

École polytechnique de Louvain

Interoperability of Ingenuous and Dodona

Author: **Simon PONCHAU**

Supervisor: **Kim MENS**

Readers: **Peter DAWYNDT, Charles PECHEUR, Anthony GÉGO**

Academic year 2018–2019

Master [60] in Computer Science

Abstract

Inginious and Dodona are two platforms meant to automatically test programming code written by students. Inginius is developed by the UCLouvain and Dodona by the University of Ghent. For each exercise on each platform, the professor has to write an exercise statement, and some files to test the code. These files can be scripts, for Inginius, or input-output files, for Dodona.

The files are not the same for each platform, which means that if we want to create an exercise for the two platforms, we have to write the exercise twice. We cannot reuse the exercises of one platform for the other, in order to increase the number of exercises available. A need to make exercises of each platform compatible with the other was therefore felt.

Both platforms are written in Python. We can therefore try to create scripts of an acceptable difficulty that make the exercises compatible between the two platforms. To achieve that goal, our solution is to convert the exercises into the format of the other platform. As both platforms need different files to manage the exercises, we will need specific scripts to go from one platform to the other. After the transformation, the exercises will be compatible with the platform on which we want to run it.

A generic format could also be foreseen in a later stage to make exercises compatible with other platforms as well, provided the creation of appropriate conversion scripts.

Contents

1	Introduction	6
2	Related Work	8
2.1	Different code grading platforms	8
2.1.1	Tools from private companies	8
2.1.2	Tools developed by universities	9
2.1.3	Other open-source tools for the evaluation of program code .	12
2.2	Comparison between platforms	13
2.3	Existing standards	13
2.3.1	Learning objects	14
2.3.2	PExIL	15
2.4	Conclusion	17
3	Problem Statement	18
3.1	Presentation of Inginious and Dodona	18
3.1.1	Global	18
3.1.2	Inginious	18
3.1.3	Dodona	23
3.1.4	Similarities and differences between Inginious and Dodona .	28
3.1.5	Comparison	30
3.2	Needs	31
3.3	Conclusion	31
4	Solution	32
4.1	Dodona to Inginious	33
4.1.1	Implementation	33
4.1.2	How to use it?	38
4.2	Inginious to Dodona	39
4.2.1	Implementation	39
4.2.2	How to use it?	45
4.3	Conclusion	45

5	Validation	46
5.1	Inginious to Dodona	46
5.2	Dodona to Inginiuous	49
6	Future Work	50
6.1	New Format	50
6.2	Dodona judge in Inginiuous	52
7	Conclusion	53

Acknowledgments

We would like to thank Kim Mens, our supervisor, for his tremendous follow-up, remarks and advice throughout the year.

We also express our gratitude to Anthony G go for his technical input on the Inginius platform, his good advice on the tools to use and his support during the drafting of our thesis.

We are grateful as well to the University of Ghent for its technical input on the Dodona platform.

Finally, we extend our gratitude to Peter Dawyndt and Charles Pecheur who agreed to read over this thesis.

Chapter 1

Introduction

Assessing programming code can be long and tedious. The UCLouvain and the University of Ghent, like many others, have implemented their own platform to automate the correction of student programming code. For UCLouvain and the University of Ghent, those platforms are respectively called Ingenious and Dodona. On these platforms, exercises will be created and each student, provided they have access to the exercise course, may submit code that will be evaluated automatically. These exercises are in principle implemented by professors or assistants of the computer science or engineering departments of each university.

But each platform has its own way of implementing exercises and its own way of correcting them, which is a shame, because exercises implemented for one platform cannot be reused on the other platform.

A desire to be able to recover exercises already available on the other platform was also felt. This would expand the set of exercises available to students. It would lead to more training opportunities for the students to carry out evaluations, or will increase the number of exercises available to teachers.

After a through analysis, it turned out that the technologies used by these two platforms were quite similar. They are both implemented in Python and based on Docker containers. These similarities suggest that it may be possible to develop exercises compatible from one platform to another. This thesis aims at analyzing the feasibility of this interoperability, and to implement it if feasible.

In order to make this work as generic as possible, and to be able to use it through other platforms, we will also define a common generic format to define these exercises. So, if we want to add another platform in the future, we only need to worry about exporting and importing the exercises from and into this format.

We will mainly work with Python exercises throughout this thesis. The principle for other languages remains the same and can be implemented later.

Many other platforms with a similar purpose exist. Some of these are open-source, which allows us to analyze them, while others are not. Given their number, a standard exercise format has been defined [1]. This standard has been created before 2011 and it is based on learning objects. This format could make the various platforms compatible. However, we did not find a single platform complying with this standard. The format is still well thought. We will, based on it, create a practical new format better suited to our case. As we will see, this previous format is poorly documented and not always perfectly adapted.

The remainder of our thesis is structured as follows: the next two chapters provide the necessary background material and a description of the different automatic code evaluation platforms already in existence as well as the standards already created. Next, we will explore the specifications and characteristics of the two platforms we will be dealing with: Ingenious and Dodona. To conclude, we explained how we made the two platforms compatible and why we did it this way. The software architecture supporting the implementation of that solution is detailed, and illustrated. We will also describe how we test the interoperability between both platforms and will show the good level of performance and will provide a series of examples. Finally, the conclusion emphasizes the main contributions of this research and explores avenues for the future.

Chapter 2

Related Work

2.1 Different code grading platforms

Since the thesis deals with the interoperability between two program code evaluation platforms, it is interesting to analyze already existing platforms. It will allow to see if certain patterns are repeated between the platforms and whether standards are put in place. It will also give an idea on how to do things in the most efficient possible way.

2.1.1 Tools from private companies

There are many other similar platforms for automatically testing code, written by users. Most have been developed by companies in order to assess the skills of workers or future recruits.

Among these we find :

- Ans [2]: It is a platform that aims to evaluate students.
- Mettl [3], Hackerrank [4]: These tools test the skills of certain people to allow employers to check the level of (potential future) employees.
- Sololearn [5]: This platform is intended to enable the learning of programming.
- Devskiller [6]: Tests the level of some people in order to hire them. This platform has the particularity to allow to test on frameworks and libraries in addition to the different programming languages.
- Talview [7]: This international platform aims at finding young talent, or at testing in an academic setting.

Unfortunately the code and architecture of these platforms are not available. So we could not analyze them and see if a pattern emerges out of exercises definitions.

2.1.2 Tools developed by universities

- **Open-source tools**

- **Autolab** [8]: Autolab is an open-source platform for automatic assessment of programming code created in 2010 and used by a large number of universities (currently 17). It allows, just like Ingenious and Dodona, to evaluate the programming skills of students. It makes it possible to evaluate student code and to give them comments directly in their code. It is also interesting to detect fraud or plagiarism.

This platform works with a RESTful Web service that allows you to run tests. It also works with docker containers.

The files needed to create an exercise are:

- * a file containing the solution code
- * a script to launch the codes
- * a makefile for the solution code
- * a template file for the student
- * a makefile for the student code

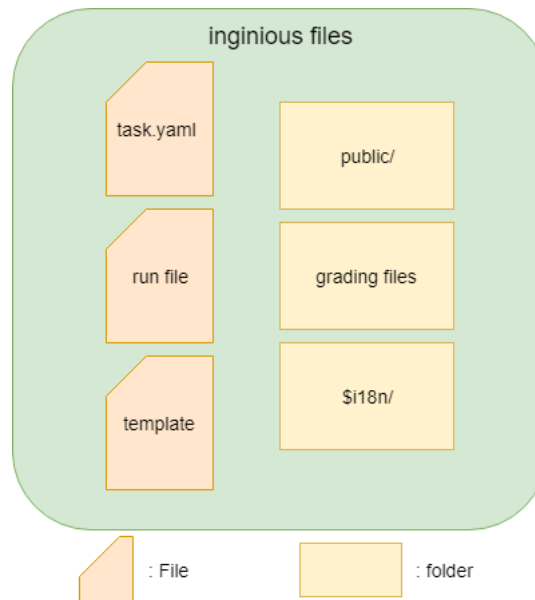
The tests are compiled (if necessary) and then run via the makefile. The result is then returned via a status code.

- **Ingenious** [9]: Ingenious is an automatic student code testing platform created by the UCLouvain in an educational setting. This platform is built in Python. The tests are launched in Docker containers. Ingenious allows teachers to create their own test scripts.

The exercises are placed in a folder containing all the files related to the exercise.

It contains the following files and folders shown on diagram 2.1:

Figure 2.1: Inginius files



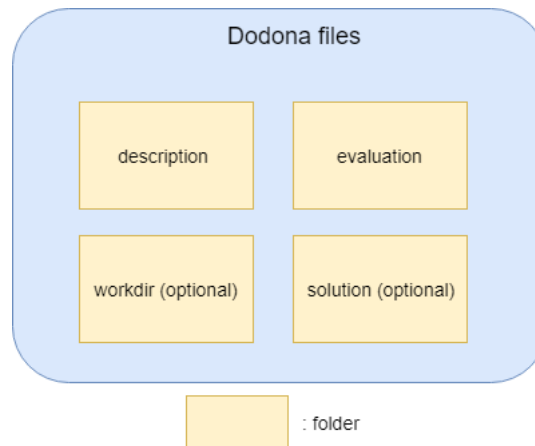
- * **task.yaml** : This file will contain all the information related to the exercise. In this one, we will find the statement of the exercise, but also the various parameters which are related to it.
- * **run file** : This file written, in either bash or Python is the entry point of the tests. In this file, all scripts/tests files will be launched. It will also collect all the results and transmit them to the front-end to give feedback to the student.
- * **template file** : When the student writes a piece of code, the teacher may want to place it in a specific place in a program before performing the tests. The program around the student code is the template. This file is not required.
- * **public/** : In this folder, we will find all the files that must be accessible directly to the student. For example, the images displayed in the statement of the exercise.
- * **grading files** : All files launched by the run file and used to verify that the student code is correct.
- * **\$i18n/** : This folder is about language management. It will contain the files necessary to the translation of the exercises. The pybabel package is used to handle all this.

- **Private tools**

- **Dodona:** This platform was created by the University of Ghent for educational purposes. This platform is built in Python. When testing, a new docker container is created for the run. The test system is based on the principle of a series of inputs that are compared with a series of outputs.

Regarding the exercises, they are placed in a repository. In this repository, one finds, at the root, the file `dirconfig.json` containing the parameters related to all the exercises of the repository. We also find all the folders containing the exercises. In each of these folders we have a `config.json` file containing the parameters of the exercise. These folders accommodate also various folders for each exercise as shown on the diagram 2.2 :

Figure 2.2: Dodona files



- * **description folder** : It holds the files with the statements of the exercise and the files associated to them (images, CSS, ...). These will be stored either in HTML or markdown format and can be in several languages.
- * **evaluation folder** : In this file we find all the files necessary for the evaluation of the student's code (based on an input-output system). The content will be made available to the evaluation system.
- * **workdir folder (optional)** : Code accessible to the code submitted by the student.
- * **solution folder (optional)** : The solution can be included in this folder.

- **UPyLab** [10]: This tool was created by the ULB for educational purposes. Student submissions are tested through a reference solution. The platform works with Python.
- **Etropy** : This platform was created by the VUB. It is also based on Python.

2.1.3 Other open-source tools for the evaluation of program code

- **Kattis** [11]: Kattis is a judge that allows you to test and check code. It is developed in Python and it is based on Docker. For each exercise, we have files to test the inputs and outputs. There will also be a description of the exercise and a configuration file. One of the advantages of Kattis is that once we have defined an exercise, it will work with all types of languages supported by the platform. This is possible thanks to the input output system. Each exercise has its own folder. The exercises of the platform are structured in this way:
 - **problem.yaml file** : This file contains the parameters of the exercise.
 - **data folder** : This folder contains the input and output files of the exercise.
 - **input_format_validators folder** : In this folder we find the scripts to verify that the files in the data folder are properly formed.
 - **problem_statement folder** :The descriptions of the exercises are stored in this folder. They are written in $\text{\LaTeX}$ and can be available in several languages.
 - **submissions folder** : In this folder we have examples of successful code, but also examples that fail, and generate errors ...
- **Domjudge** [12]: Domjudge is a platform created for the purpose of making programming contests. It evaluates the user codes and gives a rating. It is also an open-source project. For evaluation of the code, Domjudge is based on Kattis.

The data to be imported for an exercise are mainly an input file, an output file and a description file. A script will then be run to test if both match with the code provided by the user. It is also possible to carry out further

tests, provided you follow the format imposed by Kattis.

Verification scripts are run by judgedaemons. It is possible to run one or more on each server according to the parameters that are defined.

The platform has been developed in Python.

- **Submittity** [13]: This platform was developed by Rensselaer Center for Open Source (RCOS). It allows automatic testing of all programming languages available on GNU/Linux. It allows advanced testing with tools such as static analysis, JUnit, code coverage, memory and debuggers.

To launch the tests, an input python script is called. It will launch the various scripts (tests) needed to evaluate the exercise. A feedback is then returned.

2.2 Comparison between platforms

We can see that most platforms work with similar technologies and principles. In most cases we find an infrastructure written in python and tests launched in containers. The structure of an exercise is also more or less similar. We have test scripts, a description file, and sometimes solution files.

We can also see that many universities have decided to develop their own platform. Unfortunately, while most seem to be based on similar technologies, their code remains private.

The Autolab platform is open-source and operated by many universities. However, we will not use this format as a reference, because to create a simple exercise there are many files to be created. With the format we are going to establish it will be much faster and easier to create a simple exercise. But it should not jeopardize the possibility to later create scripts to make the format of Autolab compatible with ours.

2.3 Existing standards

More and more self-correcting exercise platforms are being developed.

A standard has been developed to organize things and make the different platforms compatible with each other.

2.3.1 Learning objects

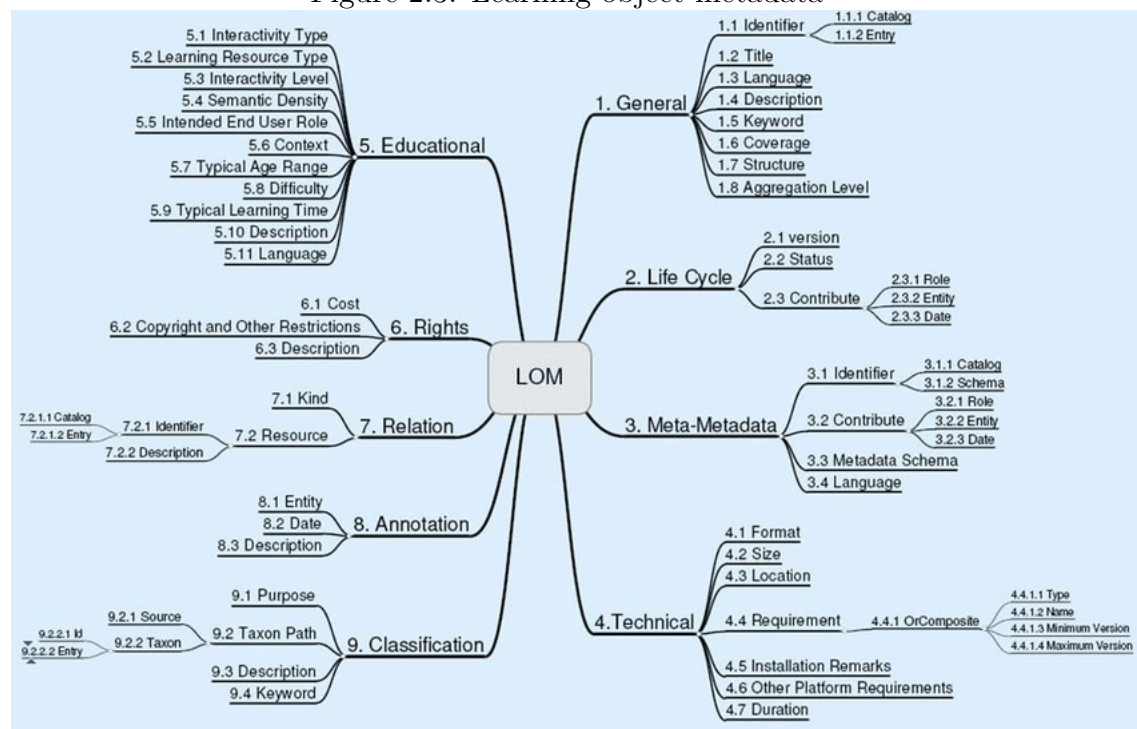
This standard is based on Learning Objects (LO) [14, 15, 16, 17] These are containers include educational files as well as meta-data describing them. There are many parts in LO that describe different types of content depending on how you want to use it. Each part is described by a different standard.

Data is available and learning objects (i.e. the exercises description) contain the associated data and the structure of these data.

A learning object is composed of two parts:

- The description of the exercises: The structure of these exercises is represented by diagram 2.3
- The data: Their structure is present in the description of the exercises.

Figure 2.3: Learning object metadata



The most widely used standard for LO is the IMS Content Packaging standard (IMS CP) [18]. It works with a manifest written in XML describing the contents of a zip, based on the IEEE standard. The zip will contain learning objects (LOM).

However, learning objects are not suitable for automatic exercise correction. Indeed, there is no way to describe the role of specific resources, such as test cases or solutions. But fortunately, the IMS CP standard has been planned to be easily extended.

For automatic multiple-choice or short-text responses, there is the QTI standard that extends IMS CP. This standard is designed to allow for the evaluation of questions with predefined answers, such as multiple choices, fill-in-the-blanks and short text questions. However, it is still not sufficient for the automatic correction of programming code.

2.3.2 PExIL

That's why the PExIL [1] standard was created. PExIL is based on XML. It allows you to describe all the necessary data in the life cycle of the exercise correction.

This description is divided into three parts:

- Textual: This section will contain the general information about the exercise. We will find for example the title of the exercise, the date of creation, the author... The statement of the exercise will also be located here.
- Specification: This part is used to describe the input and output data. We will check if passing some data inputs, expected outputs are likely be good.
- Programs: Here we will find the references to the source files of the program. We will identify the correction scripts, the corrected exercises...

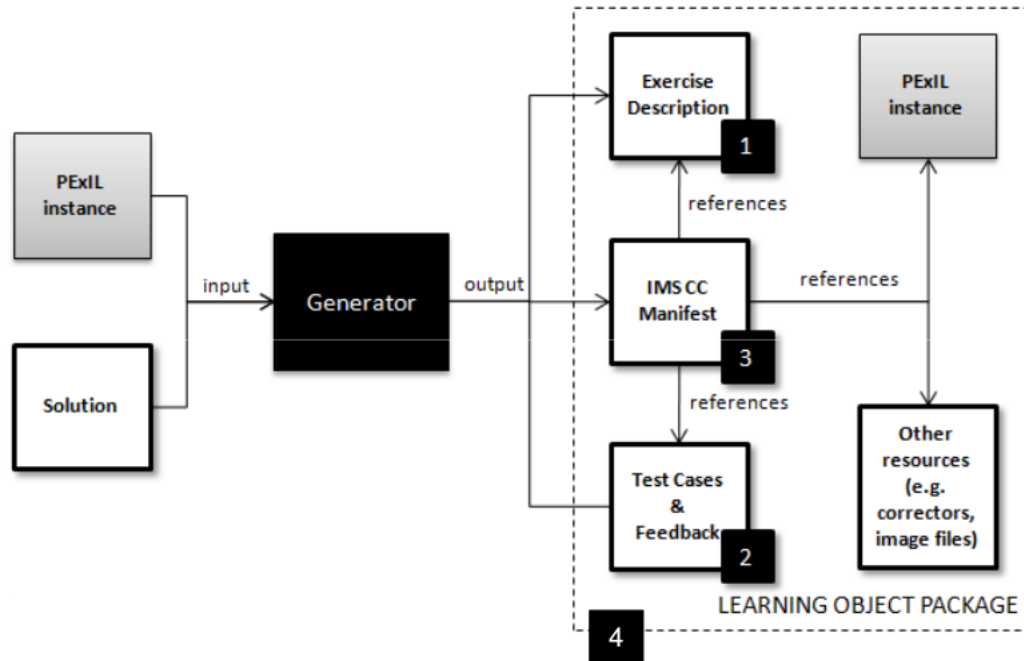
From this file, and from the solution program of the exercises, we will be able to generate various files necessary for the evaluation of exercises. All of these generated files are called the IMS CC LO package.

The following files will be generated:

- the description of the exercise
- different cases to test
- feedback files
- the IMS CC manifest : Containing references to all other files in the package.

Diagram (2.4), created by Ricardo Queirós and José Paulo Leal [1] describes the process of creating an exercise from a PExIL file and a solution:

Figure 2.4: PExIL



As we will see later, this format includes everything we need, to create a new, more generic format for Dodona, Inginious, and other platforms. We will have one or more description files, exercise-related parameters, and other exercise-related resources, such as images or the corrected code. However, for the part of the tests and feedback, we will need to be able to take into account both the execution of unit tests/scripts and a correction type input/output.

This is why we will need a format that is a little bit modified and specific to our case, while remaining as open as possible to allow other platforms to join this format.

On the other hand, the documentation relating to the PExIL format is rather limited, and we did not find any documentation to concretely use the standard.

2.4 Conclusion

We have seen in this chapter different existing platforms for the automatic evaluation of student code. We could spot the similarities and the differences between them. It induced an overview of the existing tools and allow us to see how exercises are described on these platforms.

We also covered existing standards. We described the interesting features and we determined why these standards do not apply in our case.

In the next chapter (problem statement), we will discuss more in detail the functioning of Ingenious and Dodona, their similarities and differences, as well as the needs identified which led us to draft this thesis.

Chapter 3

Problem Statement

3.1 Presentation of Ingenious and Dodona

3.1.1 Global

Ingenious and Dodona are exercise testing frameworks. Both allow managing exercise submission and correction. The platforms also manage exercise repositories. They allow teachers to create exercises, to submit them to a group of students and to automatically correct the exercise with a feedback to the student.

Both platforms can manage complex exercise like coding of programs. Ingenious can also manage simple ones like multiple choice questions, missing field missing,...

In this thesis we only concentrate on complex exercises as this has the greatest added value when exchanging exercises.

3.1.2 Ingenious

This platform was developed at UCLouvain. It was created for educational purposes. It allows the automatic correction of exercises of all kinds. Thus, it is possible for a teacher to create multiple choice questions, exercises with fields to fill in, files to upload and test automatically program code written by the student.

For automated tests, Ingenious allows teachers to write a script that will test code submitted by students. Each instance of the code will be launched in a different Docker container that will send back the result of the execution.

To do this, the professor will create a description file, a run file containing a set of unit tests that will verify the validity of the student code and the student code template.

- The description file is made up of several parts. On the one hand the description (the statement) of the exercise that will be visible to the student. On the other hand it also contains various parameters related to the exercise. For example, among these parameters we find the name, the memory limit, the time limit, the maximal number of submissions... This file is encoded in YAML.
- The run file is the entry point of the tests. The teacher has to enter the script that will execute the unit tests and produce the grading. Tests will be launched from here. This is also where we manage the feedback that is returned to the GUI. Each exercise contains its own run file. Finally, we have the unit tests performed on the code of the student. If the results obtained with the student code match those expected, Ingenious will inform the student that he has successfully completed the exercise. If not, the user gets detailed feedback from the system on how to correct his errors or improve his code.
- The student code template, which contains the code to add around the student code. This will allow the run file to access and to test the student answer.

Architecture of Ingenious

Figure 3.1 from the documentation of Ingenious [9] shows us how Ingenious works as a whole.

We can see different elements which are part of it.

To begin with, we have the **frontend**. Its role is to collect the code/answers introduced by the student. Whenever he submits his answer, it will be added to the queue.

Then we have the **backend**. It is responsible for making the transition between the frontend and the agent. It will collect the jobs in the queue and send them to the agent.

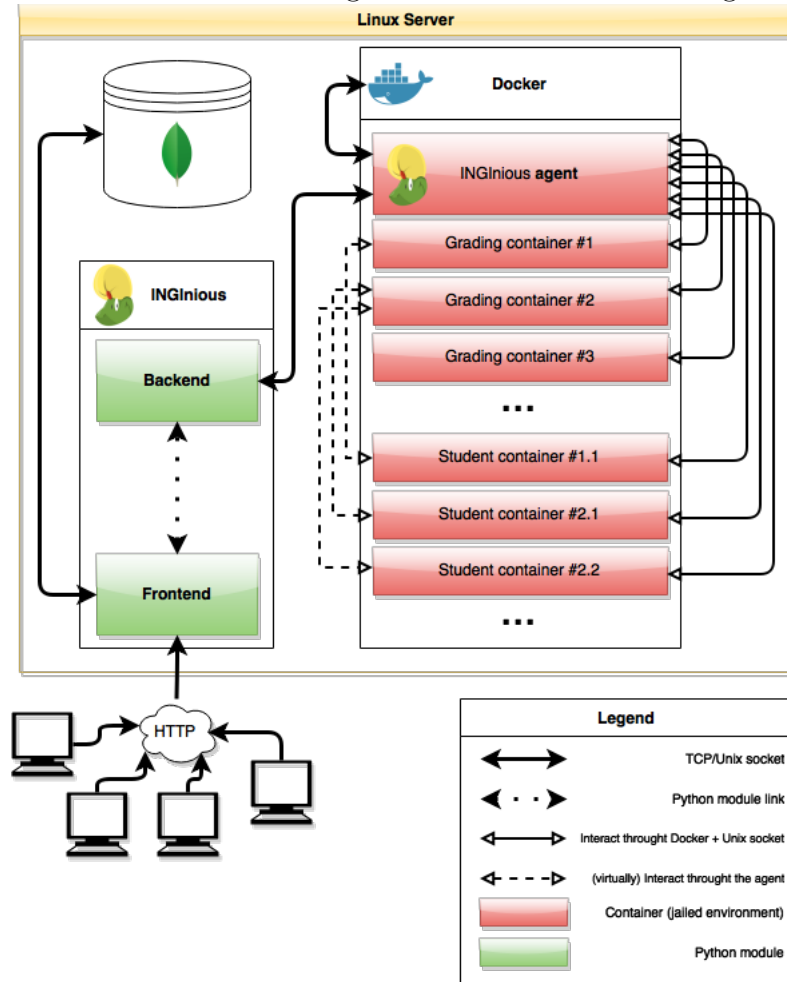
The **agent** will then retrieve all the codes of the exercise of the student and launch the evaluation containers. A container is a virtual environment (an operating system running independently on another operating system) into which the tests can be run. It will launch at least one container per exercise.

This container will be the **grading container**. It is responsible for launching the run file to evaluate the exercise. At the end of the evaluation, it will send the agent back to the backend, which will give it to the frontend which will save the results in the database and post them to the student.

It can also launch one or more **student containers** per exercise. They will run the student code separately from the rest.

The grading container and the student container communicate with each other via sockets.

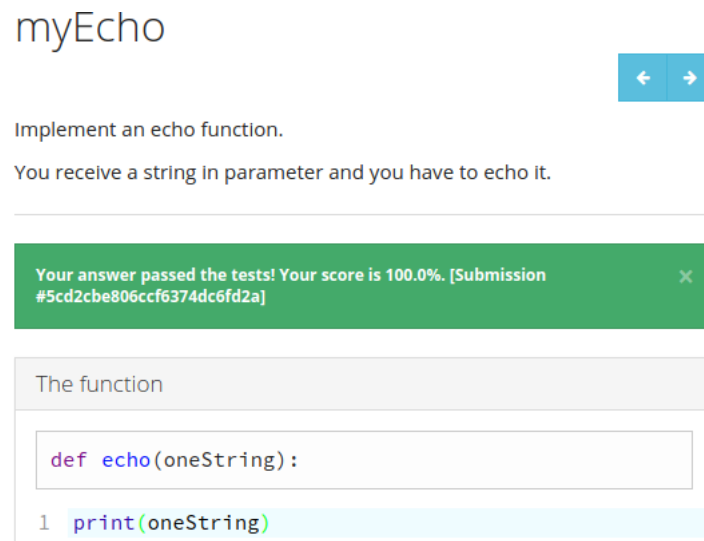
Figure 3.1: Architecture of Inginious



Example of an exercise

The figure 3.2 shows an example of a very simple exercise in Inginious allowing code evaluation.

Figure 3.2: Exercise of Ingenious



This exercise is made up of 3 different files:

- `task.yaml` : This file contains the statement of the exercise as well as the parameters. The file looks like this :

```
1 accessible: true  
2 author: Ponchau Simon  
3 context: |  
4     Implement an echo function.  
5  
6     You receive a string in parameter and you  
7     have to echo it.  
8  
9     environment: cpp  
10    evaluate: best  
11    groups: false  
12    input_random: '0'  
13    limits:  
14        memory: '100'  
15        time: '30'  
16        output: '2'  
17    name: 'myEcho'  
18    network_grading: false  
19    problems:  
20        test2:  
21            default: ''
```

```

22         type: code
23         name: The function
24         language: python
25         header: |-
26
27         .. code-block:: python
28
29             def echo(oneString):
30
31                 stored_submissions: 0
32                 submission_limit:
33                     amount: -1
34                     period: -1
35                 tags: {}
36                 weight: 1.0
37

```

This file contains, for example, the statement of the exercise in the context tag, the name of the drill in the name tag, the runtime environment (the container to be thrown) or the time and memory of this container in the limits tag.

- run file : This file is the script that is executed to run the tests. This file looks like this :

```

1      #!/bin/python3
2      import os
3      import json
4      import subprocess
5      import shlex
6      import sys
7
8      from inginius import feedback , input
9
10     #create the function code by combining the code of
the student and the template
11     input.parse_template("template.py")
12
13     #Execute the code to see if there is an error
14     myErr = subprocess.Popen( shlex.split("python3
template.py"), universal_newlines=True)
15     myErr.communicate()
16     resultat = myErr.returncode
17
18     #if 0 => there is no error
19     if resultat==0 :
20         feedback.set_global_result("success")
21     else :
22         feedback.set_global_result("failed")

```

```

23         feedback.set_global_feedback("Not the good
    answer.\n")
24     else :
25         feedback.set_global_result("failed")
26         feedback.set_global_feedback("- Your code does
    not compile.\n")
27

```

The function *input.parse_template* allows to insert the code of the student in the template provided for that purpose. The newly created file is then executed. The *feedback.set_global_result* function is used to send the status of the exercise (successful or unsuccessful) to the student. And the *feedback.set_global_feedback* function is used to convey a message to the student.

- *template.py* : The student code template looks like this :

```

1         import sys
2
3         def echo(oneString):
4             @    @test2@@
5
6             echo("test")
7

```

The student code will be inserted instead of `@@test2@@`. The number of tabs between the first two `@` will be the number of tabs that will be placed in front of each line of the student. `test2` must be the name found in the *task.yaml* file in the *problems* tag.

3.1.3 Dodona

Dodona works in the same way than Ingenious. It is also a platform for educational purposes but developed this time by the University of Ghent. It also allows an automatic correction of exercises.

Like Ingenious, it allows you to automatically correct the code written by students. The tests are written in python3 and run in a docker container. These similarities suggest that it is possible to make exercises compatible for both platforms.

Dodona has a Judge. The judge makes it possible to take inputs gathered in a file and to compare them with outputs in another file. If the inputs and outputs of the student program match, Dodona will consider the code to be good. A display

with the list of inputs/outputs of the file is given to the student. For each item in this list, the student knows if the test was successful or not.

To create an exercise, a teacher needs to create at least four files : a description file, a parameters file, an input file and an output file,.

- The description file contains the statement of the exercise. This file is encoded either in HTML or in Markdown. Several description files can exist for a single exercise, each encoded in a different language.
- There is a file including the parameters of the exercise (config.json) such as the name, the memory limit, the time limit, the level of the exercise...
- And finally the input/output files storing the expected input and output values for the student program. These files will be used by the judge.

Architecture of Dodona

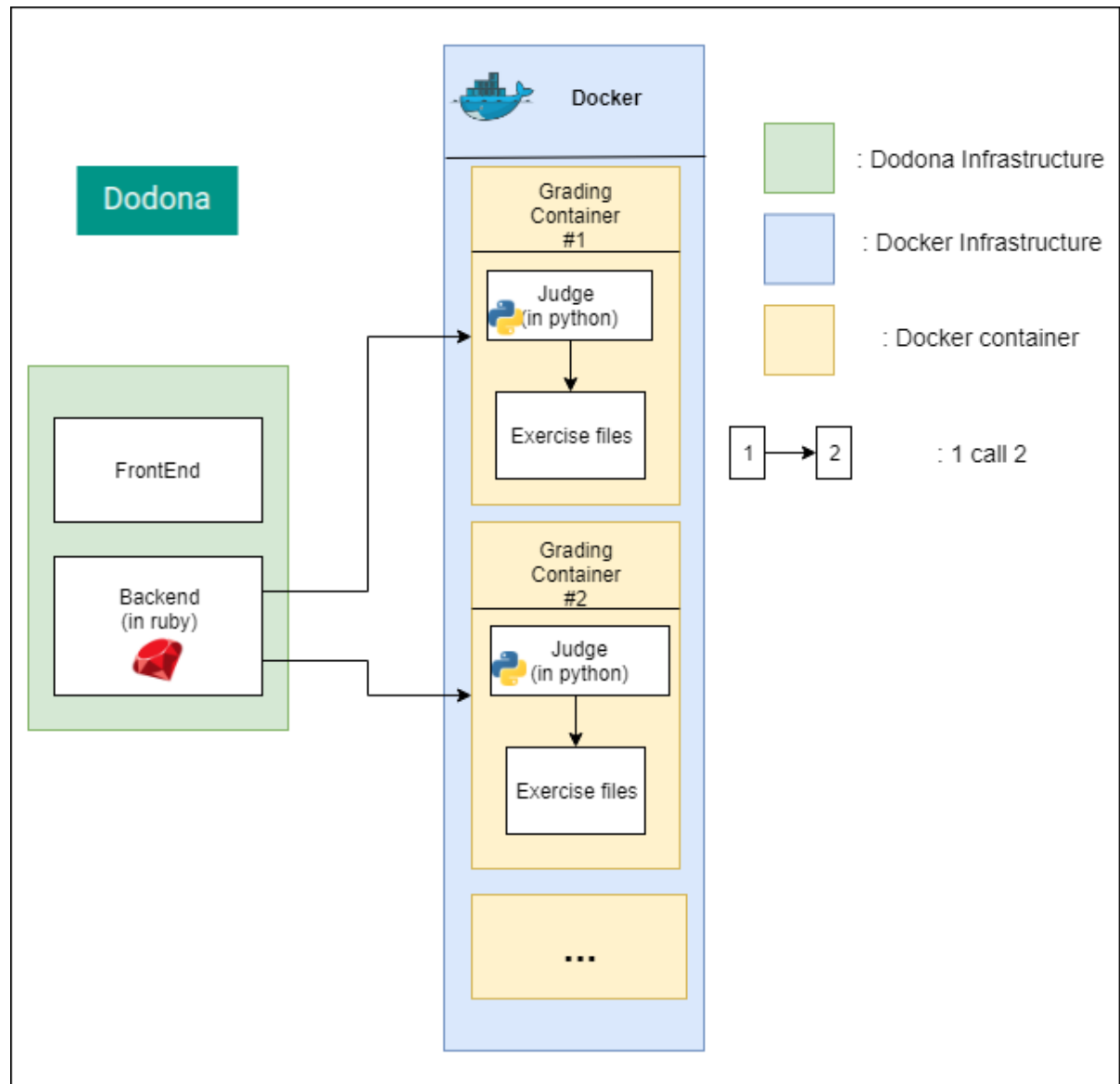
Diagram 3.3 outlines the architecture of Dodona. It consists of different parts.

First we have the frontend which allows to recuperate the code submitted by the student. This one and all the parameters necessary for the evaluation of the exercise are passed to the backend. It will launch the containers by providing the student code to evaluate the parameters associated with the exercise as well as the files related to the correction of the exercise.

A different container is launched for each exercise evaluation. In each of these, we have the judge which will evaluate the solution of the student, using the correction files that are also in the container.

In order to send the data from the containers to the backend, a standard output is used. It will transfer the data in Json format.

Figure 3.3: Architecture of Dodona



Example of an exercise

Figure 3.4 shows an example of a very simple exercise in Dodona allowing code evaluation.

Figure 3.4: Exercise of Dodona

The screenshot shows the Dodona exercise interface for an exercise titled "Echo". The interface is divided into two main sections. The top section, titled "Echo", contains the following text: "You receive one line of input and you simply have to repeat this input. You have to write a program which functions as an echo." Below this, there are two sub-sections: "Input" with the text "One line of text without any further restrictions." and "Output" with the text "The same text which can be read under input." The bottom section of the interface has a teal header with three tabs: "Hand in", "Submissions", and "Feedback". The "Feedback" tab is currently selected. Below the tabs, the following information is displayed: "User: Zeus Kronosson - zeus", "Exercise: Echo", "Handed in: less than a minute ago", "Status: Correct", and "Summary: correct answer". To the right of this information is a red circular icon with a white pencil. Below the feedback information is a table with three columns: "Correctness", "Code", and "Code". The table contains two rows of data, each with a green checkmark in the "Correctness" column. The first row shows the input "proscribable" and the output "proscribable". The second row shows the input "crystallisation" and the output "crystallisation".

Echo

You receive one line of input and you simply have to repeat this input. You have to write a program which functions as an echo.

Input

One line of text without any further restrictions.

Output

The same text which can be read under input.

Hand in Submissions Feedback

User: Zeus Kronosson - zeus
Exercise: Echo
Handed in: less than a minute ago
Status: Correct
Summary: correct answer

Correctness	Code	Code
✓	proscribable	proscribable
✓	crystallisation	crystallisation

The four minimum files required are the following :

- In the description folder, the files containing the statement of the exercise.
here is an example :

```
1 <p>You receive one line of input and you simply have to repeat  
   this input. You have to write a program which functions as an  
   echo.</p>  
2 <h3>Input</h3>  
3 <p>One line of text without any further restrictions.</p>
```

```

4 <h3>Output</h3>
5 <p>The same text which can be read under input.</p>
6

```

- config.json : Contains the configuration parameters of the exercise. Here is an example of this file :

```

1      {
2          "description": {
3              "difficulty": 0.5,
4              "names": {
5                  "en": "Echo",
6                  "nl": "Echo"
7              }
8          },
9          "evaluation": {
10             "pythia__judge": "output"
11         },
12         "pythia__id": "PROG0046",
13         "spoj__id": 9228,
14         "visibility": "open"
15     }
16

```

In this file we will find the name of the exercise, its level of difficulty or a tag to know whether it should be visible.

- In an evaluation file, we will have the input and output files (0.in, 0.out). Here is an example :
input file :

```

1      proscritable
2      crystallisation
3      overfruitful
4      sterilizable
5

```

output file :

```

1      proscritable
2      crystallisation
3      overfruitful
4      sterilizable
5
6      python input without prompt: true
7      block count: multi
8      input block size: 1
9      output block size: 1
10
11

```

3.1.4 Similarities and differences between Inginius and Dodona

In order to make the platforms compatible, it is important to analyze first the similarities and differences between them.

Similarities

- Both platforms have a frontend to manage exercise repositories as well as code collection and correction.
- Both platforms are coded in Python.
- Both platforms work with Docker containers. On each of them, when a test is launched, a new container is created and tests are performed inside it.
- The structure of both platforms is similar. As outlined by the diagram 3.5, we can compare a Dodona judge with an Inginius run File and the input/output files with unit tests.

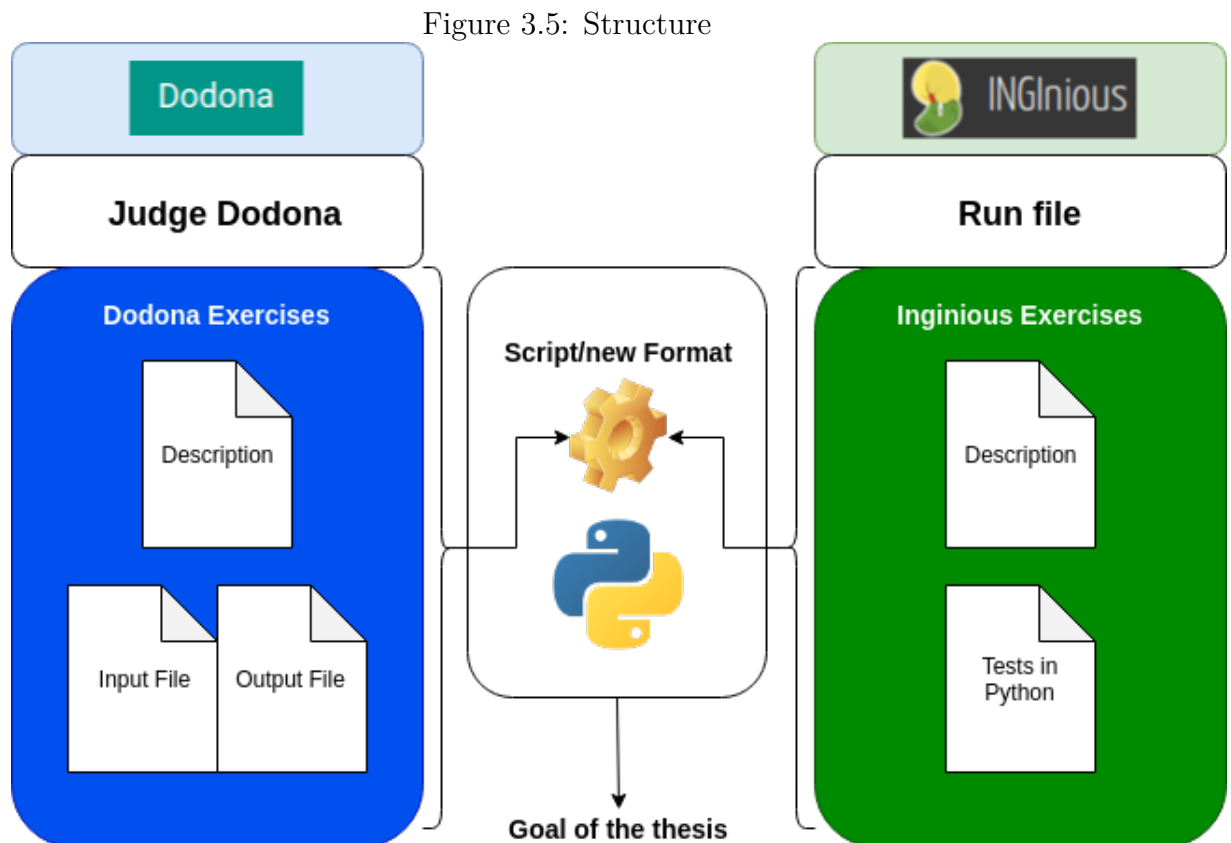
The Inginius run file with the other test script files corresponds to judge Dodona with the input/output files.

Differences

Inginious	Dodona
To return the results, the container returns a Json object stored in a variable. This one is then converted into hashmap.	To return the results, the container returns Json in the standard output.
It allows the teacher to write any code he wants in the run file. He can himself decide by himself the marks to be given to student based on the results obtained with his scripts.	Works only with an input/output system. These files are used in the Dodona judge which contains testing functions.
The advantage of Inginious is that the teacher can write any test cases he wants.	The advantage of Dodona is that a teacher is not obliged to write test cases but just input/output, which makes the use of the platform much easier. There is also the possibility to apply python functions on the inputs.
Allows to create several sub-exercises in a single exercise by introducing several fields to fill in.	Only one field is allowed.
The exercise descriptions are encoded in RST.	The exercise descriptions are encoded in HTML or in markdown.
Parameters are in yaml in the same file containing the description of the exercise.	Parameters are stored in Json.
For the feedbacks, we have only one block of text in which we can put whatever we want.	The feedback display can be divided into different blocks that will be displayed to the student.
Uses Centos as docker guest OS.	Uses Debian as docker guest OS.
An exercise description file is created by language.	Babel tools are used for language management. Thanks to this tool the translations of the character strings can be stored.

3.1.5 Comparison

Here is a diagram (3.5) that compares the infrastructure of Inginious and Dodona. We can see on the diagram that the structure of Inginious and Dodona is quite similar, even if in practice, there are some important differences.



3.2 Needs

As mentioned, there are many similarities between the two platforms. These similarities lead us to believe that it is possible to induce interoperability between both platforms at exercise level.

This interoperability has the advantage of allowing teachers/assistants to retrieve the exercises already developed on one platform in order to avoid having to implement them again, or just in order to have a wider set of exercises. This interoperability can potentially save a lot of time.

In order to go further and allow more possibilities, we will also think about the creation of a more generic format to store the exercises created by teachers/assistants in a way that is easily exportable to both platforms, as well as to other platforms, provided you create the conversion scripts. These other platforms should however be able to run Python and the language in which the exercise tests have been written (in our thesis, we focus mainly on interoperability of exercises for the Python language).

This new format is meant to be as easily editable and as intuitive as possible. The ultimate goal is to allow teachers/assistants to create and edit these exercises as quickly and easily as possible.

3.3 Conclusion

We studied more in-depth the functioning of Ingenious and Dodona. We also highlighted their infrastructure and their differences to understand the difficulties that we will face, and the issues we will have to deal with.

We also listed the needs of users.

In the next chapter, we will determine how we practically implemented the translation of the exercises from one platform to another.

Chapter 4

Solution

Both platforms have a number of differences. The challenge of this thesis will be to deal with these differences and to make Ingenious and Dodona compatible. Here are the differences that have an impact on the interoperability of the exercises on both platforms.

In order to make Ingenious's and Dodona's exercises compatible, we decided to retrieve the exercise files and load them on the other platform while trying to modify them as little as possible in order to increase the chances that they work in all cases.

In order to make things easier for the teachers, we have created scripts allowing to take an exercise folder from one system as input and to translate all these exercises at a time without the teacher's intervention on the other system format. The exercises thus translated will all be placed in the same output folder. This output folder can then be given to the corresponding platform, and will be directly interpreted correctly.

For some exercises however, due to differences in implementation on both platforms, we could not translate them for the other platform. This is the case for the Ingenious exercises where the student is asked to fill in several fields.

This chapter will be divided into two parts. The first will cover conversions of Dodona exercises into Ingenious Exercises, and the second will tackle conversions of Ingenious exercises into Dodona Exercises. In each of these two parts, we will talk about how we made the exercise statements, the exercise parameters, and the evaluation scripts compatible between the platforms.

4.1 Dodona to Inginious

In this part we will analyze how we made the Dodona exercises compatible with the Inginious platform.

4.1.1 Implementation

To execute an exercise on Inginious we need 3 files.

- the parameter YAML file : Contains the statement of the exercise and the parameters related to it.
- the run file : This file is responsible for running the tests and for returning the results.
- the student template file : This file contains code that will have to be added to the student code before tests can be started.

In Inginious, the most powerful part is the run file which can contain complex processes. Each exercise has its own run file. In Dodona the judge (which is common to all exercises), includes the processes.

For the simplest exercises of Dodona performing an assessment based on input/output files, our first idea was to create a script in Inginious to replace the Dodona judge. The script would read the input file, run the student code with this data and compare them with the outputs. This part worked well with simple exercises, but not necessarily with the more complex ones due to some parameters added to the Input/Output files. For example, a parameter that allows you to define a Python function to be applied to the Input files.

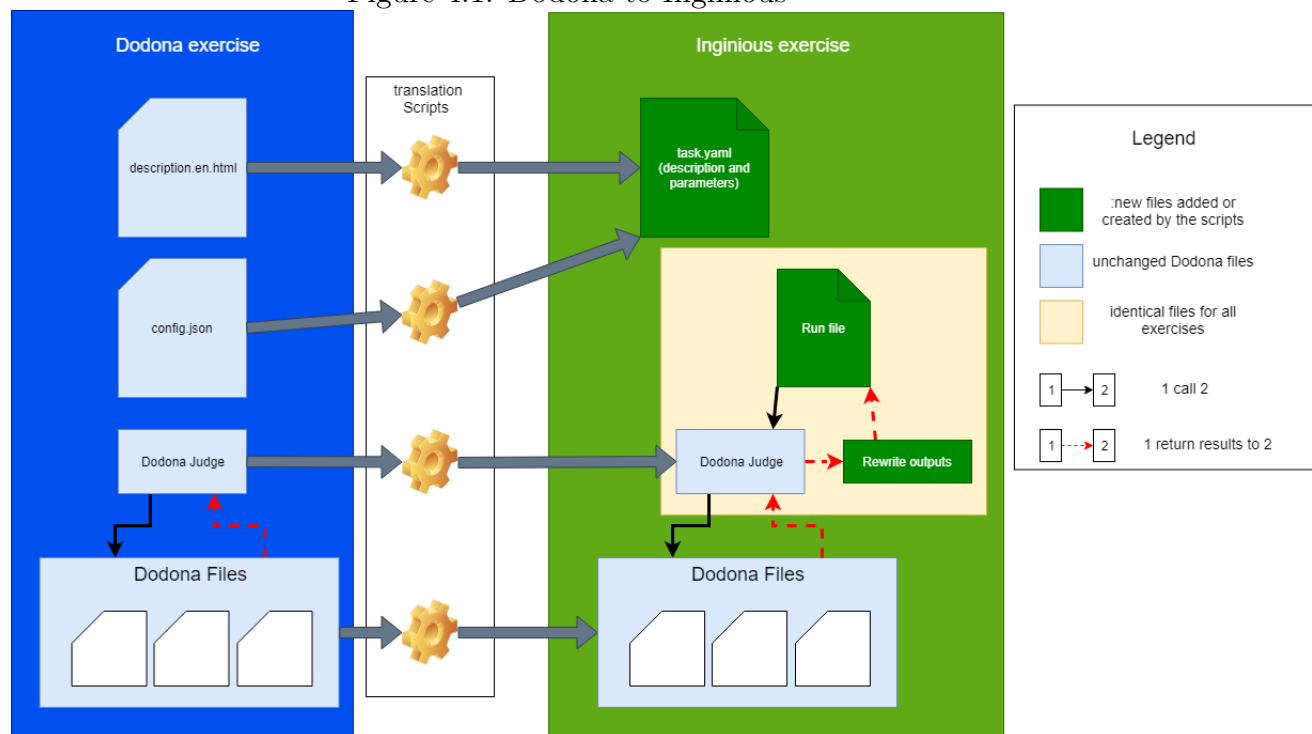
```
1
2 tab name: stressedSentence
3 <LANGUAGE code="nl">
4   <function from="stressedWord" to="woordNadruk" />
5   <function from="stressedSentence" to="zinNadruk" />
6 </LANGUAGE>
```

The names *woordNadruk* and *zinNadruk* are the function names defined in the *voorbereiding.py* file of the *preparation* folder.

This is why we applied a second method that allows the Inginious run file to launch the Dodona judge. The judge will evaluate the exercise and transform the judge's outputs into understandable returns by Inginious. So each exercise in Inginious would contain a copy of the Dodona judge.

The figure 4.1 illustrates the structure of the script which translates from Dodona to Inginiuous.

Figure 4.1: Dodona to Inginiuous



In order to carry out the translation of the exercises, we will have to dwell on several points. First, we have to reformat the statements of Dodona in an understandable format by Inginiuous.

We also retrieve the various parameters related to the exercise and add them to the parameters of Inginiuous exercises.

Once it is done, we have to allow the execution of Dodona evaluation scripts on Inginiuous. To achieve this goal, we take the Dodona judge which we introduce into the Inginiuous exercises.

We also have to adapt the judge's outputs to match those of Inginiuous.

Statement of exercises

The first problem to solve to go from Dodona to Inginiuous is the encoding of the exercise. In Dodona we have either HTML or Markdown. In Inginiuous it is encoded in RST and a YAML file. This file also contains the parameters of the exercise. We created a YAML template with the structure needed for Inginiuous to work.

In this template we put tags which will be replaced by the data recovered from Dodona.

One of these tags will contain the description of the exercise and must be in RST format. The `..raw: html` tag allows Inginius to interpret html. By inserting the Dodona HTML after this tag, it is correctly interpreted.

Here is the template file of the task.yaml file with the tags to replace.

```
1 accessible: true
2 author: Ponchau Simon
3 context: |+
4     .. raw:: html
5
6 <<<HTMLCODE>>>
7
8 environment: cpp
9 evaluate: best
10 groups: false
11 input_random: '0'
12 limits:
13     memory: <<<memoryLimit>>>
14     time: <<<timeLimit>>>
15     output: '2'
16 name: <<<name>>>
17 network_grading: false
18 problems:
19     studentCode:
20         default: ''
21         type: code
22         name: The function
23         language: python
24
25
26 stored_submissions: 0
27 submission_limit:
28     amount: -1
29     period: -1
30 tags: {}
31 weight: 1.0
```

For the management of Markdown, we chose, first, to transform it into HTML in order to insert it in the same way in the Yaml file. For the translation, we used the Python markdown tool.

Moreover, the statements of the exercises sometimes integrating $\text{\LaTeX}$, the insertion via this method led to some problems. Some characters are misinterpreted due to the addition of escape characters in the wrong place by the tools. To solve the problem, we adapted the the translation scripts to manipulate the HTML string of

the exercise description.

Another problem encountered is the management of external files, such as images. Indeed, the files and their path cannot be simply copied. Ingenuous files, to be accessible on the web page, must imperatively be in a folder named *public*. We had to move these files into this folder and modify the paths in the HTML code.

Parameters

In Ingenuous parameters of the exercise had to be in the same YAML file as the statement. We applied to them, the same principle. Tags have been placed in the template file on the right places and will be replaced during the transformation. We had to retrieve the parameters found in the files common to all exercises of a repository of Dodona exercises and those directly related to the exercise in question. To retrieve the information in the files, we used Python's Json library.

The *dirconfig.json* file is in the Dodona's exercise folder. This file contains the settings common to all exercises in the folder. The time and memory limits for each exercise are in this file. This file must be provided at the same time as the exercises to carry out the translation. However, care must be taken that the allocated memory limit does not exceed what Ingenuous can provide. Otherwise, the correction of the exercise can not be done.

Evaluation scripts

Once it was done, it was necessary to deal with the evaluation of the student code received. This part is more complex.

The first idea was to create an Ingenuous exercise to process an input file as well as an output file and compare both to return a result. The advantage of this method is that it is simple enough to set up. It works for the simple exercises of Dodona. For more complex exercises this technique is however not complete enough and does not allow to keep all the power of the judge of Dodona.

We therefore thought to integrate this judge in Ingenuous to keep all the subtleties of it.

The biggest difficulty in achieving this is that the output format on both sides is different.

In Dodona, Json writes directly on the standard output from the judge. Please refer to the example below to see what can be written. The group part will contain the returns line by line of the input/output files.

```
1 {
```

```

2     "accepted": true,
3     "description": "correct answer",
4     "groups": [
5         ...
6     ],
7     "maximal_score": 50.0,
8     "runtime_metrics": {
9         "wall_time": 0.0009954360211850144
10    },
11    "score": 50.0,
12    "status": "correct answer"
13 }

```

We developed functions in Inginious, to manage the returns. These can be called from the run file or from the teacher scripts. Please refer to the examples below :

```

1 if pass==true :
2     feedback.set_global_result("success")
3     feedback.set_global_feedback("You passed the tests \n")
4 else :
5     feedback.set_global_result("failed")
6     feedback.set_global_feedback("You failed the tests \n")

```

To overcome the problem related to different returns, we had to modify the judge so that the return format used by the judge corresponds to the format expected in Inginious. It can then be called by the run file.

In the judge, the file *pythia_judge.py* has been modified. This one is responsible for recovering all the entries and files necessary for the evaluation, to start the tests with all the data, then to return to the standard output the results encoded in Json format.

The first change was to point links to files and settings to the correct files. Once done, the tests can be started normally as before.

Here is the code to adapt the outputs :

```

1 #translate the global result
2 def set_global_result(result) :
3     if result=="AC" :
4         feedback.set_global_result("success")
5     elif result=="WA" :
6         feedback.set_global_result("failed")
7     elif result=="CE" :
8         feedback.set_global_result("miss")
9     elif result=="TLE" :
10        feedback.set_global_result("timeout")

```

```

11     elif result=='MLE' :
12         feedback.set_global_result("overflow")
13     else :
14         feedback.set_global_result("failed")
15
16     ...
17
18 # set the score
19 if (float(submission["maximal_score"])!=0.0):
20     feedback.set_grade((float(submission['score'])*100)/float(
21         submission["maximal_score"]))
22 else :
23     feedback.set_grade(0.0)
24
25 #display the global feedback
26 feedback.set_global_feedback(submission['description'])

```

4.1.2 How to use it?

We have created a bash script to retrieve all the exercises in a input folder and translate them into a output folder. The teacher is therefore responsible for placing the exercises to be translated into the right folder, and then retrieving them in the output folder. No further intervention is needed. If the translation fails, error messages are generated.

Normally all the exercises thus translated should be able to execute on Inginius without errors.

4.2 Ingenious to Dodona

In this section we will analyze how we made the Ingenious exercises compatible with the Dodona platform.

4.2.1 Implementation

In order to translate Ingenious exercises into Dodona exercises, the first idea was naturally to apply the same principle as for Dodona to Ingenious. Translate the exercise description and retrieve the parameters, modify the judge so that it can launch the run file of Ingenious, as well as create functions to translate return formats to match Dodona formats.

That is indeed what we did first. Unfortunately, this method does not work without modifying the exercise files and the run file (because of the problems explained below).

So that is what we did based on the run files of the exercises of the Python learning course. The exercises in this course are based on a unit test system and all have a common run file. This allows us to create a compatible Dodona run file for all these exercises. This method is the method currently implemented.

But it was not generic enough and did not work for all the exercises. To settle the problem, we considered launching the evaluation of the Ingenious exercises in a separate container launched from the Dodona judge. This container would then return the correct outputs according to the format needed for Dodona. We had no time to implement this.

Figure 4.2 represents the translation of the Ingenious exercises into Dodona exercises according to the currently implemented method (modify the judge of Dodona and the run file of Ingenious).

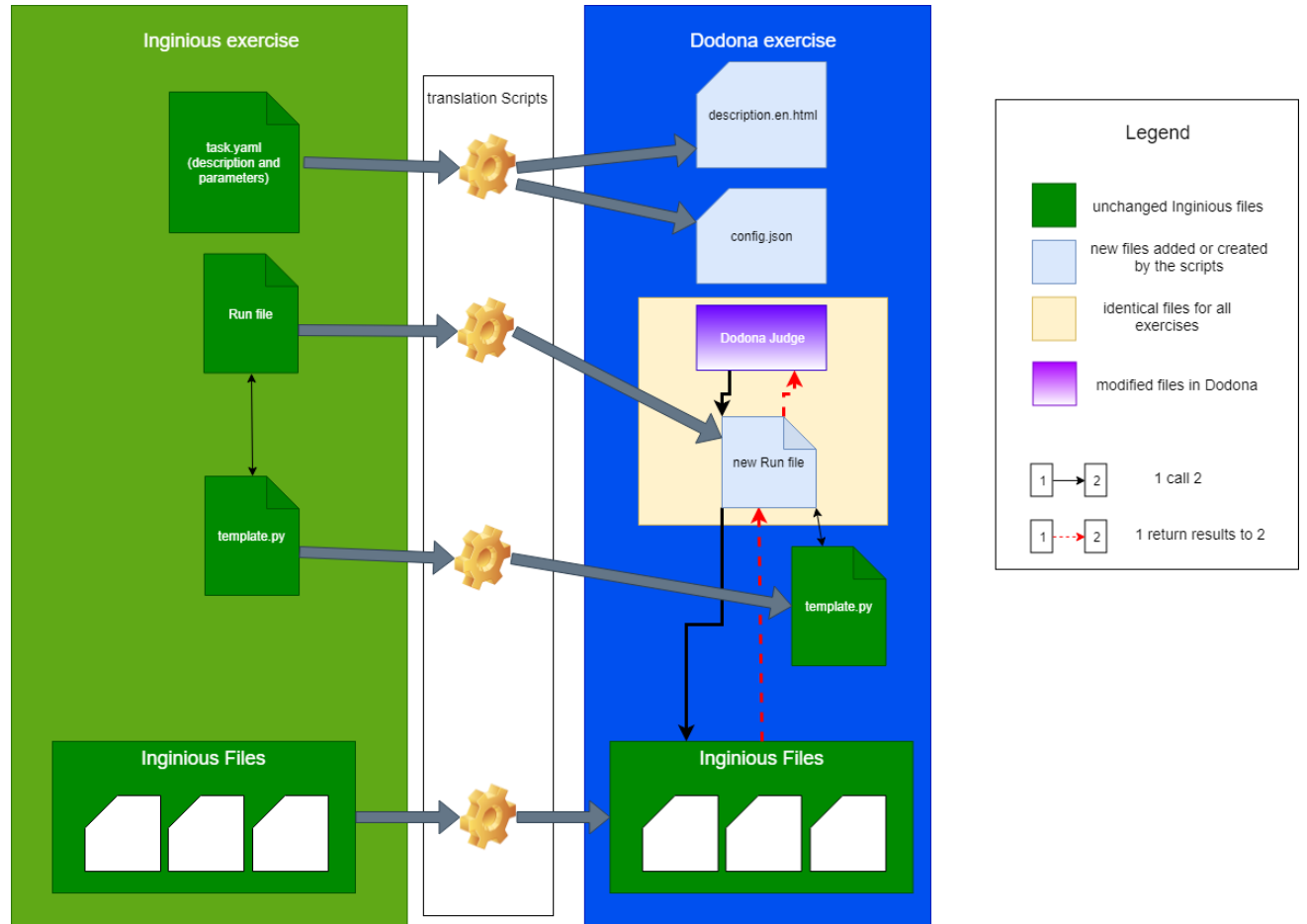
We can see the different parts that have been translated :

- The task.yaml file is translated to a description file of the exercises and a parameter file in Dodona.
- The special already modified run file is introduced in Dodona. This one will be called by a Dodona judge that we created to run these specific Ingenious exercises. This judge is then added in parallel with the other judges directly in the Dodona platform. A function present in this new Dodona judge makes it possible to translate the Ingenious exits in returns understandable by the Dodona platform.

- The `template.py` file is kept identical and is processed by the run file.
- Finally, the various Inginiuous scripts that are normally called by the run file on Inginiuous are copied in Dodona and will be called by the new run file.

This method, due to the fact that the run file is a manual modified file, can only work with the exercises of the Python learning course based on the compatible run file. The compatible run file was created based on Inginiuous Python exercises running on unit tests. So any exercise performed with this run file on Inginiuous should also work in Dodona. Inginiuous exercises also running with unit tests, but with a different run file, may work, but without any certainty. Exercises based on another evaluation system will not work. Thus, if the Inginiuous run file is different from the one processed, a message will appear informing the user that the exercise might not work on Dodona.

Figure 4.2: Inginiuous to Dodona



Statement of exercises

For the translation of the statements we first had to recover the RST format to translate it into HTML. We first extract the data from the YAML file using the Python's YAML library. When it was done, we were able to recover the part related to the description of the exercise. For the translation of the RST into HTML we used the `core.publish_parts` function of the docutils package. Once the HTML recovered, we wrote it in the right file, with the right name in the right place.

Parameters

The parameters of the exercise also present in the YAML file, are recovered at the same time than the statements. We have therefore directly rewritten them in the Json files of Dodona collecting the parameters. To do this, we used the same method than the one used for the other way round: we created a template in Json using the structure, inserting tags in the locations to be replaced. The scripts will replace these tags with the appropriate values retrieved from the YAML file.

This is how the settings template look like :

```
1 {  
2   {  
3     "description": {  
4       "difficulty": 0.5,  
5       "names": {  
6         "en": "###name##",  
7         "nl": "###name##"  
8       }  
9     },  
10    "evaluation": {  
11      "pythia_judge": "output"  
12    },  
13    "pythia_id": "PROG0046",  
14    "spoj_id": ##id##,  
15    "visibility": "open"  
16  }  
17 }
```

Evaluation scripts

Then comes the translation of the student's assessment. Our first idea was to launch the Inginious evaluation script files directly from the previously created judge and rewrite the functions present in the Inginious run file to return the result. Other functions such as Inginious template management or language management also needed to be rewritten.

- feedback.py
- input.py
- rst.py

This method, however, could not be applied as such, due to several reasons.

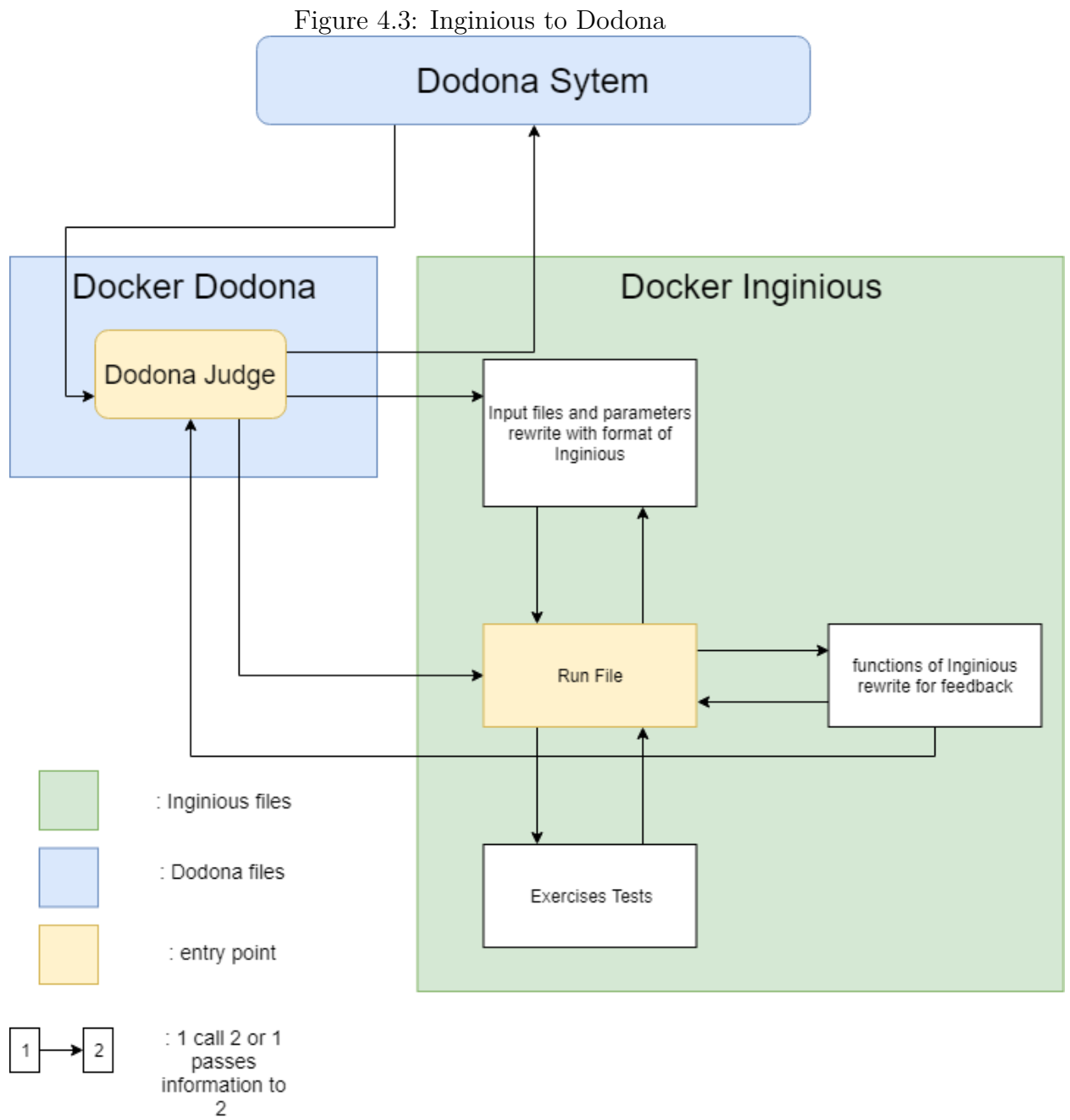
Problems encountered

- The first is that in Dodona the whole exercise is started in one and the same container. But the installation of the image of the Dodona container and that of Inginiuous are not the same and do not have the same packages installed. If we want a single image containing all the packages, we will have to modify one of them. But this is not the only problem. The OS on the two images are different (Centos and Debian), and the Python code has slightly different reactions depending on whether it is run on one container or the other. As the Python code needs to be modified to work, we can't import the Python files as is.
- Another problem is that Dodona uses the standard output for his returns. Some Inginiuous evaluation scripts use also the standard output to convey information. Both overlap and create errors.

For a lot of exercises on Inginiuous, a common run file had been created (for the Python learning course). The idea to fix all these exercises was to create a new run file that can be executed on the Dodona container and to include all the features specific to Inginiuous. We also created a script to replace all the places using standard output by writing and reading to a file. Although this method works well for Python class exercises, it does have a big disadvantage: it is not generic enough and will not work with exercises using another run file.

To overcome this problem, another solution has been considered. It consists of throwing a second container in parallel with the one that runs the judge. In the second one (running with Inginiuous image) we have the Inginiuous evaluation scripts. We also have to rewrite the Inginiuous functions so that the output of the Container corresponds to the Dodona outputs. This has not been implemented yet

Figure 4.3 illustrates this infrastructure.



4.2.2 How to use it?

As for Dodona to Inginius, the translation of the exercises is done via a bash script which takes an input folder containing all the exercises to be translated and which places them in an output folder once translated.

The teacher is therefore responsible for placing the exercises to translate into the right folder, and to recover them in the other folder once they are converted.

In this direction it is possible that the translation of the exercises does not work due to the differences of functionalities of the platforms. Indeed, the platform Dodona does not allow for the moment to manage exercises with several fields to fill in. To allow this type of exercises, the Dodona platform has to be adapted to to manage multiple field exercises. If this problem is encountered while translating the exercises, a message will be issued to warn the user that the translation can not be done.

4.3 Conclusion

We studied in this chapter how to practically implement the translation of the Inginius exercises to Dodona and vice versa. We listed the problems faced, and how to solve them. We equally explained how to use the translation tools and concluded that not all exercises can be translated.

In the next chapter, we will see how to check that the conversion scripts work properly.

Chapter 5

Validation

Once the translation scripts are finished, it was necessary to test their proper functioning. For this, a good method is to test a large number of exercises, choosing among them a maximum of limit cases that may not pass the tests.

5.1 Ingenious to Dodona

To perform the tests from Ingenious to Dodona, we took the LINFO1101-LEPL1401-ingenious Python course exercises with which the scripts are supposed to work. This folder is composed of 60 different exercises.

The conversion scripts translate all the exercises in the same folder. So, we simply put all the exercises of this course in the input folder. All the exercises could thus be translated at once.

When the scripts were done, we found that out of the 60 exercises, 4 returned indicating that the translation was impossible and 2 displayed a message warning that the exercise might not work. The first 4 impossible translation are due to the fact that the exercise requested several fields to submit. This functionality not present in Dodona. The 2 warnings are due to the fact that the run file for these exercises are different, which means that the exercise might not work.

For exercises flagged as impossible to translate by the script, we checked them by hand. As expected it were all exercises with several fields to fill in. Find below the simplified content of the problem section of the task.yaml file for one of these exercises:

```
1 problems:
2   q1:
3     header: |-
4       Implement the function “memoization(fun, n)” in Python.
5     type: code
```

```

6         language: python
7         name: Memoization implementation
8     q2:
9         header: Implement a function ‘‘fibonacci(n)’‘ in Python which
            will be used
10            by and use ‘‘memoization’‘.
11            name: Fibonacci implementation
12            type: code
13            language: python

```

For both exercises generating a warning, one of them had a completely different run file based on another evaluation method and could therefore not be translated to Dodona. The other warning was due to a typing error in all the run files (and therefore also the reference run file), which was corrected only in this exercise. Despite this warning, the exercise had therefore been translated correctly.

The next step was to test if the exercises thus generated, return the good outputs. So we tested 20 different generated drills, plus one generating a warning. For each exercise tested we:

- verified that the statement of the exercise was being displayed properly
- checked that the return was good (for a correct, wrong and crashed answer)

Thanks to the tests, we also noticed that the student code was directly executed in the judge to verify that there is no compilation error. This was a problem since in some cases the code is intended to be embedded in a template and therefore does not compile before. To solve the problem, we have to delete the check at this level in the modified Dodona judge, since a check is already done in Inginius after embedding in the template.

The diagram 5.1 shows us an exercise that generated this error :

Figure 5.1: Exercise with header

The method `average(list)` returns the arithmetic mean of the elements contained in `list`, except if `list` is empty in which case it should return `None`.

header

Provide the body of the following method:

```
def average(list):
```

Hand in

Submissions

Feedback

```
1 if len(list)!=0 :
2     average=0
3     counter=0
4     for i in list :
5         average+=i
6         counter+=1
7     average/=counter
8     return average
9 else :
10    return None
```

After the correction and regarding all the tests carried out, they performed well except for the one which had a completely different run file and posted a warning. For that one, Dodona can run the exercise, but considered it always as correct because there are no unit tests to perform.

5.2 Dodona to Inginius

For this part, we applied the same method. We converted a maximum of different exercises. We made sure to take as many different exercises as possible for the tests.

We tested the translation on 729 Dodona exercises (all those received). We had a problem with the names of the exercise records. When they contained a space, the scripts crashed.

Another problem related to language has also occurred. When English was not available, the switch to another language was not done correctly.

We have corrected these errors. After that, only one was not done correctly, due to the fact that the input file was not found in the exercise on Dodona.

For 20 exercises, we then verified that the evaluation was done correctly. In these 20 exercises, we selected both simple and complicated exercises. As mentioned before, for each of them we :

- verified that the statement of the exercise was being displayed properly
- checked that the return was good (for a correct, wrong and crashed answer)

During the tests we have discovered a problem related to links to the images present in the exercise statement of Dodona. In Inginius images must be in a specific folder and the links must point to this folder. The translation tool was adapted to manage this.

Out of the 20 exercises tested, we discovered for two of them, problems with exercises using f-strings. The Inginius Python version is too old to handle them. For example, this line does not work in Inginius:

```
1 print(f'period: {days} days, {hours} hours and {minutes} minutes')
```

After correcting these errors, no other problem was detected.

Chapter 6

Future Work

6.1 New Format

It is possible to think in a more global way. We can imagine a more generic format. This one would be a standard and would make compatible all the platforms deemed to be interested. To do this, we must think about creating a format with a well-defined structure and input-output. It should also be easy to implement an exercise, the files and the parameters should be easy to export to another format with a view to comply with the specifications of the platform.

Once this format is created, it will be possible to create repositories of exercises that will be common to all platforms and can be reused by each of them.

Each platform is different and has its own inputs/outputs for its exercises. So, they will need either to create a script to convert the input and output formats of this new format, or to modify the platform to natively manage these exercises.

The idea would be to create a format with all the parts that may be needed to evaluate an exercise. For an exercise to be complete, we need to fulfill three conditions :

- First of all we need to provide the files that describe the exercise to the user.
- We also need to provide one configuration file.
- We also need scripts to evaluate solutions. For this, we thought about creating a kind of black box with standardized inputs and outputs. Inside, we can place what we want as long as it returns a correct format. This method makes it possible to keep the evaluation scripts already available on the platforms. It is therefore necessary to create scripts allowing to convert the entries and

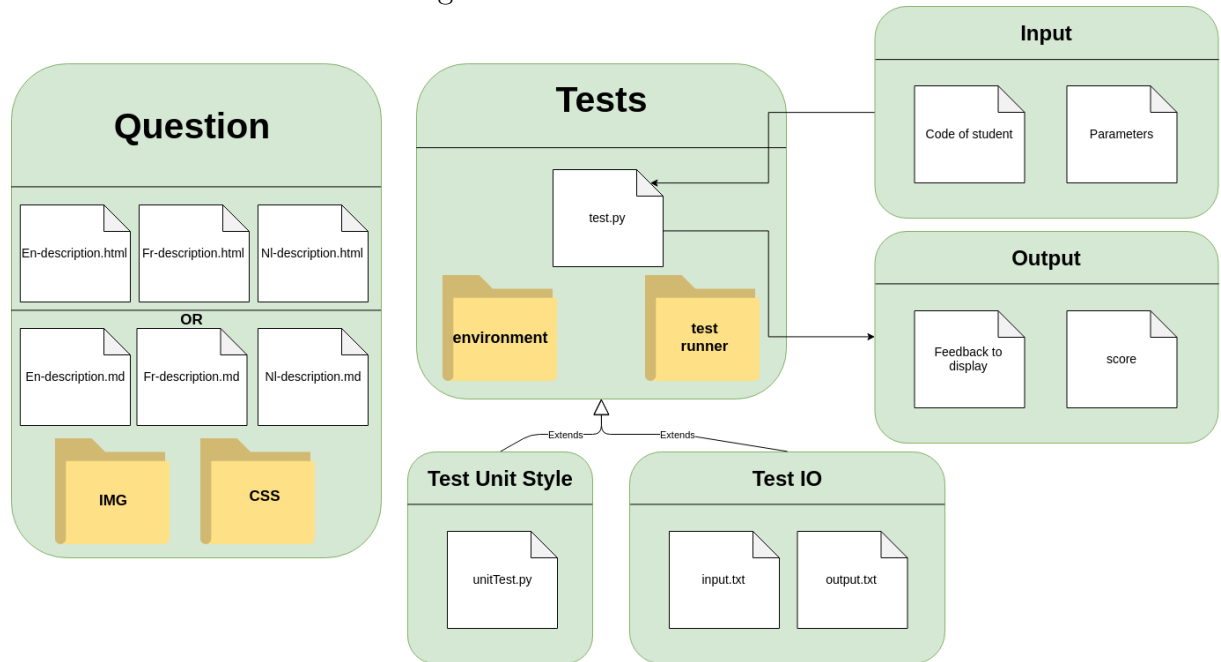
exits of the scripts so that they match with the format.

To make the creation of exercises easier, scripts to evaluate from inputs/output or unit tests can be proposed.

- If the person, who designed the exercise, wishes, he can also place the solution of the exercise.

As we can see on diagram 6.1, this new format will be divided into several folders, each containing files for different parts of the evaluation.

Figure 6.1: new format



This diagram is divided into several parts. Different parts are necessary to define an exercise:

- **Question**: This folder will contain all the files related to the statement provided to the student. We thus find there the different instructions of the exercise. Files can be encoded in either html or markdown. The designer of the exercise will also have the possibility to place files written in different languages. To do this, he will have to start the name of his file with the abbreviation of the language followed by '-'. For example `En-Description.html` for English, or `Fr-Description.html` for French. The designer will also be able to place all the files necessary for the description of the exercise in this folder. We can also find here images or CSS files.

- Tests : This folder will contain all the scripts needed to evaluate the exercises. It must contain a file (test.py) which will be the entry point of the tests. In this file there will be a function *evaluate* which will take in parameter the path to a file containing the parameters of the exercise as well as the code of the student.

The *evaluate* function will run scripts written by the teacher. It will then return the feedback in Json format, according to a structure equivalent to that present on Dodona.

In this test.py file, the *getScore ()* function will also be present. It will allow the teacher to write code to give the score he wants according to his own criteria.

It can also be interesting to provide two predefined structures for the evaluation of exercises. This would allow teachers to create exercises much faster, since the evaluation code is already (or largely) written.

- unit tests : Since the code for running the unit tests is already written, the teacher would only have to write the unit tests that will apply to the student code.
- Input/Output tests : To create an exercise correction based on this principle, the teacher would only have to provide the input file and the output file.

6.2 Dodona judge in Ingenious

Currently when translating Ingenious's exercises to Dodona, the Dodona judge is duplicated in each exercise. It can be interesting to add the judge directly in Ingenious, which would allow to have only one iteration of the judge and thus facilitate the modification while limiting the overlapping.

Another possible improvement would be to use Ingenious's HTML code reading function for exercise statements instead of translating this statement.

Chapter 7

Conclusion

The initial problem was to be able to make the Inginiuous and Dodona platforms compatible with each other. To this end, the exercises of one of the two platforms should be used on the other platform.

In order to understand the mechanisms of automatic exercise evaluation platforms, we first analyzed other existing platforms. The aim was to eventually try to find the most generic method possible. We also analyzed the PExIL format which is a coding exercise encoding standard.

We then analyzed how the Inginiuous and Dodona platforms work, as well as the structure of their exercises. In this way, we could spot the similarities and the differences between both platforms and could think about a way to make the exercises compatible between both platforms.

Once these steps were completed we were able to think about and offer a solution. The solution applied for the direction Inginiuous-Dodona was based on the modification of the files of the exercises and the change of the judge of Dodona. The solution for the direction Dodona-Inginiuous added the Dodona judge on the Inginiuous platform as well as the translation of the returns.

This method achieved much of the basic goal of interoperability of exercises between Inginiuous and Dodona for python exercises. Indeed, all the tested Dodona exercises can work in Inginiuous and Inginiuous's python lesson exercises, or exercises with the same run file, can also work on Dodona. The next step would be to recover the other exercises from Inginiuous to Dodona as well. We have to define another method, which is not yet implemented.

In order to make the exercises as generic as possible and to bearing in mind the need to reuse them on other platforms, we also defined a generic format. We did not implement it in the framework of this thesis.

Bibliography

- [1] Ricardo Queirós and José Paulo Leal. Pexil: Programming exercises interoperability language. 2011.
- [2] Ans. <https://www.ans-delft.nl>, Last consulted on 15/03/2019.
- [3] Mettl. <https://mettl.com/en/>, Last consulted on 15/03/2019.
- [4] Hackerrank. <https://www.hackerrank.com>, Last consulted on 15/03/2019.
- [5] Sololearn. <https://www.sololearn.com>, Last consulted on 15/03/2019.
- [6] Devskiller. <https://devskiller.com>, Last consulted on 15/03/2019.
- [7] Talview. <https://talview.com>, Last consulted on 15/03/2019.
- [8] Autolab. <http://www.autolabproject.com>, Last consulted on 15/03/2019.
- [9] Inginious' documentation. <https://inginius.readthedocs.io/en/latest/index.html>, Last consulted on 15/03/2019.
- [10] Upylab. <http://di.ulb.ac.be/verif/tmassart/Upylab.html>, Last consulted on 15/03/2019.
- [11] Kattis. <https://github.com/Kattis/problemtools>, Last consulted on 15/03/2019.
- [12] Domjudge. <https://www.domjudge.org/download>, Last consulted on 15/03/2019.
- [13] Submittity. <https://submittity.org>, Last consulted on 15/03/2019.
- [14] Learning object. http://edutechwiki.unige.ch/en/Learning_object, Last consulted on 16/03/2019.
- [15] Learning object standard. http://edutechwiki.unige.ch/en/Learning_object_standard, Last consulted on 16/03/2019.

- [16] Learning object metadata. https://fr.wikipedia.org/wiki/Learning_Object_Metadata,
Last consulted on 8/04/2019.
- [17] Learning technology standards
. https://www.imsglobal.org/metadata/mdv1p3/imsmd_bestv1p3.html1621610,
Last consulted on 8/04/2019.
- [18] Ims content packaging. http://edutechwiki.unige.ch/en/IMS_Content_Packaging,
Last consulted on 16/03/2019.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | www.uclouvain.be/epl