École polytechnique de Louvain (EPL)

Improving INGInious

Labeling mechanism to better identify difficulties of students

Dissertation presented by
Olivier Martin

for obtaining the Master's degree in
Computer Science

Supervisors
Olivier Bonaventure, Anthony Gégo

Readers
Olivier Bonaventure, Anthony Gégo, Olivier Goletti, Charles Pecheur, Chantal Poncin

Academic year 2017-2018

Abstract

This master thesis covers the implementation and the utilization of a labeling mechanism in the INGInious platform in order to better find difficulties of students. Technologies evolve every day in many domains and education is one of these domains. Courses begin to have online contents such as interactive syllabuses or exercises in the cloud that can create a lot of interesting data to process in order to improve courses and detect difficulties of students. However, research shows that *"there is a lack of tools combining both orchestration and evaluation in order to detect missing skills"* and that many difficulties encountered by students come from the lack of their prior knowledge. Being able to detect difficulties of students is essential in order to help them in a more targeted way and to create courses with an appropriate level of difficulty.

INGInious is an online automatic code assessment platform currently used at Université catholique de Louvain. This work aims to improve this platform to allow teachers to easily mark and record what students are doing and mistakes they make when doing exercises on this platform and then, compute statistics based on this. The implementation has already been used to reveal difficulties of students in several courses given at Université catholique de Louvain.

Acknowledgements

We would like to thank Olivier Bonaventure, our thesis supervisor who gave us valuable advices and comments throughout the year as well as the opportunity to tutor a course and work on INGInious tasks to enrich this thesis.

We thank Anthony Gégo who helped us with the implementation of our solution as well as his constructive feedback on our pull requests, which allowed us to provide better quality code and to improve our way of coding in python.

We also thank Olivier Goletti who provided us very useful documents for the realization of this thesis as well as his useful tips and comments.

Finally, we thank Chantal Poncin and Charles Pecheur who agreed to read this thesis.

Contents

1 Introduction **4**
1.1 State of the art . . . 4
1.1.1 Multiple choice questions . . . 4
1.1.2 Automatic assessment grading tool . . . 5
1.2 Objectives . . . 7

2 Submission and task labeling **9**
2.1 Motivations . . . 9
2.1.1 INGInious as a student . . . 9
2.1.2 INGInious as a task writer . . . 9
2.1.3 INGInious as a tutor . . . 10
2.2 Overview . . . 10
2.3 Types of tags . . . 11
2.3.1 Skill tag . . . 11
2.3.2 Misconception tag . . . 11
2.3.3 Category tag . . . 12
2.3.4 Auto tag . . . 12
2.4 API . . . 12
2.4.1 Skill and misconception tags . . . 13
2.4.2 Auto-tags . . . 13
2.5 Example . . . 13
2.6 Implementation: for users . . . 14
2.6.1 Task editor . . . 15
2.6.2 Task page . . . 17
2.6.3 Course page . . . 18
2.6.4 Submission viewer . . . 19
2.7 Implementation: for INGInious developers . . . 23
2.7.1 Tags . . . 23
2.7.2 Course . . . 23
2.7.3 Task . . . 24
2.7.4 Submissions . . . 26
2.7.5 Docker Agent . . . 26
2.7.6 INGInious API . . . 26
2.7.7 Submission viewer . . . 27
2.7.8 Statistics . . . 27
2.8 Test, evaluation and results . . . 29
2.8.1 LFSAB1401 exam . . . 29

2.8.2 LSINF1252 tasks 40
2.9 Guidelines 42
2.9.1 Links between tags 42
2.9.2 Hide useless skill tag 42
2.9.3 How to integrate tag in an existing test suite 42

3 Randomization of exercises **44**
3.1 Motivations 44
3.2 Overview 45
3.3 Examples 45
3.3.1 Hypotenuse 45
3.3.2 Calculate the area of a shape 47
3.4 Implementation 49
3.4.1 Task editor 49
3.4.2 Task page 49
3.4.3 Submission manager 50
3.5 Test, evaluation and results 50
3.6 Conclusion 50

4 Conclusion **53**
4.1 Links 53
4.1.1 INGInious 53
4.1.2 Tasks samples 53
4.1.3 LSFIN1252 tasks 54
4.1.4 LFSAB1401 exam 54

CHAPTER 1

Introduction

1.1 State of the art

In the world of education, providing clear, motivating, attracting and interesting lessons combined with clear and useful homeworks is not an easy job [Les16]. Having an adequate level of difficulty is important so that students can understand lessons. Too easy lessons might induce students not to come to class or not to read support material because lessons would be annoying. At the opposite, too difficult courses might also induce students not to come to class or not to read support material because students may be discouraged, less motivated or they may totally give up the course.

Professors create their courses and have to judge themselves the difficulties of the newly created course, trying to be as objective as possible about the level of students and the expected difficulty of the course. They can level the difficulty by observing other similar courses when these last exists, adapt the difficulty years after years based on the previous exam, by doing a survey about the course or even by simply using previous teaching experience. The initial knowledge of students has also to be taken into account when creating courses since many difficulties encountered by students come from the lack of their prior knowledge [TC16].

Another solution would be to analyze the behavior of students automatically when a course disposes of online material like some automatic multiple choice questions (MCQs) or tools to automatically grade assignments [HPb]. Thanks to these tools, we should be able to get information about answers of students and process them to reveal useful information that may be used to improve the content of the course [ALV16]. However there is a lack of tools that combining both orchestration and evaluation in order to detect missing skills [HPa]. We also need to be careful about plagiarism [Cig18] that occurs in these tools since plagiarism influences responses of students and therefore the statistics that flow from them.

1.1.1 Multiple choice questions

There exist many tools that allow professors to build online multiple choice questions such as Moodle. Used in combination with an exam, assignment or even non-mandatory exercise, MCQs can help professors to detect some weakness in the course material by observing answers of students. However, MCQs are limited due to its natural format: we can not see the reasoning of a student through MCQs since all the professor sees is the box checked by the student and not the intellectual reflexion behind. In some domains like computer sciences, MCQs assessments do not fit with learning expectations: students

must be able to create by themselves a solution to a problem and checking a box is far from implementing a correct algorithm from scratch. Since many tools already exist for creating, managing, and analyzing students through MCQs, we focus on the second solution: a tool to automatically grade assignments in a more generic and diversified way.

1.1.2 Automatic assessment grading tool

An alternative to MCQs is automated code assessment tools. There exist some interesting platforms like Stepik [aut18a] or INGInious [cdL18].

Stepik

Stepik [aut18b] is an educational engine and platform, focused on science, technology, engineering, and mathematics open lessons. Stepik allows for easy creation of interactive lessons with a variety of automated grading assignments. Stepik aims to apply data mining techniques to make education more efficient and to improve the way people learn and teach. Adaptive and personalized learning are one of the key priorities of the platform. The platform is suitable for many kind of e-learning activity, since there are about 20 different types of assignments, including programming, data analysis, peer review tools and Linux exercises with automated grading and real time feedback. Stepik platform is completely free for open educational content but private course are chargeable. Stepik also proposes some statistics on submissions and courses such as:

- For each assignment we can view the success rate and the number of students succeeding in completing the assignment.
- For each lesson included in a course there is information on the average time of completion, number of views and students who fully completed the lesson.
- In every course we can find all the statistics on course enrollment and the success rates of modules under the settings category 'Statistics'.
- In the Grade book we can see and also create an Excel sheet containing all the information about course enrollments and number of points obtained by the students.
- We can observe the course performance based on the success rate of students and comments on the course in the Instructor's Dashboard.

Stepik is a great tool allowing professors to create many types of small exercise however, for more ambitious exercise Stepik is quite limited due to its high level of abstractions to create very precise exercises. In other words, we can not do what we want. As a professor, this is important to dispose of the most generic tool to create every exercises we wants for students.

INGInious

INGInious [aut17] provides a simple and secure way to execute test and grade untrusted code written by students. It has been developed by the INGI[1] department at the Université

[1] https://uclouvain.be/fr/instituts-recherche/icteam/ingi

catholique de Louvain[2] as an open source[3] project to automatically grade programming assignments. INGInious can be used to create exercises for all language and is able to run any Linux programs thanks to Docker[4] containers.

How does INGInious work? INGInious is based on the concept of courses and tasks. A course is composed of several tasks and each task can contain one or more subquestions. Each task is composed of at least a `run` file, a special script responsible for providing feedback and grading code provided by users by compiling, executing and applying any form of checking and testing. This grading script is run securely inside a docker grading container, that completely jails the execution of the script and provide isolation between containers and host machine.

Architecture The architecture of INGInious is divided in three parts: the frontend, the backend, and the agents running docker containers. This division in multiple parts permits to run each parts on different machines and replicate them if needed for elasticity and scalability.

The general operation of INGInious is described in figure 1.1. As we can see, when a user submit something to be graded, the backend receives the input of the students and sends it to an agent. Then the agent is then responsible to start a grading container and to execute the `run` script. This script associated with the INGInious feedback API[5] generates a submission containing several fields such as the result, the grade, a feedback to help the student, etc. This submission is then returned to the frontend by passing back through the agent and the backend. Once the frontend receives the results of the job, it saves these results in the database and provide feedback to the user so that they can correct and improve his code.

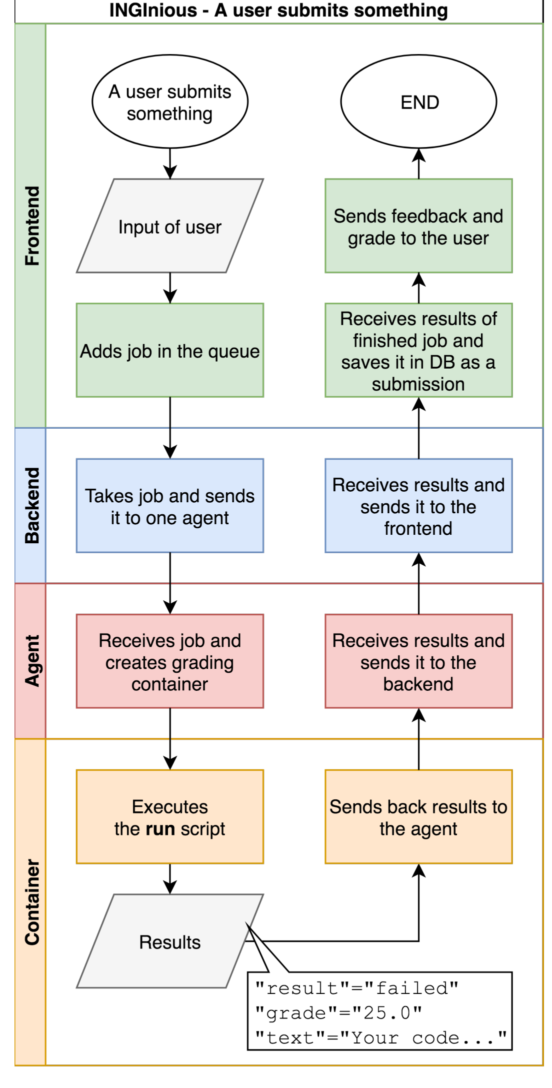
Roles INGInious is designed to have different user profiles with different permissions on the platform: student, tutor, administrator, and super admin. Each of these roles permits to have a good management and access rights on the platform. Students are the base account and can submit for tasks of courses for which they are registered. Tutors are assigned to a course and can view submissions of students registered to the same course. Administrators are also assigned to a course and can edit tasks, permissions and settings of this course. These permissions are hierarchical: administrators also dispose of tutor rights and tutors dispose also of student rights. We regroup tutors, administrators and super admin as the *staff* denomination for later usages in this document.

Statistics on students INGInious proposes some simple statistics in the frontend on students and their submissions:

- The number of task tried, the number of task done and the current grade for each student.
- The number of view, the number of attempted and the number succeeded attempts for each tasks.

[2] http://www.uclouvain.be
[3] GNU Affero General Public License v3.0
[4] https://www.docker.com
[5] https://inginious.readthedocs.io/en/latest/teacher_doc/run_file.html#feedback-commands

Figure 1.1: Operation of INGInious



- Some similar statistics on classrooms.

As we can see INGInious already proposes some statistics on students, but in a too weak manner to be effective in improving content of courses.

1.2 Objectives

INGInious is a nice tool to grade code made by students and to provide them feedback. However, feedback from INGInious to professors about what students do is nearly unexisting. For this master thesis we developed new features and tools in the INGInious platform:

- to allow task creators to tag what students do in tasks of INGInious.
- to provide feedback about students to tutors and professors based on tags.

- to query and filter submissions based on tags and other criteria.
- to better prevent plagiarism.

The rest of this master thesis is divided in two main chapters: chapter *Submission and task labeling*, explains, details and develops the first three objectives cited above while chapter *Randomization of exercises* concentrates on the last objective of plagiarism.

CHAPTER 2

Submission and task labeling

In this chapter, we explain our solution and implementation about tags that allow the labeling of tasks and submissions to provide feedback to professors and tutors.

2.1 Motivations

To better aim and highlight possible improvements to the INGInious platform, we used it for several months and we assumed three different roles in real situations: a student, a task creator, and a tutor. In the three following sections, we explain for each role concrete use cases we lived and problems we found.

2.1.1 INGInious as a student

Some courses contain many tasks. For example, there are more or less 200 tasks for the LFSAB1401 [ea18a] Java course given at Université catholique de Louvain. Consequently, searching for particular tasks is a laborious process. Moreover, some students do not want to complete all of these tasks and would prefer to focus on some specific tasks depending on their difficulties or content.

A mechanism that would allow students to filter the task lists based on criteria would be great. For example, a student who wants to train himself with arrays may only want to have the lists of tasks containing exercises on arrays.

2.1.2 INGInious as a task writer

We worked on the refactorization of Java tasks for the LFSAB1401 course. We had to rewrite tasks in a more maintainable way, correct errors in existing tasks and improve feedback shown to students. During this process, we had to test our new implementation of tasks with old submissions made by students to ensure that the compatibility and the feedback given by our new tasks was still correct. To perform this process, we were looking at the old submissions lists, we picked randomly some submissions, we replayed them, and we ensured that the feedback of new re-factored tasks was consistent with old submissions.

Unfortunately, during this process, many randomly chosen submissions did not compile. This is bad because we were looking for submissions that compile but with a failed status because we had to test feedback given by new tasks. There is no interesting feedback when the submission does not compile (error of the java compilator) and no feedback when the submission succeed.

We then thought of a mechanism that would allow us to tag submissions that do not compile or more generally, a mechanism to tag and filter submissions based on several criteria. Such mechanism would have saved us time.

2.1.3 INGInious as a tutor

Each week, when we tutored the LFSAB1401 course, we had to correct manually submissions made by students for the "mission of the week". Then, we had to prepare a manual feedback for students based on what they implemented for the mission. Unfortunately, this was a laborious process because there were many submissions with strange and complicated code from students (first-year students do not always make very clear code) and we had an average of 1 hour to review about 10 submissions. We then thought of a mechanism that would allow tutors to quickly have interesting statistics on what students are doing and difficulties they have encountered. For example, let us consider a task where students have to find and return the maximum integer of an array. As a tutor, it would be very interesting to know:

- how many students passed the test where all elements are positive?
- how many students passed the test where there are some negative element in the array?
- how many students printed the result instead of returning it?
- how many students made an `ArrayIndexOutOfBoundsException`?
- how many submissions did not compile?

With such mechanism, tutors could be able to target easily difficulties encountered by their students and then focus on these difficulties with their students. Moreover, when consulting students submissions on INGInious, there is only a huge list of submissions containing names of authors, id, date, and results of submissions. As a tutor, this is not very interesting. A mechanism that would allow tutors to consult what they want (only submissions where the grade is not 100%, only submissions that compile, etc) would be great. A concrete example was an assistant that wanted to consult submission for his classroom, for several tasks related to a white exam where submissions shown are the evaluation submissions only (the best one submission for each student) with all submissions are sorted by task id.

2.2 Overview

Our idea to provide feedback on students to professors is to register more information in submissions when grading and evaluating the task. These additional information is what we call *tags* or labels. So, we can additionally save facts in the submission like the fact that the code does not compile, that the student passed a specific test case, that the student made a certain error or more generally, nearly every facts we want. These tags can then be processed to generate statistics and highlight facilities and difficulties of students.

To implement this mechanism, we created new feedback API functions that can be used in the `run` script to allow tasks authors to save information they want in the submission when grading the code of the student as shown in green in figure 2.1. We also modified

some graphical interfaces of INGInious to take these tags into account. Main changes were at page to edit tasks and at the page to consult submissions. For the task editor, we added the possibility to define tags for each task so that tags can have a visual behavior in GUI of INGInious. For the consultation of submissions, we added statistical tables based on tags for tutors and professors and we added filtering options for these submissions.

Figure 2.1: INGInious operation with tag

2.3 Types of tags

We have defined four kinds of tags. Each tag has a different purpose, behavior, visibility and graphical representation.

2.3.1 Skill tag

Skill tag should be used when we want to label the success or failure of a concept or a typical test case. Here some examples of useful concepts that may be labeled.

Examples

- `Return type`: when a student has to write a function in Java, he has to choose a return type for his function (e.g. void, int, float, double, ...). Students often made errors on the return type and it may be interesting to tag this.
- `Open` and `Close`: in tasks using files, students have to open the file and then close it. Students often forget to close the file.

More generally, each test case of a test suite can be associated with a tag so that these information are marked in the submission and can be used to compute statistics.

Display By default, skill tags are represented by a blue badge. When the tag is validated, the badge becomes green. A tooltip shows the description of the tag.

2.3.2 Misconception tag

Misconception tag can be used to mark that the student made an error.

Examples We can tag all common errors depending on the programming language used. In Java we can tag all tasks with `NullPointerException` if the student made an invalid operation on a null object or `ArithmeticException` if the student made a division by zero. In C, we may use `Segmentation Fault` or `Double free` tags. We can also use more general tags like `Not Compile` if the code does not compile or `Timeout` if the code time out (probably due to an infinite loop).

Display By default, misconception tags are hidden. When the tag is obtained, the badge appears in a red style. A tooltip shows the description of the tag.

2.3.3 Category tag

Category tag have an organizational role and can be used to add some category on tasks to better manage a huge set of tasks.

Examples We can tag tasks with some difficulty tags (e.g. `Easy`, `Hard`), some time line depending on the course organization (e.g. `Week 1`, `Week 2`), the abstract topic of a task (e.g. `File`, `Arrays`, `Linked List`) or something relative to the organizational aspect of a course (e.g. `Exam`, `Mandatory`, `Optional` or `Bonus points`).

Display The tag name is just listed in some menu.

2.3.4 Auto tag

The last tag we define is the *auto tag*. They are quite different from other tags presented above and should not be used often. However, they are useful if we want to tag runtime information that can not be tagged with skill and misconception tag since skill and misconception tags can only mark boolean information. They can also be used for debugging.

Examples We may want to tag:

- the execution time of the code of the student (useful in algorithmic exercises).
- the number of lines of code (if the student provides a correct solution that is 30 lines of code when there is a way to do it in 3 lines, the student probably has some problems of comprehension).
- some variables used randomly in the test suite.
- error messages for debugging.

Display Auto tags are only visible by staff members. When an auto tag is present, it appears as a dark blue badge. If the name of the tag is too long, the exceeded part is visible in a tooltip.

2.4 API

We introduced two new API functions in the feedback API of INGInious.

2.4.1 Skill and misconception tags

The first function, `feedback.set_tag` is used when we want to tag a submission. If we call the function multiples times for the same tag, only the value of the last call is saved.

```
def set_tag(tag, value):
    """
    Set the tag 'tag' to the value True or False.
    :param tag: the id of the tag. Can not starts with '*auto-tag-'
    :param value: should be a boolean
    """
```

The same function is available in bash with `set-tag <tag_id> <False|True>`. Notice that there is a difference between setting the tag to `False` and not use any call to the function. In the first case, the tag is registered in the submission with a value equals to `False`. In the second case, nothing is registered in the submissions. We should take care of it if we plan to manipulate CSV of submissions with tags.

2.4.2 Auto-tags

The second function, `feedback.tag` is used to save an auto-tag to the submission.

```
def tag(value):
    """
    Add an auto-tag with generated id.
    :param value: everything compliant with the str() function
    """
```

The same function is available in bash with `tag <value>`.

2.5 Example

To better understand how tags work, their purpose, and how to use them as a professor or a tasks creator, we developed a sample task using tags.

Let us consider a task where students have to implement a function that computes the surface of a circle (figure 2.2). It may be interesting to know if the code of the student compiles and if the student correctly defined the function (return type, name, and parameters). To be able to extract these information from submissions of students, we need to use and define tags:

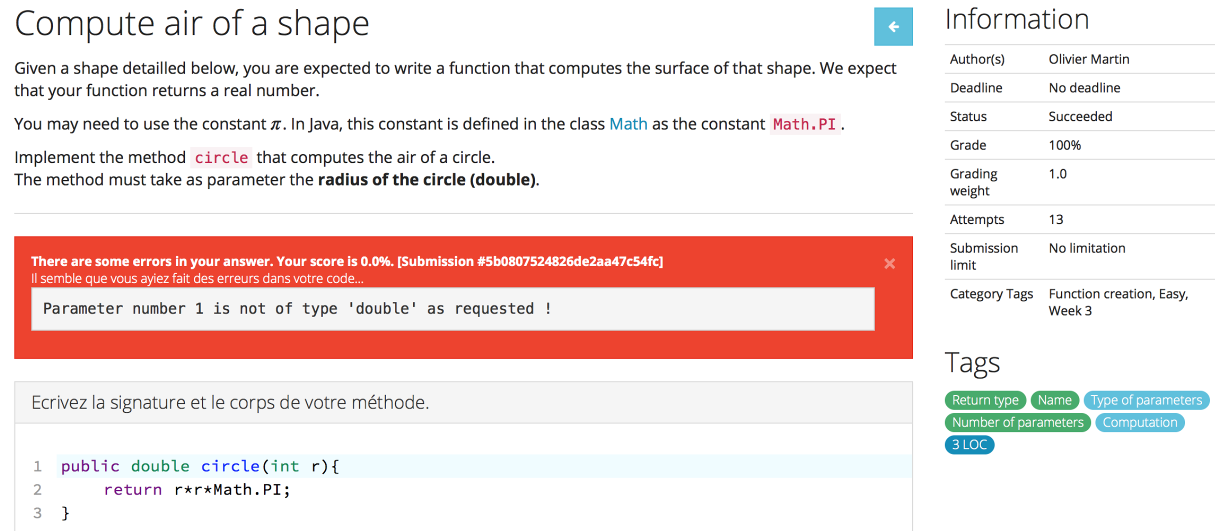
- Five **skill** tags to label that the function has the correct return type, the correct name, the correct parameters type, the correct number of parameters, and that the computation of the surface is correct.
- A **misconception** tag to label if the code does not compile.
- Some **Categorical** tags to label the task is about function creation and that the task is an easy one.
- An **auto-tag** to register the number of lines of code.

After, we need to define when and how to tag a submission. This has to be done in the `run` script of the task using our new API described in previous section. Students can now accomplish the task and tags will be recorded in their submissions. We can see on figure 2.2 that the student:

- acquired the tag `Return type` since he correctly defined the return type of his function (`double`).
- acquired the tag `Name` since he correctly defined the name of his function (`circle`).
- acquired the tag `Number of parameters` since his function takes as input one parameter.
- did not acquire the tag `Type of parameters` since his function takes as input an `integer` while the function should take a `double`.
- did not acquire the tag `Computation`. In fact his computation is correct but the test suite requires the function to be correctly defined in order to evaluate the calculation.
- acquired the tag `3 LOC` because the code is three lines.

Later, by visiting our new submission viewer page, we can consult statistics (figure 2.3) about recorded tags like the percentage of submissions that do not compile or the percentage of submissions that did not correctly define the return type of the function.

Figure 2.2: Tags used in a task



The following sections will explain in depth how to use tags, their corresponding API, changes in INGInious from the user point of view, changes from the developer point of view and finally, evaluations, tests, and results obtained by using tags in real cases.

2.6 Implementation: for users

In this section, we focus on the practical aspect of tags from the professors and tasks writers point of view. The integration of tags adds new functionalities in some places of INGInious. Each following subsection explains a different location.

Figure 2.3: Stats of tags used in a task

Tag statistics

Tag	All submissions	Best submissions
Return type	69.2 %	100.0 %
Name	76.9 %	100.0 %
Type of parameters	53.8 %	100.0 %
Number of parameters	84.6 %	100.0 %
Computation	38.4 %	100.0 %
Not compile	15.3 %	0.0 %

2.6.1 Task editor

We added a graphical tag editor on the task edition page. The interface allows tasks authors to create, modify and remove tags[1] for a specific task. This tag editor is accessible with a new tab called simply *tags* accessible via `Edit task > Tags`. The interface proposes two tabs: one for tag creation, modification and deletion and the second for importing tags already defined in other tasks. We can find on figures 2.4 and 2.5 the interface for the tag edition and tag importation based on a task of the LSINF1252 course [ea18b].

Tag editor

The purpose of this interface is to create, edit and remove tags from the task. By default, the interface is empty. To create a tag we need to click the *"New tag"* button. Clicking on it adds a new empty row in the interface. Each tag is composed of five fields: an id, a name, a description, a visibility checkbox and a type.

- **Id** the goal of the id is to give a unique identifier to a tag independently of its name. The id allows us to link the tag with the API used in the run file of the task. Since only the id value is used in the API, we can change other fields of a tag without changing scripts of the task. The id of a tag is never displayed in any interface, so its value does not matter. The tag id must be unique among all other tags defined for the task and can contain only letters, numbers, underscores and dashes. The id field is mandatory and can not be empty. When saving, if the id field is not defined, the tag will not be saved in the task. Therefore, to remove an existing tag we can simply remove the id field.

- **Name** we have to give a name to a tag with this field. The name will be displayed in interfaces using tags in the form of a badge. The field can contain everything but we recommend to only use letters, digits, and spaces. We should keep the name as short as possible (e.g. max 20 characters) in order to avoid graphical glitches since the tag name is displayed in other interfaces of INGInious. The name field is mandatory and can not be empty. Again, when saving, if the name field is not defined, the tag will not be saved in the task. Therefore, to remove an existing tag we can simply remove the name field.

[1] skill, misconception, and categorical tags

Figure 2.4: Interface for editing tags (example)

Basic settings | Container setup | Subproblems | Tags | Task files

Tag editor | Import tags

Id	Name	Description	Show to students	Type
close	Close	This tag validate that your close are	☑	Common
open	Open	This tag validate you are able to open	☑	Common
sum	Sum	The student has correctly computed	☐	Common
memory	Memory Exceeded		☑	Antitag
not_compile	Not compile	You code does not compile.	☑	Antitag
sigsegv	Segmentation Fault		☑	Antitag
timeout	Timeout	Your code exceeds the maximum	☑	Antitag
failure_handling	Failure handling	You do not manage failures properly	☑	Antitag
banned_funcs	Banned functions	You use some banned functions.	☑	Antitag
double_free	Double free	Your code produced a double free.	☑	Antitag
sigfpe	Floating Point Exception		☑	Antitag
	S5		☑	Organisation
	File		☑	Organisation

+ New tag

- **Description** we can give an optional description to the tag with this field. The description appears in a tooltip (a pop-up with the description appears when we hover the tag with the mouse). The description can contain the tag purpose, a hint or some advice for students. We recommend keeping the description as short as possible (e.g. max 300 characters).
- **Show to students** This check-box allows students to see the tag or not in task pages and course pages. Tutors and administrators always see all tags. We may want to disable the tag visibility to students if the tag contains sensitive information, if the tag does not help students or useless for them.
- **Type** the select menu allows to choose the tag type among *skill*, *misconception* and *category* types.

Import tags

If several tasks use the same tag, we recommend using the import tag feature. This avoids redefining several times the same tags across many tasks. This tab shows the complete tag list of all tasks of the course sorted by type and then by name. Clicking on the bottom arrow import the corresponding tag to the current task. After clicking, if we come back to

Figure 2.5: Interface for importing tags (example)

Basic settings | Container setup | Subproblems | Tags | Task files

Tag editor | Import tags

Import	Id	Name	Description	Show to students	Type
↓	close	Close	This tag validate that your close are correctly performed.	True	Common
↓	copy	Copy	This tag validate you are able to copy a file.	True	Common
↓	open	Open	This tag validate you are able to open a file and handle errors with open().	True	Common
↓	permission	Permission	This tag validate that you are able to copy permission of a file.	True	Common
↓	sum	Sum	The student has correctly computed and returned the sum.	False	Common
↓	banned_funcs	Banned functions	You use some banned functions.	True	Antitag
↓	double_free	Double free	Your code produced a double free.	True	Antitag
↓	failure_handling	Failure handling	You do not manage failures properly during read operations.	True	Antitag
↓	sigfpe	Floating Point Exception		True	Antitag
↓	malloc_fail_memory_size	Wrong malloc memory size	The allocated memory has not the correct size.	False	Antitag
↓		Command line		True	Organisation
↓		Data structures	Task dealing with data structures and linked list	True	Organisation
↓		File		True	Organisation
↓		Malloc	Usage of malloc()	True	Organisation
↓		Pointer	Task about pointers and their use	True	Organisation
↓		S1	Première semaine	True	Organisation
↓		S2		True	Organisation

the *Tag editor* tab, we should see the imported tags at the bottom of the list. Notice that two tags are considered equal if they share the same name and same id. The description is not taken into account when building the list displayed in the *Import tags* tab.

2.6.2 Task page

The task page is the page where students can write their code and submit it. We added a new `Tags` section (figure 2.6) for the display of skill tags, misconception tags and auto-tags. We also added the section `Information > Organizational Tags` to list categorical tags.

Students can only see tags with visibility status enabled. Auto-tags are never displayed to students. For the staff, all tags will always be displayed.

We also added a badge on each submission present in the "Submission history" table (figure 2.7). The badge indicates the number of validated skills tags. The default color of the badge is blue, but when the student has acquired all tags he can see (the non-visible skills tags are not taken into account), the badge becomes green. If the student has acquired at least one tag but not all tags, a tooltip indicates names of validated tags. This should help students if they need to consult an old submission they made. For example, a student acquired a tag in the past but now the tag is not validated anymore. The student can retrieve this submission with the help of tags.

Figure 2.6

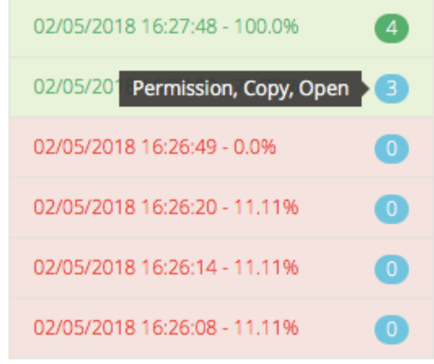
Information

Author(s)	Olivier Martin
Deadline	No deadline
Status	Succeeded
Grade	100%
Grading weight	1.0
Attempts	62
Submission limit	No limitation
Organisational Tags	File, S5

Tags

Close Permission Copy Open
Failure handling

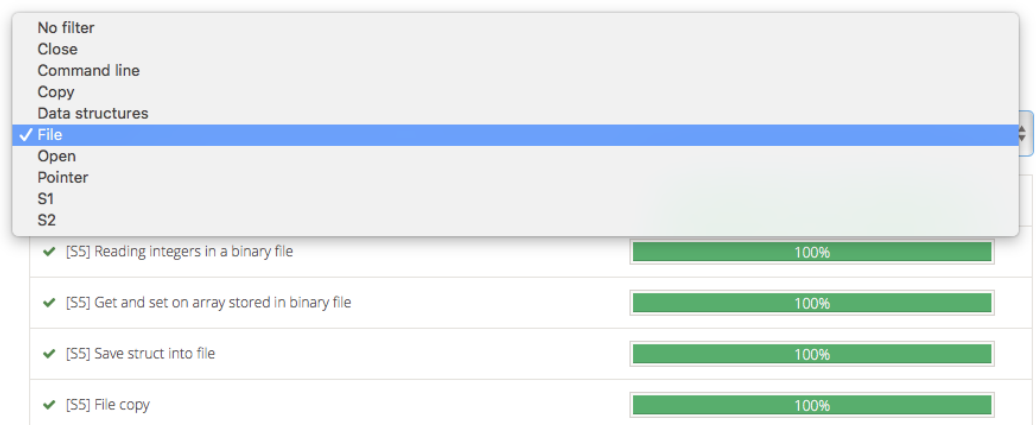
Figure 2.7



2.6.3 Course page

The course page is the page where we can consult the complete list of tasks of a course. We added a select menu to allow users to filter tasks by tags (figure 2.8). Of course, only tags with visibility enabled appear in the menu. The menu contains the skill tags and the categorical tags.

Figure 2.8: Filtering the tasks of a course



We chose to put skill tags in the menu to allow students to train themselves on specific course points. For example, a student who wants to train with exercises dealing with writing in a file can use the filter menu to only have the list of exercises where we can write to files. This is why we need to define consistent tag. If we want to tag tasks with skill tags not relative to a concept or skill (e.g. a `Unit test 1` or `Sum`) skill tag, we recommend

to set the visibility of the tag to hidden to avoid that students have uninteresting tags in the filter list.

Categorical tags are also present in the menu because this is their main goal: easily navigate through a huge number of tasks. Consider the huge number of LFSAB1401 tasks (more than 100). Each week, students of this course have to do the *final assessment question* of the week. Finding the corresponding question in the list is very annoying and time-consuming (need to scroll a lot). We do the test and it took 8 seconds. This is too much. Now imagine that all *final assessment question* are tagged with the categorical tag `Final Assessment Question`. By selecting this tag in the menu, we would only see a subset of all tasks that is 8. Now finding the *final assessment question* would be easier.

As a general advise when defining tags: do not hesitate to set the visibility of less interesting tags to hidden to avoid the growth of the filter list. The filter list was designed to easily navigate through a huge number of tasks. If the list itself contains a huge number of tags, we have the snake biting its tail problem, finding a tag in the list will be also time consuming.

An idea of improvement would be to use a kind of folders and sub-folders organization of tasks to allows the refinement of categories instead of having everything at the same place (like now with categorical tags). Indeed, a tree representation of tasks would scale better than a list representation (think of a file system).

2.6.4 Submission viewer

We modified the interface to consult submissions of students to add two new features: statistics on tags and submissions and a new interface to query submissions based on several criteria.

Submissions statistics

We think that adding general statistics on submission (without considering tags) is a useful feature. As a tutor of the LFSAB1401 course, we often wanted to know how many submissions we would have to correct. However, no interface of INGInious displays this value. We had to manually count the number of evaluation submission of our classroom to obtain the value (we only correct the best submission of each group of students). This is of course not practical and time-consuming. This is why we added some general statistics on submissions. Statistics displayed on the `Submissions statistics` box are:

- `Number of submissions`: the actual number of selected submissions.
- `Evaluation submissions (Total)`: the actual number of selected submissions but only the evaluation submissions, that is submissions marked with a star in the submission list.
- `Evaluation submissions (Succeeded)`: the actual number of selected submissions but only the evaluation submissions whose result is succeeded. Beware, `succeeded` does not mean the grade is 100%.
- `Evaluation submissions (Failed)`: the actual number of selected submissions but only the evaluation submissions whose result is `failed`. Beware, failed does not mean the grade is 0%.

Figure 2.9 shows how the interface is rendered in the actual version of INGInious.

Figure 2.9: Interface of submissions statistics box

Submissions statistics

Number of submissions	3055
Evaluation submissions (Total)	501
Evaluation submissions (Succeeded)	456
Evaluation submissions (Failed)	45

Tag statistics

To finalize our mechanism about tags, we need an interface to get our expected statistics on tags. Therefore, we added another box showing statistics on tags (figure 2.3). For more information about how these results are computed, go to section 2.7.8. Notice that all tags listed are those that are present in the actual selection of tasks (see section 2.6.4). By moving the mouse on a tag, the tag description will be shown in a tooltip. This is why the description of a tag is very important: if a staff member visits this page and wants to consult statistics, we should be able to understand the tag through the description if the tag name is not obvious.

Query builder

INGInious had only two static pages to consult submissions of students. The first page allow us to consult all submissions of a **specific** student for a unique task. The second to consult all submissions of a **classroom/team** for a unique task. As we can see, this is very restrictive and we may want to consult all submissions for a subset of students, for many tasks at the same time, etc. This is why we build this tool: a tool to graphically build a query to get, filter, and consult desired submissions. Figure 2.10 shows the interface to build queries.

The tool is composed of two parts. The first part to build the foundations of the request and the second to build more advanced queries (figure 2.11). The advance query box is by default hidden because many staff members should not use it often and it makes the GUI heavy.

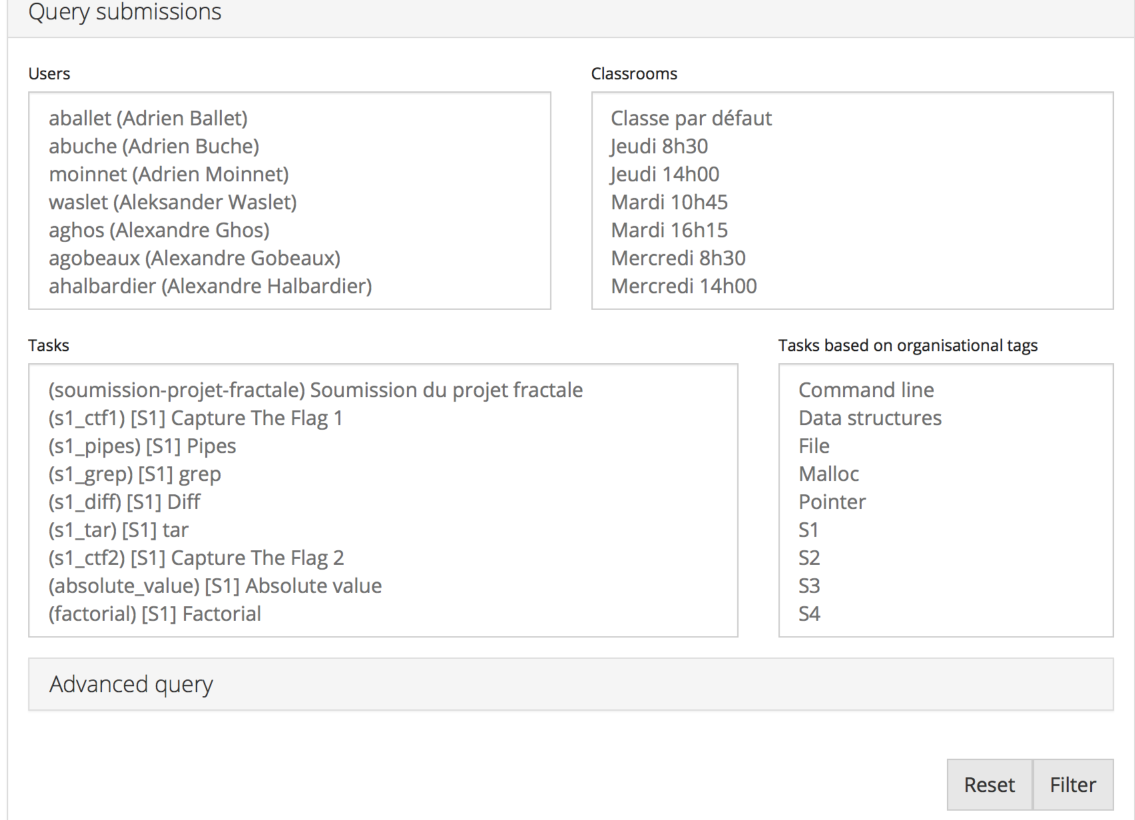
The first interface (figure 2.10) proposes four *multiple*[2] select menu. *Multiple* means that we can select more than one element in these menus.

- `Users`: this menu regroups all students and all staff members, sorted by complete name[3]. Each entry of the menu shows the username and then the complete name in parenthesis.
- `Classrooms` or `Teams`: this regroups all classrooms or teams.
- `Tasks`: the list of tasks of the course, sorted by the identifier of the task. Each entry

[2] https://www.w3schools.com/TAGS/att_select_multiple.asp

[3] The complete name is the first name followed by the last name

Figure 2.10: Interface to query submissions



shows in parenthesis the identifier of the task and then the complete name of the task.

- `Tasks based on categorical tags`: the list of all categorical tags founded in all tasks of the course. Selecting items from this list allows you to select multiple tasks in one operation.

The built request is the union $\cup$ of all selected parameters. For instance, if we select several students and one classroom, the result is $Users \cup Classrooms$. It is not possible to use an intersection $\cap$ instead.

The second box (figure 2.11) proposes many more parameters to query and filter submissions in a more accurate way. We explain below all these parameters, when they can be useful and their design choice:

- `Only evaluation submissions`: if checked, only the evaluation submissions of students will be selected.

- `Show tags`: if checked, skill, misconception, and auto tags will be displayed in each submission. This is useful if you need to manually inspect some submissions. It will also reveal four `"The tag is present"` fields after clicking the Filter button.

- `Show submission id`: previously, INGInious showed the submission if for each submission. However, this information is often useless (a tutor who wants to see submis-

Figure 2.11: Interface to query submissions (Advanced)

sions of his students does not care about the submission id). Therefore, we decided to hide the submission id by default to ease the interface.

- `Show task name`: the task id is not always human readable. We can take as example the open source Java tasks [ea18a] developed for INGInious where the identifiers of tasks are `m01Q1`, `m01Q2`, etc. Having the real task name instead of the task id is obvious in this case.
- `Show student name`: previously, INGInious only showed the username of students and there was no way to easily see the complete name of students. In many cases, tutors know the first name of their students but not their username.
- `Min grade`/`Max grade`: to select a range for the grade. The min and max value are included in the range.
- `After date`/`Before date`: to select a range for the date.
- `The tag X is present`: we included four fields to select only submission that **have** or that **have not** a specific tag. This can be useful for example when we have dependencies between tags. The proposed list of tags contains all skill tags sorted by name and then all misconception tags sorted also by name. Notice that these fields only appear if the check-box `Show tags` is checked (we need to click the filter button to refresh the page).
- `Sort by`: we can sort the displayed list of submission based on four fields: the date of the submission, the user, the grade and the identifier of the task.
- `Order`: To choose whether the submission list should be sorted in ascending or descending order.

- `Limit`: to reduce to the loading time of the web page, we can reduce the maximum number of displayed submissions. Of course, this does not affect statistics.

- `Statistics`: we can choose here how the statistics are computed. We can choose between three options: `With statistics` to have non-weighted statistics, `With weighted statistics` to have weighted statistics and `No statistics` to disable the computation of statistics. Since this computation is resource demanding, disabling statistics can be useful in some cases. The non-weighted statistics compute the mean of tags on all submissions while the weighted statistics introduce a mechanism so that each student and each task have the same weight in the computation of statistics[4]. You can find more information about weighted and non-weighted statistics in section 2.7.8.

2.7 Implementation: for INGInious developers

In this section, we focus on the practical aspect of tags form the INGInious developer and maintainer point of view. Many files across code of INGInious were modified to implement the tag mechanism. We explain in this sections major changes we performed, the location of these changes and implementation choices.

2.7.1 Tags

A new class (`inginious/common/tags.py`) representing a tag was added to handle tag operations and management. Therefore, tags are objects with variables and methods. Tags were initially simple tuple `(id, name)` but for extensibility, we chose to use objects instead.

2.7.2 Course

On the main page of courses (the page where all tasks are displayed), we added a select menu to filter tasks by skill tags and category tags. This filter works with JavaScript at the user side. When obtaining the page, each task contains some embedded hidden HTML `div` tag for each tag. Selecting an element in the list simply hide tasks that do not have the tag. The changed files are:

- `inginious/frontend/templates/course.html`
- `inginious/frontend/pages/course.py`

To make the web app more efficient when using tags, four caches were added at the level of the courses object of INGInious (`inginious/frontend/courses.py`). Indeed, we very often need to get the list of all tags of a course, for instance, when we visit the main page of a course, there is a complete list of tags to filter tasks by tags, when we want to import tags in the task editor, or when we want to query submissions. Computing this list requires many operations because we need to loop through all tasks of a course and then, for all tasks loop through all tags. By adding a cache, we can compute this list once and then keep it in memory to allows the next retrieve in constant time. For instance for the LSINF1252 course with 333 tags and 85 tasks, the computation time of the complete list

[4] A student who made many submissions will not have a bigger weight in the statistics than a student with only one submission

of tags takes 0.4ms[5] when the cache is empty. The computation time might seem minimal here, but if we consider a course with 200 tasks, 10000 tags and 1000 students loading the main course page at the same time (e.g. during an exam), the cache will be very helpful since the computation will be instantaneous.

We will now define the purpose of the four caches: Notice that the caches are cleaned and refreshed when a task is modified by someone, namely when we click on the "Save changes" button in the task editor. Therefore, if a task is modified by another manner (e.g. with automatic sync from GitHub) the cache may become inconsistent.

2.7.3 Task

Task editor

Tags are stored in the *task.yaml* file of tasks, more precisely in the `tags` entry. The GUI of INGInious was modified by adding a new tab in the task editor to allow the creation and management of tags. We have an example of two tags stored in a yaml file in figure 2.12. When saving the task, some operations concerning tags are performed:

- Uncompleted tags are removed: if the id or name field is missing, the tag will not be saved in the task.
- Throw an error if the id of a tag does not contain only alphanumerical characters, dashes or underscores.
- Throw an error if some tags have the same id.
- All tags are sorted by type. This allows to retrieve tags rapidly (by humans, for example to update or edit an existing tag) in the GUI when the task contains many tags.

Some JavaScript is also used in the tag editor for three purposes. First, we have to blur the id field when the selected type is *category* since category tags have no id. Secondly, expand the description field when the user clicks on this field. This permits us to have a compact interface while having a place to enter a long description. The last thing is the button to add new rows for new tags. In fact, a hidden row is present in the HTML of the page. When someone clicks on the add row button, the hidden row is copied and then added at the end of the table with a different visibility status. The purpose of using a hidden row is for maintainability reasons; if someone has to perform modifications in the tag editor, he would just have to edit the hidden row. The import tag feature also uses some JavaScript: when clicking on the arrow to import a tag, a new empty row is added using the method described above, but with an extra parameter specifying that the new row has to copy its value field from an existing tag. Concerned files:

- `inginious/frontend/pages/course_admin/task_edit.py`
- `inginious/frontend/templates/course_admin/edit_tabs/tags.html`
- `inginious/frontend/static/js/studio.js`

[5] On a laptop with 2 cores at 2.6 GHz and an SSD

Figure 2.12: Tags saved in task.yaml file

```yaml
tags:
    '0':
        id: bhd
        type: 0
        name: Bounds Handling
        visible: true
        description: You are able to loop through an
            array without going out of the bounds of
            the array.
    '1':
        description: An array has been accessed with
            an illegal index. The index is either
            negative or greater than or equal to the
            size of the array. Try to make easy
            exercises having the "Bounds Handling" tag.
        visible: true
        id: oob
        name: Out Of Bounds
        type: 1
```

There is no way to edit a tag for several tasks (e.g. update the description of all tags with name `compile`). We recommend to build and use separate scripts to perform these kinds of operations. Notice that the numbers at the second layer (`'0'` and `'1'` in the example) are not used by INGInious (the tag order is the definition order in the yaml). Their unique goal is to separate tags.

Task page

Concerned files:

- `inginious/frontend/templates/task.html`
- `inginious/frontend/pages/tasks.py`

When a student submits something and after INGInious finished a job, a JSON object is returned to the task page with some results like the grade, the feedback, the status, etc. We needed to add tag states in this JSON to allow the JavaScript on the page to highlight, hide or show tags. We also paid particular attention to not include in the JSON information about tags not visible to students. Indeed, it would be annoying if students have access through the JSON to the tags that they are not supposed to consult. The presence or not of these tags is based on the debug mode when submitting.

We added two new sections for tags in the HTML of the page. The first section for category tags which simply list all category tags of the task. The second section includes all skill tags and misconception tags, but these latter are hidden. Thanks to that, we are able to show or hide misconception tags and change the style of skill tags with some JavaScript

(inginious/frontend/static/js/task.js). When receiving the results of a finished job, depending on the tag type and the status of the tag (true or false), we perform some modification of the visual display of tags: when a skill tag is present with a true status, we swap the style of skill tags to green. When a misconception tag is present with a true status, we simply show it to users. A new badge is also generated for each auto-tag.

2.7.4 Submissions

All tags are stored in the tests entry of submissions.

2.7.5 Docker Agent

The docker agent is responsible to handle job management in the backend. When a job ends, the agent gets the result of the job running in the container in the form of a dictionary. However, the agent imposes restrictions on key and sub-keys present in this dictionary. The restriction imposes the key to only contain alphanumerical characters, dashes or underscores. Since ids of auto-tags start with a star like *auto-tag-123, an exception was added to accept this kind of id for auto-tags. The star in the id is mandatory to avoid users to set auto-tags manually, namely without using the corresponding API function.
Concerned files:

- inginious/agent/docker_agent/__init__.py
- inginious/common/base.py

2.7.6 INGInious API

Previously, INGInious had a mechanism called definetest to define some key-value pairs when grading the task. However, this feature was not really used and was only used for debugging. The tag mechanism enhances and replaces the old definetest that was removed. The tag mechanism introduces four new API functions available in the base container of INGInious, where two are for python and two for bash:

- feedback.set_tag(tag, value)
- feedback.tag(value)
- tag <value>
- set-tag <tag_id> <value>

Some changes were also performed to allow this mechanism to work in all cases because the old definetest was originally designed to be activated in debug mode only. In INGInious, the debug mode is only activated for administrators of the course. Since the tag mechanism has to work also with code of students, these changes were mandatory.
Concerned files are located in folders:

- base-containers/base/bin/
- base-containers/base/inginious/

2.7.7 Submission viewer

A new page to allow tutors and administrators to query submission based on tags was added. This page replaces two old pages in only one. Previously, INGInious had one static page to consult submissions for **one student** and another page to consult submissions for a **team/classroom**. Since these two pages were very similar and because we did not want to add a form to filter submissions by tags in these two pages (code duplication is bad), we decided to merge them in a single one. This is also beneficial for the maintainability of INGInious. This page inherits from `INGIniousAdminPage`. In other words, this page is only accessible for tutors and administrators. If a student tries to access this page, he will get a 404 error.

The page allows users to filter or query submissions based on a wide panel of options. When clicking on the `Filter` button, all data of the form are sent to the web app of INGInious with a POST request.

Next, the web app will generate a huge Mongo request containing all data present in the form. We paid attention to sanitize inputs of the form to avoid some Mongo injection.

After the processing of the Mongo query, some statistics can be computed based on the choice of the users. By default, statistics are computed without weights. Finally, just before sending the results to the user, the list of submissions is truncated to 500 (without affecting statistics of course). This allows the web page to load faster. Indeed, generating a web page containing many submissions is very resource demanding, but also useless for the user since he will anyway not inspect several thousand of submissions by hand.

Tag statistics are displayed with a one digit accuracy after the decimal point and rounded to the floor. Notice that we display $> 0.0\%$ when the real number is not 0.0% but the round operation transform it to 0.0%. Indeed, we need to make a difference between tags that **never** appear and tags that appear **rarely**.

Finally, we added support to download the current selection of submission as a CSV, download submissions in an archive and replay submissions.

2.7.8 Statistics

A new utility (`inginious/frontend/pages/course_admin/statistics.py`) containing simple functions to compute statistics on submissions and tags was built. There are two utility function.

The first to compute statistics like the total number of submission, the number of evaluation submissions, the number of evaluation submissions with a succeeded status or the number of evaluation submissions with a failed status. This function was designed to be easily extensible so that, we could add new basic statistics such as the mean/median/min/max grade on evaluation submissions, the average number of submissions for each student, etc. These additions can be done in `inginious/frontend/pages/course_admin/statistics.py` in the function `fast_stats()`.

The second function computes statistics on tags. This function introduces two modes of computations: weighted statistics and non-weighted statistics. Depending on the use case, one may prefer one over the other. To illustrate the difference between these two modes, let us consider a concrete example: we have two students and two tasks, each task is tagged with the tag `not_compile`, indicating that the code of the student does not compile. The first student had many difficulties and made 100 submissions where 90 of these did not compile for the task 1. For the task 2, he made 1 submission that did not compile. The

second student is very good and made only 1 correct submission for each task. We may expect two types of results on statistics about "how many submissions do not compile?": 88% or 48%. In the first case, the student who made many more submissions has more weight in the statistics because statistics are literally computed on the total number of submissions. In the second case, each student **and** each task have the same weight in the statistical computation. Therefore, a student who made many submissions in a task will not overweight statistics.

Figure 2.13: Computation of statistics for a single tag

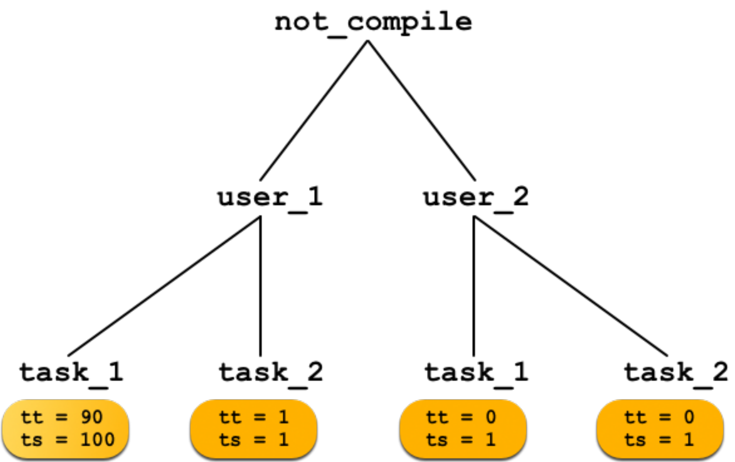


Figure 2.13 presents a detailed explanation of how statistics are computed: for each tag, a tree is generated based on users and tasks. When looping through each submission and each tag of these submissions, we increment two variables in the corresponding tree depending on the presence of the tag or not. `tt` is the total number of the tag and `ts` is the total number of submissions. Once all trees are built, we can compute statistics in two different manners to have weighted statistics or not. If we want non-weighted statistics, we use the equation 2.1. If we want weighted statistics, we use equation 2.2. In these two equations, L_i is the ith leaf of the tree and n is the total number of leaves.

$$\frac{\sum_{i=1}^{n} L_{i_{tt}}}{\sum_{i=0}^{n} L_{i_{ts}}} \tag{2.1}$$

$$\frac{\sum_{i=1}^{n} \frac{L_{i_{tt}}}{L_{i_{ts}}}}{n} \tag{2.2}$$

For example, with our example above, we would compute for non-wighted statistics:

$$(90 + 1 + 0 + 0)/(100 + 1 + 1 + 1) = 0.88$$

and for weighted statistics:

$$(90/100 + 1/1 + 0/1 + 0/1)/4 = 0.48$$

2.8 Test, evaluation and results

2.8.1 LFSAB1401 exam

In this section, we integrate tags in several exam questions to highlight how we can use tags efficiently, what kind of information can be derived from tags and how to use this information to improve the content of the course and the tasks themselves. We chose to use three questions from the exam of the LFSAB1401 Java course during the January 2017 session. Three facts motivated this choice: this course relies on an online exam on INGInious, secondly, we tutored this course, and finally, more than 500 students registered for this exam, what is interesting for the statistical part. For the three chosen questions, we explain the general statement of the question, define some tags related to the statement and the test suite, show and interpret results and finally, suggest some improvements for the content of the course and/or the task. Notice that we do not modify the behavior of the original test suite during the integration of the tags. We only perform some additions to the original test cases of the test suite to support the tag mechanism.

Equals method

Statement In this question, the student has to implement the `public boolean equals(Object o)` method of the class `Pass`. The statement students had was the following:

```java
/**
 * A ski pass.
 * Contains a number, the name of the pass holder, and a period of validity,
 * in the form of [start time, end time].
 */
public class Pass {
    private int number;
    private String name;
    private long startTime;
    private long endTime;

    /**
     * @pre -
     * @post return {true} if {o} is a pass of the same number and
     *       same name as this pass, {false} otherwise.
     */
    public boolean equals(Object o) {
        // QUESTION 1
    }
}
```

To answer this question, we expect students to first check if the object is an instance of `Pass`, then cast `Object o` in a `Pass` object. Finally, check that numbers of the two pass are equal and that the names are equal.

Tag definition Since the test suite already exists and we do not want to modify its behavior, we simply add one skill tag for each test case. So, we label the task with the following skill tags:

- `InstanceOf`: the student correctly uses the `instanceof` operator. Notice that it is simply done by giving as parameter to the `equals` function of the student a null object and then an object of another type.
- `Base Case`: the base case checks if the code of the student works correctly when the two objects are the same (same reference) when the two objects are the same (with different references) and finally when the two objects are completely different (all fields different).
- `Required Fields Used`: this tag checks that the `number` and the `name` are indeed used in the comparison. This tag depends[6] on the `Base Case` tag.
- `Only Required Fields Used`: this tag checks that the `startTime` and the `endTime` are not used in the comparison. This tag depends on `Required Fields Used`.
- `String Equals Used`: checks that the student uses the `name.equals(String)` method instead of the == operator. This tag depends on `Only Required Fields Used`.

[6] *Depends on* means that the tag cannot be obtained if the other tag is not obtained.

Table 2.1: Weighted tag results: `equals`

Skill tag	All submissions	Best submission
InstanceOf	64.9 %	83.0 %
Base Case	66.1 %	84.8 %
Required Fields Used	65.5 %	83.8 %
Only Required Fields Used	63.5 %	81.8 %
String Equals Used	42.2 %	51.6 %
Misconception tag	**All submissions**	**Best submission**
Not Compile	29.5 %	6.9 %
Try Catch Block Detected	0.1 %	0.7 %
No InstanceOf	3.6 %	9.1 %
StackOverflowError	0.5 %	1.1 %
ClassCastException	0.2 %	0.3 %
ArrayIndexOutOfBoundsException	0.0 %	0.0 %
NegativeArraySizeException	0.0 %	0.0 %
ArithmeticException	0.0 %	0.0 %
NullPointerException	2.9 %	5.7 %
StringIndexOutOfBoundsException	$>$ 0.0 %	0.1 %
Exception	3.8 %	7.5 %

Concerning misconception tags, we chose to define:

- `Try Catch Block Detected`: when the student tries to resolve this question with a try-catch block. This tag is given by using two `grep` on the code of the student.
- `Not Compile`: when the code does not compile. Notice that all skills and misconceptions tags are dependent on this tag.
- `No InstanceOf` when the student does not try at all to use the `instanceof` operator. This tag is given by using a `grep` on the code of the student.
- Some common exceptions of the Java language.

Results Results are presented in table 2.1.

Interpretation and suggestions

String Equals Used As we can see, results for the first four skill tags are good: more than 80% of students are able to write an `equals` method. However, the score for the `String Equals Used` tag drastically decreases. Since we have a dependency between tag `Only Required Fields Used` and `String Equals Used`, we need to re-filter submissions and keep only those that have the `Only Required Fields Used` tag acquired. The new result is 62.1% for the best submissions. This means that only 62.1% of the students who passed the `Only Required Fields Used` test use the `equals` method of the `String` class. This reveals a problem in form students compare strings. This may come from the fact that some tasks [ea18a] of INGInious did not check properly that the student uses `equals`

instead of ==. This is a known problem and was corrected[7] in some tasks during the year but the damage was done: some students performed the task before the correction and got no feedback about this misconception. Moreover, there are still probably some tasks that do not check properly the comparison...

Instance of We can see through the `No InstanceOf` tag that 9.1% of the students did not use the `instanceof` operator for their final submission either because they ignore this operator, because they ignore how to use it correctly or because they simply forgot to use it. This quite high absence of `instanceof` operator in submissions of students may be explained by the fact this operator is not often used in the course except for the `equals` method. Indeed, the course could give more example of usages of `instanceof` and more explanations on the operator itself during the lecture. For example, by passing through the slides of lectures on classes and objects [BP18b] [BP18a], nothing explains the existence of the `instanceof` operator and how to use it.

It may be also interesting to verify the effectiveness of the `InstanceOf` tag/test case. Thanks to our tags, we can use the filtering of submissions to keep only those where the tag `InstanceOf` is present and the tag `No InstanceOf` is also present. We would expect that no submissions are found since this is paradoxical to have at the same time the opposite tag. However, the result outputs that there are 9 submissions tagged with these two tags... By looking at these submissions by hand, we can see a weakness for the `InstanceOf` test case since a submission doing just `return false;` validates this test case and gives a score of 20% to the student.

Try-Catch We expect that students resolve this task without any try-catch block since this is absolutely not necessary to implement an `equals` method in Java. Thanks to the tag `Try Catch Block Detected`, all submissions where students use a try-catch block are marked. We can see that only 0.7% of the final submissions contain a try-catch block but, if we look carefully to these submissions, we find four of them where the student has the maximum grade by implementing something like

```
1  try {
2    Pass p = (Pass) o;
3    if(o != null && number == p.getNumber() && name.equals(p.getName()))
4      return true;
5    else
6      return false;
7  }
8  catch(Exception e){
9    return false;
10 }
```

This works but this is one of the ugliest ways to implement an equals `method`. After discussing with one of the assistants of the course, we agreed that this kind of solution should not be graded 100%. A simple improvement to the INGInious task would be to reject submissions containing a try-catch block.

[7] github.com/UCL-INGI/CS1-Java/commit/9db6fe5d5888665ac6224a90a60142752b44d9be

Delete suffix of a sorted list

Statement In this question, the student has to implement a method to delete a suffix of a sorted list of nodes. The statement proposed to the students was the following:

```
1  /**
2   * A log of scanned ski passes, in the form of a simply linked list. A
3   * list item is a recording including the scanned ski pass, the
4   * scan status and the time it was scanned. Recordings
5   * are ordered by scan time, most recent at the top of the list.
6   */
7  public class Log {
8
9      private class Node {
10         long hour; // The timestamps of the scan
11         Pass pass; // The scanned pass
12         int stat;  // The status of the scan
13         Node next; // The next node
14     }
15
16     private Node first; // head of the list
17
18     /**
19      * @pre hour > 0
20      * @post deletes records whose time is less than {hour} from the log.
21      */
22     public void deleteBefore(long hour) {
23         // QUESTION 5
24     }
25 }
```

To answer this question, we expect students to iterate over the list, stop when the timestamps of the next node is smaller than the given timestamps and then cut the rest of the list by setting the next pointer to NULL. For instance, if the list contains [5,4,3,2,1] and the time to cut is 3, the list should become [5,4,3].

Tag definition We define skill tags based on test cases as:

- `Erase before all`: List=[4,3,2,1], Hour=0, we should remove nothing.
- `Erase after 1`: List=[4,3,2,1], Hour=5, we should remove all nodes.
- `Erase after 2`: List=[4,3,2,1], Hour=3, remove the last two nodes
- `Erase between 2 and 3`: List=[4,3,2,1], Hour=2.5, remove the last two nodes.
- `Erase after 3`: List=[4,3,2,1], Hour=2, remove the last node.
- `Erase empty`: List=[], Hour=1, remove nothing since the list is empty.
- `Erase before result`: List=[4,3,2,1], Hour=1, we only check that the code of the student does not throw an exception. The result of the code of the student is ignored.

The misconception tags are the same as the previous question and are self-explained.

Table 2.2: Weighted tag results: delete suffix of a sorted list

Skill tag	All submissions	Best submission
Erase before result	43.9 %	63.3 %
Erase before all	40.8 %	57.4 %
Erase empty	39.0 %	56.3 %
Erase after 1	26.9 %	40.8 %
Erase between 2 and 3	26.3 %	40.6 %
Erase after 3	21.6 %	33.6 %
Erase after 2	20.6 %	31.2 %
Misconception tag	**All submissions**	**Best submission**
Not Compile	31.0 %	5.8 %
Infinite Loop	5.8 %	6.9 %
Try Catch Block Detected	0.7 %	1.7 %
StackOverflowError	0.0 %	0.0 %
NullPointerException	38.9 %	51.3 %
ArrayIndexOutOfBoundsException	0.0 %	0.0 %
ClassCastException	0.0 %	0.0 %
NegativeArraySizeException	0.0 %	0.0 %
StringIndexOutOfBoundsException	0.0 %	0.0 %
ArithmeticException	0.0 %	0.0 %
Exception	40.9 %	54.5 %

Results Results are presented in table 2.2.

Interpretation and suggestions

Skill tags As we can see, results for skill the tags are well distributed: simple test cases have better scores than specific test cases. We can see here the interest of defining several independent test cases where each of them can give some points to the student. Indeed, students who have difficulties will still be able to get some points through simple test cases like Erase `before result`, `Erase before all` and `Erase empty`.

Null pointer exception We can see through the tag `NullPointerException` that the code of more than half of students provokes some `NullPointerException` for their final submissions. This reveals that students have many difficulties to handle null objects. An improvement to the course would be to better focus on these exceptions, explain in depth how to prevent them and do exercises on how to handle null objects correctly.

Try-Catch 8 students, handle `NullPointersException` by surrounding their code with a big `try{...}catch(NullPointersException){}`. This confirms that students have difficulties to handle these exceptions. Moreover, like for the previous exam question, students use the exception handling mechanism in an incorrect context: we can avoid `NullPointersException` by simply implementing a check to detect that a reference is null instead of using a try-catch block. To overcome this problem, we should better inform students about when to use or not to use try-catch blocks and why this is bad to use them

in an incorrect context. We may also forbid the usage of try-catch blocks in INGInious tasks where this is not necessary to train students to not use them.

Infinite loop The exam contained a bug that allowed students to get a score of 100% by doing an infinite loop. Hopefully, students were not able to detect this issue during the exam since tasks did not display the grade to students. However, some students still made infinite loops without doing it on purpose and therefore obtained the maximal score instead of 0%. Thanks to the tag `Infinite loop`, we can see that 6.9% of the students (32 students) are concerned by this bug.

Iterating on a list and writing in file

Statement In this question, the student has to implement a method to save the content of the nodes of a list in a file. The statement students had was the following:

```java
/**
 * A log of scanned ski passes, in the form of a simply linked list. A
 * list item is a recording including the scanned ski pass, the
 * scan status and the time it was scanned. Recordings
 * are ordered by scan time, most recent at the top of the list.
 */
public class Log {

    private class Node {
        long hour; // The timestamps of the scan
        Pass pass; // The scanned pass
        int stat;  // The status of the scan
        Node next; // The next node
    }

    private Node first; // head of the list

    /**
     * @pre filename != null
     * @post Save the content of the log as text in the file
     *       {filename}. In case of error when accessing the file,
     *       interput the saveing and print "ERROR" on standar output.
     *
     *       The format of the file is one line per record formatted as:
     *       "{hour} {number} {stat}", where {hour} is the time formatted by
     *       {Horloge.stringOf}, {number} is the number of the pass, ans {stat} the
     *       status. For example:
     *            8/01/70-1:13:00 100 1
     *            4/01/70-1:12:40 102 0
     *            4/01/70-1:12:20 101 2
     *            4/01/70-1:12:00 100 0
     *            2/01/70-1:01:20 100 3
     *            1/01/70-1:00:20 100 1
     */
    public void save(String filename) {
        // QUESTION 6
    }
}
```

To answer this question, we expect students to initialize a `PrintWriter`, iterate on the list until the end and for each node, write the time, the number and the status of the node in the file and finally close the file. The student has also to use a try-catch block to intercept possible `IOException` that will occur and print a message to the standard output.

Tag definition For this question, we chose to define the following skill tag:

- `Exception Handled`: checks that the student handles IOException. This is checked by giving an invalid filename to the student: when the student tries to create the file with this invalid filename, the system throws an `IOException`.

- `Exception MSG Printed`: same as the tag `Exception Handled` but we check also that the student wrote "ERROR" to the standard output.
- `Close Called`: checks that the student called the `close()` method.
- `File Exists`: simply checks that the student creates the expected file. Notice that the content of the file does not matter here.
- `Empty Log`: checks that the student writes no line in the file when the log is empty.
- `Log 4 Lines`: checks that the student writes four lines in the file when the log contains four entries. Notice that we only count the number of lines, we do not care of the content of these lines.
- `Log Line Format`: checks that the first line written in the file respects the defined format.

Again, these tags are simply based on the actual test cases of the task. Concerning misconceptions tags, we defined:

- `System exit`: if students use `System.exit()` in try-catch block.
- `File Not Accessible`: if the test suite can not open the file created by the student. For example because the file was not created, because the name of the file is incorrect or because of incorrect permissions.

Results Results are presented in table 2.3.

Interpretation and suggestions

Easy skill tags Let us start by some skill tags: we can see that for the most simple test cases, students reached 78.6% of success to create the file and 76.8% were able to use a try-catch block to intercept the `IOException`. We can also see a lower score of about 18% concerning test cases `Empty Log`, `Log 4 Lines` and `Log Line Format`. This is normal since these test cases are more difficult and require iterating over the list and writing in the file (not simply create it).

Exception message printed and System exit We can see that the score of the test case `Exception MSG Printed` is very low. In fact, this is due to the fact that students call in abundance the `System.exit()` function: when looking by hand at some submissions tagged with `System exit`, we can see that these students first printed the expected message and then called `System.exit()`. However, the test case `Exception MSG Printed` is not validated if `System.exit()` is used. This explains the lower score for the `Exception MSG Printed` test case.

Let us now analyze the `System.exit()` tag: 27.6% of students called `System.exit()` in their final submissions! We first thought this was a bug in the test cases since the score for this tag is very high but after verifications, this was not the case. This high score reveals a problem for students to use and understand how, when, and why using `System.exit()`. Indeed, the statement of the question do not require to use `System.exit()` but 27.6% of the students use it anyway. We think that this problem, may come from that students

Table 2.3: Weighted tag results: write file

Skill tag	All submissions	Best submission
File Exists	35.5 %	78.6 %
Exception Handled	34.5 %	76.8 %
Empty Log	28.0 %	59.8 %
Log 4 Lines	25.8 %	58.3 %
Log Line Format	25.8 %	58.5 %
Exception MSG Printed	15.5 %	34.5 %
Close Called	0.0%	0.0 %
Misconception tag	**All submissions**	**Best submission**
System exit	12.6 %	27.6 %
File Not Accessible	3.7 %	8.8 %
Not Compile	56.5 %	9.5 %
Infinite Loop	3.8 %	5.8 %
ArrayIndexOutOfBoundsException	0.0 %	0.0 %
StringIndexOutOfBoundsException	0.0 %	0.0 %
ClassCastException	0.0 %	0.0 %
ArithmeticException	0.0 %	0.0 %
StackOverflowError	0.0 %	0.0 %
NegativeArraySizeException	0.0 %	0.0 %
NullPointerException	7.6 %	18.3 %
Exception	8.3 %	20.3 %

do not really know the difference between a program that ends by itself (because we are at the end of the program) and a program that ends because `System.exit()` was called explicitly.

Close Called As we can see, absolutely no student have passed this test case. Two reasons can explain this results: the test case is too difficult or there is a problem that prevents it from succeeding. Therefore, we analyzed the test code of the task by hand and found that it failed, no matter what we do. This illustrates how we can use tags to find errors in test suites.

Not Compile 9.5% of the final submissions even do not compile. Not even being able to write a code that compiles in an exam context is relatively unfortunate. This problem probably comes from students who are unable to understand the compilation output when their code does not compile. Since this an introductory course, students should learn how to interpret compilation errors. However, during practical work session with a tutor who can help them to understand compilation error, students use paper to write their code and there are no compilation errors on paper. Maybe this explains why one in ten students submits code that does not compile at the exam.

Conclusion

We saw through the three exam questions above what kind of information can be derived from tags.

Concerning students, we found out that 51.6% students use the `==` operator instead of `String.equals(String)` function when comparing two strings, that 1.2% of students do not really know when to use the exception mechanism, that 51.3% of the students made `NullPointerException` when playing with lists, that 27.6% of the students have difficulties regarding the `System.exit()` function and use it when it is not necessary, and finally, that 7.4%[8] of the evaluation submissions do not compile.

Regarding the tasks themselves, we found that some test cases were too weak like the `instanceof` test of the first exam question where simply `return false;` validates the test. We also discovered that infinite loop gave a 100% grade and thanks to tags we were able to measure this impact easily. Finally, we detected that a test case was infeasible since no students were able to solve it. We can also confirm that test suites with several independent easy and hard test cases allow to give some points to students who have more difficulties than the others.

[8] Computed on the three exam question

2.8.2 LSINF1252 tasks

At the end of January, some students were hired to write INGInious tasks in C programming for the LSINF1252 computer systems class [ea18b] and the new tag mechanism was used. Tasks were divided into five categories, one for each week. The week 1 was about Linux commands, week 2 about strings and bitwise operations, week 3 about pointers, week 4 about data structures (list, tree, queue) and finally week 5 about files.

During this week, no bug or problem were found concerning the implementation of the tag mechanism in the production version of INGInious. On the other hand, some usability problems were found in the web app concerning tags:

- The button to import tags in the *Import* tags panel was located at the right. However, authors of tasks use mainly the id and the name of tags which are located at the left to choose which tag they want to import. Consequently, the button is far from the useful information and can produce miss clicks or loss of time because we have to ensure we click on the correct row. This does not respect the 7th rule of the Jakob Nielsen's 10 general principles for interaction design [Nie95]. The button is now located at the left.

- The description of tags are not properly shown on the task page in the web app due to problems in the CSS. However, the CSS is not our job and this is out of the scope of this master thesis. The problem may disappear when the CSS of INGInious will be updated.

Concerning the tag mechanism, we used it as following during the job:

- All tasks were tagged with the misconception tag `Timeout`, `Segmentation Fault`, `Not compile`, `Memory Exceeded`, `Floating Point Exception`, `Double free` and `Banned functions`. This was done easily because these tags are set directly in the run script of the task and the *CTester* library which is common to all tasks. Therefore, writers of tasks had just to import these tags and did not have to beware about their behavior. Notice that tasks of the week 1 were not tagged since these last do not use the *CTester* library.

- All tasks were tagged with some category tags named `S1`, `S2`, `S3`, `S4` or `S5`, depending on the week of the task.

- Finally, some tasks were tagged in depth with skill tags and many misconception tags. For example, the task *[S5] File copy*[9] was tagged with skill tags `Open`, `Close`, `Copy`, `Permissions` and the misconception tags `Original file modified` and `Failure handling`.

We would have liked to use tags more in depth on many other tasks but, the lack of time did not allow us to implement it.

Results

Table 2.4 reports results about general tags obtained from the submission viewer page. Blanks in the table are equivalent to 0.0%. There is not much to say about these results

[9] https://github.com/UCL-INGI/LSINF1252/tree/master/s5_file_copy

Table 2.4: LSINF1252 general weighted tag results

Tag	2 all	2 best	3 all	3 best	4 all	4 best	5 all	5 best
Timeout	0.9%	0.1%	1.1%		3.2%	0.4%	1.1%	0.1%
Seg. Fault	2.2%	0.2%	9.2%	1.3%	14.4%	5.9%	0.9%	
Not compile	29.5%	0.8%	42.2%	1.0%	40.8%	2.1%	35.6%	0.7%
Memory Exceed			0.1%		0.1%			
Float Pt Except								
Double free								

except that week 3 and 4 have the highest compilation errors and segmentation faults, but this is normal since these are the most difficult weeks. Moreover, these exercises were done by students during practical work sessions supervised by tutors. Therefore, this is normal that the evaluation submissions have a very small rate of misconception tags since tutors could help students to resolve their problems. This is why we should interpret results given by tags carefully since tutors interfered in the results as they helped the students.

As said above, we also tag in depth tasks of week 5. After consulting results and rates of tags for these tasks, it appears that no major difficulties are highlighted by tags. For example, for a task where students have to copy a file, for their best submissions, 98% opened a file correctly, 91.5% closed the file correctly, 85.7% were able to perform the file content copy and 84.4% handled permissions of the copied file. Concerning misconceptions, 4.5% of students did not manage failures properly during the copy and 0.6% of the students modified the original file. Results for other tasks of week 5 are similar.

However, during the academic quadrimester we consulted the statistics of the tasks of the week 5 and we saw that the tag `failure handling` had a low rate of acquisition. We therefore checked that the tasks of week 5 correctly evaluated this tag. So we filtered the submissions by selecting only those submissions that did not have the `failure handling` tag and sorting them by grade, to get the best students in the top of the list. We looked at their submissions and realized that they were correct. The problem was therefore the task, which we immediately corrected.

2.9 Guidelines

In this section, we provide some guidelines for tasks authors that would like to label their tasks with our tag mechanism.

2.9.1 Links between tags

Try to associate skill tags with misconception tags. For example, a skill tag called `Bounds Handling` and a misconception tag called `Out Of Bounds`. The description of the `Out Of Bounds` tag can then refer to the `Bounds Handling` tag as shown in table 2.5. In this manner, students can be guided across different exercises if they make redundant mistakes. Consider a task where students have to sort an array. If they get the tag `Out Of Bounds` many time, the tag should advise them to make or remake easier exercises tagged with `Bounds handling`; like a simple task where we have to find the maximum element in an array.

Table 2.5: Associated tags

Id	Name	Description	Show to students	Type
bhd	Bounds Handling	You are able to loop through an array without going out of the bounds of the array.	Yes	Skill
oob	Out Of Bounds	An array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array. Try to make easy exercises having the "Bounds Handling" tag.	Yes	Misc

2.9.2 Hide useless skill tag

We can use skill tags to register in the submissions the success or failure of all test cases of a test suite. If these tags do not represent a useful concept for students, hide these tags. By doing that, students will not be disturbed by strange tags and we and we will still be able to see tag statistics. For instance, if we define a tags called `Test1`, we will hide this tag to student since it means nothing for them. We also recommend to write a nice description for your tag so that all staff member that may consult statistics can understand the tag goal.

2.9.3 How to integrate tag in an existing test suite

If a test suite is well designed with several and independent test cases, it should be easy to integrate tags in it. We provide below an example of a Java task with JUnit.

For skill tag, we can set a tag in each test case.

```
@Test
public void TestSum() {
  if (Student.studentMethod(2, 3, "sum") != 5)
```

```
    fail("The sum is not correct");
  else
    set_tag("sum", true);
}

@Test
public void TestNeg() {
  if (Student.studentMethod(10, 4, "neg") != 6)
    fail("The negation is not correct");
  else
    set_tag("neg", true);
}
```

For misconception tags, we should have a place where we catch exceptions that the code of the student throws. If this is the case, we can simply set misconception tags in this location.

```
try {
  // Call code of student
} catch (Throwable e) {
  set_tag("exception", true);
  set_tag(e.getClass().getSimpleName().toLowerCase(), true);
  fail("Exception: " + e.toString() + ": " + e.getMessage());
}
```

CHAPTER 3

Randomization of exercises

This chapter describes a mechanism we called *random inputs* to randomize INGInious exercises to prevent students to copy-paste solutions of these exercises. Section 3.1 explains the need for such mechanism, section 3.2 briefly describes *random inputs*, section 3.3 illustrates how to use our mechanism through two examples of exercises, section 3.4 shows how we implement this mechanism in INGInious and finally section 3.5, briefly explains how we test *random inputs* and some directions for improvements.

3.1 Motivations

As a tutor, we sometimes have to face copy/paste made by students in INGInious tasks. Indeed, students can easily share between them solutions of exercises (e.g. with Facebook, Dropbox or Google Drive) and then copy-paste these solutions on INGInious to get a 100% grade without really doing the exercise. This may affect the entire classroom and the tutors may be less effective. Here are some bad points of copy-paste:

- Tutors may think that their classroom has correctly understood the content of the course because students have a nice grade on INGInious, but in fact, many students simply copy-paste code found online. Tutors may then speed up and students will be completely lost.
- When students use copy-paste, students and tutors lose their time. The student lost his time to perform the copy-paste and learned nothing. The tutor may lose his time if he has to manually review some code to provide personalized feedback.
- We cannot motivate students to do exercises on INGInious by giving them some bonus point. Students may simply copy-paste solutions to get the bonus point. We may use a plagiary detector but if there are many students doing copy-paste operations, penalizing many students is not the most effective solution. In other words, we can say that with a plagiarism detector, we can **detect** plagiarism, but we can not **prevent** plagiarism.
- Tags described in the previous chapter will also be affected: students will copy-paste and will acquire many tags. Consequently, tags will not be able to highlight efficiently difficulties of students since a tutor consulting statistics will only see nice scores for each tag.

3.2 Overview

Our idea to prevent students from copy-pasting solutions of INGInious exercises would be to randomize these exercises by changing in the statement small elements so that each student will have to provide a different solution than his neighbor.

For instance, if two students A and B visit the task page of the same task, both students will see a different exercise but, that looks very similar. We could thus imagine an exercise in Java where the student A must implement a method to calculate the area of a rectangle and the student B a method to calculate the area of a circle. Students will therefore not be able to copy-paste their answers because they will not be the same.

To implement this mechanism, we allow tasks to generate random numbers for each student. These numbers can then be used to modify the context of exercises with some JavaScript and will also be made available to the `run` script to evaluate the task.

3.3 Examples

In this section, we provide two detailed examples of a task using our mechanism to randomize its statement and we show that if a student simply copies and pastes the answer of his neighbor, the copied solution will not work. We will also provide the code used to create the task to illustrate how we can use random inputs.

3.3.1 Hypotenuse

Statement

The first example consists of computing the length of the hypotenuse of a right-angled triangle using the famous formula $c^2 = a^2 + b^2$. The use of random inputs fits well in this kind of exercise where we have numerical data like a and b regarding the formula. So, in this exercise, we will use two random inputs that will allow us to vary the value of the initial a and b for each student. For example, a student will have to compute the length of the hypotenuse with a triangle whose $a = 3$ and the $b = 24$ and another student will have $a = 5$ and $b = 6$. We can see on figures 3.1 and 3.2 two illustrations of this exercise for two different students in the graphical interface of INGInious. Of course, if we consider a third student he will also have different values.

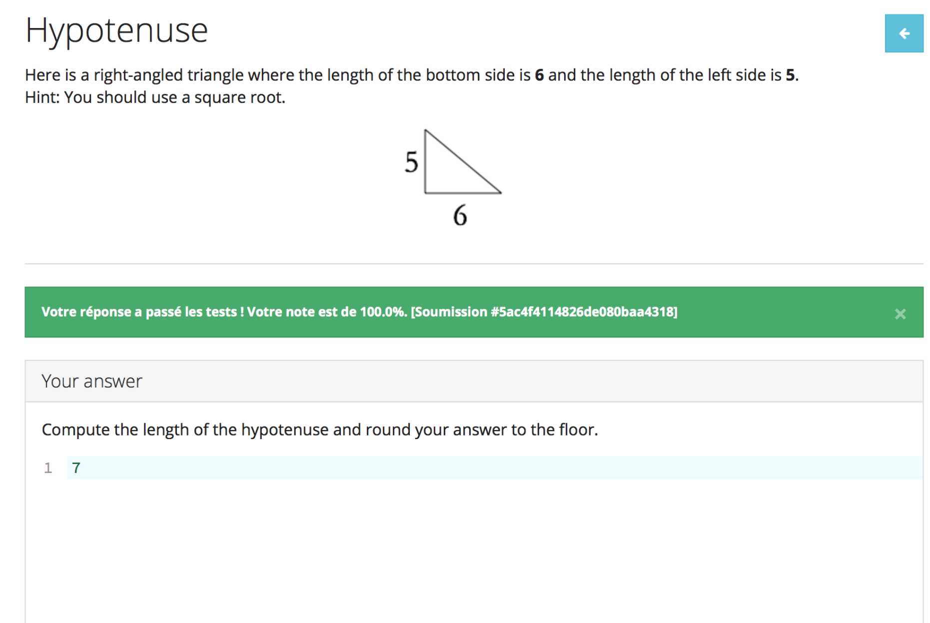
As we can see on figures 3.1 and 3.2, if student B copies the answer of student A and pastes it in his exercise, the answer will be incorrect because the expected answer is different thanks to the usage of random inputs.

Implementation

This task is composed of three parts: the main `run` script that will evaluate the answer of the student, an additional JavaScript file placed in the folder `/public` of the task that will update and modify the HTML based on our random inputs and finally the general statement of the task that uses raw-html to allow us to insert hidden HTML tag.

Statement The context of the task is presented in figure 3.3. As we can see, we have hidden in the context three HTML tag thanks to the `..raw::html` rst directive: one for the width of the triangle (line 3), another for the height of the triangle (line 4) and the

Figure 3.1: The statement on hypotenuse for student A



last, an HTML canvas to draw the triangle (line 6). Then, we have inserted our JavaScript script (line 8) that will edit the two values of the triangle and draw the triangle.

Additional JavaScript file To perform some live modifications of the context of the task, we need to write a small JavaScript script that will edit the HTML tags in the content of the task in function of generated random inputs. The script is present in figure 3.4. As we can see, we retrieve random inputs using `input["@random"][0]` (line 2) and `input["@random"][1]` (line 3). Since all random inputs generated by INGInious are floating point numbers in the range $[0.0, 1.0[$, we need to perform some mathematical tricks to choose the maximum and minimum width/height of the triangle. Then, we need to edit the two `span` tag by setting their content to the value of the width (line 7) end height (line 8). Finally, we draw the triangle (line 11 until the end) and we place two text label to describe graphically the width/height.

This script **must** be placed in the `/public`[1] folder of the task so that it is publicly accessible since the context of the task will fetch this file and insert it in its content (line 9 of figure 3.3).

Run file Figure 3.5 shows the `run` script of the task and how to retrieve random inputs (line 6 and 7) to be able to grade the answer of students based on the values generated by random inputs of INGInious.

[1] http://inginious.readthedocs.io/en/v0.4/teacher_doc/share_files.html

Figure 3.2: The statement on hypotenuse for student B

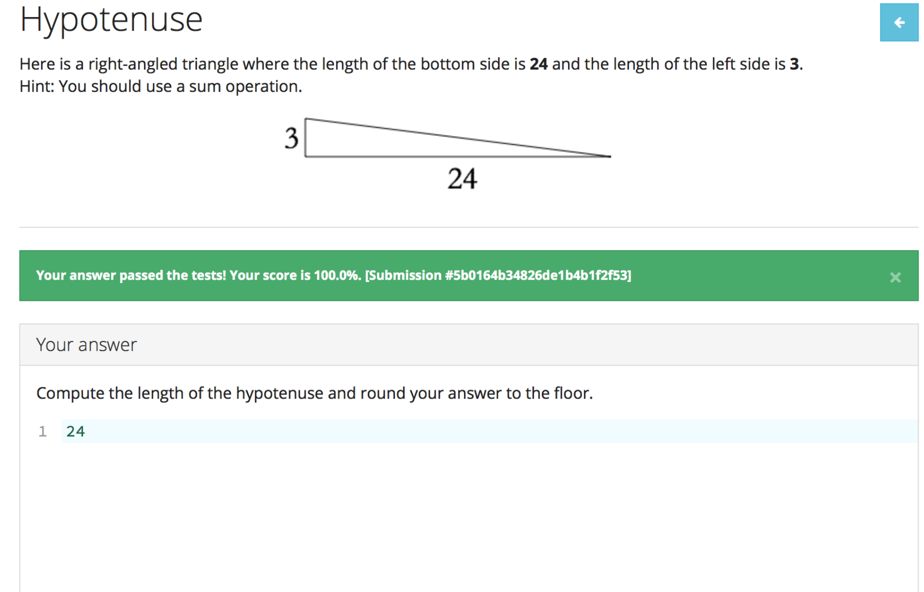


Figure 3.3: The context of the task

```
1  .. raw:: html
2      Here is a right-angled triangle where the length of the bottom side is
3          <b><span id="ipr1"></span></b> and the length of the left side is
4          <b><span id="ipr2"></span></b>.<br>
5      <div style="text-align:center">
6          <canvas id="canvas" width="100%"></canvas>
7      </div>
8      <script src="task_input_random_01/input_random.js"></script>
```

3.3.2 Calculate the area of a shape

Statement

The second example is a task where students have to write a function in Java with its signature to compute the area of a shape. This exercise fits well with random inputs since we can vary the shape that the student has to compute. We defined three shapes in the exercise that can vary: a square, a rectangle, and a circle. Of course, we can easily add more shapes but for this example, we limit ourselves to three shapes. We can see on figures 3.6, 3.7, and 3.8 the three different generated statements thanks to random inputs, one for each shape. Again, we can see through these three figures that random inputs allow us to vary a little bit the statement so that students will have different similar exercises but different solutions.

Implementation

Since the implementation contains lots of code and files because of the huge `run` file, the usage of the JUnit framework and Java reflection libraries, we will not explain here the

Figure 3.4: Additional JavaScript file

```
1   // Retrieve random inputs
2   var a = parseInt(input["@random"][0] * 25 + 5);
3   var b = parseInt(input["@random"][1] * 7 + 2);
4
5   // Retrieve HTML tag we placed in the statement
6   // and replace them by the generated random value
7   document.getElementById("ipr1").innerHTML = a;
8   document.getElementById("ipr2").innerHTML = b;
9
10  // Draw the triangle
11  var canvas = document.getElementById("canvas");
12  var size = 12;
13  var margin = 20;
14  canvas.width  = a*size+margin*3;
15  canvas.height = b*size+margin*3;
16
17  var context = canvas.getContext("2d");
18  context.moveTo(margin,margin);
19  context.lineTo(margin+a*size,margin+b*size);
20  context.moveTo(margin,margin);
21  context.lineTo(margin,margin+b*size);
22  context.moveTo(margin,margin+b*size);
23  context.lineTo(margin+a*size,margin+b*size);
24  context.stroke();
25
26  context.font = "30px Garamond";
27  context.fillText(a, margin+a*size/2 - 10, margin+b*size + 30);
28  context.fillText(b, 0, margin+b*size/2 + 10);
```

complete implementation of this task. Notice this task is based on the task `m03Q5` from the `CS1-Java` [ea18a] repository.

However, it is interesting to notice the usage of a `json` file placed in the `/public` folder of the task (figure 3.9). This file contains all possible strings displayed in the context of the task and is useful if we want to do more complex things like using strings in the context in contrary to the previous exercise that uses only numbers. The idea here is to use only one random input that indicates which object will be picked in the `json` to be displayed in the context (figure 3.10).

Figure 3.5: The `run` file

```python
#!/bin/python3
import subprocess, shlex, re, os, math
from inginious import feedback, rst, input

# Retrieve random inputs
a = int(input.get_input("@random")[0] * 25 + 5)
b = int(input.get_input("@random")[1] * 7 + 2)

try:
    response_user = int(input.get_input("q1"))
    if (response_user == int(math.sqrt(a*a+b*b))):
        feedback.set_grade(100)
        feedback.set_global_result("success")
    else:
        feedback.set_global_feedback("The hypotenuse is not " + str(response_user))
        feedback.set_grade(0)
        feedback.set_global_result("failed")
except ValueError:
    feedback.set_global_feedback("Your input is invalid. Enter a number rounded to the floor.")
    feedback.set_grade(0)
    feedback.set_global_result("failed")
```

3.4 Implementation

In this section, we describe how we modified INGInious.

3.4.1 Task editor

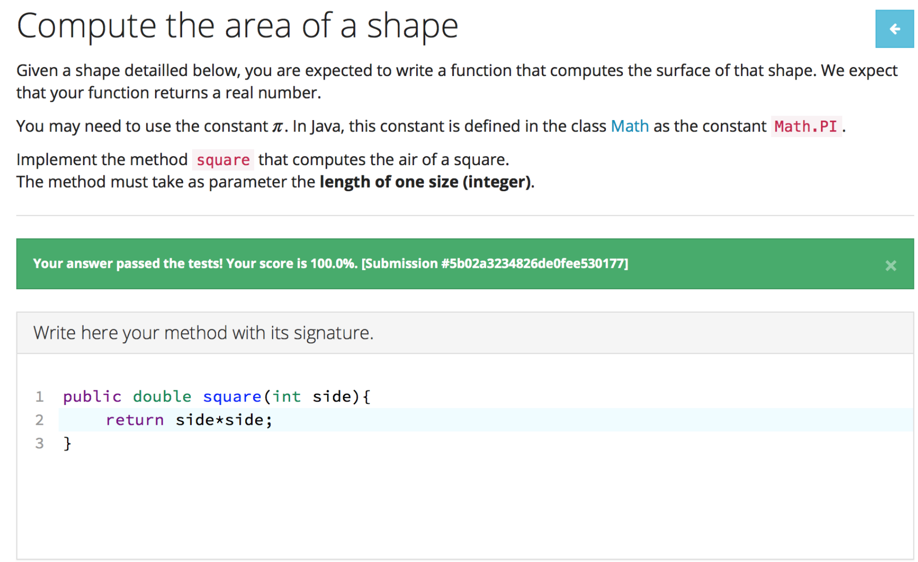
We first add in the task editor two new entries: one to specify the number of random inputs that the task will use and the second to specify if random inputs have to be regenerated each time a student refreshes the page (figure 3.11). Indeed, our mechanism has two "modes". The first mode permits to generate random inputs once for a student. If this student leaves the page and comes back again, we will have the same statement with the same random inputs. The second mode allows students to regenerate new random inputs each time they reload the page. This can be useful if students want to train themselves multiple times when the task is quick to do and the number of random inputs is very large like for example our previous example with the hypotenuse where we can generate hundred of different triangles.

3.4.2 Task page

We implemented two modifications in the task page.

The first modification was to generate random inputs when the task page is loaded. These random inputs are in fact simple floating point numbers in the range $[0.0, 1.0[$. The generation starts by defining a seed depending on the username, the id of the task and the id of the course (and also the time if random inputs have to be regenerated). Next, we generate a list of random numbers, in function of the number of random inputs configured

Figure 3.6: Calculate the area of a square



in the task (figure 3.11). This list is then saved in the database to keep the frontend stateless.

The second modification of the task page is the addition of a JavaScript script before the context of the task. This script in fact just contains an object with three fields: `"@lang"`, `"@username"` and `"@random"`. This variable called `input` is mandatory so that the JavaScript used to edit the context of the task based on random inputs has access to these random values (see figure 3.4, line 2 and 3).

3.4.3 Submission manager

When sending the input data of the user to the backend, the frontend also adds in the submission random inputs fetched from the database under the `"@random"` key. These values will then be accessible as a list in the `run` script by using `input.get_input("@random")` (see figure 3.5, line 6 and 7).

3.5 Test, evaluation and results

To test our mechanism, we prepare two samples of tasks. These tasks behave well in the latest version of INGInious. Moreover, Brandon Naitali is already using our `random inputs` mechanism to integrate it in conjugation exercises for his master thesis on adaptive testing in INGInious.

3.6 Conclusion

We saw in this chapter, how we designed and implemented an innovative solution to randomize exercises in INGInious in order to prevent plagiarism. We also provide two examples of exercises using random inputs to help future authors of tasks to integrate our mechanism in their tasks.

Figure 3.7: Calculate the area of a rectangle

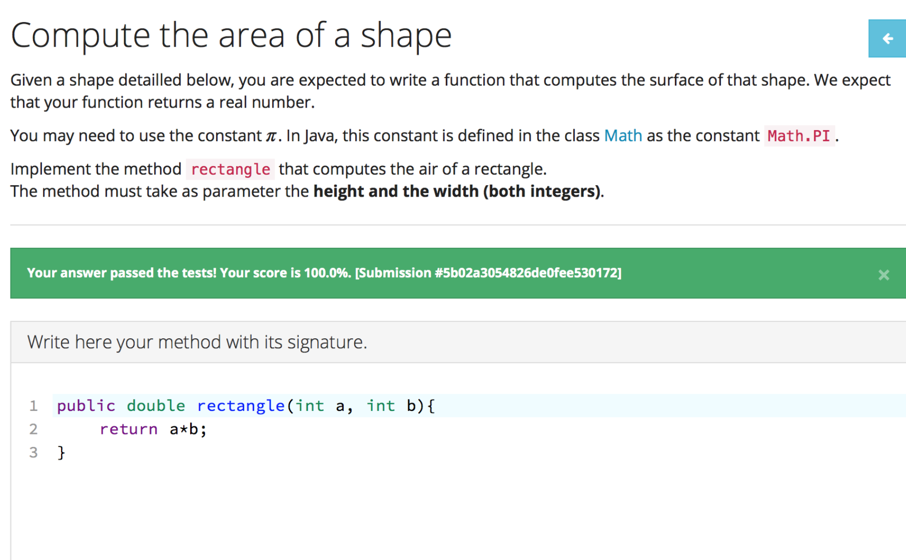


Figure 3.8: Calculate the area of a circle

Compute the area of a shape

Given a shape detailled below, you are expected to write a function that computes the surface of that shape. We expect that your function returns a real number.

You may need to use the constant π. In Java, this constant is defined in the class Math as the constant `Math.PI` .

Implement the method `circle` that computes the air of a circle.
The method must take as parameter the **radius of the circle (double)**.

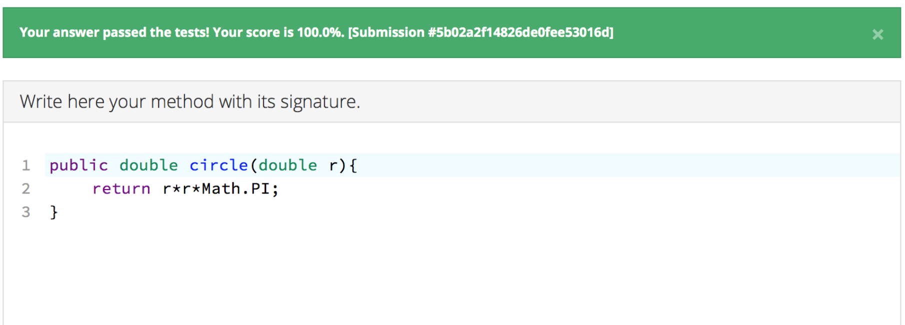


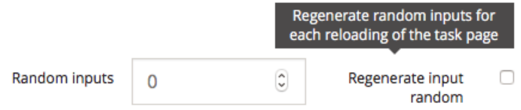
Figure 3.9: the /public/data.json file

```json
{
  "shapes": [
    {
        "name": "square",
        "parameters": "length of one size (integer)"
    },
    {
        "name": "rectangle",
        "parameters": "height and the width (both integers)"
    },
    {
        "name": "circle",
        "parameters": "radius of the circle (double)"
    }
  ]
}
```

Figure 3.10: The js file to edit the context

```javascript
// Load the JSON with JQuery
$.getJSON("m03Q5/data.json", function(data) {
    // We use the random input to get a random element of the JSON.
    var i = parseInt(input["@random"][0] * data.shapes.length);
    document.getElementById("method_name").innerHTML = data.shapes[i].name;
    document.getElementById("shape_name").innerHTML = data.shapes[i].name;
    document.getElementById("parameters").innerHTML = data.shapes[i].parameters;
});
```

Figure 3.11: New fields in the task editor for random inputs



CHAPTER 4

Conclusion

INGInious, a secure cloud platform providing automatic code grading to assess programming assignments, was extended to provide tools to allow tasks creators to label what students do, to give feedback about students to tutors and professors, to query accurately the submissions database, and finally to prevent plagiarism.

Our investigation of the Java course shows that tags are a very useful mechanism to identify difficulties of students and to improve the content of exercise thanks to statistics directly accessible on INGInious, calculated on the basis of these tags.

We provide below links to all work done for this master thesis.

4.1 Links

4.1.1 INGInious

Our implementation in INGInious contains several pull requests made on the public GitHub repository of INGInious (https://github.com/UCL-INGI/INGInious). All these pull requests have already been merged and put into production in the INGInious instance (https://inginious.info.ucl.ac.be) of Université catholique de Louvain.

- INGInious: tags, https://github.com/UCL-INGI/INGInious/pull/260, +833 −100.
- Submission viewer, https://github.com/UCL-INGI/INGInious/pull/275, +766 −453.
- Input random 1, https://github.com/UCL-INGI/INGInious/pull/300, +45 −2.
- Input random 2, https://github.com/UCL-INGI/INGInious/pull/311, +66 -33.

Our work also contains five other minor pull requests for bug fix and improvements.

4.1.2 Tasks samples

We prepared some task examples to help future authors of tasks to use our new mechanisms. These tasks are available on our personal GitHub repository: https://github.com/BIG-COW/INGInious_tasks-tag-random_inputs.

4.1.3 LSFIN1252 tasks

The repository https://github.com/UCL-INGI/LSINF1252 contains tasks written in C for the LSFIN1252 course given at Université catholique de Louvain. The tag mechanism was integrated in several of these tasks.

4.1.4 LFSAB1401 exam

We can not give sources of tasks used for the statistical part of the LFSAB1401 since these sources are confidential.

Bibliography

[ALV16] Alireza Ahadi, Raymond Lister, and Arto Vihavainen. On the number of attempts students made on some online programming exercises during semester and their subsequent performance on final exam questions. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, pages 218–223. ACM, 2016.

[aut17] The INGInious authors. What is INGInious? https://inginious.readthedocs.io/en/v0.4/teacher_doc/what_is_inginious.html, 2014-2017. [Online; accessed 2-May-2018].

[aut18a] The Stepik authors. Stepik. https://welcome.stepik.org, 2013-2018. [Online; accessed 2-May-2018].

[aut18b] The Stepik authors. What is Stepik? https://stepik.org/lesson/9184/step/1, 2018. [Online; accessed 2-May-2018].

[BP18a] Olivier Bonaventure and Charles Pecheur. Informatique 1: Mission 5 (restructuration), mission 6 (introduction). 2018.

[BP18b] Olivier Bonaventure and Charles Pecheur. Informatique 1: Mission 6 (restructuration), mission 7 (introduction). 2018.

[cdL18] Université catholique de Louvain. INGInious. https://www.inginious.org, 2014-2018. [Online; accessed 2-May-2018].

[Cig18] Richardson Ciguene. Génération automatique de sujets d'évaluation différenciés : vers une distribution optimisée en salle d'examen. In *Septièmes Rencontres Jeunes Chercheurs en EIAH (RJC EIAH 2018)* , Besançon, France, April 2018.

[ea18a] Olivier Bonaventure et al. INGInious tasks for introductory Java course. https://github.com/UCL-INGI/CS1-Java, 2017-2018. [Online; accessed 2-May-2018].

[ea18b] Olivier Bonaventure et al. INGInious tasks for the LSINF1252 course. https://github.com/UCL-INGI/LSINF1252, 2018. [Online; accessed 2-May-2018].

[HPa] Souleiman Ali Houssein and Yvan Peter. État de l'art des outils de soutien à l'enseignement/apprentissage de la programmation.

[HPb] Souleiman Ali Houssein12 and Yvan Peter. Outils d'assistance et les difficultés d'enseignement/apprentissage de la programmation, quelle aide?

[Les16] Daniel Lessner. Attitudes towards computer science in secondary education: Evaluation of an introductory course. In *International Conference on Informatics in Schools: Situation, Evolution, and Perspectives*, pages 53–64. Springer, 2016.

[Nie95] Jakob Nielsen. 10 usability heuristics for user interface design. *Nielsen Norman Group*, 1(1), 1995.

[TC16] Zsuzsanna Szalayné Tahy and Zoltán Czirkos. "why can't i learn programming?" the learning and teaching environment of programming. In *International Conference on Informatics in Schools: Situation, Evolution, and Perspectives*, pages 199–204. Springer, 2016.

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