

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

Inhoudsopgave

1. Educational software testing	3
1.1. Inleiding	3
1.2. Wat is Dodona?	3
1.3. Beoordelen van oplossingen	3
1.3.1. De judge	3
1.3.2. De beoordeling zelf	4
1.4. Probleemstelling	5
1.5. Opbouw van de thesis	8
2. TESTed	9
2.1. Overzicht	9
2.1.1. Architecturaal ontwerp	9
2.1.2. Stappenplan bij een beoordeling	9
2.2. Beschrijven van een oefening	13
2.2.1. Het testplan	13
2.2.2. Dataserialisatie	14
2.2.3. Functieoproepen en assignments	20
2.2.4. Controle ondersteuning voor programmeertalen	23
2.3. Oplossingen uitvoeren	23
2.3.1. Testcode genereren	23
2.3.2. Testcode uitvoeren	25
2.3.3. Beoordelen van gedrag	25
2.4. Oplossingen beoordelen	28
2.4.1. Generieke evaluatie	30
2.4.2. Aangepaste evaluator	31
3. Case-study: configureren van een programmeertaal in TESTed	35
4. Case-study: nieuwe oefening	36
4.1. ISBN	36
4.1.1. Voorbereiding	36
4.1.2. Opgave	37
4.2. Testplan	37
4.2.1. Afsluiting	44
5. Beperkingen en toekomstig werk	45
5.1. Performance	45
5.2. Functies	45
A. Specificatie van het serialisatieformaat	46

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:

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

²Informatie 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”-oefening³, 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

```
1 > loterij()  
2 '2 - 17 - 22 - 27 - 35 - 40'  
3 > loterij(aantal=8)  
4 '5 - 13 - 15 - 31 - 34 - 36 - 39 - 40'  
5 > loterij(aantal=4, maximum=38)  
6 '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

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

```

1  import java.util.HashSet;
2  import java.util.Set;
3  import java.util.concurrent.ThreadLocalRandom;
4  import java.util.stream.Collectors;
5
6  class Main {
7
8      public static String loterij(int aantal, int maximum) {
9          var r = ThreadLocalRandom.current();
10         var result = new HashSet<Integer>();
11         while (result.size() < aantal) {
12             result.add(r.nextInt(1, maximum + 1));
13         }
14         return result.stream()
15             .sorted()
16             .map(Object::toString)
17             .collect(Collectors.joining(" - "));
18     }
19
20     public static String loterij(int aantal) {
21         return loterij(aantal, 42);
22     }
23
24     public static String loterij() {
25         return loterij(6, 42);
26     }
27 }

```

Codefragment 1.1.: Voorbeeldoplossing in Java.

```

1  from random import randint
2
3
4  def loterij(aantal=6, maximum=42):
5      getallen = set()
6      while len(getallen) < aantal:
7          getallen.add(randint(1, maximum))
8
9      return " - ".join(str(x) for x in sorted(getallen))

```

Codefragment 1.2.: Voorbeeldoplossing in Python.

Java-judge junit-testen vereist. Anderzijds lijkt dit ook tot verschillende versies van dezelfde oefening, wat het onderhouden van de oefeningen moeilijker 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 constatacie 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 wiskunde⁴. 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 programmeertaalafhankelijk 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:

⁴De 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 programmeertaalafhankelijk 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:

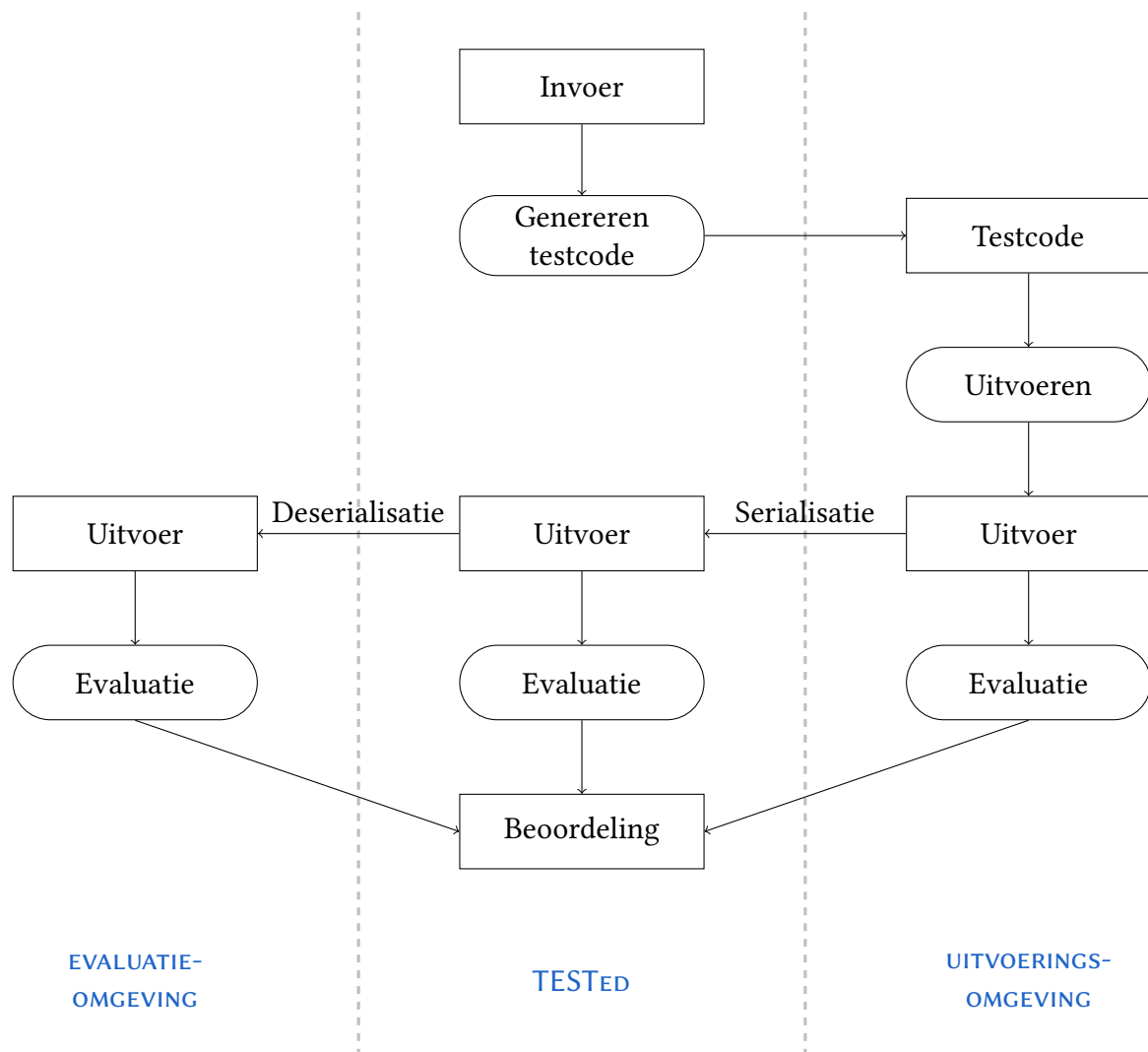
1. De Docker-container voor *TESTed* wordt gestart. Dodona stelt de invoer ter beschikking aan de container: het testplan komt uit de oefening, terwijl de ingediende oplossing en de configuratie uit Dodona komen.
2. Als eerste stap wordt gecontroleerd dat het testplan de programmeertaal van de ingediende 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.

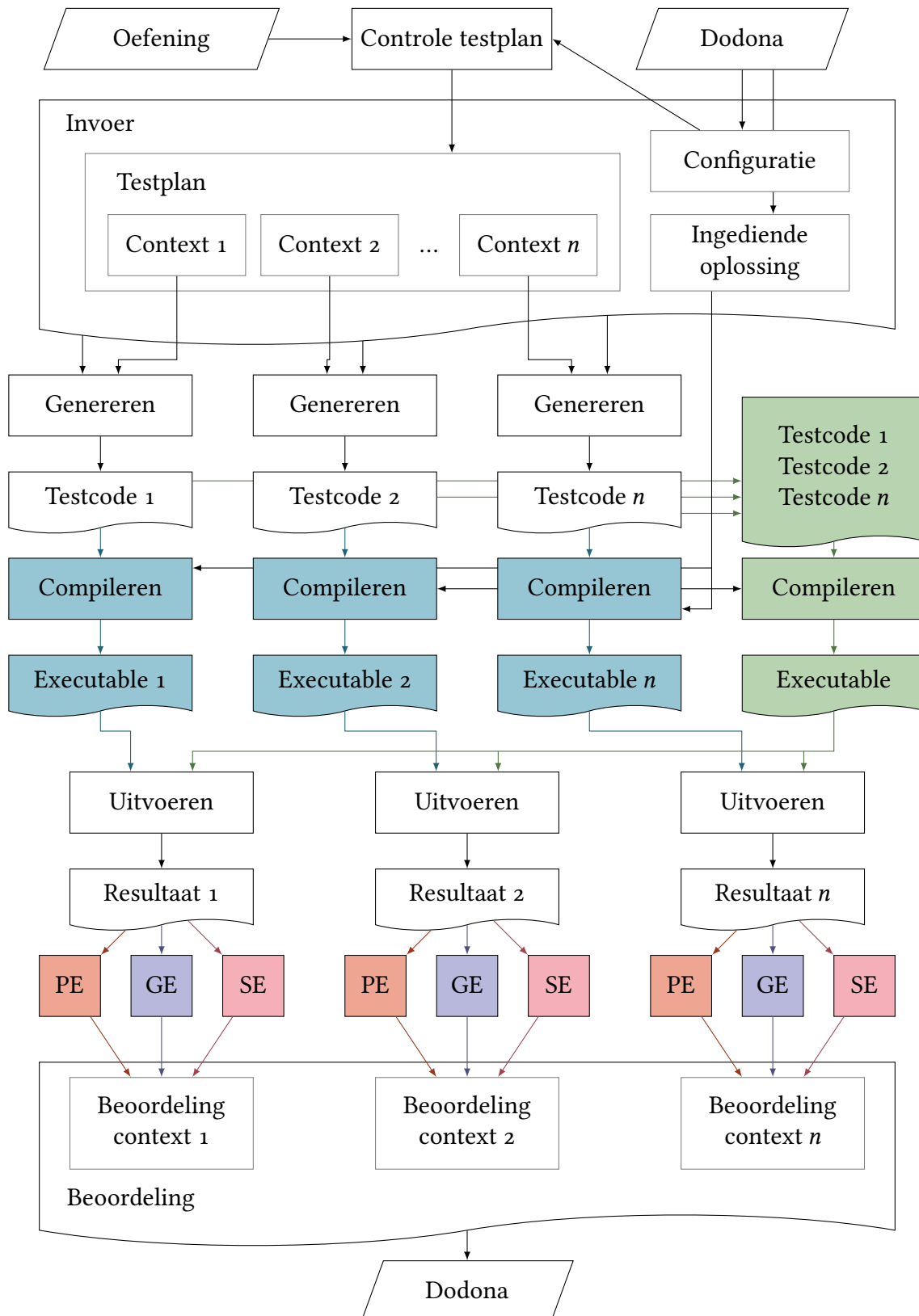
3. De testcode wordt gegenereerd voor elke context uit het testplan, wat resulteert in de testcode voor elke context. Deze stap is de overgang naar de *uitvoeringsomgeving*.
4. De code wordt optioneel gecompileerd. Dit kan op twee manieren gebeuren (meer details in ??):
 - a) Batchcompilatie: hierbij 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 het 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 parallelisatie binnen een judge is.
6. De uitvoering van de executable in de vorige stap resulteert in resultaten (voor elke context), zoals de standaarduitvoerstream, de standaardfoutstream, 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 programmeertaalafhankelijke 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 programmeertaalafhankelijke 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 *tabs* 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 *contexten*. 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 *testcases* 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 main-functie (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 programma-argumenten 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 *tests*, die elk één aspect van het testgeval controleren. Met aspect bedoelen we het standaarduitvoerkanal, het standaardfoutkanal, opvangen 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
- programmeertaalafhankelijk 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 programmeertaalafhankelijk 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*),

```

1 {
2   "tabs": [
3     {
4       "name": "Feedback",
5       "contexts": [
6         {
7           "input": {
8             "function": {
9               "type": "top",
10              "name": "loterij",
11              "arguments": [
12                { "type": "integer", "data": 6 },
13                { "type": "integer", "data": 15 }
14              ]
15            }
16          },
17          "output": {
18            "result": {
19              "value": {
20                "type": "text",
21                "data": "1 - 6 - 7 - 11 - 13 - 14"
22              },
23              "evaluator": {
24                "type": "custom",
25                "language": "python",
26                "path": "./evaluator.py",
27                "arguments": [
28                  { "type": "integer", "data": 6 },
29                  { "type": "integer", "data": 15 }
30                ]
31              }
32            }
33          }
34        ]
35      }
36    ]
37  }
38 }

```

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 programmeertaalafhankelijke 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 encodings 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 serialisatie-formaat 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 encoding 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.

```

1 {
2   "type": "sequence",
3   "data": [
4     {
5       "type": "integer",
6       "data": 5
7     },
8     {
9       "type": "integer",
10      "data": 15
11    }
12  ]
13 }

```

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. Hier-voor 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.

```

1 {
2   "type": "object",
3   "properties": {
4     "data": {
5       "type": ["number", "string", "boolean", "object", "array", "null"]
6     },
7     "type": {
8       "type": "string",
9       "enum": ["<types>"]
10    }
11  }
12 }

```

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 *Gevenstypes* van paragraaf 2.2.2.

Gevenstypes

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 mogelijkheid 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. Gevenstypes in deze categorie kunnen enkel gebruikt worden bij het aangeven welk gegevenstype verwacht wordt, niet bij de eigenlijke encoding 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 aliases van short, unsigned, ...

```

1 {
2   "type": "object",
3   "properties": {
4     "data": {
5       "type": ["number", "string", "boolean", "object", "array", "null"]
6     },
7     "type": {
8       "type": "string",
9       "enum": ["<types>"]
10    }
11  }
12 }
```

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.

```

1 var variabele = new DoorDeStudentGemaakteKlasse();
2 assert variabele.testfunctie1() == 15;
3 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 nietodeloos 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 programmeertaalafhankelijke 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:

```
1 {  
2   "name": "Naam van de variabele",  
3   "expression": "<Object voor functieoproep>",  
4   "type": "Optioneel type"  
5 }
```

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.

```
1 {
2   "name": "name",
3   "expression": {
4     "type": "identity",
5     "arguments": [
6       {
7         "type": "text",
8         "value": "Dodona"
9       }
10    ]
11  }
12 }
```

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 programmeertaalafhankelijke 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 toekenningsoopdracht 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 optreedt 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 *uitvoerkanal* genoemd. TESTed verzamelt volgende uitvoerkanalen:

- De standaarduitvoerstream. Dit wordt verzameld als tekstuele uitvoer.
- De standaardfoutstream. 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.

```

1  import values
2  import sys
3
4  value_file = open(r"Ogdcj0QzN_values.txt", "w")
5  exception_file = open(r"Ogdcj0QzN_exceptions.txt", "w")
6
7  def write_delimiter(delimiter):
8      value_file.write(delimiter)
9      exception_file.write(delimiter)
10
11 def send(value):
12     values.send_value(value_file, value)
13
14 def send_exception(exception):
15     values.send_exception(exception_file, exception)
16
17 def e_evaluate_main(value):
18     send_exception(value)
19
20 def v_evaluate_0(value):
21     send(value)
22
23 def e_evaluate_0(value):
24     send_exception(value)
25
26 # Import code.
27 try:
28     from submission import *
29 except Exception as e:
30     raise e
31
32 # Context 0-0.
33 try:
34     v_evaluate_0(    loterij(6, 15)    )
35 except Exception as e:
36     e_evaluate_0(e)
37
38 sys.stderr.write("--Ogdcj0QzN-- SEP")
39 sys.stdout.write("--Ogdcj0QzN-- SEP")
40 write_delimiter("--Ogdcj0QzN-- SEP")
41
42 value_file.close()
43 exception_file.close()

```

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

```

1 public class Context_0_0 {
2
3     private final FileWriter valueWriter;
4     private final FileWriter exceptionWriter;
5
6     public Context_0_0() throws Exception {
7         this.valueWriter = new FileWriter("TDm75Wrze_values.txt");
8         this.exceptionWriter = new FileWriter("TDm75Wrze_exceptions.txt");
9     }
10
11     private void send(Object value) throws Exception {
12         Values.send(valueWriter, value);
13     }
14
15     private void sendE(Exception exception) throws Exception {
16         Values.sendException(exceptionWriter, exception);
17     }
18
19     private void vEvaluate0(Object value) throws Exception { send(value); }
20
21     private void eEvaluate0(Exception value) throws Exception { sendE(value); }
22
23     void execute() throws Exception {
24         try {
25             this.vEvaluate0(Main.loterij(6, 15));
26         } catch (Exception e) {
27             this.eEvaluate0(e);
28         }
29         System.err.print("--TDm75Wrze-- SEP");
30         System.out.print("--TDm75Wrze-- SEP");
31         valueWriter.write("--TDm75Wrze-- SEP");
32         exceptionWriter.write("--TDm75Wrze-- SEP");
33     }
34
35     void close() throws Exception {
36         this.valueWriter.close();
37         this.exceptionWriter.close();
38     }
39
40     public static void main(String[] a) throws Exception {
41         var context = new Context_0_1();
42         context.execute();
43         context.close();
44     }
45 }

```

Codefragment 2.6.: Gegenerateerde 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.

```

1  class Selector {
2      public static void main(String[] a) throws Exception {
3          var name = a[0];
4          if ("Context_0_0".equals(name)) {
5              Context_0_0.main(new String[]{});
6          }
7          if ("Context_0_1".equals(name)) {
8              Context_0_1.main(new String[]{});
9          }
10     }
11 }

```

Codefragment 2.7.: Gegenerateerde 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.

```

1  workdir                                //Werkmap van TESTed
2  |   common                            //Gemeenschappelijke code
3  |   |   context_0_0.py                //Broncode voor context 0 van tabblad 0
4  |   |   context_0_0.pyc               //Gecompileerde code voor context 0 van tabblad 0
5  |   |   context_0_1.py
6  |   |   context_0_1.pyc
7  |   |   ...
8  |   |   context_0_49.py
9  |   |   context_0_49.pyc
10 |   |   ...
11 |   |   context_1_49.py                //Broncode voor context 49 van tabblad 1
12 |   |   context_1_49.pyc
13 |   |   submission.py                 //Ingediende oplossing
14 |   |   submission.pyc
15 |   |   values.py                     //Values-module
16 |   |   values.pyc
17 |   context_0_0                        //Map voor context 0-0
18 |   |   FaLd6WGRN_exceptions.txt      //Uitzonderingskanaal
19 |   |   FaLd6WGRN_values.txt          //Kanaal voor returnwaarden
20 |   |   context_0_0.pyc               //Code voor context 0-0
21 |   |   submission.pyc
22 |   |   values.pyc
23 |   ...
24 |   evaluators                        //Aangepaste evaluatoren
25 |   |   buzzchecker_Aao9H18Ve         //Map voor elke context
26 |   |   |   buzzchecker.py            //Code aangepaste evaluator
27 |   |   |   evaluation_utils.py
28 |   |   |   evaluator_executor.py
29 |   |   |   values.py
30 |   |   buzzchecker_B5WViK0zQ
31 |   |   |   buzzchecker.py
32 |   |   |   evaluation_utils.py
33 |   |   |   evaluator_executor.py
34 |   |   |   values.py
35 |   ...

```

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.

```

1  {"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 volstaat de generieke evaluatie door 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. Momenteel zijn er drie soorten ingebouwde evaluatiemethoden, die hieronder besproken worden.

Tekstevaluatie

Deze evaluator vergelijkt de verkregen uitvoer van een uitvoerkanaal (standaarduitvoer, return-waarde, ...) met de verwachte uitvoer uit het testplan. Alle data worden als string behandeld. Deze evaluator biedt enkele opties om het gedrag aan te passen:

ignoreWhitespace Witruimte voor en na het resultaat wordt genegeerd.

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

tryFloatingPoint De waarde moet geïnterpreteerd worden als een zwevendekommagetal (*floating point*), waarbij rekening gehouden wordt met de foutmarge.

applyRounding Of zwevendekommagetallen afgerond moeten worden. Indien wel wordt het aantal cijfers genomen van de optie `roundTo`.

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

Bestandsevaluator

Hiermee kan een geproduceerd bestand vergeleken worden met een gegeven bestand uit het testplan. Het gaat om tekstuele bestanden. Deze evaluator kan werken in drie modi:

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.

values Elke regel wordt geïnterpreteerd als een tekstuele waarde en vergeleken met de tekstevaluator. In deze modus worden kunnen ook alle opties van de tekstevaluator gebruikt worden.

Waarde-evaluator

Deze evaluator vergelijkt twee waarden, zoals gedefinieerd door het serialisatieformaat. De twee waarden moeten exact overeenkomen, met uitzondering van zwevendekommagetallen.

```

1 def evaluate(expected, actual, arguments):
2     """
3     :param expected: The expected value from the testplan.
4     :param actual: The actual value produced by the student's code.
5     :param arguments: Arguments from the testplan.
6     """
7     pass

```

Codefragment 2.10.: De definitie van de aangepaste evaluator.

2.4.2. Aangepaste evaluator

Voor de aangepaste evaluator moet een bestand geschreven worden in een programmeertaal naar keuze. Het resultaat van de uitvoering wordt vervolgens geserialiseerd en gedeserialiseerd naar het evaluatieproces. Hoe een evaluator moet geïmplementeerd worden, hangt af van de programmeertaal.

In Python bestaat de aangepaste evaluator uit een module met een functie die voldoet aan de definitie, zoals gegeven in codefragment 2.10. De judge 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 2.11.

In de Java-implementatie is de situatie gelijkaardig: het gaat om het implementeren van een abstracte klasse, die ook dienst doet als de module van Python. Deze klassen en haar ouder staan in codefragmenten 2.12 en 2.13.

Taalspecifieke evaluator

De taalspecifieke evaluator lijkt sterk op de aangepaste evaluator. is de eenvoudigste: deze neemt een codefragment met daarin één functie `evaluate`, die één argument aanvaardt, de geproduceerde waarde. Waar de geproduceerde waarde bij de aangepaste evaluator in het serialisatieformaat moet kunnen, is dit hier niet het geval: de functie wordt rechtstreeks opgeroepen tijdens de uitvoering. In Python wordt dit codefragment 2.14, in Java codefragment 2.15. Om het resultaat van de evaluatie aan de judge te geven, wordt dezelfde `evaluated`-functie als bij de aangepaste evaluator gebruikt (respectievelijk codefragmenten 2.11 en 2.12)


```

1  import values
2  import sys
3
4  from typing import List, Optional
5
6
7  def evaluated(result: bool,
8               readable_expected: Optional[str] = None,
9               readable_actual: Optional[str] = None,
10              messages: Optional[List[str]] = None):
11      """
12      Report the result of an evaluation to the judge. This method should only
13      be called once, otherwise things will break.
14
15      :param messages: Optional list of messages to be shown to the student.
16      :param readable_actual: A string version of the actual value. Optional; if
17                           not given, the judge will produce one on a best-
18                           efforts basis.
19      :param readable_expected: A string version of the expected value. Optional;
20                           if not given, the judge will produce one on a
21                           best-efforts basis.
22      :param result: The result of the evaluation.
23      """
24      if messages is None:
25          messages = []
26
27      values.send_evaluated(sys.stdout,
28                          result, readable_expected, readable_actual, messages)

```

Codefragment 2.11.: De implementatie van de module evaluation_utils

```

1  import java.io.Closeable;
2  import java.io.IOException;
3  import java.io.OutputStreamWriter;
4  import java.util.Collection;
5  import java.util.List;
6
7  abstract class AbstractEvaluator implements Closeable {
8
9      protected final OutputStreamWriter writer;
10
11     public AbstractEvaluator() {
12         this.writer = new OutputStreamWriter(System.out);
13     }
14
15     @Override
16     public void close() throws IOException {
17         this.writer.close();
18     }
19
20     /**
21      * Report the result of an evaluation to the judge. This method should only
22      * be called once, otherwise things will break.
23      *
24      * @param result          The result of the evaluation.
25      * @param readableExpected Optional string version of the expected value.
26      * @param readableActual  Optional string version of the actual value.
27      * @param messages        Optional list of messages to pass to the student.
28      */
29     protected void evaluated(boolean result, String readableExpected,
30                             String readableActual,
31                             Collection<String> messages) throws IOException {
32         Values.evaluated(writer,
33             result, readableExpected, readableActual, messages);
34     }
35
36     protected void evaluated(boolean result, String readableExpected,
37                             String readableActual) throws IOException {
38         Values.evaluated(writer,
39             result, readableExpected, readableActual, List.of());
40     }
41 }

```

Codefragment 2.12.: De implementatie van de klasse AbstractEvaluator.

```

1  import java.io.IOException;
2  import java.util.List;
3
4  abstract class AbstractCustomEvaluator extends AbstractEvaluator {
5
6      abstract void evaluate(Object expected,
7                             Object actual,
8                             List<Object> arguments) throws IOException;
9  }

```

Codefragment 2.13.: De implementatie van de klasse AbstractCustomEvaluator.

```

1  def evaluate(actual):
2      """
3      :param actual: The actual value produced by the student's code.
4      """
5      pass

```

Codefragment 2.14.: De definitie van de taalspecifieke evaluator.

```

1  import java.io.IOException;
2  import java.util.List;
3
4  abstract class AbstractSpecificEvaluator extends AbstractEvaluator {
5
6      abstract void evaluate(Object actual) throws IOException;
7  }

```

Codefragment 2.15.: De implementatie van de klasse AbstractSpecificEvaluator.

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 Dodona¹.

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:

```
1 {
2   "description": {
3     "difficulty": 2.0,
4     "names": {
5       "en": "ISBN",
6       "nl": "ISBN"
7     }
8   },
9   "access": "public",
10  "programming_language": "python",
11  "evaluation": {
12    "plan_name": "plan.json"
13  }
14 }
```

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.

¹Hier 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:

```
1 isbn
2 └─ description
3     └─ description.en.html
4     └─ description.nl.html
5     └─ media
6         └─ 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.

```
1 import random
2 import string
3 import json
4
5 # De implementatie van het serialisatieformaat.
6 from isbn.preparation import values
7 # De voorbeeldoplossing.
8 from isbn.solution import solution
9
10 # Met een vaste seed krijgen we deterministische resultaten.
11 random.seed(123456789)
12
13 # We halen de naam van het testplan op uit de configuratie van de oefening.
14 with open("../config.json", "r") as config_file:
15     config = json.load(config_file)
16 testplan_name = config["evaluation"].get("plan_name", "plan.json")
```

```

17
18 # Of alle testgevallen in dezelfde context moeten plaatsvinden of niet.
19 # Aangezien ze onafhankelijk zijn, doen we dit niet.
20 # Meerdere contexten hebben echter wel een performantiekost, waardoor we de
21 # optie toch voorzien.
22 ONE_CONTEXT = False
23
24
25 def check_digit10(code):
26     """Bereken het controlecijfer voor een ISBN van lengte 10."""
27     check = sum((i + 1) * int(code[i]) for i in range(9)) % 11
28     # Zet het controlecijfer om naar een string.
29     return 'X' if check == 10 else str(check)
30
31
32 def check_digit13(code):
33     """Bereken het controlecijfer voor een ISBN van lengte 13."""
34     check = sum((3 if i % 2 else 1) * int(code[i]) for i in range(12))
35     # Zet het controlecijfer om naar een string.
36     return str((10 - check) % 10)
37
38
39 def random_characters(length, alphabet):
40     """Genereer een aantal willekeurige tekens uit een alfabet."""
41     return ''.join(random.choice(alphabet) for _ in range(length))
42
43
44 def generate_code():
45     """Genereer een ISBN-code, met zowel lengte 10 als 13."""
46     length = random.choice([10, 13])
47     code = random_characters(length - 1, string.digits)
48     if length == 10:
49         if random.random() < 0.5:
50             code += check_digit10(code)
51         else:
52             code += random.choice(string.digits + 'X')
53     else:
54         if random.random() < 0.5:
55             code += check_digit13(code)
56         else:
57             code += random.choice(string.digits)
58     return code
59
60
61 def generate_is_isbn():
62     """
63     Genereer de contexten voor de "is_isbn"-functie.
64     :return: De gegenereerde contexten.
65     """

```



```

66 contexts = []
67
68 # Genereer eerst de argumenten voor de functie "is_isbn".
69 # We beginnen met wat vaste combinaties.
70 args = [
71     ('9789027439642', False),
72     ('9789027439642', True),
73     ('9789027439642', None),
74     ('080442957X', None),
75     ('080442957X', False),
76     (9789027439642, None),
77 ]
78 # Voor de rest vullen we aan met willekeurige argumenten.
79 while len(args) < 50:
80     code = generate_code()
81     args.append((code, random.choice([None, True, False])))
82
83 # Genereer de eigenlijke contexten.
84 for code, isbn13 in args:
85     # Eerst doen we de functieoproep. We geven zeker de ISBN mee.
86     function_arguments = [
87         values.encode(code)
88     ]
89     # Indien nodig geven we ook het tweede argument mee.
90     if isbn13 is not None:
91         function_arguments.append(values.encode(isbn13))
92
93     # Bereken het resultaat met de gegeven argumenten.
94     result = solution.is_isbn(code, isbn13 if isbn13 is not None else True)
95
96     # Ons testgeval bevat de functieoproep als invoer, en de berekende waarde
97     # als verwachte uitvoer.
98     testcase = {
99         "input": {
100             "function": {
101                 "type": "top",
102                 "name": "is_isbn",
103                 "arguments": function_arguments
104             }
105         },
106         "output": {
107             "result": {
108                 "value": (values.encode(result))
109             }
110         }
111     }
112
113     # Steek het testgeval in een context.
114     context = {

```

```

115         "normal": [testcase]
116     }
117     contexts.append(context)
118
119     return contexts
120
121
122 def generate_are_isbn():
123     """
124     Genereer de contexten voor de "are_isbn"-functie.
125     :return: De gegenereerde contexten.
126     """
127
128     # Vaste invoerargumenten die we zeker in het testplan willen.
129     codes = [
130         '0012345678', '0012345679', '9971502100', '080442957X', 5, True,
131         'The Practice of Computing Using Python', '9789027439642', '5486948320146'
132     ]
133     codes2 = ['012345678' + str(digit) for digit in range(10)]
134     args = [
135         (codes, None),
136         (codes, True),
137         (codes, False),
138         (codes2, None),
139         (codes2, True),
140         (codes2, False),
141     ]
142     # Vul opnieuw de rest aan tot we aan 50 zitten.
143     while len(args) < 50:
144         codes = [generate_code() for _ in range(random.randint(4, 10))]
145         args.append((codes, random.choice([None, True, False])))
146
147     # Genereer de eigenlijke contexten.
148     contexts = []
149     for index, (codes, isbn13) in enumerate(args):
150         index += 1
151         # Deze keer pakken we het iets anders aan: de lijst van ISBN's kennen we
152         # eerst toe aan een variabele, en geven het niet rechtstreeks mee als
153         # argument. Maak het testgeval voor de assignment.
154         assignment_testcase = {
155             "input": {
156                 "assignment": {
157                     "name": f"codes{index:02d}",
158                     "expression": {
159                         "type": "identity",
160                         "arguments": [
161                             values.encode(codes)
162                         ]
163                     }
164                 }
165             }
166         }

```

```

164         }
165     }
166 }
167
168 # Stel de functieargumenten op. We geven opnieuw sowieso de variabele die
169 # we eerst hebben aangemaakt mee als argument.
170 function_arguments = [
171     {
172         "type": "literal",
173         "data": f"codes{index:02d}"
174     }
175 ]
176
177 # Voeg het tweede argument toe indien nodig.
178 if isbn13 is not None:
179     function_arguments.append(values.encode(isbn13))
180
181 # Bereken het resultaat.
182 result = solution.are_isbn(codes, isbn13 if isbn13 is not None else None)
183
184 # Maak het normale testgeval. We hebben opnieuw als invoer de functie en als
185 # uitvoer de verwachte waarde.
186 testcase = {
187     "input": {
188         "function": {
189             "type": "top",
190             "name": "are_isbn",
191             "arguments": function_arguments
192         }
193     },
194     "output": {
195         "result": {
196             "value": (values.encode(result))
197         }
198     }
199 }
200
201 # Voeg beide testcases toe aan een context.
202 # Elke testcase heeft hoogstens één functieoproep of assignment, maar niet
203 # beide. Daarom hebben we nu twee testgevallen. Het eerste testgeval voor
204 # de assignment heeft enkel de standaardtests, wat wil zeggen dat er bv.
205 # geen uitvoer op stderr mag zijn.
206 context = {
207     "normal": [assignment_testcase, testcase]
208 }
209
210 contexts.append(context)
211 return contexts
212

```

```

213
214 def flatten_contexts(contexts):
215     """Voeg alle testgevallen samen tot 1 context."""
216     testcases = [context["normal"] for context in contexts]
217     flat = [item for sublist in testcases for item in sublist]
218     new_context = {
219         "normal": flat
220     }
221     return new_context
222
223
224 # Creëer de contexten.
225 tab_1_contexts = generate_is_isbn()
226 tab_2_contexts = generate_are_isbn()
227
228 # Creëer het testplan met de twee tabbladen.
229 plan = {
230     "tabs": [
231         {
232             "name": "is_isbn",
233             "contexts": tab_1_contexts
234         },
235         {
236             "name": "are_isbn",
237             "contexts": tab_2_contexts
238         }
239     ]
240 }
241
242 # Indien we alles in één context willen, doen we dat.
243 if ONE_CONTEXT:
244     new_tab1_context = flatten_contexts(plan["tabs"][0]["contexts"])
245     plan["tabs"][0]["contexts"] = [new_tab1_context]
246     new_tab2_context = flatten_contexts(plan["tabs"][1]["contexts"])
247     plan["tabs"][1]["contexts"] = [new_tab2_context]
248
249 # Terugvallen op individuele modus is niet nuttig in Python, dus laten we dat niet
250 # toe. Indien het terugvallen niet nuttig is, is het sneller om het uit te zetten.
251 plan["configuration"] = {
252     "allow_fallback": False
253 }
254
255 # Schrijf het testplan.
256 with open(f"../evaluation/{testplan_name}", 'w') as fp:
257     json.dump(plan, fp, indent=2)

```

```

1 isbn
2 |─ config.json
3 |─ description
4 |   |─ description.en.html
5 |   |─ description.nl.html
6 |   └─ media
7 |       └─ ISBN.gif
8 |─ evaluation
9 |   └─ plan.json      # Gegenereerd door generator.py
10 |─ preparation
11 |   |─ generator.py
12 |   └─ values.py
13 |─ solution
14 |   └─ 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 worden².

²De 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>.