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

1.1. Inleiding

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

1.2. Wat is Dodona?

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

Dodona:

- Opgaves (of oefeningen), opgesteld door lesgevers - Oplossingen, opgesteld door studenten - Judge beoordeelt oplossing van een opgave, door Dodona-team

1.3. Beoordelen van oplossingen

1.3.1. De judge

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 de invoer, bestaande uit onder andere de programmeertaal van de oplossing, het pad naar het oplossingsbestand en 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 een 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 student 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] 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 vindt u op https://docs.docker.com/storage/bind-mounts/

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 standaardinvoer).
3. De judge moet het resultaat van zijn beoordeling uitschrijven naar standaarduitvoer, 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. Ook heeft elke judge een eigen manier waarop de testen voor een oplossing opgesteld moeten worden. Zo worden in de Java-judge jUnit-testen gebruikt, terwijl de Python-judge doctests en een eigen formaat ondersteunt.

1.3.2. De beoordeling zelf

De beoordeling van een oplossing van een student laat zich beschreven als het volgende stappenplan:

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 invoer.
5. De judge beoordeelt de oplossing aan de beoordelingsmethodes opgesteld door de lesgever (d.w.z. de jUnit-test, de doctests, …). Sommige judges voeren ook bijkomende taken, 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 het standaarduitvoerkanaal.
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.

1.4. 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], met volgende opgave:

> 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'
> ```

Oplossingen voor deze oefening staan in codefragmenten 1.1 en 1.2, voor respectievelijk Python en Java.

Implementatie 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

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

```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 1.1.: Voorbeeldoplossing in Java.

```python
from random import randint


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

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

Codefragment 1.2.: Voorbeeldoplossing in Python.

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.

Kijken we naar onze Lotto-oefening, merken we 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.

We kunnen tot eenzelfde constatatie komen bij meer complexe oefeningen die zich concentreren op algoritmen: ook daar zijn de concepten belangrijker dan in welke programmeertaal een algoritme 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 vandaag al beschikbaar in Java en Python op Dodona, maar dan als afzonderlijke 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 gegenereerd door de oplossing te vergelijken met een op voorhand vastgelegde verwachte uitvoer. De geproduceerde uitvoer zal moeten gecontroleerd worden met code, specifiek gericht op deze oefening, die de verwachte vereisten van de oplossing controleert. Deze evaluatiecode moet momenteel voor elke programmeertaal en dus elke judge opnieuw geschreven worden. In de context van de Lotto-oefening controleert deze code bijvoorbeeld of de gegeven getallen binnen het bereik liggen en of ze gesorteerd zijn.

Implementatie van judges

Een tweede nadeel aan de 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

We beschouwen het eerste nadeel als het belangrijkste nadeel, en vatten het samen als de onderzoeksvraag waarop deze thesis een antwoord wil bieden:

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

> 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 willen we dat eens een oefening opgesteld is, deze niet meer gewijzigd moet worden wanneer talen toegevoegd worden aan de judge.

Als bijzaak zijn we ook geïnteresseerd of de judge uit de onderzoeksvraag een voordeel kan bieden voor het implementeren van judges zelf (het tweede nadeel).

De aandachtige lezer zal opmerken dat de opgave voor de Lotto-oefening programmeertaalspecifieke en taalspecifieke elementen bevat. Zo zijn de voorbeelden in Python en zijn de namen van functies en argumenten in het Nederlands. Beide zaken worden voor deze thesis expliciet als *out-of-scope* gezien en zullen niet behandeld worden.

1.5. Opbouw van de thesis

Hoofdstuk 2 handelt over het antwoord op bovenstaande vraag, waar een prototype van een dergelijke judge wordt voorgesteld. Daarna volgt ter illustratie een gedetailleerde beschrijving van hoe een oefening opgesteld moet worden voor deze judge. Nadien volgt een beschrijving van hoe een nieuwe programmeertaal moet toegevoegd worden. Daar deze twee hoofdstukken voornamelijk ten doel hebben zij die met de judge moeten werken te informeren, nemen deze hoofdstukken de vorm aan van meer traditionele softwarehandleidingen. Tot slot volgt met een hoofdstuk over beperkingen van de huidige implementaties, 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 van de judge toe te lichten, waarna elk onderdeel in meer detail besproken wordt. Tot slot wordt afgesloten met een samenvattend overzicht.

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 beoordelingscode wordt uitgevoerd. Deze moet niet in dezelfde programmeertaal als de oplossing of TESTed geschreven zijn.

2.1.2. Stappenplan bij 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. De testcode wordt gegenereerd voor elke context uit het testplan, wat resulteert in de testcode voor elke context. Deze stap is de overgang naar de *uitvoeringsomgeving*.

4. De code wordt optioneel gecompileerd. Dit kan op twee manieren gebeuren (meer 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 goed is voor de performantie.

   b) Contextcompilatie: hierbij wordt de testcode voor elke context afzonderlijk gecompileerd tot een uitvoerbaar bestand.

   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 he testplan wordt in een afzonderlijk subproces uitgevoerd, teneinde het delen van informatie tegen te gaan. De onafhankelijkheid van de contexten laat ons ook toe om contexten in parallel uit te voeren. Toch moeten we opmerken dat het parallel uitvoeren vooral een theoretische mogelijkheid is: in werkelijkheid gebruiken er meerdere gebruikers tegelijk Dodona, waardoor er ook meerdere oefeningen tegelijk beoordeeld worden, wat op zijn beurt ervoor zorgt dat er geen grote ruimte voor snelheidswinst door parallellisatie binnen een judge is.

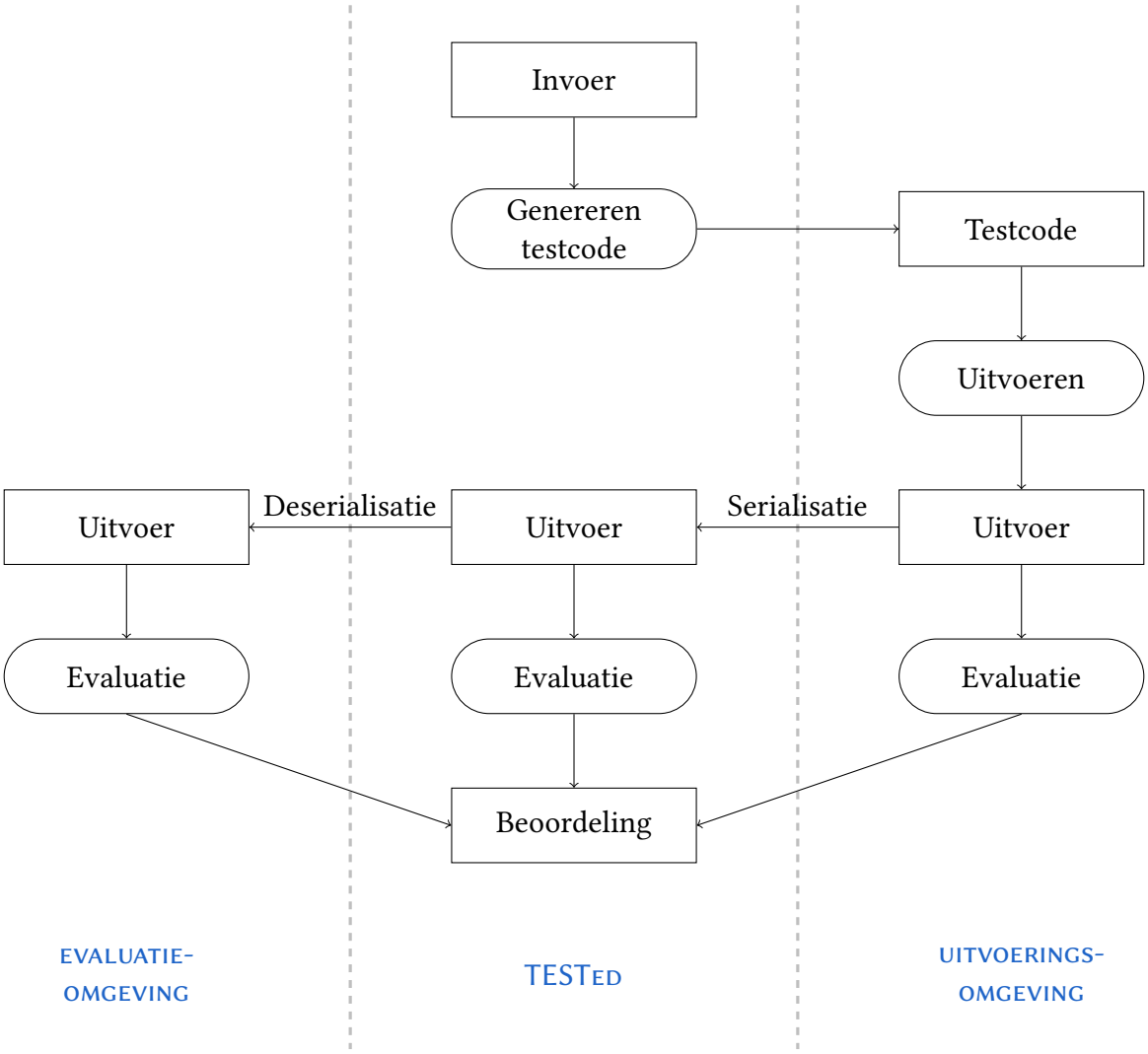
6. De uitvoering van de executable in de vorige stap resulteert in resultaten (voor elke context), zoals de standaarduitvoerstroom, de standaardfoutstroom, returnwaardes, exceptions of exitcodes. Deze bundel resultaten wordt nu geëvalueerd op juistheid. Hiervoor zijn drie mogelijke manieren:

   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.

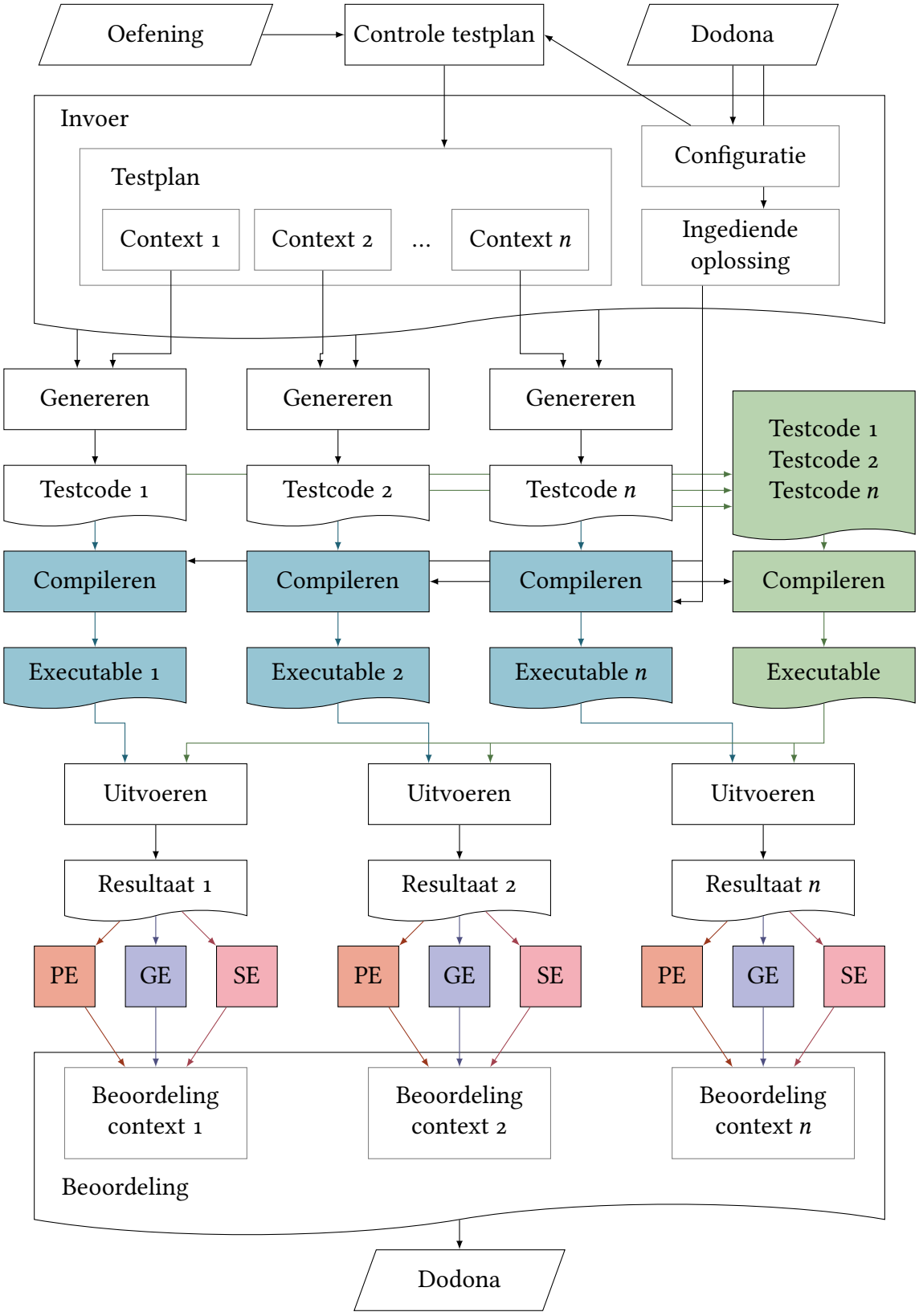
   b) Geprogrammeerde evaluatie (afgekort tot PE in de flowchart). Hierbij is er evaluatiecode geschreven die los staat van de oplossing, waardoor deze evaluatiecode ook in een andere programmeertaal geschreven kan zijn. De code ter uitvoering van de geprogrammeerde evaluatiecode wordt gegenereerd en dan 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, en slechts een van de mogelijkheden gebruikt wordt. 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, zowel voor de TESTed zelf als voor programmeertaalspecifieke dingen.

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 (dus 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 het standaardinvoerkanaal 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 het standaarduitvoerkanaal, het standaardfoutkanaal, opgevangen uitzonderingen (*exceptions*), de teruggegeven waarden van een functieoproep (returnwaarde) of de inhoud van een bestand. De exitcode is ook mogelijk, maar enkel in het hoofdtestgeval.

Bij de keuze voor een formaat voor het testplan (json, xml, ...), hebben we 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 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*.

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*),

```json
{
  "tabs": [
    {
      "name": "Feedback",
      "contexts": [
        {
          "input": {
            "function": {
              "type": "top",
              "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.

- 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 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, (Wikipedia-bijdragers 2020). 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 (*Apache Avro™ 1.9.1 Documentation* 2019).

Apache Parquet Minder relevant, dit is een bestandsformaat voor Hadoop (*Apache Parquet* 2020).

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, bv. 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 (*Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation* 2015).

Bencode Schema gebruikt in BitTorrent. Het is gedeeltelijk binair, gedeeltelijk in text (Cohen 2017).

Binn Binair dataformaat (Ramos 2019).

BSON Een binaire variant op json, geschreven voor en door MongoDB (*BSON 1.1* 2019).

CBOR Een lichtjes op json gebaseerd formaat, ook binair. Heeft een goede standaard, ondersteunt redelijk wat talen (Bormann en Hoffman 2013).

FlatBuffers Lijkt op ProtocolBuffers, allebei geschreven door Google, maar verschilt wat in implementatie van ProtocolBuffers. De encodering is binair (Oortmerssen 2019).

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 (*Information technology – Generic applications of ASN.1: Fast infoset* 2005).

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. (*Amazon Ion* 2020).

MessagePack Nog een binair formaat dat lichtjes op json gebaseerd is. Lijkt qua types sterk op json. Heeft implementaties in veel talen (Furuhashi 2018).

OGDL Afkorting voor *Ordered Graph Data Language*. Daar het om een serialisatieformaat voor grafen gaat, is het niet nuttig voor ons doel (*OGDL 2018.2* 2018).

OPC Unified Architecture Een protocol voor intermachinecommunicatie. Complex: de specificatie bevat 14 documenten, met ongeveer 1250 pagina's (*OPC unified architecture - Part 1: Overview and concepts* 2016).

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 bv. json (Lengyel 2017).

ProtocolBuffers Lijkt zoals vermeld sterk op FlatBuffers, maar heeft nog extra stappen nodig bij het encoderen en decoderen, wat het minder geschikt maakt (*ProtocolBuffers* 2019).

Smile Nog een binaire variant van json (Jackson JSON team 2010).

SOAP Afkorting voor *Simple Object Access Protocol*. Niet bedoeld als formaat voor dataserialisatie, maar voor communicatie tussen systemen over een netwerk (Mitra en Lafon 2007).

SDXF Binair formaat voor data-uitwisseling. Weinig talen ondersteunen dit formaat (Wildgrube 2001).

Thrift Lijkt sterk op ProtocolBuffers, maar geschreven door Facebook (Slee, Agarwal en Kwiatkowski 2007).

UBJSON Nog een binaire variant van json (*Universal Binary JSON* 2018).

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 echter 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.

Enerzijds vervult dit dataschema een gelijkaardige functie als JSON Schema (of XML Schema voor xml-bestanden): het dataschema legt de structuur van het serialisatieformaat vast. Anderzijds is er een belangrijk verschil: het dataschema schrijft geen verwachte gegevenstypes voor, maar beschrijft bestaande data. Het dataschema is dus altijd gekoppeld aan concrete data. Dit laatste zorgt ervoor dat het dataschema eenvoudig kan blijven, wat als voordeel heeft dat implementaties eenvoudiger zijn. Complexe types als `union[string, int]` zijn bijvoorbeeld niet mogelijk, maar ook niet nodig: concrete data kan niet zowel een `string` als een `int` zijn.

Tot slot bevat codefragment 2.3 het schema van een waarde in het serialisatieformaat, in een vereenvoudigde versie van JSON Schema. Hierbij staat `<type>` voor een van de gegevenstypes die hierna besproken worden.

```
{
  "type": "object",
  "properties": {
    "data": {
      "type": ["number", "string", "boolean", "object", "array", "null"]
    },
    "type": {
      "type": "string",
      "enum": ["<types>"]
    }
  }
}
```

Codefragment 2.3.: Het schema voor een waarde in het serialisatieformaat, in een vereenvoudigde versie van JSON Schema. Hierbij is `<type>` een van de gegevenstypes die besproken worden in subparagraaf *Gegevenstypes* van paragraaf 2.2.2.

Gegevenstypes

Het systeem om de gegevenstypes aan te duiden vervult een dubbele functie:

- Het wordt gebruikt om het gegevenstype van concrete data aan te duiden (beschrijvende modus).
- Het wordt gebruikt om te beschrijven welke gegevenstype verwacht wordt (voorschrijvende modus).

Bij het ontwerp van het systeem voor de gegevenstypes zorgen deze twee 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 beschrijven 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 taal een eigen niveau van details bij gegevenstypes. Python heeft bijvoorbeeld enkel `integer` voor gehele getallen, terwijl C beschikt over `int`, `unsigned`, `long`, enz.

Om deze redenen zijn de gegevenstypes opgedeeld in vier categorieën:

1. De basistypes. Deze gegevenstypes zijn bruikbaar in zowel beschrijvende als voorschrijvende modus. De lijst van basistypes omvat:

   `integer` Gehele getallen, zowel positief als negatief.

   `rational` Rationale getallen.

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

   `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.

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. Naast de basistypes van hierboven, bevat TESTed een hele reeks bijkomende types. Deze gegevenstypes staan toe om meer details te gebruiken. Een voorbeeld is de lijst van types in tabel 2.1, dat 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 vanzelf omgezet worden naar een `list`. Er is ook de mogelijk dat implementaties voor programmeertalen expliciet een bepaald type niet ondersteunen. Zo zal de Java-implementatie geen `uint64` (een unsigned 64-bit integer) ondersteunen, omdat er geen equivalent bestaat in de taal.

3. De gegevenstypes worden ook gebruikt om de types van functie-argumenten aan te duiden. Hier is er nood aan een bijkomend type:

   `identifier` Een verwijzing naar een eerder benoemd element (zoals variabelen of functies). Bij het gebruik van dit gegevenstype stel de data de naam van de identifier voor.

   Het is logisch dat dit type niet gebruikt kan worden voor het encoderen van data, daar dit bijvoorbeeld onmogelijk als returnwaarde kan voorkomen.

4. 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 echt expliciet gekozen om dit soort types niet te ondersteunen, door de moeilijkheid om dit te implementeren in statisch getypeerde talen, zoals Java of C. Een type dat wel ondersteund wordt is:

   `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).

2.2.3. Functieoproepen en assignments

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

Tabel 2.1.: 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	`Integer`	`Integer`	`Integer`	`Integer`	`Integer`	`Integer`	`Integer`	`Integer`

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

```
{
  "type": "object",
  "properties": {
    "data": {
      "type": ["number", "string", "boolean", "object", "array", "null"]
    },
    "type": {
      "type": "string",
      "enum": ["<types>"]
    }
  }
}
```

Codefragment 2.4.: Het schema voor een functieoproep in het testplan. Hierbij is `<waarde>` een waarde, zoals besproken in subparagraaf *Dataschema* van paragraaf 2.2.2.

een opgave kunnen bestaan uit het implementeren van een klasse. Bij de evaluatie dient dan een 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";
```

Concreet is ervoor gekozen om het testplan niet uit te breiden met generieke statements of expressies, maar de ondersteuning te beperken tot assignments en functieoproepen. Dit om de implementatie van de vertaling van het testplan naar de ondersteunde programmeertalen niet nodeloos ingewikkeld te maken.

Codefragment 2.4 toont de structuur van een functieoproep in een vereenvoudigde versie van JSON Schema. Het bestaat uit volgende onderdelen:

`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 functie op een object (bv. naar een statische functie in Java).

`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 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.

`identity` Dit is een speciaal geval: er mag geen functienaam gegeven worden en er moet exact één argument gegeven worden. Dat ene argument zal de returnwaarde van de functie zijn. In de implementaties wordt dit vaak ook niet vertaald als een functie, maar wordt de waarde rechtstreeks gebruikt. De bestaansreden van dit soort functies is het toekennen van waarden aan variabelen, om redenen die we later zullen bespreken.

`namespace` De namespace voor functies van het type `namespace`.

`name` De naam van de functie.

`arguments` De argumenten van de functie. Dit is een lijst van waarden, waarbij de waarden zijn zoals beschreven bij het serialisatieformaat in paragraaf 2.2.2.

Een beperking is dat het niet mogelijk is om rechtstreeks een functieoproep te doen als argument voor een andere functie, of toch niet op een programmeertaalonafhankelijke manier. Een oproep als `oproep(hallo(), 5)` is niet mogelijk. Bij dergelijke dingen zal de functieoproep eerst aan variabele moeten toegekend worden, bv. `var param = hallo()`, waarna deze variabele als argument met type `literal` kan gegeven worden aan de oorspronkelijke functie: `oproep(param, 5)`. De aandachtige lezer zal opmerken dat met die functieargumenten van het type `literal` rond deze beperking kan gewerkt worden, aangezien de tekstuele waarde van een dergelijk argument letterlijk in de taal komt. We raden deze omweg echter ten sterkste af: dit maakt het testplan taalafhankelijk, want niet elke programmeertaal implementeert functieoproepen op eenzelfde wijze.

Dit brengt ons bij de variabeletoekenning of *assignment*. In ons testplan beperkt dit zich tot het toekennen van een naam aan het resultaat van een functieoproep. Dit is ook meteen de reden voor het bestaan van de functiesoort `identity`: via deze weg blijft het testplan eenvoudiger (de waarde van een assignment is altijd een functieoproep), maar toch kunnen gewoon waarden toegekend worden aan een variabelen. Concreet ziet een variabeletoekenning er als volgt uit:

```
{
  "name": "Naam van de variabele",
  "expression": "<Object voor functieoproep>",
  "type": "Optioneel type"
}
```

De `name` is de naam die aan de variabele gegeven zal worden. Het veldje `expression` moet een object zijn dat een functieoproep voorstelt, in het formaat zoals hiervoor besproken. In een beperkt aantal gevallen kan de judge het type van de variabele afleiden uit de functieoproep, maar in veel gevallen is het nodig om zelf het type mee te geven. Dit type moet een van de ondersteunde types zijn uit het serialisatieformaat, zij het dat er ondersteuning is voor eigen types (zoals een klasse die geïmplementeerd moest worden door de student).

Een gecombineerd voorbeeld staat hieronder. Hier wordt de string `'Dodona'` toegekend aan een variabele met naam `name`. De judge kan het type afleiden, dus we moeten niet opgeven dat `name` een `text` is.

```
{
  "name": "name",
  "expression": {
    "type": "identity",
    "arguments": [
      {
        "type": "text",
        "value": "Dodona"
      }
    ]
  }
}
```

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. Concreet gebeurt dit door voor elk item in het testplan af te leiden wat de programmeertaal dient te ondersteunen om met dat item uit het testplan te kunnen werken. 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. Het afleiden van wat de programmeertaal moet ondersteunen gebeurt volledig automatisch aan de hand van het testplan.

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 (Bayer e.a. 2020). 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.

`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 2.5 en 2.6 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.

Indien de configuratie het toelaat, worden de contexten parallel uitgevoerd. Om te vermijden dat bestanden of uitvoer overschreven worden, wordt alle relevante gecompileerde code voor een context gekopieerd naar een aparte map waar het uitvoeren gebeurt. Codefragment 2.8 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.

```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_main(value):
    send_exception(value)

def v_evaluate_0(value):
    send(value)

def e_evaluate_0(value):
    send_exception(value)

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

# Context 0-0.
try:
    v_evaluate_0(    loterij(6, 15)    )
except Exception as e:
    e_evaluate_0(e)

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

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

Codefragment 2.5.: Gegenereerde testcode in Python voor de eerste context uit het testplan van de voorbeeldoefening Lotto.

```java
public class Context_0_0 {

    private final FileWriter valueWriter;
    private final FileWriter exceptionWriter;

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

    void close() throws Exception {
        this.valueWriter.close();
        this.exceptionWriter.close();
    }

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

Codefragment 2.6.: 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 2.7.: Gegenereerde selector in Java voor twee contexten uit het testplan van de voorbeeldoefening Lotto.

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.

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 2.9.

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 echt 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.

```
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
    ├── buzzchecker_Aao9H18Ve        //Map voor elke context
    │   ├── buzzchecker.py           //Code aangepaste evaluator
    │   ├── evaluation_utils.py
    │   ├── evaluator_executor.py
    │   └── values.py
    ├── buzzchecker_B5WViK0zQ
    │   ├── buzzchecker.py
    │   ├── evaluation_utils.py
    │   ├── evaluator_executor.py
    │   └── values.py
    ...
```

Codefragment 2.8.: 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 2.9.: Voorbeeld van het uitvoerkanaal voor returnwaarden na het uitvoeren van de eerste context uit de voorbeeldoefening Lotto.

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()` [1] 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 2.10 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:

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

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

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

`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.

Een voorbeeld van hoe dit eruitziet is codefragment 2.11. 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 2.8 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:

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

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.

   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[2] gevolgd worden. Eén uitzondering is zwevendekommagetallen, waarvoor opnieuw `math.isclose()` gebruikt wordt in plaats van $==$.

[2] 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 2.12.: Fragment uit een testplan dat de uitvoerspecificatie van de returnwaarde voor een testgeval toont, waarbij de waarde-evaluatie gebruikt wordt.

Bij deze evaluatievorm zijn geen configuratieopties. Een voorbeeld van het gebruik binnen een testplan is codefragment 2.12. Hier wordt als returnwaarde een verzameling met drie elementen (5, 10 en 15) verwacht.

2.4.2. Geprogrammeerde evaluatie

Bij oefeningen, zoals de voorbeeldoefening Lotto, met niet-deterministische resultaten, kunnen de verwachte waarden niet in het testplan komen. Ook andere oefeningen waar geen directe vergelijking 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 2.13. 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 2.14

```python
def evaluate(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 2.13.: 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 expected 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 2.14.: 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 2.15.: De implementatie van de klasse `AbstractCustomEvaluator`.

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 2.15 en 2.16.

Een geprogrammeerde evaluatie wordt gebruikt in de voorbeeldoefening Lotto. Het gebruik in het testplan wordt getoond in codefragment 2.17, 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 2.18.

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 X. 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 2.14 en 2.16). Het gebruik in het testplan is codefragment 2.21.

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

abstract class AbstractEvaluator implements Closeable {

    protected final OutputStreamWriter writer;

    public AbstractEvaluator() {
        this.writer = new OutputStreamWriter(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 2.16.: 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 2.17.: Fragment uit het testplan van de voorbeeldoefening Lotto, waar een geprogrammeerde evaluatie gebruikt wordt.

2.4.4. Performantie

Zoals reeds vermeld in onder andere 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. Deze subparagraaf beschrijft de evaluatie van de implementatie van TESTed vanuit het perspectief van de performantie.

Jupyter-kernels

Het eerste prototype van TESTed gebruikte Jupyter-kernels voor het uitvoeren van de code. Jupyter-kernels zijn de achterliggende technologie bij de Jupyter Notebooks (**jupyter2016**). De werking van een Jupyter-kernel kan als volgt samengevat worden: een Jupyter-kernel is een lokale server, die 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 voor elke taal veel implementatiewerk te verrichten: 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.

```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, f"volgende lottogetallen zijn groter dan de maximale " \
                      f"waarde {maximum}: {listing(wrong)} "

    if wrong := [number for number in nrs if number < 1]:
        return False, f"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, f"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 2.18.: De evaluatiecode voor de geprogrammeerde evaluatie van de voorbeeldoefening Lotto.

```python
def evaluate(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.

```java
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`.

```
{
  "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 2.21.: Fragment uit een testplan, waar een programmeertaalspecifieke evaluatie gebruikt wordt.

- Eerde werk (**petegem2018**), dat gebruik maakt van Jupyter-kernels voor een gelijkaardig doel, rapporteert geen problemen.

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":  {
                "function": {
                    "type":      "top",
                    "name":      "is_isbn",
                    "arguments": function_arguments
                }
            },
            "output": {
                "result": {
                    "value": (values.encode(result))
                }
            }
        }

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

```
            "normal": [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": {
                "assignment": {
                    "name": f"codes{index:02d}",
                    "expression": {
                        "type": "identity",
                        "arguments": [
                            values.encode(codes)
                        ]
                    }
```

```
                }
            }
        }

        # Stel de functieargumenten op. We geven opnieuw sowieso de variabele die
        # we eerst hebben aangemaakt mee als argument.
        function_arguments = [
            {
                "type": "literal",
                "data": 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":  {
                "function": {
                    "type":      "top",
                    "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 = {
            "normal": [assignment_testcase, testcase]
        }

        contexts.append(context)
    return contexts

```

```python

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 = {
        "normal": 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]

# Terugvallen op individuele modus is niet nuttig in Python, dus laten we dat niet
# toe. Indien het terugvallen niet nuttig is, is het sneller om het uit te zetten.
plan["configuration"] = {
    "allow_fallback": False
}

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

```
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.

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/

5. Beperkingen en toekomstig werk

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

Korte samenvatting

5.1. Performance

-> Uitleg over eerste implementatie met jupyter kernels

-> Uitleg over verschillende stadia van codegeneratie (alles apart -> zoveel mogelijk samen)

5.2. Functies

-> Dynamisch testplan –> Dingen meerdere keren uitvoeren –> Dingen wel of niet uitvoeren op basis van vorige uitkomst

-> Functies/assignments –> Functie als functie-argumenten zonder tussenstap met assignments

-> Meertaligheid

-> Beschrijving van de oefening (programmeertalen)

A. Specificatie van het serialisatieformaat

TODO: hier misschien json-schema?

Bibliografie

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

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

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

Bayer, Michael e.a. (20 januari 2020). *Mako Templates for Python*. URL: https://www.makotemplates.org/.

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

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

Cohen, Bram (4 februari 2017). *The BitTorrent Protocol Specification*. URL: http://bittorrent.org/beps/bep_0003.html.

Furuhashi, Sadayuki (17 september 2018). *MessagePack*. URL: https://msgpack.org/.

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

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

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

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

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

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

Oortmerssen, Wouter van (24 april 2019). *FlatBuffers*. Google. URL: https://google.github.io/flatbuffers/.

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

ProtocolBuffers (13 december 2019). Google. URL: https://developers.google.com/protocol-buffers/.

Ramos, Bernardo (25 september 2019). *Binn*. URL: https://github.com/liteserver/binn.

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

Universal Binary JSON (25 februari 2018). URL: http://ubjson.org/.

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

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