dus kunnen zijn om het testplan uit te breiden met meer taalconstructies, waarbij het testplan dan dienst doet als een soort AST (abstract syntax tree). Afhankelijk van hoe ver men hierin gaat, begint dit wel een universele programmeertaal te worden (wat ook expliciet buiten het bestek van deze thesis valt). In dat geval loont het de moeite om

[3] Zie https://docs.python.org/3/library/gettext.html

te onderzoeken of geen eenvoudige, bestaande programmeertaal *transpiled* zou kunnen worden naar het testplan, in plaats van zelf een nieuwe taal op te stellen. Hiervoor moet het testplan wel uitgebreid worden met functies van een AST of IR (intermediary language). Ook TESTed zelf zal uitgebreid moeten worden, aangezien er meer zal moeten omgezet kunnen worden naar de programmeertalen (de sjablonen zullen dus uitgebreid moeten worden).

5.3.5 Programmeertaalonafhankelijke programmeerparadigma

TESTed vertaalt geen programmeerparadigma tussen verschillende programmeertalen. Dit kan ervoor zorgen dat bepaalde oefeningen in sommige programmeertalen op onnatuurlijke wijze opgelost moeten worden. Stel als voorbeeld de ISBN-oefeningen. In Python is het *pythonic* om hiervoor twee top-level functies te schrijven (`is_isbn` en `are_isbn`) In Java zal TESTed dan twee statische functies verwachten, terwijl deze opgave in de Java-wereld ook vaak opgelost zal worden met bijvoorbeeld een klasse `IsbnValidator`, met twee methoden `check` en `checkAll`.

Er zijn meerdere denkpistes om hier een oplossing voor te bieden:

- Voorzie binnen TESTed ook vertalingen van programmeerparadigma. Zo zou voor een oefening opgegeven kunnen worden of de Java-oplossing een klasse of statische methodes moet gebruiken.
- Maak een systeem met hybride oefeningen, waarbij de invoer programmeertaalafhankelijk is, terwijl de evaluatie van resultaten programmeertaalonafhankelijk blijft. In het voorbeeld hierboven zou een lesgever dan voor elke programmeertaal opgeven hoe een resultaat bekomen moet worden (in Python twee functieoproepen, in Java een instantie van een klasse maken en twee methoden oproepen), waarna TESTed overneemt om een programmeertaalonafhankelijke evaluatie te doen van de resultaten.

Het omgekeerde, programmeertaalonafhankelijk invoer en programmeertaalspecifieke beoordeling, bestaat al binnen TESTed (zie paragraaf 2.4.3). Als tot slot zowel de invoer als de uitvoer verschilt van programmeertaal tot programmeertaal, kan er niet meer gesproken worden van een programmeertaalonafhankelijke oefeningen. In dat geval is het beter een bestaande programmeertaalspecifieke judge van Dodona te gebruiken.

5.3.6 Kleinere functionaliteit

Er zijn in de loop van de thesis verschillende kleinere features naar boven gekomen, waarvoor ook ondersteuning zou toegevoegd kunnen worden:

- Ondersteuning voor Python Tutor. Dit is een visuele debugger, origineel geschreven voor Python [14]. Ondertussen worden ook andere talen ondersteund, zoals Java, C++, JavaScript en Ruby. In Dodona ondersteunt momenteel enkel de Python-judge de Python Tutor. Het zou een mooie toevoeging zijn indien dit bij TESTed voor meerdere programmeertalen kan toegevoegd worden.
- Interpreteren van output van de compiler. Momenteel ondersteunt TESTed linting (zie paragraaf 2.6), maar bij sommige talen worden ook veel nuttige waarschuwingen getoond bij het compileren van de code. Deze uitvoeren wordt momenteel getoond in TESTed, en er is ondersteuning om deze uitvoer om te zetten in annotaties op de code van de oplossing. Dit laatste is momenteel geïmplementeerd voor de compilatiestap in Python. De ondersteuning

hiervoor zou uitgebreid kunnen worden naar andere talen en flexibeler gemaakt worden. Zo wordt de uitvoer van de compiler altijd getoond, ook al is die uitvoer omgezet naar annotaties (dezelfde informatie wordt dus dubbel getoond).

- Foutboodschappen beter afhandelen. Als momenteel een fout optreed wordt de stacktrace zonder verdere verwerking getoond aan de gebruiker. Het is echter nuttig hier een verwerkingsstap toe te voegen, zodat:
  - De code gegenereerd door TESTed weggefilterd kan worden. Het kan verwarrend zijn voor studenten als er in de stacktrace code staat zij niet ingediend hebben.
  - Het aanbrengen van links naar specifieke plaatsen in de ingediende code. Als er bijvoorbeeld in de foutboodschap staat „fout op regel 5, kolom 10", dan zou een snelkoppeling naar die plaats in de code nuttig zijn. Dit is ook iets onder andere door de Python-judge geïmplementeerd is.
- Lange uitvoer beperkt tonen. TESTed bevat momenteel een zeer eenvoudige limiet: de uitvoer wordt beperkt tot 20 regels en de regellengte is beperkt tot 150 tekens. Deze limiet houdt echter geen rekening met de aard van de uitvoer (bijvoorbeeld een boodschap van TESTed of de uitvoer van de compiler). Zo zou een stacktrace anders ingekort kunnen worden.
- Waarden mooier weergeven (*pretty printing*). Enerzijds gaat het om grote verzamelingen verkort weer te geven. Is de uitvoer bijvoorbeeld een lijst van duizend elementen, zouden enkel de eerste tien getoond kunnen worden. Momenteel worden alle waarden verbatim aan de studenten getoond. Anderzijds gaat het ook om het beter weergeven van waarden om het vergelijken met de verwachte uitvoer eenvoudiger te maken. Een voorbeeld hier zijn verzamelingen: door de verzameling gesorteerd te tonen wordt het zoeken naar bijvoorbeeld ontbrekende waarden een stuk eenvoudiger.
- Tijdslimieten op het niveau van de contexten. Momenteel is het enkel mogelijk om op het niveau van een oefening een tijdslimiet in te stellen. De implementatie hiervan zou eenvoudig moeten zijn: intern werkt TESTed al met een tijdslimiet per context, namelijk de totale tijdslimiet van de oefening minus de verstreken tijd.
- Gerelateerd aan de tijdslimieten is binnen Dodona een status voor een test die niet uitgevoerd is[4]. Momenteel toont TESTed de niet-uitgevoerde testen met de status „fout". Het is echter wenselijk deze oefening als „niet-uitgevoerd" aan te duiden. Ook visueel zou er een verschil kunnen zijn, door bijvoorbeeld een grijze achtergrond te gebruiken in plaats van een rode achtergrond.

[4]Zie de issue: https://github.com/dodona-edu/dodona/issues/1785

Bibliografie

[1] *Amazon Ion*. Amazon. 15 januari 2020. URL: https://amzn.github.io/ion-docs/ (bezocht op 27-01-2020).

[2] *Apache Avro™ 1.9.1 Documentation*. The Apache Foundation. 9 februari 2019. URL: http://avro.apache.org/docs/1.9.1/ (bezocht op 27-01-2020).

[3] *Apache Parquet*. The Apache Foundation. 13 januari 2020. URL: https://parquet.apache.org/documentation/latest/ (bezocht op 27-01-2020).

[4] Ben Bastiaensen en Jan De Craemer. *Zo denkt een computer*. Vlaamse Overheid, 2017. Hfdstk. Computationeel denken, p. 8–9.

[5] Michael Bayer e.a. *Mako Templates for Python*. 20 januari 2020. URL: https://www.makotemplates.org/.

[6] Carstenn Bormann en Paul Hoffman. *Concise Binary Object Representation (CBOR)*. RFC 7049. Internet Engineering Task Force, oktober 2013. URL: https://tools.ietf.org/html/rfc7049.

[7] *BSON 1.1*. MongoDB. 19 juli 2019. URL: http://bsonspec.org/ (bezocht op 17-01-2020).

[8] Bram Cohen. *The BitTorrent Protocol Specification*. 4 februari 2017. URL: http://bittorrent.org/beps/bep_0003.html.

[9] Phill Conrad, Cory Bart en Stephen Edwards. *Programming Exercise Markup Language*. Standards, Protocols, en Learning Infrastructure for Computing Education (SPLICE). URL: https://cssplice.github.io/peml/ (bezocht op 31-03-2020).

[10] Wikipedia Contributors. *Comparison of data-serialization formats. Wikipedia, The Free Encyclopedia*. 25 januari 2020. URL: https://en.wikipedia.org/w/index.php?title=Comparison_of_data-serialization_formats&oldid=937433197.

[11] „Decreet van 14 december 2018 betreffende de onderwijsdoelen voor de eerste graad van het secundair onderwijs". In: *Belgisch Staatsblad* 26/04/2019 (14 december 2018), p. 40558. URL: http://www.ejustice.just.fgov.be/mopdf/2019/04/26_1.pdf#page=44.

[12] Marc Duranton e.a. *HiPEAC Vision 2019*. Januari 2019. Hfdstk. Impact of computing technology on society, p. 110–111. ISBN: 9789090313641.

[13] Sadayuki Furuhashi. *MessagePack*. 17 september 2018. URL: https://msgpack.org/.

[14] Philip J. Guo. „Online Python Tutor: Embeddable Web-Based Program Visualization for Cs Education". In: *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*. SIGCSE '13. Denver, Colorado, USA: Association for Computing Machinery, 2013, p. 579–584. ISBN: 9781450318686. DOI: 10.1145/2445196.2445368.

[15] *Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation*. Recommendation X.608. International Telecommunications Union, augustus 2015. URL: https://www.itu.int/rec/T-REC-X.680-201508-I/en.

[16] *Information technology – Generic applications of ASN.1: Fast infoset*. Recommendation X.881. International Telecommunications Union, 14 mei 2005. URL: https://www.itu.int/rec/T-REC-X.891-200505-I/en.

[17] Jackson JSON team. *Smile Data Format*. 2010. URL: https://github.com/FasterXML/smile-format-specification.

[18] Hieke Keuning, Johan Jeuring en Bastiaan Heeren. „Towards a Systematic Review of Automated Feedback Generation for Programming Exercises". In: *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*. ITiCSE '16. Arequipa, Peru: Association for Computing Machinery, 2016, p. 41–46. ISBN: 9781450342315. DOI: 10.1145/2899415.2899422.

[19] Thomas Kluyver e.a. „Jupyter Notebooks – a publishing format for reproducible computational workflows". In: *Positioning and Power in Academic Publishing: Players, Agents and Agendas*. Red. door F. Loizides en B. Schmidt. IOS Press. 2016, p. 87–90. DOI: 10.3233/978-1-61499-649-1-87.

[20] Adrian Kosowski, Michał Małafiejski en Tomasz Noiński. „Application of an Online Judge & Contester System in Academic Tuition". In: *Advances in Web Based Learning – ICWL 2007*. Red. door Howard Leung e.a. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, p. 343–354. ISBN: 978-3-540-78139-4. DOI: 10.1007/978-3-540-78139-4_31.

[21] Eric Lengyel. *Open Data Description Language (OpenDDL)*. 17 januari 2017. URL: http://openddl.org/.

[22] Andrew Luxton-Reilly e.a. „Introductory Programming: A Systematic Literature Review". In: *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*. ITiCSE 2018 Companion. Larnaca, Cyprus: Association for Computing Machinery, 2018, p. 55–106. ISBN: 9781450362238. DOI: 10.1145/3293881.3295779.

[23] Nilo Mitra en Yves Lafon. *SOAP Version 1.2 Part 0: Primer (Second Edition)*. TR. W3C, april 2007. URL: http://www.w3.org/TR/2007/REC-soap12-part0-20070427/.

[24] *OGDL 2018.2*. 25 februari 2018. URL: https://msgpack.org/ (bezocht op 28-01-2020).

[25] Wouter van Oortmerssen. *FlatBuffers*. Google. 24 april 2019. URL: https://google.github.io/flatbuffers/.

[26] *OPC unified architecture - Part 1: Overview and concepts*. IEC TR 62541-1:2016. International Electrotechnical Commission, 10 mei 2016. URL: https://webstore.iec.ch/publication/25997.

[27] Charlotte Van Petegem. „Computationele benaderingen voor deductie van de computationele complexiteit van computerprogramma's". Universiteit Gent, 2018. URL: https://lib.ugent.be/catalog/rug01:002479652.

[28] *ProtocolBuffers*. Google. 13 december 2019. URL: https://developers.google.com/protocol-buffers/.

[29] Ricardo Queirós en José Paulo Leal. „Pexil: Programming exercises interoperability language". In: *Conferência Nacional XATA: XML, aplicações e tecnologias associadas*. ESEIG. 2011, p. 37–48. ISBN: 9789899686311. DOI: 10400.22/4748.

[30] Bernardo Ramos. *Binn*. 25 september 2019. URL: https://github.com/liteserver/binn.

[31] Mark Slee, Aditya Agarwal en Marc Kwiatkowski. „Thrift: Scalable cross-language services implementation". In: (2007). URL: http://thrift.apache.org/static/files/thrift-20070401.pdf.

[32] *Universal Binary JSON*. 25 februari 2018. URL: http://ubjson.org/.

[33] Dienst Curriculum & vorming. *Computationeel denken in Zin in leren! Zin in leven!* Katholiek Onderwijs Vlaanderen, 25 september 2017.

[34] Max Wildgrube. *Structured Data Exchange Format (SDXF)*. RFC 3072. Internet Engineering Task Force, maart 2001. URL: https://tools.ietf.org/html/rfc3072.