TAT: teacher assisting tools

An application to manage teacher-student interactions

James Daniel Trevarton Jane

August 2023

supervised by
Dr. Pierre-Alain Cherix

`https://github.com/danjane/GymInf`

UNI FR
UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Gyminf
Département d'Informatique
Université de Fribourg (Suisse)

GymInf
Informatikausbildung für Gymnasiallehrkräfte
Formation en informatique destinée aux enseignant·e·s au gymnase

Contents

Acknowledgements

A big thank you to Pierre-Alain Cherix for his suggestions and advice. His years of experience with computers, teaching and teachers led to a very different, and much better, project than I had first envisaged.

Thanks to Chris Lawrence for discussions about test-driven development, Pat Jane for hilarious stories of systems design disasters, and Ruskin Constant for the basics of UX. I also thank my family: Sarah for her support, and Gabriel and Julia for destruction testing the GUI (and everything else in the house).

1 Introduction

From the classic teaching literature, Bucheton and Soulé have described the act of teaching as a multi-agenda game of postures requiring good preparation and excellent micro-decisions [BS09]. In their model, they identify the crucial roles of the teacher in controlling the cadence, the atmosphere, the scaffolding and the relationships during the class, and how each supports the learning objective. Many of the teacher's most time-consuming tasks do not take place in the classroom: good preparation and strong follow-ups (auto-reflection, marking, parent-teacher interactions) work to support and complement the overall success of the student learning process.

For example, scanning the offers of teaching posts we see that teacher responsibilities include the ability to

- plan and implement effective classroom management practices,
- design and implement effective strategies to develop independent learners,
- engage students in active, hands-on, creative problem-based learning,
- build students' ability to work collaboratively with others,
- maintain a safe, orderly environment conducive to learning,
- adapt instruction/support to students' differences in development, learning styles, strengths and needs, and
- write student reports to guide changes in instruction and practice, and to improve student learning.

As A. Sweigart points out in his 2015 book "Automate the boring stuff" many of the teaching tasks listed above are ripe for automation [Swei15]. I would also accept that some of these tasks should **not** be automated (even if they can be). As John Hattie explains in *Visible Learning* [Hat12], *"Expert teachers monitor learning and provide feedback."* In my opinion writing student reports are a perfect example of a necessary evil: although time consuming (and potentially stressful) for the teacher, writing a student report forces the teacher to reflect on the progress of the student and at the same time manage the expectations of all partners - student, teacher, management and parent.

So which tasks should be automated? Why? And for whom? When I first started teaching, my natural character led to two bad teaching practices: I found it difficult to engage with the quieter, more reserved students; and I was so busy answering student questions that I left little time for taking notes. I felt my teaching (and so hopefully also my students' learning experience) would benefit from a tool which tracked my interactions in class in an attempt to shift the focus away from the "louder" students.

"The best climate for learning is one in which there is trust. Students often don't like to make mistakes because they fear a negative response from peers. Expert teachers create classrooms in which errors are welcome and learning is cool." [Hat12]

J. Hattie finds evidence that answering quickly has a positive effect on how the students view the teacher, but basically no effect on student learning [Hat12, p.184]. A better practice is throwing the question back to the class and then allowing them time to think: J. Rowe identified many advantages in extending the "wait-time" from one second to about five seconds [Rowe74]. By taking the time to record the interaction at my desk I force myself to give the class this wait-time.

But if I start recording a brief comment at an opportune moment after a positive (or negative) interaction with a set of students, I could also use this to build a reminder of the interactions per student: a useful capability when planning lessons, writing reports, and especially for parent evenings. These reports would include a list of the pertinent interactions with the student in question as well as his exam notes.

For the Master's thesis project undertaken for the GymInf formation, I chose to build a suite of tools to support a range of teacher tasks including capturing key interaction information, building individual student reports, suggesting teacher-student interactions for upcoming classes, organising seating plans, and creating spreadsheets of marks. I called this system the **teacher assisting tools**, or TAT for short.

I will now explain the layout of this thesis. In section 3 I outline the planned architecture of the system, explaining how it can be built incrementally. While I sometimes take the opportunity to explore alternative solutions, in general I mainly explain and defend my decisions.

The tools will be coded in Python. This choice of language is mainly because Python is the language we teach our students, see section 4, and I would like to take this opportunity to consolidate my Python skillset. For the same reason, in section 6, I will defend my choice of PyGame for the graphical user interface (GUI), despite its many disadvantages. I then explain a little about the theory behind GUI design and how I planned the frontend of the TAT system.

I also wanted to use this thesis to improve my coding style, exploring the industry technique of *test-driven development* as explained in section 5. The majority of my contemporaries from university who ended up in industry have highly recommended this coding style. As discussed later, while it has some disadvantages there are strong reasons for having *test-driven development* as another tool in the toolkit[Amman16, p. 98].

In section 8 I have written a mock instruction booklet for the TAT system as it

currently operates. I based the instructions around user intentions, explaining how each action is achieved. All the functionality described in the instruction booklet has been delivered.

Ideally a TAT user need not be proficient with a computer. That is, any teacher who wanted to use the TAT system would be able to, regardless of their computing ability. This is why the system has a graphical user interface rather than relying on scripts run from the command line. I was asked to provide a single executable file and in section 7 I describe how this can be delivered.

Given the limited resources of this masters' thesis and my enthusiastic imagination, not all of the desired functionality has been completed. These missing features are described in section 9. Please note that the current codebase was designed with these features in mind, and that existing code should require minimal changes to incorporate these features as they are added.

Finally, in section 10 I conclude the thesis with a reflection on the strengths and weaknesses of the project, on what I've learnt and what I would have done differently.

The latest version of the code and this thesis can be found at

`https://github.com/danjane/GymInf`

2 Project scope

As Robinson points out in *BOSCARD: a scoping tool* [Rob19], for a project to be effective and efficient it is necessary (but alas not sufficient) to be clear about the project's aims from the start. BOSCARD is an acronym for background, objectives, scope, constraints, assumptions, risks, and deliverables. Although an industry standard, Robinson claims that the origins of this approach to project planning are unclear but likely originated with the consulting company Cap Gemini in the 1980s [Rob19, p. 181].

2.1 Background

Provide background information that includes the reasons for creating the project and mentions the key stakeholders who will benefit from the project result.

I am doing master's thesis in Computing and have chosen to create a suite of tools to aid teachers. These will be implemented through the *teacher assisting tools* system, called TAT. This will be aimed at assisting myself and other teachers in their daily work.

2.2 Objectives

Describe the project goals.

There are common mistakes made by many new teachers, for example

- answering student questions immediately rather than leaving time for the class to think,
- allowing a subset of students to monopolise class interactions, and
- taking inadequate notes during class.

I will build the TAT system to help teachers to overcome these issues. As well as recording student interaction information, the system will also build individual student reports, suggest future teacher-student interactions for upcoming classes, suggest seating plans (descoped), and create spreadsheets of marks.

2.3 Scope

Provide a high-level description of the features and functions that characterize the product, service, or result the project is meant to deliver.

TAT will be able to:

- Actions during teaching.
  - Record, during class, an interaction with a set of students.
  - Suggest students for focus, see section 3.2.1.
  - Manually modify the seating plan as required.
- Actions outside of teaching.
  - Prepare student reports.
  - Suggest a seating plan (descoped).
  - Review previous seating plans (descoped).
  - Prepare an empty spreadsheet for marking and noting an exam.
  - Collate marks and calculate semester note.
- Actions to set up the class.
  - Add a class to the list of classes taught.
  - Configure the class lists.
  - Suggest a seating plan (descoped).

The TAT backend will be linked to a GUI, run from a single executable file, with a local instance per teacher.

TAT will not connect to the internet. TAT will not cover data privacy concerns beyond what is currently used in standard teaching practices in Geneva, see section 3.3.10.

All development and testing will be done on my personal machine, a MacBook Air running the OS Ventura 13.4 (22F66) on an Apple M1 chip. This should be compatible with the examiner's personal computer, running Windows, but any work required to migrate to the school system will not be in the scope of this thesis.

2.4 Constraints

Identify the specific constraints or restrictions that limit or place conditions on the project, especially those associated with project scope.

The thesis must be defended and marked by September 8^{th}, 2023. Therefore the TAT system must be delivered to my thesis advisor before August 14^{th} 2023. I am writing this thesis and the code for the TAT system alone: while a basic functionality is essential for all objectives, I anticipate future enhancements and even functional additions will occur.

The TAT system will eventually run on the school computers. However, for this project all testing will be done on my local machine.

2.5 Assumptions

Specify all factors that are, for planning purposes, considered to be true. During the planning process, these assumptions will be validated.

I will assume that classes have at most 24 students, and that the seating plan is arranged in the standard three-by-four blocks of pairs.

2.6 Risks

Outline the risks identified at the start of the project. Include a quick assessment of the significance of each risk and how to deal with them.

Given the constraints, and especially the time limit, there is a large risk of some features being dropped from the first version. I want to first deliver basic functionality with a graphical user interface, and then add as many features as possible in the timeframe.

2.7 Deliverables

Define the key deliverables the project is required to produce to achieve the stated objectives.

The TAT system will provide a basic GUI over a backend handling the tasks covered in the scope. In particular, the system should at least allow notes to be taken against student names and then suggest pertinent students for focus.

3 System plan

In his 2017 book "Clean Architecture" Robert Martin[1] is clear about why we invest time in the planning phase of an IT system:

> *"The goal of software architecture is to minimize the human resources required to build and maintain the required system."* [Mart17, p. 5]

We can analyse a system by connecting its (physical or virtual) inputs and outputs. In this project, we have

Inputs

- Class lists
- Commentary on students
- Seating plan used in class
- Exam notes and weights

Outputs

- Suggested students for focus
- Suggested seating plans
- Average notes for the semester
- Individual student reports

The inputs clearly contain sensitive information, and the relevant laws and best practices with regards to student data will be discussed further in Section 3.3.10. It is also worth pointing out that the processed data (data during calculations and the outputs) is also sensitive. In general it is good practice to store as little personal data as possible.

3.1 Explanation of inputs

3.1.1 Class lists

For each class of interest, we need to have a list of students who are members of this class. For each student, it is often useful to store a "given" name with which they like to be called in class. Other information is not essential (gender, age, etc.), and so we will not store it.

[1] Robert Martin likes to be called "Uncle Bob".

3.1.2 Commentary on students

During the teaching process, the teacher makes useful judgements about students and subgroups of students. The teaching process obviously includes contact time during classes, but it can also include thoughts and decisions during the planning process, while marking homework or exams, or while evaluating a lesson *ex-post*.

Comments could take the form

- Excellent definition of Pythagoras' Theorem
- Good explanation of photosynthesis on blackboard
- Very quick with past continuous exercises
- Chatting
- DNF *(homework not done)*
- Seemed unashamed that he did not know the formula for the area of a triangle.

To each comment should be associated the student, the class and the date. Thinking ahead, It would be very useful to also include whether or not the comment was positive or negative: see the "focus" and "report" outputs, respectively sections 3.2.1 and 3.2.4.

3.1.3 Seating plan used in class

Especially at the start of the year, but also just after holidays, it can be difficult to remember the given names of students. By asking the students to follow a seating plan *and then having this seating plan to hand,* the teacher has an easy way to refer to students by their given name. At the start of the year this also helps the teacher in learning the names. See figure 1 on page 11 for an example of a seating plan created using the tikz package of LaTeX. As far as possible, I followed the format of seating plans used during exams at Collège Rousseau : thus the seating plan layout should be familiar to both students and teachers.

3.1.4 Exam notes

The individual exam results will be needed in order to calculate the average note for each student for each semester, section 3.2.3. We will also need the weighting associated with each exam, and by including the date of the exam we can check the progression of students, section 3.2.4.

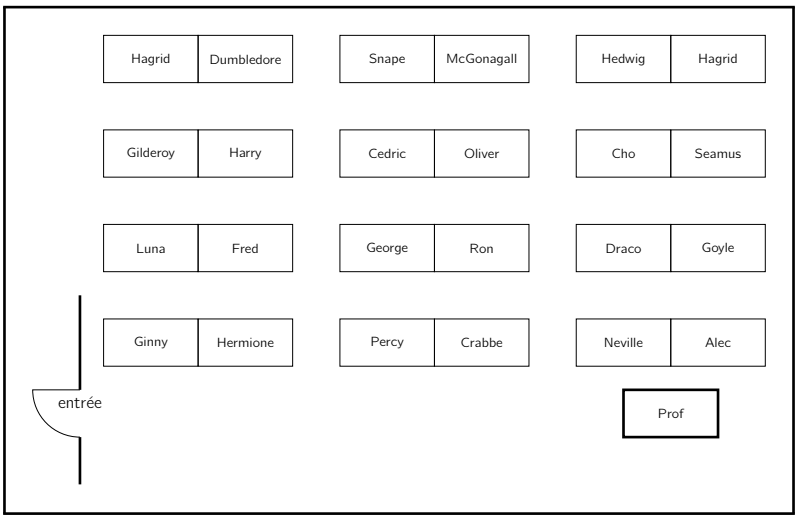


Figure 1: Example of a seating plan, created with the Tikz package of LaTeX

3.2 Explanation of outputs

3.2.1 Suggested students for focus

As a student teacher, I found it hard to balance my time equitably among the students in a class. Certain students, perhaps those who were louder or more confident, tended to capture my attention. By keeping a record of my interactions with students, it was possible to estimate an ordering of students by recent interactions, and so indicate students who require teacher *focus* during the next set of teacher-student interactions (probably during the next class).

This *focus* can be very simple: these are the students to whom the teacher addresses the opening questions. These opening questions can revise topics from the last lesson or prepare the students for the ideas to be tackled in this lesson. J. Larkin and H. Pines point out that students who respond positively to an early question are much more likely to volunteer answers later in the class [Larkin05]. They also suggest supporting shy students by announcing the question in advance of the lesson.

Let us model the interactions with a given student by simply counting the number of interactions on a given day.

Comments:
01/06/2023 Harry gives conclusion of Pythag
01/06/2023 Harry, Ron correct calc litt
04/06/2023 Ron calculates angles in triangle
08/06/2023 Harry calculates hypotenuse

We define a function $f_H : \text{Dates} \to \mathbb{N}$ to count the interactions with Harry

$$f_H(d) = \begin{cases} 2 & \text{if} \quad d = 01/06/2023 \\ 1 & \text{if} \quad d = 08/06/2023 \\ 0 & \quad \text{otherwise.} \end{cases}$$

and a corresponding function f_R for the interactions with Ron

$$f_R(d) = \begin{cases} 1 & \text{if} \quad d = 01/06/2023 \\ 1 & \text{if} \quad d = 04/06/2023 \\ 0 & \quad \text{otherwise.} \end{cases}$$

On a given date t we would like to create a weight operator, W_t, which would map these functions to a real number. This operator should measure the number of recent interactions, giving more weight if there have been more interactions. For example, Harry has had more interactions than Ron and so we expect $W_t(f_H) > W_t(f_R)$.

Consider calculating the students for focus just before the holidays or just after the holidays. Assuming the list of comments has not changed, then ideally the **ordering** of the students by weight would not change. Thus the students for focus should not be dependant on when the algorithm was run. A straightforward property with this behaviour is that all weights change by the same constant. In probability, this is called the *memoryless* property [Norr98] and implies an exponential weighting of the counts:

$$W_t(f) = \sum_{d \in \mathcal{D}} f(d)\, e^{k(d-t)}$$

where we sum over all dates[2] and $k > 0$ is a constant. Then recalculating the weights after n days gives

$$W_{t+n}(f) = \sum_{d \in \mathcal{D}} f(d)\, e^{k(d-(t+n))} = e^{-kn} \cdot \sum_{d \in \mathcal{D}} f(d)\, e^{k(d-t)} = e^{-kn} \cdot W_t(f),$$

so all weights are rescaled by e^{-kn} which does not depend on f.

To pick the constant k, consider the desired relative weighting between a student to whom we made a comment yesterday, and another who received two comments quite a while ago (but two comments on the same day). How much

[2] As the number of comments is finite this is really a finite sum.

time should pass in order that both students receive the same weight? Completely subjectively, I assume one week should pass, which gives

$$1e^{0k} = 2e^{-7k} \Rightarrow k = \frac{1}{7}\ln 2 \approx 0.1.$$

Ideally negative interactions should **not** be counted in this weighting; otherwise "difficult" students would be unlikely to ever be chosen for positive interactions.

3.2.2 Suggested seating plans

Most classes have 24 students arranged in 12 pairs, with the 12 pairs at desks arranged in 3 columns by 4 rows, see figure 1 on page 11. There are two obvious seating arrangements : using the ordering from the class list[3] or using a random seating plan. For testing and repeatability it is much easier to use a deterministic algorithm, and so I have preferred using the class list as a default seating plan.

As the year progresses we have more information about the students: pairs that work well together, students who work at similar speeds, students which have different strengths. Could we use the student comments and the exam results to suggest seating plans suited to particular lessons?

3.2.3 Average notes for the semester

The notes in Genevan institutions range between 1.5 and 6.0. The average note for the semester is a weighted sum of the individual notes of the marked exams of the semester. The individual notes are rounded to the nearest half, whereas the *semestriel* notes are rounded to the nearest tenth. Mathematically,

$$n_s = R_{0.1}\left(\frac{\sum_{i\in\mathcal{I}_s} w_i \cdot n_i}{\sum_{i\in\mathcal{I}_s} w_i}\right)$$

where the index runs over the marked exams of the semester, and R_s is the function that rounds to the nearest multiple of s.

It is worth pointing out that the rounding function is *not* continuous. While this is obvious, it can make the semestriel note surprisingly sensitive to small changes in individual exam notes. This also true for the end of year note, which is the **rounded** average of the semestriel notes:

$$n_y = R_{0.1}\left(\frac{n_{s_1} + n_{s_2}}{2}\right).$$

In general the pass mark is 4.0, and the rounding functions are a big advantage for weak students. Consider a concrete example, where before rounding a student

[3]The class list is alphabetic on surname. However, due to data privacy concerns the system does not have access to the surname: it only has the email username, see section 3.3.10. The alphabetic ordering of the usernames is not necessarily the same as the alphabetic ordering of the fullnames. To avoid this problem I use the class list ordering directly rather than the usernames.

receives 3.85 for the first semester and 3.95 for the second. Then the final end of year note is

$$n_y = R_{0.1}\left(\frac{R_{0.1}(3.85) + R_{0.1}(3.95)}{2}\right) = R_{0.1}\left(\frac{3.9 + 4.0}{2}\right) = R_{0.1}(3.95) = 4.0$$

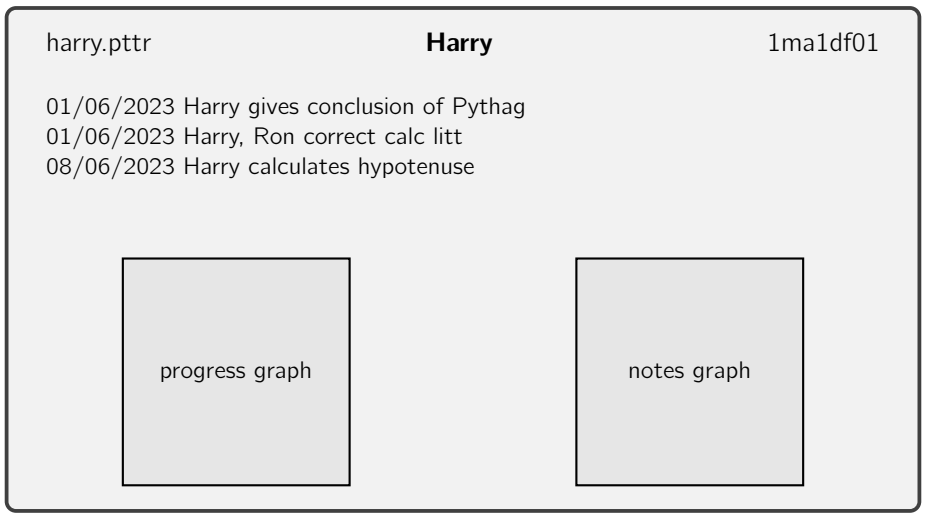
and the student passes! Fingers crossed this also applies to Master's theses.

I would like the system to calculate the semestriel and end of year notes for each student.

3.2.4 Individual student reports

It is often useful to have a quick overview of an individual student's progress. Perhaps beforehand, while planning which students will tackle which activities, or afterwards, when analysing the efficacy of a sequence of lessons. More concretely, such an overview would be very useful when meeting parents, when giving notes, and when asked to comment on their progress in the *conseils de classes* at the end of each semester.

Information that would be useful would be the student name, their given name, the class, a list of the comments concerning this student, their exam notes over time, and a graphical visualisation of their progress. Roughly, the overview would include an A4 page per student which looks something like this:



3.3 Implied structure of backend

By "backend" I mean the data access layer of the system. By connecting the inputs and outputs we can assess what intermediate processing will be required,

and what functionality can be shared. The schematic in figure 2, on page 15, is a *Data Flow Diagram* for the TAT system. I have used the Yourdon-Coad notation where the processes (functions) are circles and data is represented by rectangles. Flow of control (transfer of information) is represented by arrows, see [Coad91].

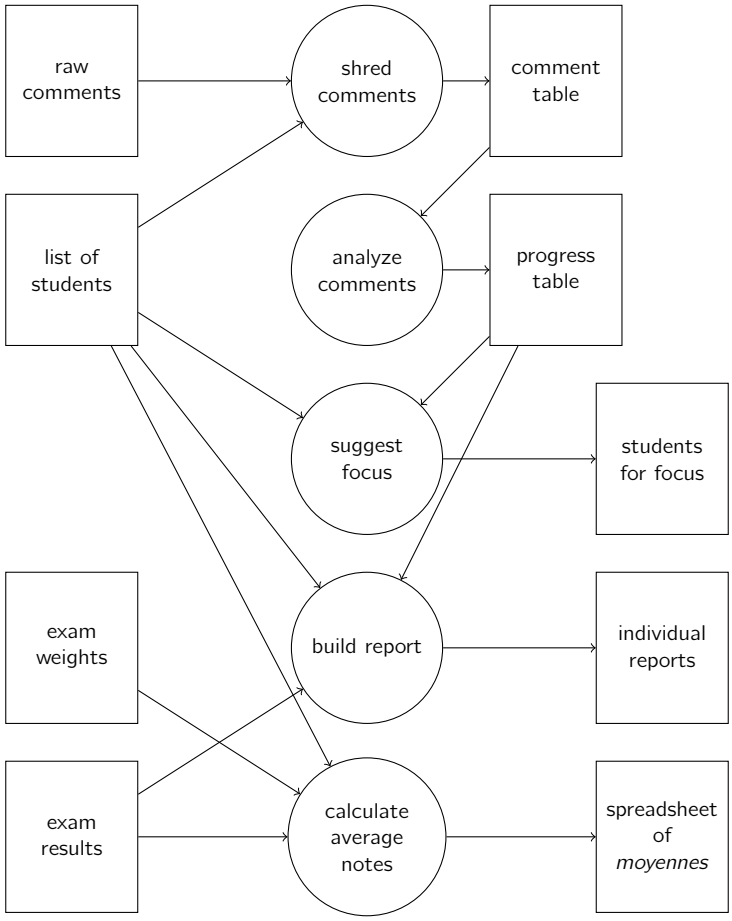


Figure 2: Data flow diagram for the backend of the TAT system.

The data flow diagram in figure 2, on page 15, is a very high-level description of a possible system. To make concrete decisions about how to implement the TAT system it will help to define the expected inputs and outputs. In this section I will outline a basic backend based on files. An implementation based on files is simpler and quicker, and so more suited for a master's project, but in the future I would expect the backend to handle updates via ACID database queries. I will briefly explain the advantages of databases over files in subsection 3.3.9. I will finish this section by touching on the data privacy laws that apply in Geneva; in

particular the system will use only a minimum amount of personal data.

3.3.1 The list of students

The **list of students** is a list of students in each class. I will use a text file for each class, storing a list of user-ids for each student. For example, for the class 1in1dfb01 I would create a textfile called "1in1dfb01.txt":

1in1dfb01.txt

```
harry.pttr
ronald.wsly
marie.cr
richard.fynmn
```

This also seems a pertinent place to store the given name of each student. Currently the user-id starts with the first name of the student followed by a fullstop followed by (roughly) the consonants of the surname,

```
user_id = firstname + "." + consonants(surname).
```

A regular expression of the form `^[a-z\-]+` strips the student's first name from their user-id. In Python, we use the `re` package, for example

```
firstname = re.search(r"^[a-z\-]+", user_id).group(0).capitalize()
```

Applying this function to the students above would yield

```
["Harry", "Ronald", "Marie", "Richard"]
```

However, there are two issues with assuming that this function will always yield the given name of a student. Firstly, we do not control how the user-ids are created, and so we cannot guarantee that this function will work in the future. Secondly, a student might want to be called by a different name (for example their second name).

I decided to add an optional comma-separated second item which designates the given name of a student in a class: this will be the name shown on the seating plan. For example, if Ronald Weasley wished to be called "Ron" and Richard Feynman preferred "Dick", then the file for the class list would be

1in1dfb01.txt

```
harry.pttr
ronald.wsly, Ron
marie.cr
richard.fynmn, Dick
```

3.3.2 Raw comments

The **raw comments** was originally stored in a text file designed to be read and updated by the teacher. For this project, I have used the same design. A comment is written on a new line in the file, prefaced with a "+" or "-" depending on whether the comment is positive or negative. Then I include a list of students, followed by a comment.[4]

Examples of positive comments:

```
+Harry gives conclusion of Pythag
+Harry, Ron correct calc litt for $(a+b)^2$
```

Examples of negative comments:

```
-Harry moaning about scar
-Harry, Ron chatting
```

Before lines of comments there must be a line with the name of the class and, independently, the date. So a valid state for the comments file might be something like the following

comments_file_v0.1.txt

```
01Apr2023
1indfb01
+Harry gives conclusion of Pythag
+Harry, Ron correct calc litt for $(a+b)^2$

2indfb01
+George TN for Djikstra

02Apr2023
1indfb01
-Ron DNF
+Harry, Ron TN for Pythagoras
```

(Note that the vertical spaces are optional and aid human readability.)

(TN is my shorthand for *"l'élève explique au tableau noir"*.)

Given that this is a first implementation, but that nevertheless we want an efficient backend, we can improve the parsing of this file by demanding that the first character in each line defines the information that this line will contain:

- "+" or "-" will be followed by a comma separated list of students and a comment.
- "d" will be followed by a date in "ddMMMyyyy" format.
- "c" will be be followed by a class name.

[4] The text in the comment can handle LaTeX commands

comments_file_v1.1.txt

```
d01Apr2023
c1indfb01
+Harry gives conclusion of Pythag
+Harry, Ron correct calc litt for $(a+b)^2$

c2indfb01
+George TN for Djikstra

d02Apr2023
c1indfb01
-Ron DNF
+Harry, Ron TN for Pythagoras
```

3.3.3 Comment table

The **comment table** is a pandas data frame[5] created from the raw comments. The *shred comments* functions read the raw comments from top to bottom, scraping the information in a comment along with the associated student, date, the course name, and the sentiment (whether the comment is positive or negative). A comment in the raw comments file can concern multiple students, so if necessary a line in the data frame is duplicated for each individual student. An example is given on page 19, restricting to the columns "Student", "Date", "Course", "Info" and "Sentiment".

NB: I had hoped that the TAT system would also keep track of missed homeworks (*devoirs non-fait* in French, hence "DNF" in the comments). I have decided to punish students with a electronic *renvoi* for every second homework missed. Of course, it is quite a lot of work to track which students have missed a second homework. The TAT system can easily count the DNFs through time which eliminates this bureaucracy (it also avoids mistakes and so is fairer.) The counting of DNFs is implemented in the backend but not yet wired up in the GUI, and so I do not mention it elsewhere in this thesis.

[5] https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.html

comments_file_v1.1.txt

```
d01Apr2023
c1indfb01
+Harry gives conclusion of Pythag
+Harry, Ron correct calc litt for $(a+b)^2$

c2indfb01
+George TN for Djikstra

d02Apr2023
c1indfb01
-Ron DNF
+Harry, Ron TN for Pythagoras
```

comments_table_DataFrame $\subset$ progress_table_DataFrame

Student	Date	Course	Info	Sentiment	Weight	Progress
harry.pttr	01Apr2023	1indfb01	+Harry gives conclusion of Pythag	1	0.9	1
harry.pttr	01Apr2023	1indfb01	+Harry, Ron correct calc litt for $(a+b)^2$	1	0.9	2
ronald.wsl	01Apr2023	1indfb01	+Harry, Ron correct calc litt for $(a+b)^2$	1	0.9	1
george.wsl	01Apr2023	2indfb01	+George TN for Djikstra	1	0.9	1
ronald.wsl	02Apr2023	1indfb01	-Ron DNF	-1	0.0	0
harry.pttr	02Apr2023	1indfb01	+Harry, Ron TN for Pythagoras	1	1.0	3
ronald.wsl	02Apr2023	1indfb01	+Harry, Ron TN for Pythagoras	1	1.0	1

3.3.4 Progress table

In the **comments table** the columns were "Student", "Date", "Course", "Info" and "Sentiment". To these columns we adjoin the "Weight" and "Progress" columns to create **progress table**, which (like the comments table) is also implemented as a pandas DataFrame. See the example on page 19.

The *weight* of a positive comment is a memoryless function of the time duration between now and the date associated with the comment, see subsection 3.2.1 on page 11:

$$\text{weight} = e^{-kd}$$

where d is the age of the comment in calendar days and $k = 0.1$ is a constant. For example, the weight of a comment made today is $e^{-0.1 \cdot 0} = 1$ while the weight of a comment made yesterday is $e^{-0.1 \cdot 1} \approx 0.9$. Comments with a negative sentiment have a weight of 0 (forgive and forget).

The *progress* is the running cumulative sum of sentiments filtered on that student in that class. If we filter the progress table on `harry.ptter` in course `1indfb01` then we see

Date	Info	Sentiment	Weight	Progress
01Apr2023	+Harry gives conclusion of Pythag	1	0.9	1
01Apr2023	+Harry, Ron correct calc litt for $(a+b)^2$	1	0.9	2
02Apr2023	+Harry, Ron TN for Pythagore	1	1.0	3

3.3.5 Students for focus

We sort the set of students in a given course to return a list of **students for focus**, a list ordered in increasing need for teacher-student interactions. This "need" is approximated by a function of the distribution of the comments over time, as explained in subsection 3.2.1 on page 11. In practice, it is simply the sum of the weights in the progress table. Referring to the example on page 19, the sum of weights for the students in class `1indfb01` is

$$\{\texttt{"harry.pttr": 2.8, "ronald.wsl": 1.9}\}.$$

So as a list ordered on increasing weight, the students for focus would be

$$\texttt{["ronald.wsl", "harry.pttr"]}.$$

Of course, other students in this course have a default weight of zero if there are currently no comments associated to them. So if the course also included `"hermione.grngr"` then the ordered list of students for focus would be

$$\texttt{["hermione.grngr", "ronald.wsl", "harry.pttr"]}.$$

3.3.6 Report

The **report** is a pdf document containing an A4 page per student. On each page we have the student-id, the student's given name, the course, and a list of comments associated with this student. The TAT system first creates a text-based .tex document, and then calls a system function which executes pdflatex[6] to typeset the report as a .pdf document.

3.3.7 Spreadsheet of moyennes

The TAT system can shred spreadsheets of individual exams to create a table with the exam name, exam date, course, student-ids and associated notes. It then creates a **spreadsheet of moyennes** with a worksheet per course and a table of notes with the exams as columns and the student-ids as rows.

So far, the description above explains how a spreadsheet of static data is created (albeit static data collated in a useful single file). The TAT system goes further, automatically creating a table of cells containing formulae (rather than static data) which calculate the student *moyennes* for the provisional notes (NIPs), the first semester notes (S1), the second semester notes (S2) and the implied end of year note (EOY). This allows the teacher to fill-in or correct individual exams and see the effect on the notes to be officially declared. Further, by leaving rows for the individual exam weights the teacher has the flexibility to specify which exams are noted and which are formative, which exams count double, as well as choosing the weight of the semestriel exams.

The algebra for the calculation of the notes was given in subsection 3.2.3, on page 13. The equivalent spreadsheet formulae are

- `"RAWNOTE=SUMPRODUCT(EXAMWEIGHTS, EXAMNOTES)/SUM(EXAMWEIGHTS),`
- `"NOTE_Sx=ROUND(RAWNOTE*10)/10"`, and
- `"NOTE_EOY=ROUND(AVERAGE(NOTE_S1;NOTE_S2)*10)/10"`.

I also made the TAT system create the workbook with conditional formatting applied to the cells, so that notes less than the passing mark of 4.0 are flagged in red. To handle missing notes (for example, when students were absent), I use a default value of -100 as I did not want to expose non-scientific users to the joys of `NaN`, the representation of the idea of "Not a Number". The value of -100 is sufficiently negative to cause obvious problems throughout the calculations.

3.3.8 Exam results

In order to shred individual exams that TAT system assumes that the notes are stored in spreadsheets named with the convention `EXAMDATE_EXAMNAME_Notes` with extension either `.ods`, `.xls` or `.xlsx`. The files for a single course should

[6] `https://www.tug.org/applications/pdftex/`

be grouped together in a subdirectory with the same name as the course name. Each spreadsheet must contain a cell in the first column containing the keyword "Student", followed by a column of student-ids. There should be a cell in the first row containing the keyword "Note", and the notes are in the corresponding intersection of the student-id row and the note column. These are notes for an individual exam, and so should be rounded to the nearest multiple of 0.5.[7]

So an example for an exam filename might `20Apr2020_ExampleExam_Notes.ods` found in the directory `.\1indfb01`.

3.3.9 Databases

In the Genevan *plan d'études* one third of the second year is devoted to databases (ten out of thirty contact hours). This project would have been excellent opportunity for me to practice planning, creating, updating and querying tables programmatically: the course lists, the comment table and the progress table lend themselves naturally to being stored and treated as tables in a database (indeed, that's why I named them as tables).

Using a database has a number of advantages over a system based on flat files, including:

1. handling scalability and concurrency,
2. data organisation and structure,
3. providing automated backup and recovery programs,
4. offering user authentication and enhanced data security, as well as
5. data consistency based on ACID transactions.

The first four reasons are less important for small, individual projects such as this masters. The last reason, however, would have been useful: ACID stands for Atomicity, Consistency, Isolation, and Durability. Consistency promises that the databases is always in a valid state, even if the system crashes during an update. The TAT system does not make this promise with flat text files: if the system only adds the first few letters of a course name in the raw comments file, for example, the TAT system will not be able to restart without a manual intervention.

[7] Individual exam notes are given to the nearest half except in very special cases. One such exception is for controlled exams with an external jury: where the teacher and the jury agree to disagree, the exam note is the average (rounded to the nearest tenth) of the two notes (rounded to the nearest half).

comments_file_BROKEN.txt

```
d01Apr2023
c1indfb01
+Harry gives conclusion of Pythag
+Harry, Ron correct calc litt for $(a+b)^2$

c2ind
```

In spite of these advantages I decided to leave the TAT system based on flat files. First, there is the non-negligible cost and risk of migration (although the risk is minimised with a good testing framework, see 5 on page 26). Second, for novice users the readable files make adjustments much easier while the GUI is not fully refined. Thirdly, as explained in the next section, by basing the system on files which could be reasonably expected to be found on an average teachers' computer, we are still within the remit of current data privacy recommendations. If the system began storing the data in a novel way then it would possibly need to be reassessed by the *Préposé cantonal à la protection des données et à la transparence* (PPDT)[8].

3.3.10 Data privacy

In the course "Security and Confidentiality" given by Linus Gasser at EPFL, all GymInf students have studied the most relevant laws governing IT systems. In Geneva, for example, we must be follow the Swiss "Federal Act on Data Protection"[9], called **LIPAD** by the Genevan authorities, and we should also check that we do not have to follow the much more stringent European "General Data Protection Regulation"[10], called the **GDPR**, which is extraterritorial.

Luckily, public schools in the Genevan canton do not need to follow the GDPR. This was decided by the *Préposé cantonal à la protection des données et à la transparence* (PPDT)[11] in 2018:

> *"Ainsi, ne sont en principe pas soumises au RGPD les situations suivantes... L'instruction publique accueille des élèves qui résident et/ou ont la nationalité d'un Etat membre de l'UE, sans avoir fait de promotion sur le territoire de l'UE."*[PPDT18, p. 3]

In the same document the PPDT reminds us that we should follow LIPAD with respect to sensitive personal data:

"Les institutions publiques genevoises sont soumises à la LIPAD, s'agissant du traitement des données personnelles (art. 3 LIPAD). Elles doivent donc respecter

[8] https://www.ge.ch/organisation/protection-donnees-transparence
[9] https://www.fedlex.admin.ch/eli/cc/1993/1945_1945_1945/en
[10] https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX%3A02016R0679-20160504
[11] https://www.ge.ch/organisation/protection-donnees-transparence

les dispositions prévues par cette loi dans tout traitement de données personnelles."[PPDT18, p. 1]

The LIPAD defines sensitive personal data as

1. religious, ideological, political or trade union-related views or activities,
2. health, the intimate sphere or the racial origin,
3. social security measures,
4. administrative or criminal proceedings and sanctions; and also
5. a *personality profile*: a collection of data that permits an assessment of essential characteristics of the personality of a natural person.

As implemented, the TAT system uses the same information that an average teacher would already be storing about their students in order to carry out their duties. Therefore the TAT system stays within the current framework and findings. However, if we collated much larger amounts of student information in a single database it might be argued that this personal data constituted a *personality profile*, which would be covered by the LIPAD. To avoid having to check this during the masters project, I decided not to implement a database and instead make use of existing files.

4 Choice of programming language

4.1 Previous personal experience

I have experience programming in a number of languages : I used **Basic** and a bit of **assembler** as a child, then **Delphi** (Visual Pascal) as a teenager, next **C++** as a grad student, and then I used **Java** and **Matlab** extensively when I worked in industry. I tried **Scala**, **Clojure** and **Julia** during my years transitioning to teaching, and more recently during my GymInf studies I have also had to learn **Python** and **Prolog**.

In general, apart from Prolog, these languages feel quite similar. Some are more weighted towards object-oriented programming (Delphi, Java and Scala) but in general the virtual machine can be modelled as storing the program as a text file and executing the instructions line by line. Variables are declared and assigned values, and expressions are evaluated using iterative rules. Except for Basic and Assembler, which still allow Djikstra's nemesis the `GOTO` command, the virtual machine has control of the general flow of the execution using *functions* or *events* (the latter are in some sense just functions owned or controlled by objects). If we ignore naughty tricks, the programmer can only use switch statements, `if-elif-else`, `switch`, or `case`, to adjust the execution path at a very local level.

The other distinction between the languages is whether or not they are *functional*. Functional languages do not allow the redefinition of a variable, such as Prolog, Clojure and Scala.

I considered the following factors when choosing the language for my project:

- my current proficiency,
- my interest for improving my proficiency,
- ease for finding solutions to coding problems,
- readability of the code,
- maintainability of the code, and
- future applications of improved coding abilities.

There are other factors which did not affect my decision at the start of coding : for example, creating the GUI (graphical user interface), deployability of the code, the speed and reactivity of the program, or whether the code would be scalable. I will comment on these oversights in the conclusion.

Alongside **Scala** (syntactic sugar over Java with a functional feel) and **Julia** (Matlab maths functionality built directly on C++), the other *new* languages I considered using were **Swift**, **Clojure** and **Rust**.

4.2 Python

At Collège Rousseau, where I teach, we decided to teach the students Python. This was a difficult decision, with each language having advantages and disadvantages. Like the vast majority of Secondary II schools in Switzerland, we decided that the low barrier to entry, the wide use of Python in industry, the teachers' current abilities, the availability of packages, the large support community, and the focus on readability narrowly outweighed the use of weak (duck) typing with beginners.

The teacher is expected to have a strong background in the coding language being taught :

- for the pedagogical benefits of "live coding", modelling how to create code for the students to discuss and learn from [Raj2018],
- creating well written code for students to study,
- for ease of marking and correcting (lots of) student code,
- for spotting mistakes in student code during class time, in order to suggest hints,
- for recognising bad coding practices and explaining to students the potential pitfalls and how to avoid them, and
- for understanding error messages.

5 Test-Driven Development

The process of developing complex IT systems involves various stages, from understanding client requirements to delivering and maintaining the final product. Traditionally, software development followed the waterfall method [Amman16, p. 53], where each phase was executed sequentially:

- requirements analysis,
- design,
- implementation,
- testing, and
- deployment.

In their 2016 book, Ammann and Offutt speculate that the waterfall method originated with software development for large military projects. Here software was expected to have a long lifetime with no maintenance [Amman16, p. 96]. However, in industry this approach had several drawbacks, such as a long development cycle and limited client involvement until the final stages.

Reviews with the clients at a late stage of the development cycle led to misunderstandings and mismatches between the final product and client expectations. This resulted in costly redesigns that would have been cheaper if they had been identified earlier in the process, or even projects being dropped altogether. In 2004, a study showed that 45% of IT projects in the US failed to meet user requirements [LX10, p. 88]. In this section I explore how industry practitioners have tried to adapt the waterfall process and the introduction of test-driven development.

5.1 Reducing the distance between clients and developers

As a response to costly overruns the **Agile** methodologies emerged in the early 2000s, emphasising iterative development and client collaboration [Anaya18, Beck03]. The adoption of Agile methodologies aimed to bridge the gap between clients and developers throughout the process. Agile principles emphasise frequent communication, iterative development, and the delivery of working software in short cycles known as *sprints*. I would argue that autonomous teams of developers were encouraged to take control of the waterfall method themselves, but on a timescale of weeks rather than years. This approach allowed clients to see progress regularly, provide feedback, and steer the development process in the right direction.

Note that even with Agile principles it is still difficult to ensure that the delivered software precisely meets the client's needs. Further, as G. Lee and W. Xia pointed out in 2010, much of the alleged advantages of Agile development are anecdotal rather than scientifically proven [LX10]. They found that there is a difference between being able to quickly respond to a particular change in user requirement

(response efficiency), compared to being able to respond to as many changes as possible (response extensiveness). Encouragingly, with this distinction they did find evidence that agile teams with more autonomy had greater response efficiency, and that response efficiency positively effects all measures of project success [LX10, pp. 97-98].

5.2 Testing

The shorter sprints and early involvement of clients may have led to developers becoming used to the idea of shipping beta versions of their software, rather than aiming for a final version. Another problem independently emerging in the 1990s was the increasing cost of *maintaining* delivered systems. For these reasons the **testing** of software evolved.

Ammann and Offutt state that *"faults in software are design mistakes. . . they exist as a result of a decision by a human"* [Amman16, p. 29]. Since humans make mistakes, we must assume that software contains bugs. This in turn means we cannot prove that code is correct.[12] So why do we test code? Ammann and Offutt identify five levels of increasing understanding to this question:

1. There is no difference between testing and debugging.
2. The purpose of testing is to show correctness.
3. The purpose of testing is to show that the software does not work.
4. The purpose of testing is to reduce the risk of using the software.
5. Testing is a mental discipline that helps all IT professionals develop higher-quality software.

There are different types of testing. **Unit testing** compares inputs and outputs to check the functions behave as expected in a finite set of situations. Choosing these situations to cover as many interesting use-cases as possible becomes part of mental discipline mentioned above. **Integration testing** focuses on how different modules interact. **Regression testing** compares previous outputs against current outputs after a software update. **User acceptance testing** involves clients evaluating the system to determine if it meets their expectations.

While this approach to correctness might strike mathematicians as impoverished, software is built by engineers, and guess what? Engineers are not mathematicians! [Amman16, p. 98]

But there is still a problem: once the code is developed, who is doing the testing? If the developers and the testers are different teams, then it is hard to

[12] This is a long way from E. Djikstra's ideal of proving correctness of code, but the complexity of even moderately sized programs rules this out in practice.

avoid a level 3 mindset (showing that the software does not work). But if the developers test their own code there is a risk of a level 2 mindset (testing shows correctness, which is impossible).

5.3 TDD: Test-Driven Development

To avoid these issues altogether and hence promote a level 5 mindset (that testing is a mental discipline which improves coding) K. Beck and his collaborators suggested that the developers should write their own tests *before* they wrote their code [Beck03]. This practice became known as **test-driven development**, or TDD. To help developers use this counterintuitive practice, there is a simple cycle to follow:

- **Red.** Write a failing test for a requirement.
- **Green.** As quickly as possible, write code to pass the test.
- **Refactor.** Clean the code, removing duplication and renaming functions.

Coding is driven by testing.

Let us consider an explicit example from the TAT system: running the application for the first time, when there are no files or folders pre-created. We expect the TAT application to run with no courses listed, ready for the user to add courses (and students), and we also expect that the folder structure is automatically set up to handle the future files. Currently the GUI crashes almost immediately - it does not find the expected `config.yaml` config file. So let's use TDD to write code to create a default config file.

5.3.1 Red

The desired code lives naturally in the module `config.py`, but first we write the test in `tests/config_test.py`:

```
1 import config
2 import os
3
4
5 def test_create_default_cfg_file(tmp_path):
6     f = os.path.join(tmp_path, "config.yaml")
7     config.default(f)
8     assert os.path.isfile(f)
```

Note that `tmp_path` is a PyTest fixture which creates a temporary folder for this test: PyTest handles setting up and tearing down the temporary folder. We run the PyTest suite and, of course, the test crashes with `AttributeError: module 'config' has no attribute 'default'`. However, this is not truly a failing test - the test **crashed** rather than an assertion failing. We need to write an empty function in `config.py`:

```
1  def default(f):
2      pass
```

Now PyTest fails with an `AssertionError`, so we have successfully completed the **Red** part of the cycle.

5.3.2 Green

Now we code the function `default.py`.

```
1  import os
2
3  def default(f):
4      open(f, 'w').close
```

We run PyTest again and all tests trivially pass. Yay!

5.3.3 Refactor

In this case the code is already as simple as possible. **We're done.** This feels wrong, since we know that we have created an empty file instead of a working config file. But since the tests are passing we are not (yet) forced to write the correct code and so we do not.

5.3.4 Red

Now we have the idea, we can try moving a little faster. The new default config file should contain paths to the folder which will hold the exam results. We would like a function which sets up the basic structure, so we start with the `exams` subdirectory.

```
1  import config
2  import os
3
4
5  def test_create_default_cfg_file(tmp_path):
6      f = os.path.join(tmp_path, , "config.yaml")
7      config.default(f)
8      assert os.path.isfile(f)
9
10
11 def test_basic_structure_from_cfg_file(tmp_path):
12     f = os.path.join(tmp_path, "config.yaml")
13     exam_dir = os.path.join(tmp_path, "exams")
14     config.default(f)
15     config.setup_from_cfg_file(f)
16     assert os.path.isdir(exam_dir)
```

```
def default(f):
    open(f, 'w').close()


def setup_from_cfg_file(f):
    pass
```

As well as the test, I've written a placeholder for the new function. PyTest fails with an `AssertionError`, so we have completed the **Red** phase.

5.3.5 Green

We write a truly horrific bit of code to make the test pass:

```
import os


def default(f):
    open(f, 'w').close()


def setup_from_cfg_file(f):
    os.mkdir(f.replace("config.yaml", "exams"))
```

PyTest is happy (even if you are not), so **Green** is complete.

5.3.6 Refactor

You may want to sit down at this point: there is refactoring to be done, but it's not in `config.py`. We refactor the tests in `test_config.py`:

```
import config
import os


def create_cfg_file(path):
    f = os.path.join(path, "config.yaml")
    config.default(f)
    return f


def test_create_default_cfg_file(tmp_path):
    assert os.path.isfile(create_cfg_file(tmp_path))


def test_basic_structure_from_cfg_file(tmp_path):
```

```
16      f = create_cfg_file(tmp_path)
17      config.setup_from_cfg_file(f)
18      exam_dir = os.path.join(tmp_path, "exams")
19      assert os.path.isdir(exam_dir)
```

We rerun the tests to check that the refactoring does not break any of the tests, and the **Refactor** is done.

5.3.7 Red

Before we've tried test-driven development, this feels completely wrong. How can we still have the code `os.mkdir(f.replace("config.yaml", "exams"))` ? In my opinion, we treat coding as a three-player game : code as succinctly as possible in the Green phase, refactor as much as possible in the Refactor phase, and then try to think of tests which will force the code in the right direction in the Red phase. This leads to the level 5 mindset of subsection 5.2, where *testing is a mental discipline*.

So how could we force the green player to create a real config file, and then use this to create the exam folder specified in this config file? We could change the config file with a new location for the exam folder!

```
1  import config
2  import os
3
4
5  def create_cfg_file(path):
6      f = os.path.join(path, "config.yaml")
7      config.default(f)
8      return f
9
10
11 def test_create_default_cfg_file(tmp_path):
12     assert os.path.isfile(create_cfg_file(tmp_path))
13
14
15 def test_basic_structure_from_cfg_file(tmp_path):
16     f = create_cfg_file(tmp_path)
17     config.setup_from_cfg_file(f)
18     exam_dir = os.path.join(tmp_path, "exams")
19     assert os.path.isdir(exam_dir)
20
21
22 def test_tweak_structure_from_cfg_file(tmp_path):
23     f = create_cfg_file(tmp_path)
24     cfg = config.load(f)
```

```
25      exam_dir = os.path.join(tmp_path, "exams2")
26      cfg["exam_path"] = exam_dir
27      config.setup_from_cfg_file(f)
28      assert os.path.isdir(exam_dir)
```

Now all hell breaks lose, as `config.load()` finds an empty yaml file.

5.3.8 Green

We will need to create a yaml file with the correct fields otherwise `config.load()` does not run. And then we need to really use that the value in the field `exam_path` when we create the exams directory.

```
1  import os
2
3
4  def default(f):
5      path, _ = os.path.split(f)
6      config = {
7          "courses": [],
8          "courses_path": path,
9          "exam_path": os.path.join(path, "exams")
10     }
11     save(config, f)
12
13
14 def setup_from_cfg_file(f):
15     cfg = load(f)
16     os.mkdir(cfg["exam_path"])
```

5.3.9 Refactor

Looking at the tests, it seems odd that we need to load and save the config dictionary. We should return the config dictionary directly, rather than using the filename.

I also refactor the function `default()`, which is currently storing the information for the default dictionary and also saving it. These roles become `default_config()` and `create_default()` respectively.

```
1  import config
2  import os
3
4
5  def create_cfg_file(path):
6      f = os.path.join(path, "config.yaml")
```

```
    return config.create_default(f), f


def test_create_default_cfg_file(tmp_path):
    _, f = create_cfg_file(tmp_path)
    assert os.path.isfile(f)


def test_basic_structure_from_cfg_file(tmp_path):
    cfg, _ = create_cfg_file(tmp_path)
    config.setup_from_cfg(cfg)
    exam_dir = os.path.join(tmp_path, "exams")
    assert os.path.isdir(exam_dir)


def test_tweak_structure_from_cfg_file(tmp_path):
    cfg, _ = create_cfg_file(tmp_path)
    exam_dir = os.path.join(tmp_path, "exams2")
    cfg["exam_path"] = exam_dir
    config.setup_from_cfg(cfg)
    assert os.path.isdir(exam_dir)
```

```
import os


def default_config(base_path):
    return {
        "courses": [],
        "courses_path": base_path,
        "exam_path": os.path.join(base_path, "exams")
    }


def create_default(f):
    base_path, _ = os.path.split(f)
    config = default_config(base_path)
    save(config, f)
    return config


def setup_from_cfg(cfg):
    os.mkdir(cfg["exam_path"])
```

At the point the code feels well-structured. There is a reasonable test coverage, and so I have the confidence to refactor or experiment. I continue cycling through

red, green, refactor until the GUI can run from a "cold start".

I implemented the backed of the TAT system using test-driven development. Currently I have about 575 lines of code and 350 lines of tests. The tests were written with **pytest**, which runs all of the tests automatically. I will comment on how I found this in the conclusion, see section 10 on page 48.

6 The graphical user interface

Initially I had planned to interact with the TAT system using a *Command Line Interface*, updating flat files using a text editor and then calling the Python code from the terminal. To make the system accessible to a wider audience it was suggested that I add a *graphical user interface*, or GUI (pronounced *"gooey"*), over the backend.

In W. Martinez's 2011 article [Mart11] introducing graphical user interfaces the author explains that for most users the GUI is the standard form of human-computer interaction: windows, buttons, drop-down menus, spreadsheets, progress bars, etc. provide a graphical buffer on the screen between the human and the program being executed on the computer. GUIs tend to be *event-based*, driven off user events such as mouse actions and keyboard entry.

In this section I explain how I chose to implement the GUI; both the language and the approach. I then list a few basic principles of GUI design from the literature before outlining the interface I built.

6.1 Choice of language

As in subsection 4.2, I decided to continue with Python when implementing the GUI. Again the aim was to improve the programming skills I use in class, while gaining experience in areas of computing which offer different directions for my teaching.

Even then, there are many Python packages available for building a GUI. Some packages create familiar "windows": for example **Tkinter**, **PyQt**, **wxPython** and **PyGTK**. These packages create *widgets* such as buttons, menus, and text editors with standard behaviours. Fitting the widgets into the window as well as handling mouse and keyboard events is done by the package.The widgets can be *"wired up"* to Python functions in the backend.

The disadvantage with the applications built with these packages is their "look": the GUIs feel old-fashioned which would reduce student interest.

A completely different approach, which is now very popular, is to create an interactive web application. These packages are more diverse than the standard "windows" style packages. There is the pythonic **JustPy**, lightweight packages such as **Flask** and **Remi**, and the full-stack solution **Django**. These packages can be run locally, serving a webpage which presents the user interface in a navigator,

or if hosted on a server then a user can access them from any device with an internet connection. This makes it easier to deploy and share the application with users.

I opted for a third approach: building a GUI from the ground up with **PyGame** [KK16]. PyGame is a popular Python library for game development, especially 2D games. I have already had a number of students use PyGame in class and the results are almost always impressive: the imagination of the students, their perseverance in the face of difficulties, and the end results. Allowing the students to build their own projects, with more or less scaffolding as required, helps foster internal motivation as explained in the classical literature by R. Viau [Viau94].

PyGame provides a built-in event queue that collects and manages events such as mouse clicks, keyboard presses, window resizing, etc. This event handling allows developers to create interactive applications that respond to user actions in real-time. As mentioned above GUIs tend to be event-driven and so this is pertinent for GUIs as well as for games.

I thought that the TAT system would be a good chance to improve my own PyGame skills.

6.2 Planning for a positive user experience

J. Johnson outlines some simple ideas for improved *"UX"*, user experience, in his 2020 book on "GUI Bloopers" [JJo20]. He suggests the user will feel more in control if the design elements and interactions are simple and standardised, feedback is prompt and useful (obvious button clicks, error messages, progress indicators), and actions can easily be undone. Martinez summarises Johnson's ideas into four principles [Mart11, p. 123]:

1. 'Focus on the users and their tasks, not the technology'.
2. 'Consider function first, presentation later'.
3. 'Conform to the user's view of the task: do not make it more complicated'.
4. 'Promote learning and deliver information, not just data'.

There are strong links to D. Norman's ideas in "The Design of Everyday Things" [Norm13]. Good design should be intuitive and make it easy for users to understand and interact with the objects or systems, which in turn positively influences user behaviour and promotes satisfaction. Norman introduces the concept of *"affordances,"* which refers to the perceived actions that an interface suggests to the user. Clear affordances will guide users on how to use the interface naturally.

So rather than starting with what the backend can do, the design process starts with what the user will want to achieve with the TAT system:

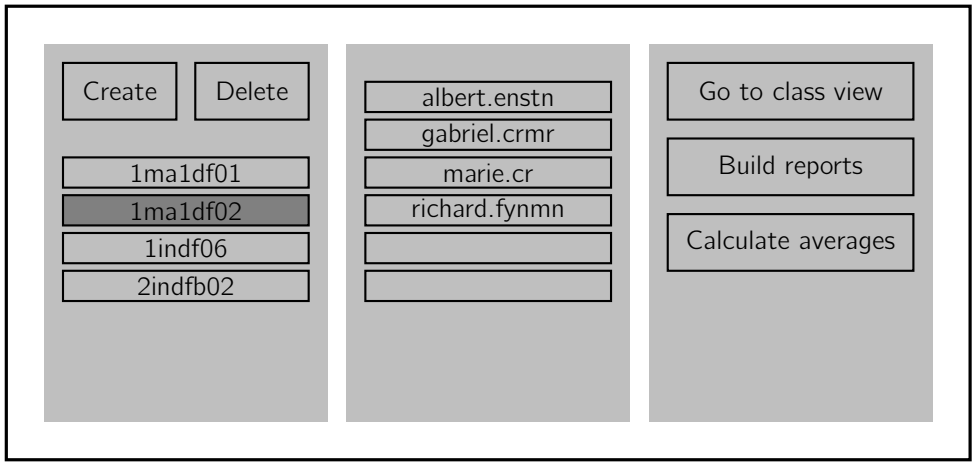
1. Review and update the list of courses.

2. Edit the list of students in a course.
3. Change the seating plan in a given course.
4. Record a comment for a set of students.
5. Suggest students for next teacher-student focus.
6. Build student reports.
7. Calculate average notes.

6.3 Views

A classic GUI structure is to use *views*, different screens designed for different use cases. I based the GUI views on the two phases defined when reviewing the types of actions in the last section: configuration and teaching. I have called the respective views the **control view** and the **class view**.

6.3.1 Control view



Possible events controlled from this view:

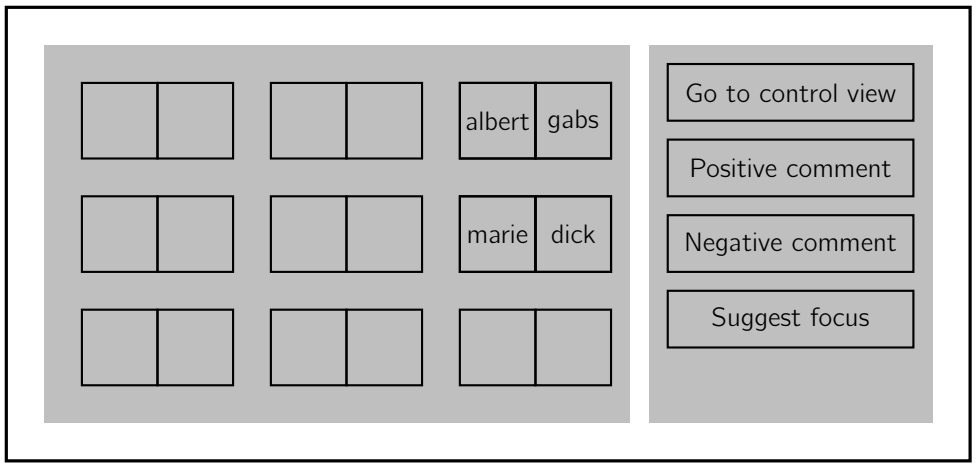
1. Create a new class.
2. Select the active class.
3. Delete the active class
4. Edit the class list.
5. Pass to the class view of the active class.

6. Build student reports.

7. Calculate average notes.

6.3.2 Class view

The class view includes the seating plan, both as a reminder for names and as a visual cue for interactions.



Possible events:

1. Pass back to the control view.

2. Change the seating plan.

3. Select/ deselect a student.

4. Record a positive comment for the selected students.

5. Record a negative comment for the selected students.

6. Suggest students for next teacher-student focus.

6.4 Implementation style

I used an object-oriented based system for the widgets that appear in the GUI, relying on inheritance and polymorphism. I quickly gave up with test-driven development, as there was too much initial investment required for this project. Indeed, as Nasser et al recently report this is a common problem when building GUIs [Nass21]. The inheritance diagram of the objects used to model Buttons in the GUI is in figure 3 on page 38, and similarly a diagram for the desks is in figure 4 on page 38.

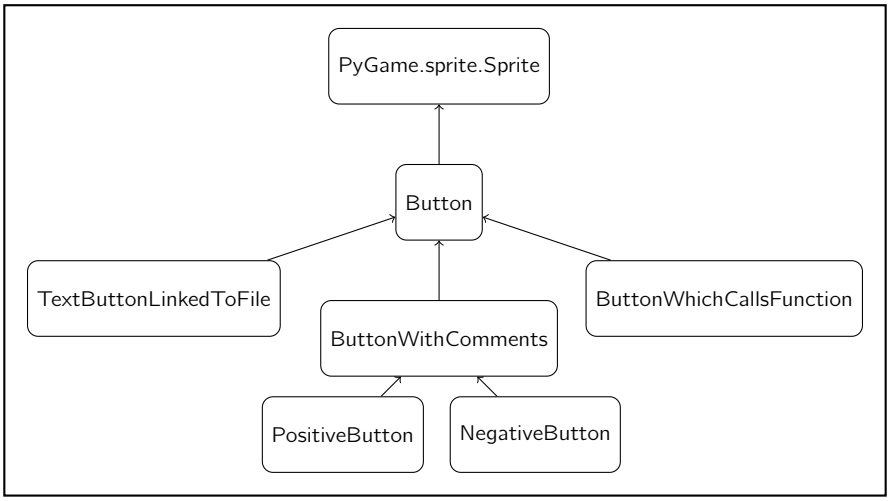


Figure 3: Inheritance diagram for the objects modelling the Buttons in the GUI.

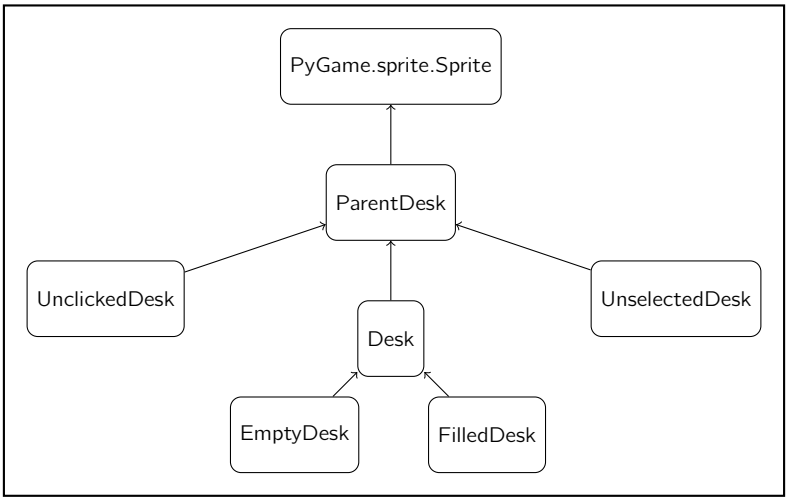


Figure 4: Inheritance diagram for the objects modelling the Buttons in the GUI.

7 A single executable file

Ideally, the TAT system would run from a single executable file. The user would launch the TAT application (for example, by double clicking on the icon) and then all interactions would happen through the GUI.

I tried using two different packages: **PyInstaller**[13] and **cx-Freeze**[14]. Both packages attempt to build standalone executables from Python scripts by bundling the application and dependencies into a single package.

Importantly, both scripts are only able to build packages for the hardware and software used when building the package. This means that **I must build the executable using the school computers,** as the executable I make locally will only run on machines identical to my own. This is a clear disadvantage with the decision to use Python, a scripting language. If I had implemented the TAT system using Java, for example, then I could have assumed the school computers used a compatible virtual machine.

Unfortunately, at this time I have not managed to automate the process of creating a standalone package. PyInstaller has trouble linking to the pygame module, and cx-Freeze dislikes subfolders. Instead I manually create the executable as follows:

1. Create an empty folder and use this as the current directory. This can be done at the command line using

   ```
   mkdir my_folder; cd my_folder
   ```

2. Copy all the TAT python scripts and the `requirements.txt` into this folder.

   - TAT.py
   - control_view.py
   - linkComments.py
   - analyseNotes.py
   - link_gui_backend.py
   - config.py
   - gui.py
   - students.py
   - class_view.py
   - icons.py
   - events.py
   - desk_functions.py
   - constants.py
   - examNotes.py
   - shredComments.py
   - analyseComments.py
   - updateComments.py
   - scrapeClassLists.py
   - createSeatingPlan.py
   - requirements.txt

3. Create a virtual Python environment for this directory.

   ```
   python3 -m venv .
   ```

[13] `https://pyinstaller.org/en/stable/`
[14] `https://cx-freeze.readthedocs.io/en/latest/index.html`

4. Add the necessary python modules.

   `pip3 install -r requirements.txt`

5. Make sure the cx-Freeze module is up to date, for example with

   `pip3 install --upgrade cx_Freeze`

6. Again at the command line in this working directory, run cx-Freeze

   `cxfreeze -c TAT.py --target-dir dist`

7. Copy the `GUI_files` directory into the new `dist` subfolder.

8. From the `dist` subfolder, zip the following files into a single archive ready for distribution:

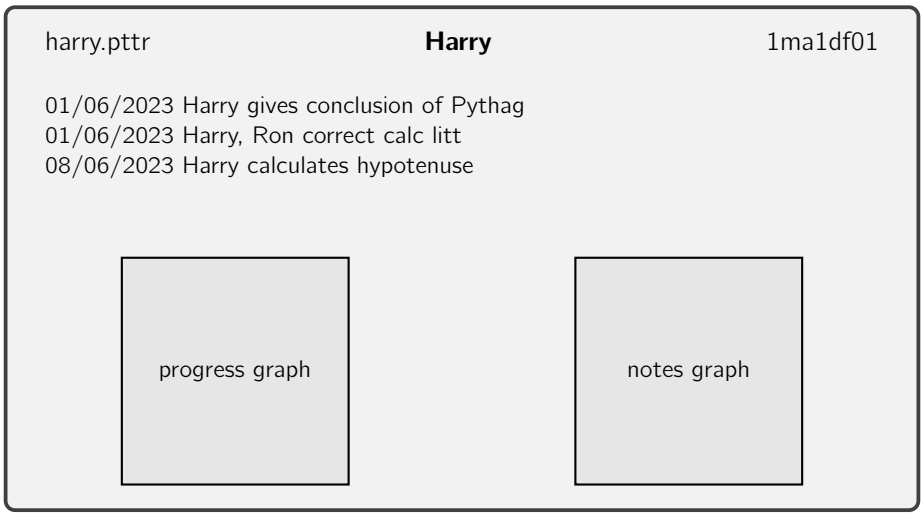
   - The application file `TAT.exe`.
   - The directory of packaged files `lib`.
   - The directory of TAT options and helper files `GUI_files`.

9. Double click on the `TAT.exe` file to launch the GUI.
   Warning: the first time the application is run there is still some set up to be done, and the GUI takes a while to appear.

8 Instruction booklet

8.1 Introducing the TAT system

The **Teaching Assisting Tools** system, TAT, is designed to improve your teaching by automating some of the boring stuff. During class, when planning lessons, or as a reminder before meeting the parents, TAT helps manage and collate a record of your interactions with students.

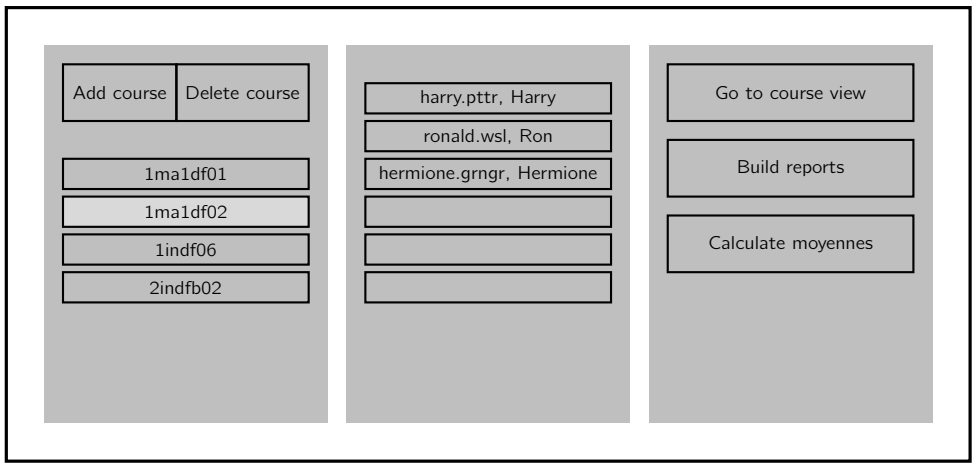


TAT can suggest the next students who deserve your focus during class, build individual student reports to help planning student feedback, and calculate your *moyennes* over the year. In class, the tool uses a seating plan arrangement which is a very helpful reminder of names at the start of the year (and after holidays).

In the TAT folder, double-click on the icon TAT.exe to launch the program. Once TAT has started, you should see the **control view**.

8.2 The control view

The **control view** allows you to configure your courses and students, as well as having the system build the individual student reports and calculating all the *moyennes* for you.



The control view is separated into three columns : the courses you teach, the students in a selected course, and the control buttons. At the top of the first column are two buttons. The left one allows you to add a new course to the list, while the right button deletes the selected course.

8.2.1 Add a new course

Click on the "Add course" button, type the name of the course, and again click on the "Add course" button to create the course.
The new course should be added as a new button in the list of courses under the "Add" and "Delete" buttons.

8.2.2 Select a course

Click on the course button.
The course button will turn blue, to indicate it is now the selected course. The list of students in the middle column will show the students in this course. If no students have been added yet, then the top student button helpfully reminds you that it expects `student.id, name`.

8.2.3 Add a student to a course

Click on the corresponding student button, type the student.id and the name of the student (separated with a comma), and again click on the student button to save the information.
Hint: the name of the student is optional. If you just enter the student.id then the TAT system will use the name in the student.id by default.

8.2.4 Delete a course

First select the offending course. Now click on the "Delete course" button.
This removes the course from the list of active courses. The list of students is still stored, and so you can recover a deleted course by adding it again.

8.2.5 Build student reports

Click on the "Build reports" button.
This will create a pdf of individual reports for each student in each class (not just the selected class). This pdf file is stored in `./latex/report.pdf` by default, and will automatically open once it has been created.

8.2.6 Collate marks and calculate semestriel notes

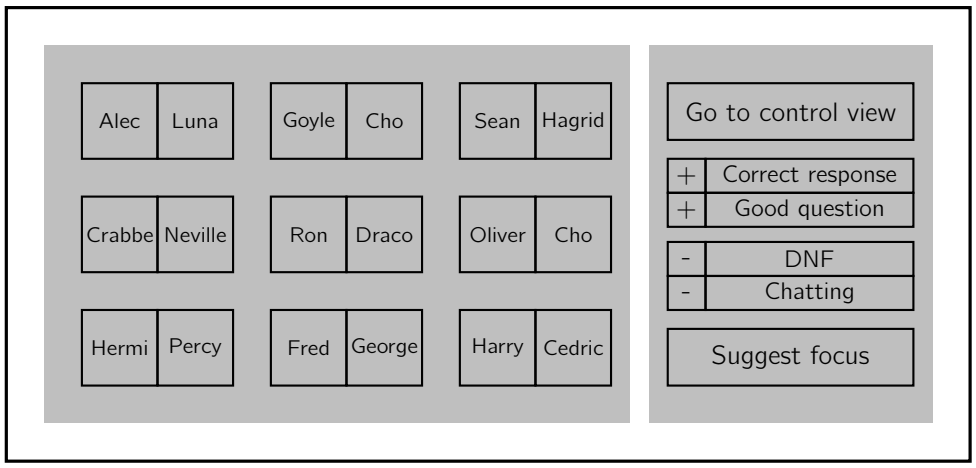
Click on the "Calculate moyennes" button.
In the spreadsheet you can adjust the weights of each exam in rows 3, 4 and 5. The rounded notes are updated automatically in the tables to the right. This xlsx file is stored in `./exams/moyennes/big_dump.xlsx` by default, and will automatically open once it has been created.

8.2.7 Switch to the course view

Select the desired course and then click on the top "Go to course" button.
This will take you to the **course view**.

8.3 The course view

The **course view** has the seating plan on the left hand side of the view, and the control buttons on the right. The seating plan uses the ordering of the students in the course list.



Note: currently the TAT system can only handle a maximum of 24 students, sat in the standard pairs format.

8.3.1 Modify the seating plan

Click and drag a student.
The student will swap places with another.

8.3.2 Select a student

Click on the desired student.
The student will turn blue, to indicate they have been selected. Hint: you can select multiple students at once.

8.3.3 Deselect a student

Click on the selected student.
The student will turn yellow again.

8.3.4 Go back to the control view

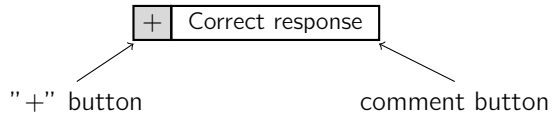
Click on the "Go to control view" button at the top.
This course will still be selected, so if you double click you return to the same course view.

8.3.5 Suggest students for focus

Click on the "Suggest focus" button at the bottom.
The TAT system will identify five students who have not received a positive comment recently and automatically select them for you (they turn blue).

8.3.6 Record a positive remark for students

Click on the corresponding "+" button.
The comment next to the associated "+" button will flash red, and then all the students will be deselected. The comment has been stored, and will appear in the student reports with today's date. An example:



These two buttons are associated. One click on the "+" button on the left will store the positive comment `"Correct response"` against all the selected students. Double-clicking on the comment button will give the same effect.

8.3.7 Update a comment option

Click on the comment button, type the new comment, and click on the same comment button again.
The new comment will be immediately saved for any selected students. Hint: the new comment is saved, and will be used next time you pass to the course view.

8.3.8 Record a negative remark for students

Click on the corresponding "-" button OR double-click the comment button.

9 Missing features

In roughly decreasing order of importance, in my opinion, here are the features I would like to see in the next implementation of the TAT system. Many of these features I had in mind as I was designing the system, but had to be descoped in this project due to time constraints.

9.1 Handling students who leave

In the current implementation, if a student leaves then they will soon be constantly suggested for focus. Unfortunately we cannot simply delete them from the list of students as the functions for "Shred Comments" would crash on an unknown student. This can be very annoying if, say, five students in the same class leave on an extramuros...

9.2 Button to create a blank exam file

On the selected course, create a spreadsheet in the correct format with the student.ids already filled in. This exists in the backend but was not wired up in the GUI.

9.3 Counting DNFs

The backend can already count the number of *devoir non fait* and return a dictionary of counts against student.ids. I did not have time to wire this into the GUI.

9.4 Migrate to databases

We should confirm with the PPDT[15] what extra security would be needed, see subsection 3.3.10 on page 23, and migrate the system onto a database instead of flat files. This raises other needs: it would have to be easier to review and change comments on students, for example, as simply opening the `comments.txt` file would no longer be an option. Also, should we connect to the official databases for student.ids and even course lists?

9.5 Stage the processed data

Currently the entire raw comments file must be shredded each time the comments are used. For example, every time the system suggests students for focus it must start from the beginning even though very little (and possibly nothing) has changed since the last calculation. Intermediate steps could be saved in the database using coherent transactions which mark which data has already been preprocessed.

9.6 Changing a name which already has comments

Although I have yet to see this happen in reality, if a student wanted to change their given name it would cause large headaches for the current TAT system. This seems like an edge case which it should be able to handle.

9.7 Persist seating plans

Currently the seating plan is lost when we switch out of the class view of the GUI.

9.8 Export seating plan to pdf

We could plan the seating plan in advance, and then we could print it off for the students or project it on to the board.

[15] `https://www.ge.ch/organisation/protection-donnees-transparence`

9.9 Suggest seating plans

We could consider, for example, how well pairs have worked previously, or how correlated students were in exams over the different questions, or whether certain students reduce the learning around them.

> *"Since it has been reasonably well established that student affect toward a class is related to student learning, student attitudes toward classroom arrangements are a matter of no small concern when determining a choice of classroom arrangement."* [MM78]

I think a simulated annealing optimisation would work well.

9.10 Review previous seating plans

It would be useful to store previous seating plans.

9.11 Configure constraints in a seating plan

I would like to be able to take into account the needs of certain students. For example, short-sighted students often ask to sit in the front row, as do students with hearing difficulties. More rarely, students have asked to sit away from the windows because of hay-fever. These constraints could be handled by specifying particular seats as "preferable" for a given student.

Pairs of students often asked to be sat together. The decision as to whether or not these requests are accepted should rest with the teacher, but it would be a nice to have when automatically generating seating plans.

9.12 Testing the GUI

It is commonly accepted that it is much harder to build a testing framework for a GUI [Nass21]. However, to ensure future maintainability would require either a manual Acceptance Testing process [Amman16, p. 53] or, ideally, an automated harness. The latter would require high initial investment.

9.13 Predictions

It would be interesting to use the data in the system to make predictions about students. For example: their behaviour in class has changed dramatically (for the better or for the worse); students who might need extra support in order to pass the year; students who should consider accelerated learning programs. There may be issues around data privacy with such an expansion though, see subsection 3.3.10.

9.14 Automate the process of creating a standalone package, which is currently manual

This would be a nice to have, especially if the system was being regularly updated.

10 Conclusion

The backend of the TAT system was coded using test-driven development, section 5, where I religiously wrote failing tests, then I wrote trivial code which passed the tests, before I finally refactored the code as much as possible. Refactoring is definitely much easier with a large suite of tests: I found I had the confidence to try much larger factorisations knowing that it was no effort to check for obvious mistakes then roll-back using version control.[16] I may be at risk of falling into a level 2 mindset of testing (testing is to show correctness), see section 5.2.

However, I quickly gave up on TDD when implementing the GUI. This means that roughly half of my code was created with TDD, and half was not. I was interested to see how much better my coding would be with TDD, and so I used a Python package called **Radon** to calculate various metrics of code "goodness" [Oman92].

The Radon package analyses text-based python scripts to calculate

- the cyclotomic complexity of each function; roughly, the number of holes in the possible logical pathways of a function,
- the number of lines of code, and
- the maintainability index; a rough estimate of how easy or hard the code will be to understand, modify and maintain[Oman92]. The higher the maintainability index the better, with a value in the range $[0, 100]$.

I ran the analysis on my own code base, and also calculated the average length of functions. Looking through the analysis I am not surprised to see certain functions being flagged - shredding the raw comments file was difficult to code, for example, and the PyGame loops running the GUI views involve overly long conditional chains on the events.

I was hoping to see a large difference between the code written with TDD and the rest.

Complexities	**Backend (TDD)**	**GUI (without)**
Mean	2.26	2.20
Median	2	2
Stdev	2.13	2.53
Maintainability	49.63	47.53
Lines of code	574	579
Avg length fns	7.97	6.36

[16] I used github for this project, `https://github.com/danjane/GymInf`

I was surprised to see that there really is no difference as far as the metrics are concerned! Rather than being disheartened notice that the backend has a strong testing framework, making changes relatively risk free. Further, we might hope that I am becoming better at naturally refactoring as I code.

> *...most people who learn TDD find that*
> *their programming practice changed for good.* [Beck03, p.xii].

Recent research by A. Santos et al has shown that *"[small projects] tend to provide less optimistic results for TDD"* and that TDD is more effective for teams.[SVDU21] My aim at the start of this project was to be able to use test-driven development, not that to show that I must use TDD to write a better standard of code.

I was happy that I used Python for this project. I am also clearly more competent with PyGame, and I feel more confident replying to student questions on this package. However, I do feel that building a GUI with PyGame required a lot more work than I had expected: probably a lightweight interactive web application built with JustPy would have been more appropriate for this masters project. Another regret was that I continued to use PyCharm as my IDE: given that I was trying to improve the skills used in class, perhaps I should have used Replit or Thonny.

The deliverables of section 2.7 were that *the TAT system will provide a basic GUI over a backend handling... notes to be taken against student names and then suggest pertinent students for focus."* In this project I have managed to complete the deliverables with only a relatively small number of missing functionalities. This project has managed to "automate the boring bits", as A. Sweigart puts in [Swei15]. The most recent UNESCO global education monitoring report raises the next question: what to do with all the free time I have now afforded myself?

> *"Technology is touted as being able to reduce the time students and teachers spend on menial tasks, time that can be used in other, educationally more meaningful activities. However, there are conflicting views on what is meaningful."* [Unesco23, p. 11]

References

[Amman16] Ammann, P., & Offutt, J. (2016). Introduction to software testing. Cambridge University Press.

[Anaya18] Anaya, M. (2018). Clean code in Python. Packt Publishing

[Beck03] Beck, K. (2003). Test-driven development by example. Addison-Wesley.

[Bloom79] Bloom, B. S. (1976). Human characteristics and school learning. McGraw-Hill.

[BCSW10] Borba, P., Cavalcanti, A., Sampaio, A., & Woodcook, J. (Eds.). (2010). Testing techniques in software engineering: Second Pernambuco Summer School on Software Engineering, PSSE 2007, Recife, Brazil, December 3-7, 2007, Revised Lectures (Vol. 6153). Springer.

[Bryant20] Bryant, J., Heitz, C., Sanghvi, S., & Wagle, D. (2020). How artificial intelligence will impact K-12 teachers. Retrieved May, 12, 2020.

[BS09] Bucheton D. & Soulé Y. (2009). Les gestes professionnels et le jeu des postures de l'enseignant dans la classe : un multi-agenda de préoccupations enchâssées.

[Coad91] Coad, P., Yourdon, E., & Coad, P. (1991). Object-oriented analysis (Vol. 2). Englewood Cliffs, NJ: Yourdon press.

[Hat12] Hattie, J. (2012). Visible learning for teachers: Maximizing impact on learning. Routledge.

[JJo20] Johnson, J. (2000). GUI bloopers: don'ts and do's for software developers and Web designers. Morgan Kaufmann.

[Jorg17] Jorgensen, P. C. (2017). The craft of model-based testing. CRC Press.

[KK16] Kelly, S., & Kelly, S. (2016). Basic introduction to pygame. Python, PyGame and Raspberry Pi Game Development. Springer.

[Larkin05] Larkin, J. E., & Pines, H. A. (2005). Asking Questions: Promoting student-faculty interchange in the classroom. APS Observer, 18.

[LX10] Lee, G., & Xia, W. (2010). Toward agile: an integrated analysis of quantitative and qualitative field data on software development agility. MIS quarterly, 34(1), 87-114.

[Mart08] Martin, R. (2008). Clean Code. Pearson.

[Mart17] Martin, R. (2017). Clean Architecture: A Craftsman's Guide to Software Structure and Design. Pearson

[Mart11] Martinez, W. L. (2011). Graphical user interfaces. Wiley Interdisciplinary Reviews: Computational Statistics, 3(2), 119-133.

[MM78] McCorskey, J. C., & McVetta, R. W. (1978). Classroom seating arrangements: Instructional communication theory versus student preferences. Communication education, 27(2), 99-111.

[MSB11] Myers, G. J., Sandler, C., & Badgett, T. (2011). The art of software testing. John Wiley & Sons.

[Nass21] Nass, M., Alégroth, E., & Feldt, R. (2021). Why many challenges with GUI test automation (will) remain. Information and Software Technology, 138, 106625.

[Norm13] Norman, D. (2013). The design of everyday things: Revised and expanded edition. Basic books.

[Norr98] Norris, J. R. (1998). Markov chains (No. 2). Cambridge university press.

[Oman92] Oman, P., & Hagemeister, J. (1992, January). Metrics for assessing a software system's maintainability. In Proceedings Conference on Software Maintenance 1992 (pp. 337-338). IEEE Computer Society.

[PPDT18] Le PPDT - Guide pratique RGPD à l'attention des institutions publiques genevoises (2018).

[Raj2018] Raj, A. G. S., Patel, J. M., Halverson, R., & Halverson, E. R. (2018, November). Role of live-coding in learning introductory programming. In Proceedings of the 18th koli calling international conference on computing education research (pp. 1-8).

[Rob19] Robinson, M. (2019). BOSCARD: A Scoping Tool for Lean Continuous Improvement Projects. In Global Lean for Higher Education (pp. 181-195). Productivity Press.

[Rowe74] Rowe, M. B. (1974). Wait-Time and Rewards as Instructional Variables: Their Influence on Language, Logic, and Fate Control. Journal of research in science teaching, 11(2), pp. 81-94.

[SVDU21] Santos, A., Vegas, S., Dieste, O., Uyaguari, F., et al (2021). A family of experiments on test-driven development. Empirical Software Engineering, 26, 1-53.

[Swei15] Sweigart, A. (2015). Automate the Boring Stuff with Python.

[Tidw10] Tidwell, J. (2010). Designing interfaces: Patterns for effective interaction design. O'Reilly Media, Inc.

[Unesco23] UNESCO (2023). Global Education Monitoring Report: Technology in education: a tool on whose terms? Paris: UNESCO.

[Viau94] Viau, R. (1994). La motivation en contexte scolaire. Éditions du Renouveau pédagogique.

A Original project proposal

Bucheton and Soulé have described the act of teaching as a multi-agenda game of postures requiring good preparation and excellent micro-decisions [BS09]. In their model, they identify the crucial roles of the teacher in controlling the cadence, the atmosphere, the scaffolding and the relationships during the class, and how each supports the learning objective. And yet many of the teacher's most time-consuming tasks do not take place in the classroom [Bryant20]: good preparation and strong follow-ups (for example auto-reflection, marking and parent-teacher interactions) work to support and complement the pillars identified in the student learning process.

Many of these tasks are ripe for automation. As Sweigart writes, "many people spend hours clicking and typing to perform repetitive tasks, unaware that the machine they're using could do their job in seconds if they gave it the right instructions" [Swei15]. For my thesis project of the GymInf formation, I intend to build a suite of tools to support a range of teacher tasks including

- organizing seating plans
  to aid the atmosphere and relationships
- suggesting teacher-student interactions for upcoming classes
  to foster relationships
- building individual student reports
  to regulate cadence and scaffolding, as well as for follow-ups
- creating spreadsheets of marks
  for follow-ups

Teachers will interact with the tool using a GUI: it should be easy to use for all teachers, not just those teachers who are computer literate.

I will implement this project using *Test-Driven Development*. This is a style of software development that grew out of the 'Extreme Programming' philosophy of the 1990s, encouraging quick development cycles and active feedback from clients. The first book written on the subject, *TTD by Example*, by Kent Beck, is still the main resource [Beck03]. Kent emphasises the need to thoroughly test the requirements in order to have confidence when refactoring or making changes, as well as the need to make incremental changes (both for confidence and for time to market). TTD programming follows the following cycle:

Red. Write a failing test for a requirement.

Green. As quickly as possible, write code to pass the test.

Refactor. Clean the code, removing duplication and renaming functions.

Coding is driven by testing. In the thesis I will explore testing in more detail.