Contents

1 Introduction

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: 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, 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.

Many of these tasks are ripe for automation [Swei15], although 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.

TODO: discussion on equality for students, and [Hat12] on answering too quickly.

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: an incredibly useful trick when planning lessons, writing reports, and especially for parent evenings.

For the Master's thesis project undertaken for the GymInf formation, I built a suite of tools to support a range of teacher tasks including organising seating plans, building individual student reports, suggesting teacher-student interactions for upcoming classes, and creating spreadsheets of marks.

I will now explain the layout of this thesis. In sections 2 and 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.

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 university have highly recommended this coding style, and while it has disadvantages (as discussed later) there are strong reasons to having it as an option [Amman16]. TODO proper link to book.

Given the limited resources of this masters' thesis, not all of the desired functionality has been delivered. These missing features are described in section 8. 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.

2 System plan: inputs and outputs

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

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 2.2.3. It is also worth pointing out that the processed data (data during calculations and the outputs) is also sensitive.

2.1 Implied structure

2.2 Data

2.2.1 Files

2.2.2 Databases

2.2.3 Data privacy

3 System plan: user actions

We can also model a system by considering the possible actions. We can then analyse the sub-processes inside the actions, which can suggest a natural architecture for the system.

3.1 Actions during teaching

3.1.1 Record a remark about a set of students

3.1.2 Suggest students for interpolation

3.1.3 Modify the seating plan

3.2 Actions outside of teaching

3.2.1 Prepare student reports

3.2.2 Suggest a seating plan

3.2.3 Review previous seating plans

3.2.4 Prepare spreadsheets for noting an exam

3.2.5 Collate marks and calculate semester note

3.3 Actions to set up the classes

3.3.1 Add a class to the list of classes taught

3.3.2 Configure the class lists

The *class list* is the list of students in a given class.

3.3.3 Suggest a seating plan

3.3.4 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.

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.

TODO discussion of functional languages.

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,
- 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.

4.2 New languages

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**.

TODO discussion of these new languages: why they were created, their advantages and disadvantages and possible

4.3 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 large support community, and the focus of 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, TODO citation needed
- 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, and
- for recognising bad coding practices and explaining to students the potential pitfalls and how to avoid them,
- for understanding error messages.

5 Test driven development

5.1 Building IT systems

5.2 The waterfall method

5.3 Reducing the distance between clients and developers

5.4 TDD

5.5 And next...?

6 The graphical user interface

6.1 Choice of language

6.2 Implementation style

6.3 Views

A classic GUI structure is to use *views*, different screens designed for different use cases.

TODO: citation needed.

I based the GUI views on the three phases defined when reviewing the types of actions in section 3: configuration, teaching, and planning. I wanted to be able to use some of the planning actions while teaching, for example suggesting students for interaction, so I moved these to the teaching view. This meant "planning" only left those actions directly related to proposing seating plans, so the "planning" phase is really only "planning the seating plan". Thus I have called the respective views the **control view**, the **class view**, and the **seating plan view**.

6.3.1 Control view

TODO : diagram

Possible events:

1. Select the active class.
2. Pass to the class view of the active class.
3. Create a new class.
4. Edit the class list.
5. Export student reports.
6. 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.

TODO : diagram

Possible events:

1. Pass back to the control view.
2. Pass to the seating plan view of this class.
3. Select/ deselect a student.
4. Record a positive comment.
5. Record a negative comment.
6. Suggest students.

6.3.3 Seating plan view

TODO : diagram
Possible events:

1. Pass back to the control view.
2. Pass to the class view of this class.
3. Change the seating plan.
4. Save the seating plan.
5. Load previous seating plans.
6. Suggest a seating plan.

7 Instruction booklet

7.1 Page 1: introduction

7.2 Page 2: control view

7.3 Page 3: class view

7.4 Page 4: seating plan view

8 Missing features

Simple executable file

instruction booklet

exam buttons

export seating plan to latex to pdf

handling students who leave

using the data 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 are lots of resources available to the students and to the teachers, and it is difficult for young teachers to suggest

9 Conclusion

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.

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

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

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

[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.

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

A 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.

B Useful quotes

"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]

"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]

`https://github.com/zedr/clean-code-python#table-of-contents`