TABLE OF CONTENTS

ANALYSIS

BACKGROUND INFO

A grading system is used by teachers to input their student's data from exams that the students had taken. The grades tend to show where a student is at in terms of knowledge of a certain subject. Over time, when a student takes different exams, there would be a change in grades whether it is a student progressing and getting better grades (showing signs of improvements and understanding) or doing not so well and getting grades which shows no revision or lack of knowledge for that subject/topics that the student was being examined on. This data tends to be important to a school as it tracks progress for a student and the knowledge gap between students doing the same course. I specifically chose my 6th form as it is important to show students in year 13 how well they are doing and where they would need to work on for them to be achieving the grades which would determine their future. The main goal for this project is to establish a user interface simple enough for students to know where they are at and how they are doing with their A-level subjects.

CURRENT SYSTEM AND ISSUES

By interviewing several teachers in my 6th form, 90% had stated that they use a spreadsheet application known as Excel. By using student information that was collected, the teacher would have to create a grade sheet for that exam period and enter all the data such as the paper and the marks. After entering the data, the spreadsheet would have to convert the marks into a percentage which would output a grade from U-A* by using the mark scheme for that specific exam. The clients for my project would be teachers as they would be the main users of the system and my end-user will be the head of year of my 6th form who has concerns regarding the current system. The issues with the current system are that it can be too complicated for teachers who have no experience with the use of spreadsheets. The volume of data tends to be large, so it is extremely time-consuming and this delay when students will be receiving their grades as students will have to wait around two to three weeks before they get their results. I would need to create a system which would be fast, efficient and easy to use by teachers.

PROPOSED SOLUTION

As a proposed solution, I am going to create a program which would allow students to view their grades for the A-levels they had chosen. When the program is run, there will be an authentication system so that the user can login or register an account. The user would have to pick if they are a student or a teacher so that when they login, they get the GUI dedicated to their role. Once registered into the database and the user login's, there would be a GUI. In a teacher's GUI, they would be able to add the classes they teach and add students to that class. This would be done via a Tkinter window where the python program has a connection to the database. As students are being added to the class, the program should subsequently be inserting those students to the database. Once the students are added to the class, teachers should be able to set those students grades based on what they received. If there are no grade boundaries, there will be a feature from which the teacher can convert the percentage into a concrete grade, for example, if 70% was inputted, it would convert then output "A". The issue with this feature may be that the grade boundary tends to vary between exam board and subjects, so grades can be inaccurate. After the grades have been inputted, the database would store the grade, subject and student. Conversely, students would have a different GUI. The student's GUI would only allow them to view the grades that were set for the subjects they take. As teachers are adding students to their classes, the database would store that into the student's database so that when the students run the program, that would appear on the menu. Overall, my program should be able to overcome the complex issues teachers had with inputting grades onto a spreadsheet.

INTERVIEW WITH END-USER

End User's name: Ms Lok Position: Head of 6th Form
Signature: 

What subjects are being taught in the 6th form?

- "A level – English literature, Mathematics, Further Mathematics, Geography, History, Sociology Psychology, Government and politics, Product design, Photography, Chemistry, Biology, Economics, Physics and Computer Science."

How many assessments are being done?

- "There are two assessment cycles for Year 13."

When does the results tend to be inputted?

- "Normally within two to three weeks of the test."

What do you usually do with the grades you get?

- "Analysis trend and diagnostic/intervention plans."

Why are grades important?

- "The students need them to attend higher education, but good grades would help them to pursue top third/Russell Group universities. This open doors to their careers and future. The market is very competitive, everyone has a degree and almost all jobs required graduate level."

How is grades currently being inputted and what is the main drawbacks of this system?

- "Currently, teachers are inputting grades into an excel spreadsheet or a paper spreadsheet. The main drawbacks to this system are that it is extremely time consuming, and that the new teacher's rarely have any understanding on how to use the system. As the system is time consuming, this delays the subject departments from moderating the papers."

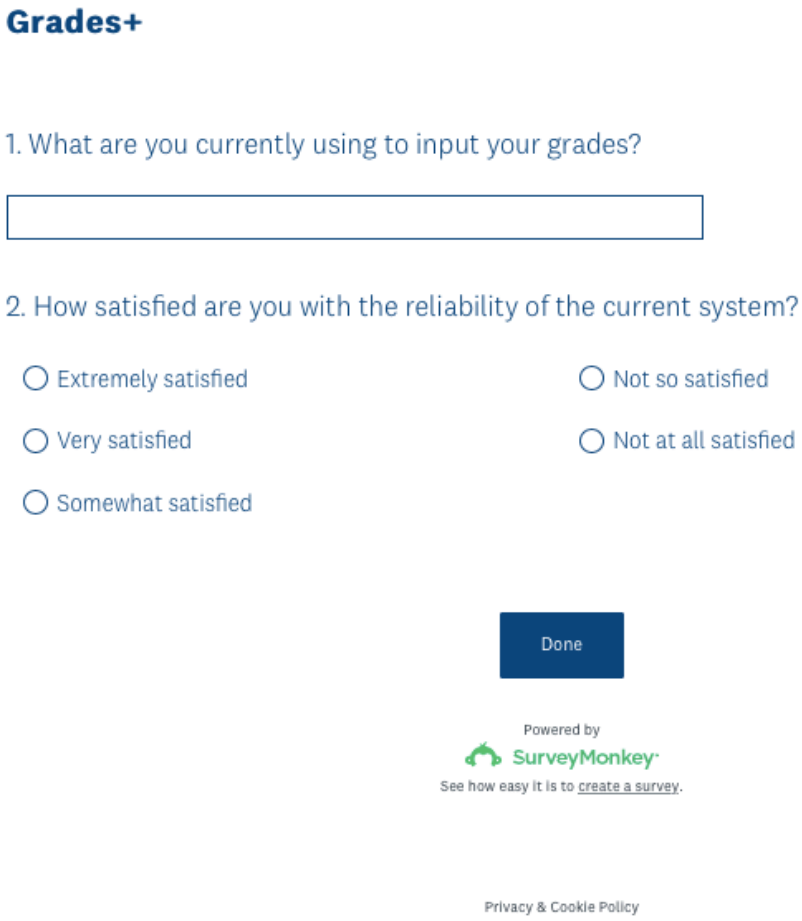
What would you like to see from the new grades system?

- "The new grades system should be user-friendly as this would be used by a variety of teachers, so whether they are experienced or not with computers, they should be able to use this system. Students should be capable of viewing their grades that were set. This is important as it helps the students track their progress from one assessment to another. To see how the year group are doing with their mock assessments, I should be able to view every student's grades. From this, I would be able to evaluate their predicted grades for their UCAS application."

SURVEY

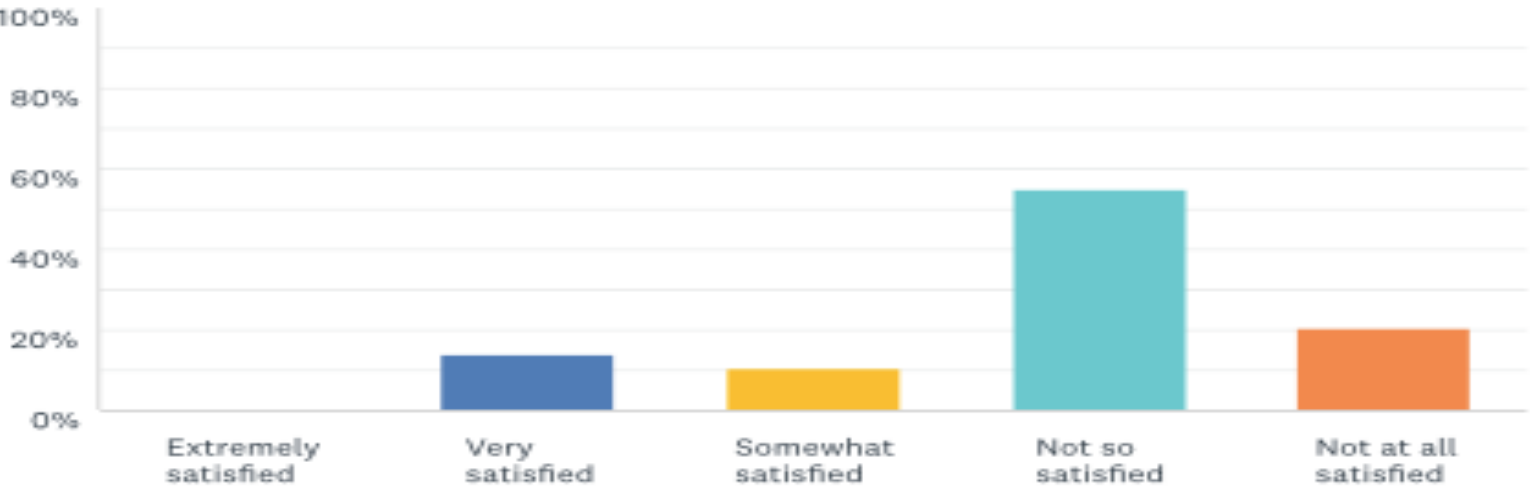
THE PROCESS

To know how satisfied the teachers are with the current system, I had used SurveyMonkey to create a short survey which allowed the clients to mention what they were using and the satisfaction they receive from it. Survey was the best research method to use as the data received can represent characteristics of a large population. SurveyMonkey is a flexible website which allows users to put their input anywhere, from mobile devices to personal computers. In total 20 teachers had participated in the survey. The survey I had created for my clients is as shown below.



REPRESENTING THE DATA

With the data I received from the teachers who have participated, I had represented it using different types of statistical graphics. This was important as graphs are used to visualise relationships which are shown in the data. For Question 1, I had to first organise the data by categorising it. A pie chart is used to show data that are classified into categories. Therefore, I chose to represent the data in Question 1 as a pie chart. The pie chart I made shows that around 85% of teachers are using excel and that 15% of teachers use other methods. However, for Question 2, the results were presented using a bar chart. This is because it is much easier to view the teacher's opinion on their methods. From the bar chart, I can see that majority of the teachers are not satisfied with the current system, hence, by creating a new, basic system with advanced functionalities, it may change their satisfaction.



LIMITS AND CONSTRAINTS

Main Limitations

The main limitation of my project is the fact that the user must be able to run the program on an IDE or Python IDLE. This is because it is hard for me to incorporate this grading system into a much simpler form such as a software application. As shown by the hardware and software requirement tables, the user must be able to meet the recommended specifications for the program to be working.

Hardware Requirements

Hardware	Justification
Intel Atom or Intel Core 13 processor	The better the processor, the faster the computer can complete the tasks.
RAM: 1GB (Recommended)	The larger the size of RAM you have, the more programs you can access with speed and efficiency.
Disk space: 1GB (Recommended)	Space is required in the hard drive, so your programs can be stored.
Keyboard	This peripheral is required so that data can be inputted into the system.
Mouse	This peripheral would be required in my system as there are buttons which would need to be clicked at times. It would also be needed to navigate around the system.

Software Requirements

Software	Justification
Windows 10 (Recommended)	The chosen operating system is required as it manages the computer's memory and processor. The users of my system are used to using Windows 10 which makes it the most suitable OS to run my system.
Python 3 (Recommended)	This software would be used to provide the system a user interface. It is required in order to run the program.
PostgreSQL 12 **PSequel (Recommended)**	The software mentioned are used to control the database that I am going to be using for my system. PSequel can be used by the teacher to create queries to get an understanding of the system's database.

IPSO

INPUT	PROCESS
- Student Name - Subject - Marks	- Teacher would add the marks along with the student's details into the spreadsheet and the spreadsheet would convert the marks into a percentage from which another column would convert that percentage into a grade.
STORAGE	**OUTPUT**
- Excel Spreadsheet	- CSV File - Graph - Grade

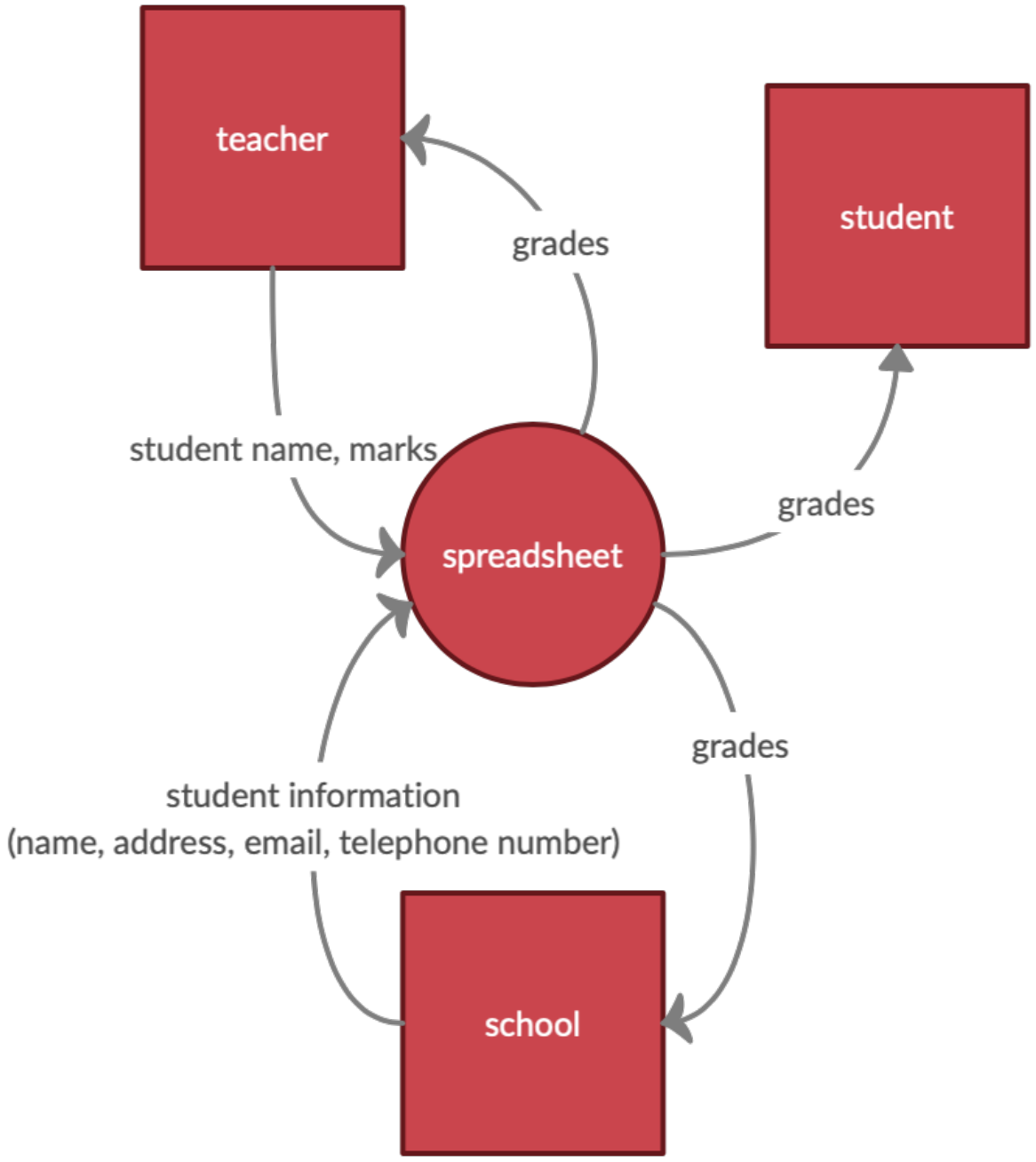
An IPSO table shows the data that is being inputted into the system, the process for the data, where it is being stored and what is being outputted. In my current system, there are a couple pieces of data of a student which is being inputted into the system by their teacher.

DATA DICTIONARY

Data Item	Data Type	Validation	Description	Data Sample
Student Name	String			“John Apple”
Teacher Name	String			“N. Daniels”
Grade	String	A*, A, B, C, D, E, F, U	The grade given for the exam.	
Marks	Integer	The mark must be in range: $0 \leq x \leq$ maximum marks available	The value accumulated from an exam.	76/100
Subject	String			
Absent	Boolean		Has the student taken the exam?	“Yes”

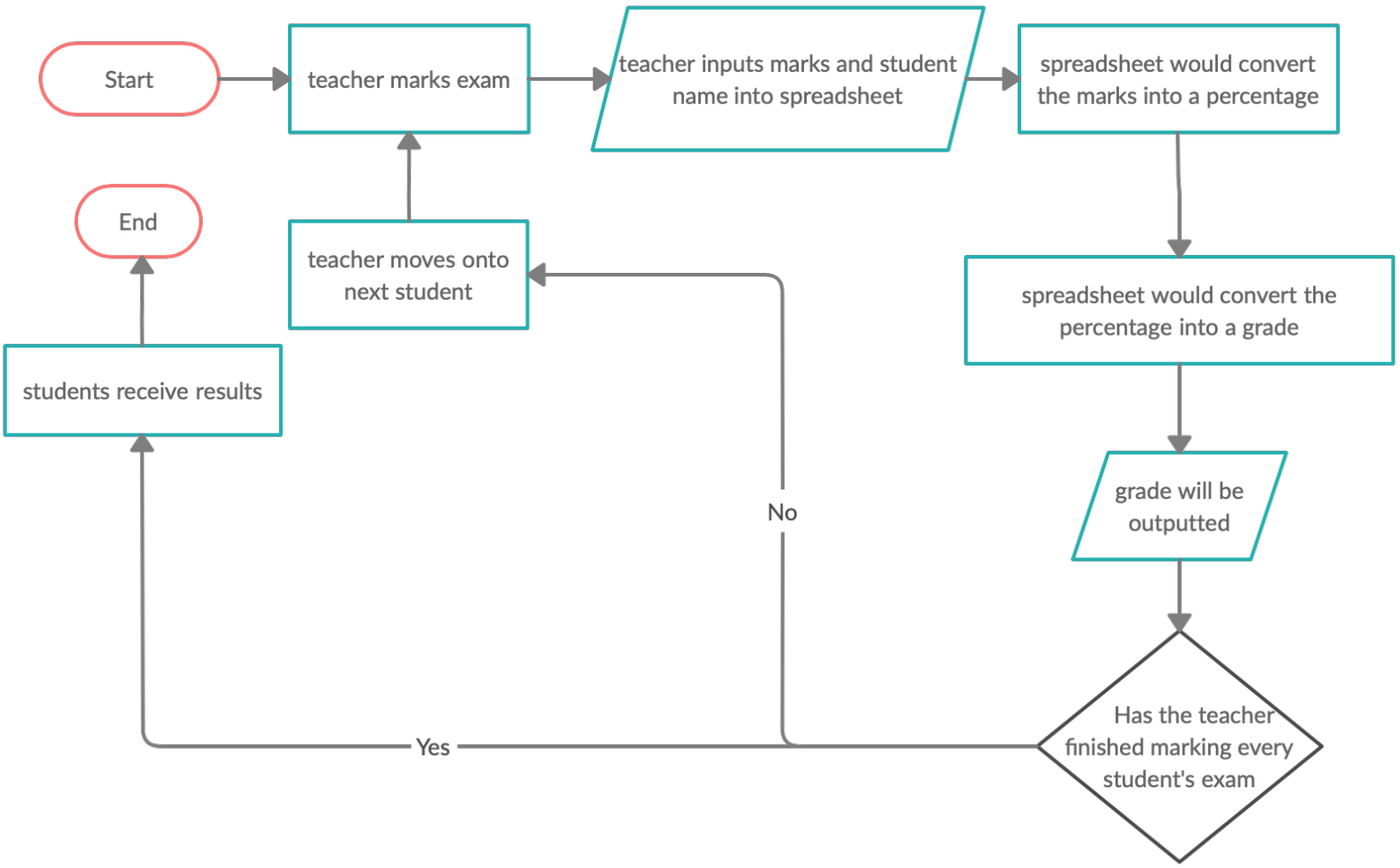
A data dictionary is a table which identifies the data that would be inputted into the system by a user. For certain data, it will need to be validated before it is stored into the system.

CONTEXT DIAGRAM



A context diagram is used to show how data goes in and out of a system. For the current system, the main data that is stored in the system is the student's information and grades. As the teacher inputs the grades into the system, the school would be able to access the grades as the data is sent to the school's main system. From the system, students would subsequently receive their grades in the form of a spreadsheet.

SYSTEM FLOWCHART



A system flowchart represents how data flows in a system and the decisions made to control the data flow. In the current system, the teacher would have to first mark the paper then the spreadsheet would tend to have a converter implemented so that the marks would be converted to a percentage. The percentage is then converted to a rough grade from a separate spreadsheet which follows the grade boundary for the paper. The teacher would have to repeat this process until all the papers have been marked before distributing the grades to the students.

OBJECTIVES

1) Program only allows authenticated users to register an account and login.
   a) There will be an option to choose if you are a student or a teacher.
   b) The students will have a code which they will be asked to enter as per to authentication.

2) Store student and teacher data in a database which would be easy to access for the program so that they can login without any issues.
   a) The system should be able to generate queries from user tasks such as setting the grades for a student.
   b) Password: make sure there are limitations for these such as min length, max length to provide a safer and securer system.

3) Hash the passwords so that they are securely stored.

4) The students should be able to view their subjects and the grades that were set. This will be done through a separate user interface.

5) The teachers should be able to view the students and set grades for students.

6) The teacher should be able to view the class's grades.

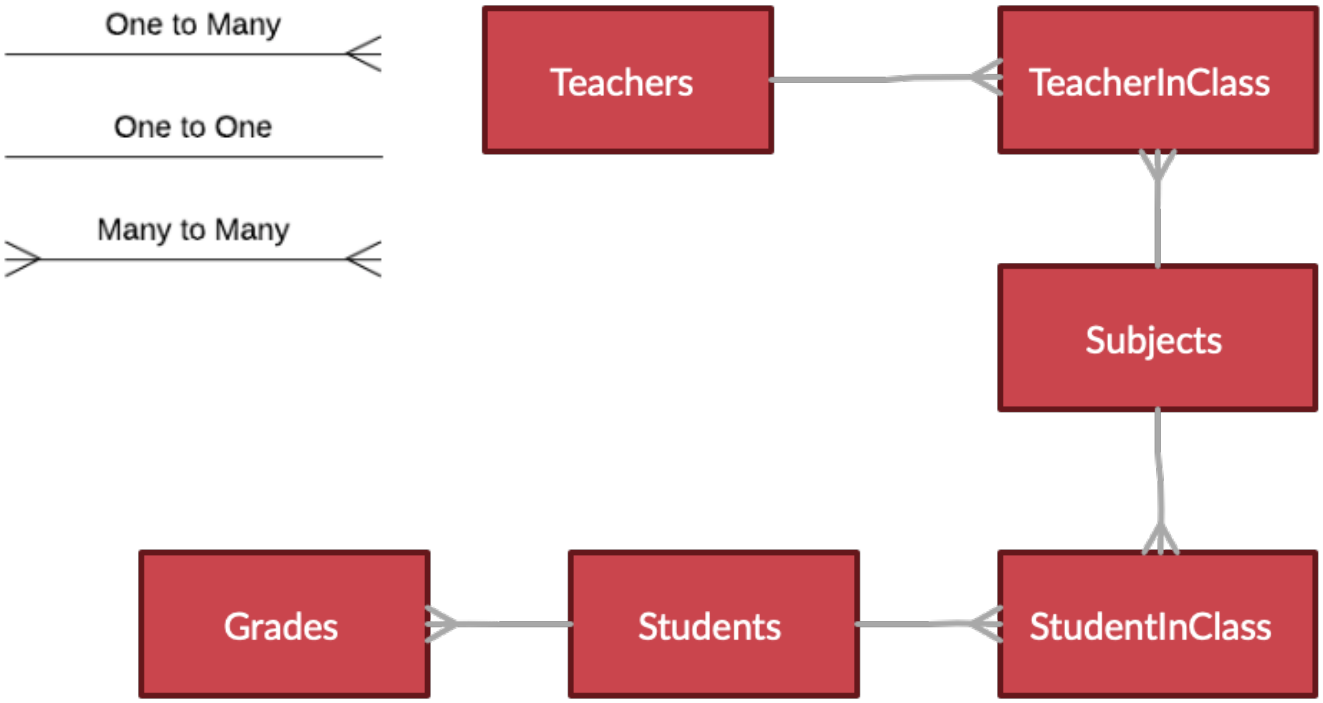
7) There will be an option for teachers to convert a percentage to a rough grade.

8) There will a dropdown for teachers to add the grade to make it easier.

DESIGN

DATABASE

ENTITY RELATIONSHIP DIAGRAM



In the entity-relationship diagram, the relationship for entities in my systems is shown. One student can have many classes. In my school, there is many students and teachers. The students are assigned to classes by their corresponding teacher who would set them their grades. A student can have many grades due to the fact they are taking multiple a-levels and there are two assessment cycles.

NORMALISATION

In order to store data for my system, I may use a database. Normalisation is the process which ensures that a database is structured efficiently. For example, by removing redundant data, we create more space in our database. Redundant data makes our database inefficient as when it is required to modify data, it is hard to select the exact data that is needed, causing data inconsistency. I am going to be normalising my database so that it is organised, and redundant date is eliminated. Table 1 shows my first attempt at creating a database-table before normalising it.

Table 1: *Students Table*

Student Name	Subject ID	Subject	Grade
Freddie James	1	Mathematics	A
	5	Physics	A
	9	Computer Science	A*
Steph Jones	7	Psychology	A
	6	Sociology	B
	4	Economics	B
Gab Fox	8	Government and Politics	B
	4	Economics	A
	1	Mathematics	A

FIRST NORMAL FORM (1NF)

In Table 1, the following attributes "Subject" and "Grade" have repeating entries which use up storage in the database. To get rid of the repeating entries, they must be separated. If a student was to be assigned multiple grades depending on the subject, then in first normal form, there would be many records associating with the student. In addition to this, I have identified a primary key "student ID" which is highlighted in Table 2. Table 2 is in first normal forms as, in comparison with Table 1, I have got rid of the repeated entries and separated them.

Table 2: *Students Table*

Student ID	Student Name	Subject ID	Subject	Grade
1	Freddie James	1	Mathematics	A
1	Freddie James	5	Physics	A
1	Freddie James	9	Computer Science	A*
2	Steph Jones	7	Psychology	A
2	Steph Jones	6	Sociology	B
2	Steph Jones	4	Economics	B
3	Gab Fox	8	Government and Politics	B
3	Gab Fox	4	Economics	A
3	Gab Fox	1	Mathematics	A

SECOND NORMAL FORM (2NF)/THIRD NORMAL FORM (3NF)

Table 3: *Students Table*

Student ID	Student Name
1	Freddie James
2	Steph Jones
3	Gab Fox

Table 4: *Subjects Table*

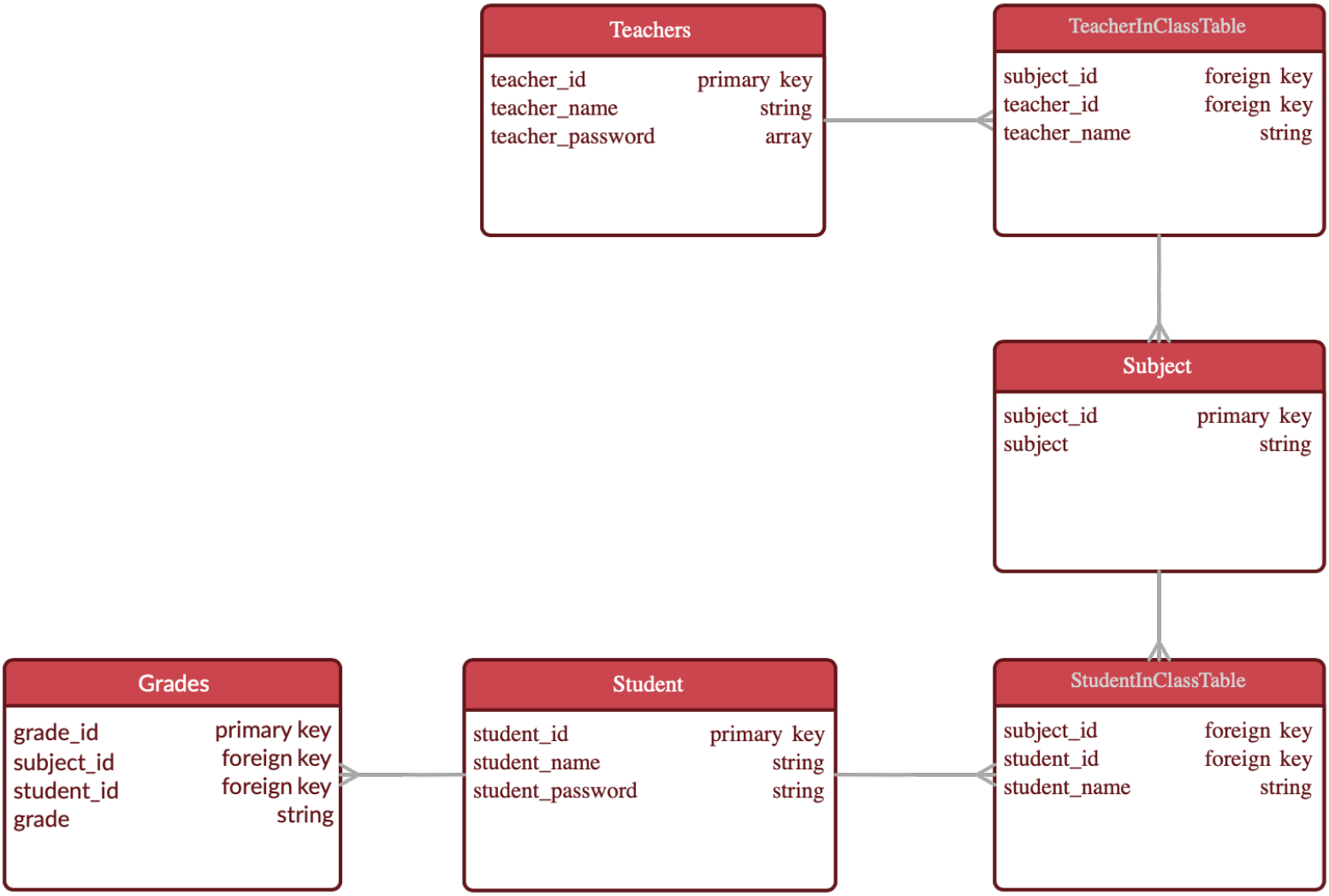
Subject ID	Subject
1	Mathematics
2	Further Mathematics
3	English Literature
4	Economics
5	Physics
6	Sociology
7	Psychology
8	Government and Politics
9	Computer Science

Table 5: *Grades Table*

Grade ID	Subject ID	Student ID	Grade
1	1	1	A
2	5	1	A
3	9	1	A*
4	7	2	A
5	6	2	B
6	4	2	B
7	8	3	B
8	4	3	A
9	1	3	A

For the database to be in second normal form, it must be already in first normal form. I have achieved this by removing repeated entries. Then, by creating multiple tables, I am separating non-key attributes apart from the primary key. In Table 2, the attributes “Grade” and “Subject” are both non-key attributes as they are not entirely dependent on the primary key which is “Student ID”. Table 3 shows that each student has a corresponding attribute which depends on the primary key showing that it is in second normal form. The attribute “Subject” has been separated from Table 2 and created a new table, Table 4, from which the primary key is “Subject ID. In Table 5, the foreign keys “Subject ID” and “Student ID” is partially dependant on the primary key “Grade ID” with the attribute “Grade” being fully dependant on it. This is because as each student is set a grade, their subject and student id would be used to identify them. As the database is now in second normal form, to achieve third normal form, we need to remove non-key attributes which are reliant upon other non-key attributes. However, my database has already got rid of it, therefore my database is fully normalised.

ENTITY ATTRIBUTE DIAGRAM



The entity attribute diagram shows all of the attributes for each table such as its primary keys and foreign keys. Attributes are details about each entity that needs to be stored, for example, a student's name and password would have to be stored in a database so that they can log into their account as the program would retrieve the data stored within the table.

DATA DICTIONARIES FOR EACH TABLE

Table Name	students		
Primary Key	student_id		
Foreign Keys	None		
Data Item	**Data type**	**Validation**	**Sample data**
student_name	String	= 150 chars	"John Adams"
student_password	String	= 10 chars	"AXHSJ12!2"

Table Name	teachers		
Primary Key	teacher_id		
Foreign Keys	None		
Data Item	**Data type**	**Validation**	**Sample data**
teacher_name	String	= 150 chars	"Blake White"
teacher_password	String	= 10 chars	"C!232LOP"

Table Name	subjects		
Primary Key	subject_id		
Foreign Keys	None		
Data Item	**Data type**	**Validation**	**Sample data**
subject	String	= 50 chars	"Mathematics"

Table Name	grades		
Primary Key	grades _id		
Foreign Keys	subject_id, student_id		
Data Item	**Data type**	**Validation**	**Sample data**
grade	String	= 2 chars	A*
date	Date	YY/MM/DD	20/01/01

Table Name	StudentInClass		
Primary Key	None		
Foreign Keys	subject_id, student_id		
Data Item	**Data type**	**Validation**	**Sample data**
student_name	String	= 150 chars	"John Adams"

Table Name	TeacherInClass		
Primary Key	None		
Foreign Keys	subject_id, teacher_id		
Data Item	**Data type**	**Validation**	**Sample data**
teacher_name	String	= 150 chars	"Blake White"

The data dictionaries for each table shows the data items that would be stored in the database. I have also identified the primary keys and foreign keys that will be required for the tables.

SAMPLE SQL STATEMENTS

Creating database:

```
CREATE DATABASE school;
```

Creating table:

```
CREATE TABLE IF NOT EXISTS students (
                    student_id SERIAL PRIMARY KEY,
                    student_name VARCHAR(150),
                    student_password VARCHAR(10)
                    );
```

Displaying a specific student:

```
SELECT student_name FROM students WHERE student_id = ?;
```

Displaying a specific teacher:

```
SELECT teacher_name FROM teachers WHERE teacher _id = ?;
```

Adding a student to the database:

```
INSERT INTO students (student_id, student_name, student_password) VALUES (?,?,?);
```

Adding a teacher to the database:

```
INSERT INTO teachers (teacher_id, teacher_name, teacher_password) VALUES (?,?,?);
```

Displaying students from a class:

```
SELECT student_name FROM studentinclass WHERE subject_id = ?;
```

Displaying teachers from a class:

```
SELECT teacher_name FROM teacherinclass WHERE subject_id = ?;
```

Retrieving students' grades from a specific subject:

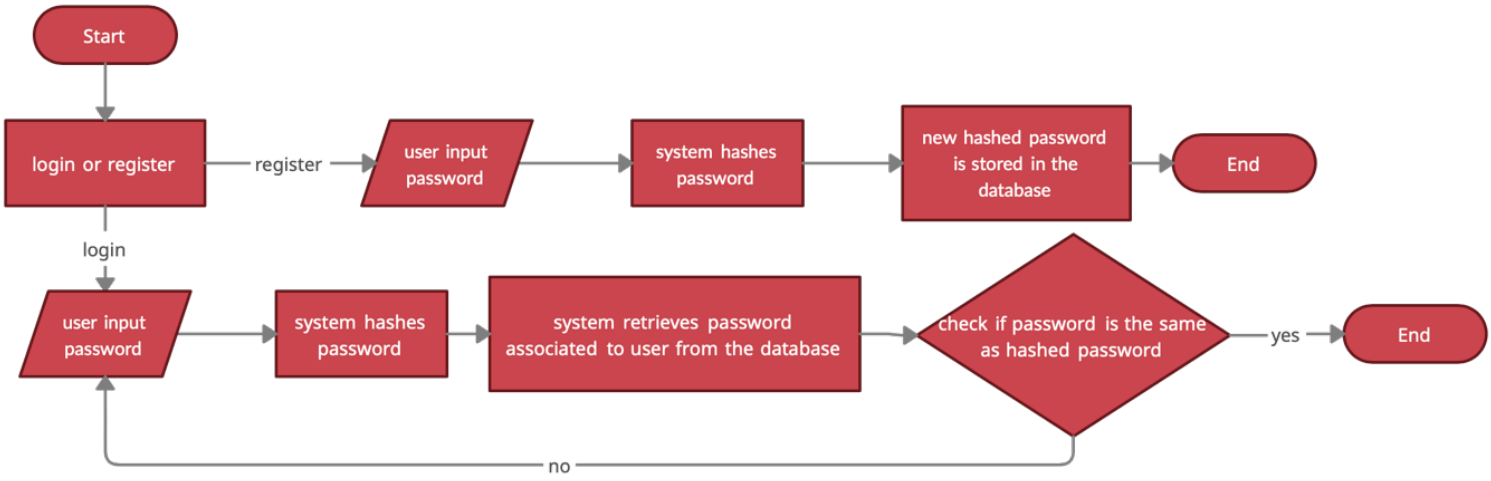
```
SELECT grade_id FROM grades INNER JOIN students ON grades.student_id =
students.student_id WHERE subject_id = ? AND student_id = ?;
```

CHOOSING BETWEEN MYSQL AND POSTGRESQL

As I am potentially going to use a database to store the given data, I would have to choose between two strong open-source relational databases which are MySQL and PostgreSQL. The unique functionality of PostgreSQL is that it supports many data types which MySQL doesn't support such as arrays, so if I were intending on storing collective data into the database, PostgreSQL is more reliable. The main difference between these two databases is that PostgreSQL is an object-relational database whereas MySQL is a purely relational database meaning that I would be able to use methods such as table inheritance. However, when it comes to speed, MySQL is faster at reading in comparison to PostgreSQL which is slower at reading but can still write large amounts of data efficiently. Due to PostgreSQL being a more advanced relational database, I am going to use it if I were to use a database for my technical solution.

ENCRYPTION VS HASHING

When a user's data such as a password is going into the database, the data must be securely stored. This is because, if the database is hacked, the data can be stolen and hackers would have everyone's data in their hands, hence, I need to decide how I would store the user's password. Encryption is the process of converting plain text into a scrambled form making it unreadable whereas hashing is one-way encryption as the data cannot be reverted to its original form once hashed. For my system, I am going to use hashing as it is more unlikely for the data to be reverted once leaked as if I used encryption, the data can still be decrypted using the key that was used to encrypt the data. In Python, I would use argon2 module to hash the password\. For example, if the user's password was "John123", it's hashed form would be "$argon2i$v=19$m=16,t=2,p=1$SjQyTUdtbWh3OWNTWTdOTw$GmblTgxOGnfx3hO7Q3u7LQ" (this was generated using https://argon2.online). The following process shows how the system would check if the password is hashed.



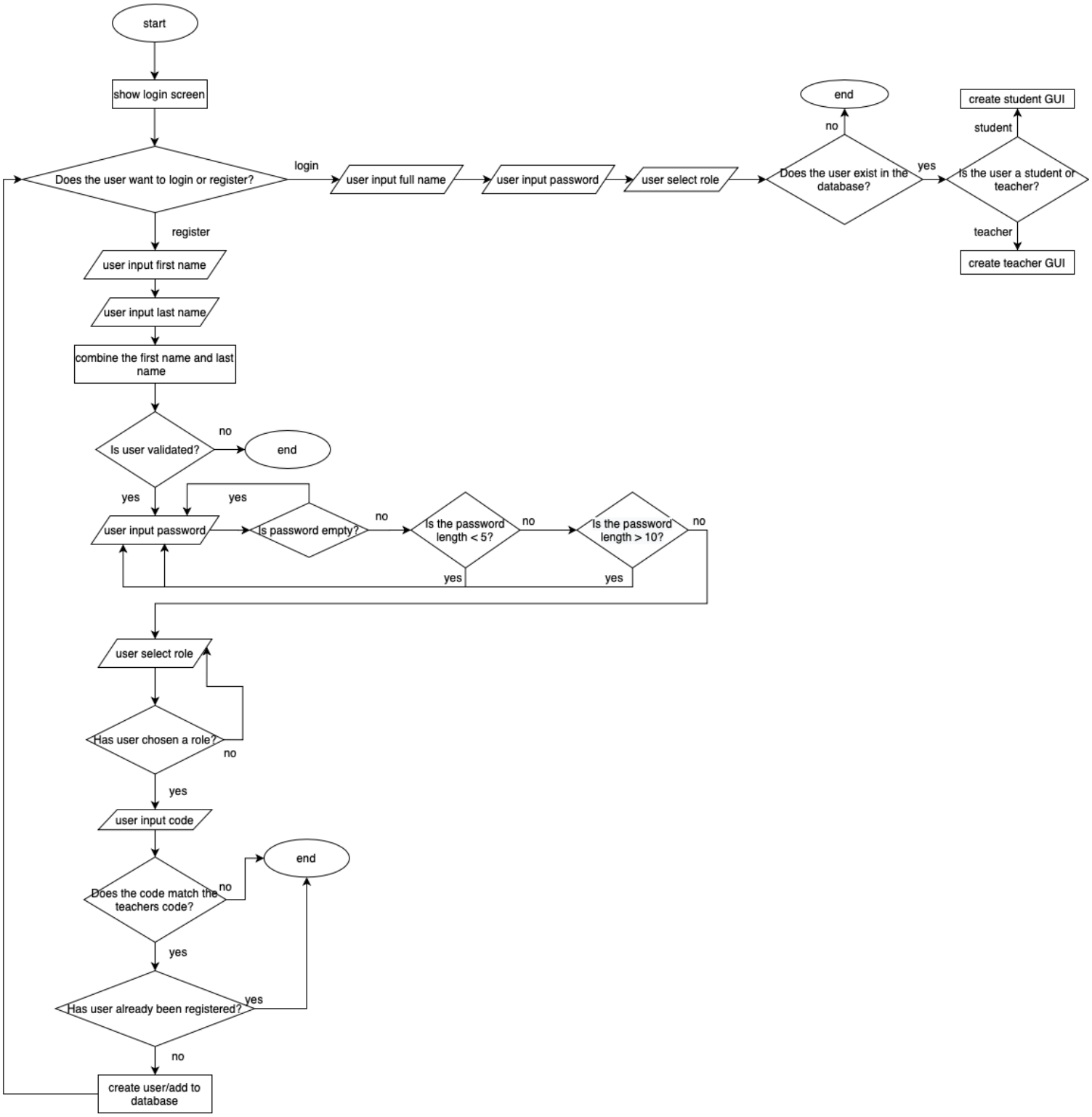
DATA DICTIONARY

Data Item	Data Type	Validation	Description	Example
First Name	String		The first name of the user.	“John”
Last Name	String		The last name of the user.	“Apple”
Full Name	String	Must be authorised.	The full name of the user.	“Gary Oaks”
Password	String	5 ≤ length of password ≤ 10	A set of characters/numbers or symbols used to gain access to the grading system through the authentication system.	“John121” “352KL!?!”
Code	String	Must be either “STARK2021” or “TEARK2021”	The code will validate if the user if authorised for the desired role.	“TEARK2021” would be for a teacher.
Role	String	Must be either “Teacher” or “Student”	The role the user identifies as.	“Teacher”
Student				
Teacher				
Validated	Boolean			
Students	Array			
Subjects	Array			
inClass	Boolean			
Grades	Array			

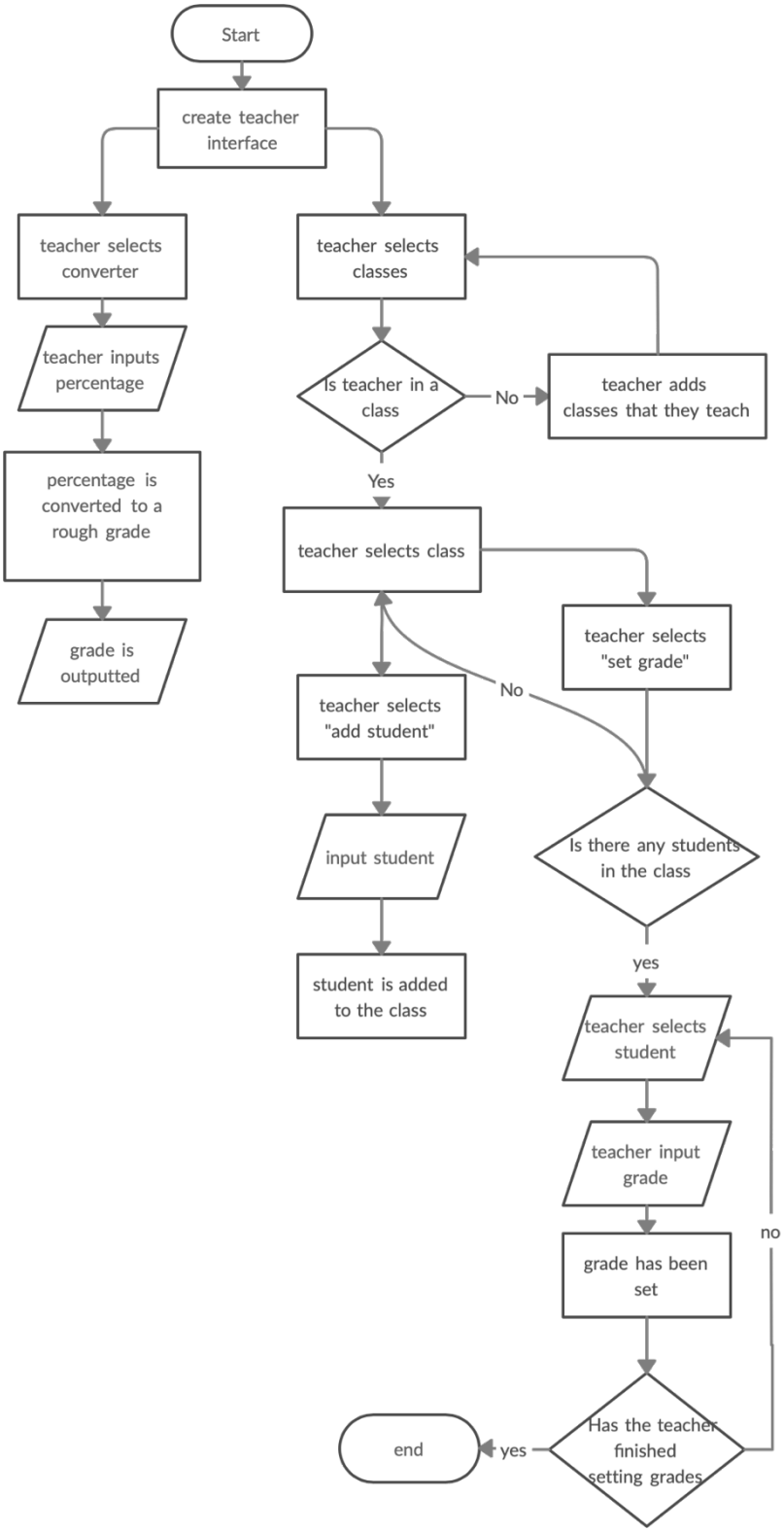
The table above contains the data which are going to be used in my system. Certain data items are going to follow a validation for my system such as when the user is registering, their password’s length must be in a certain range.

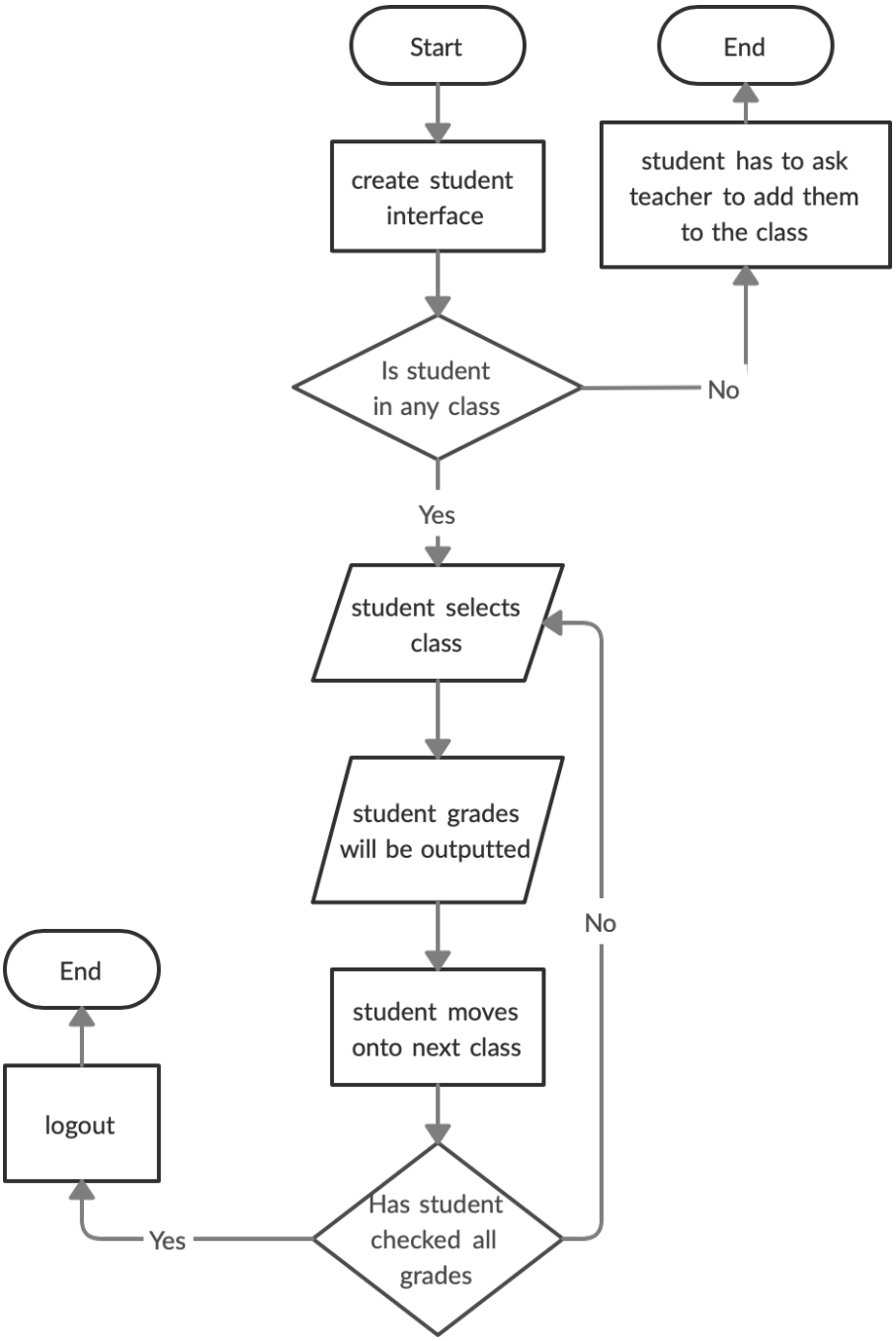
SYSTEM FLOWCHART

AUTHENTICATION SYSTEM



TEACHER INTERFACE

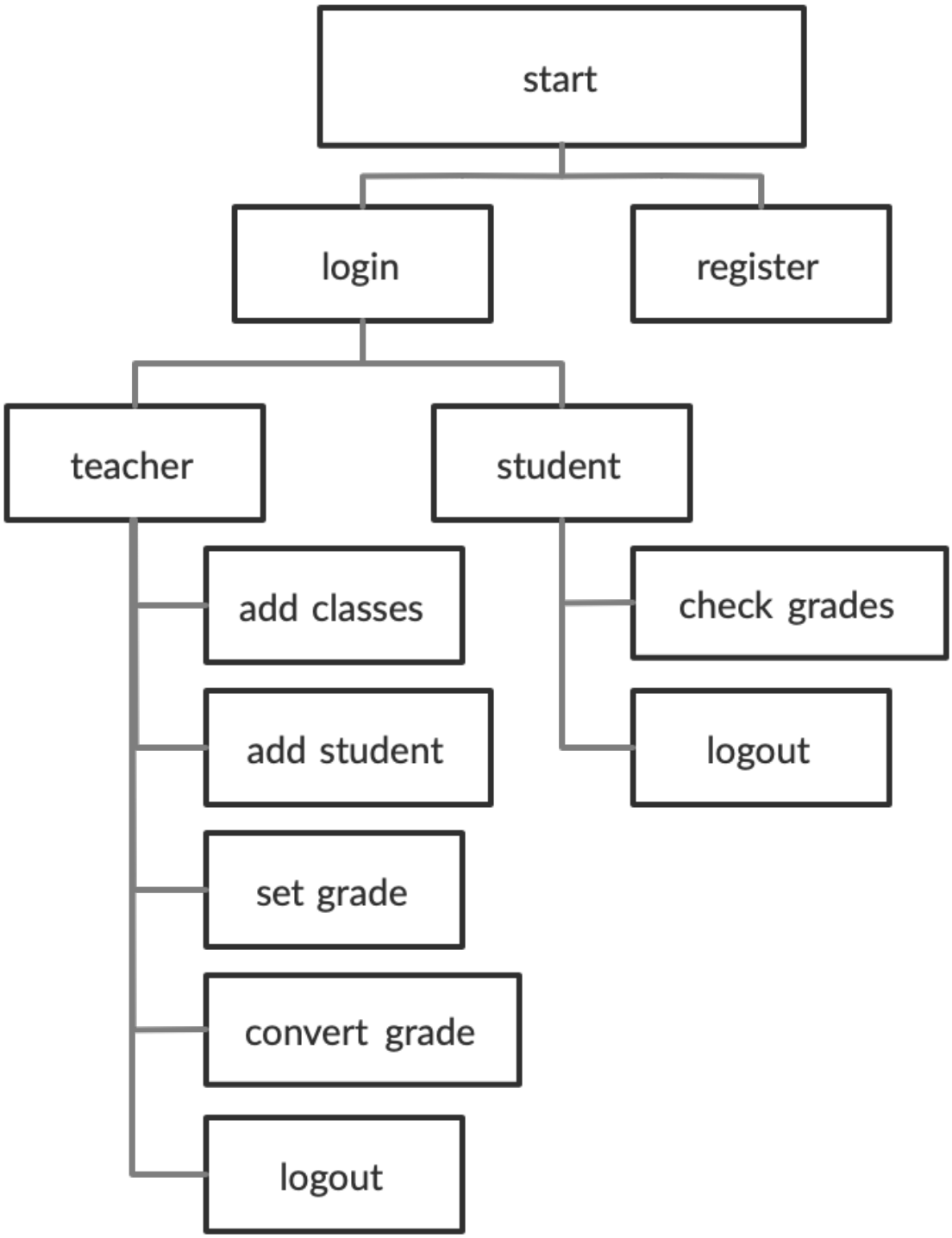




The system flowchart I have designed for the new system is much more advanced than the algorithm the old system had used. The main difference in the systems is that there is an authentication system which is used to differentiate students and teachers along with their corresponding functionalities they are limited to. In the old system, the algorithm was complicated for users who had not used a spreadsheet software before.

SYSTEM DESIGN

TOP DOWN DESIGN

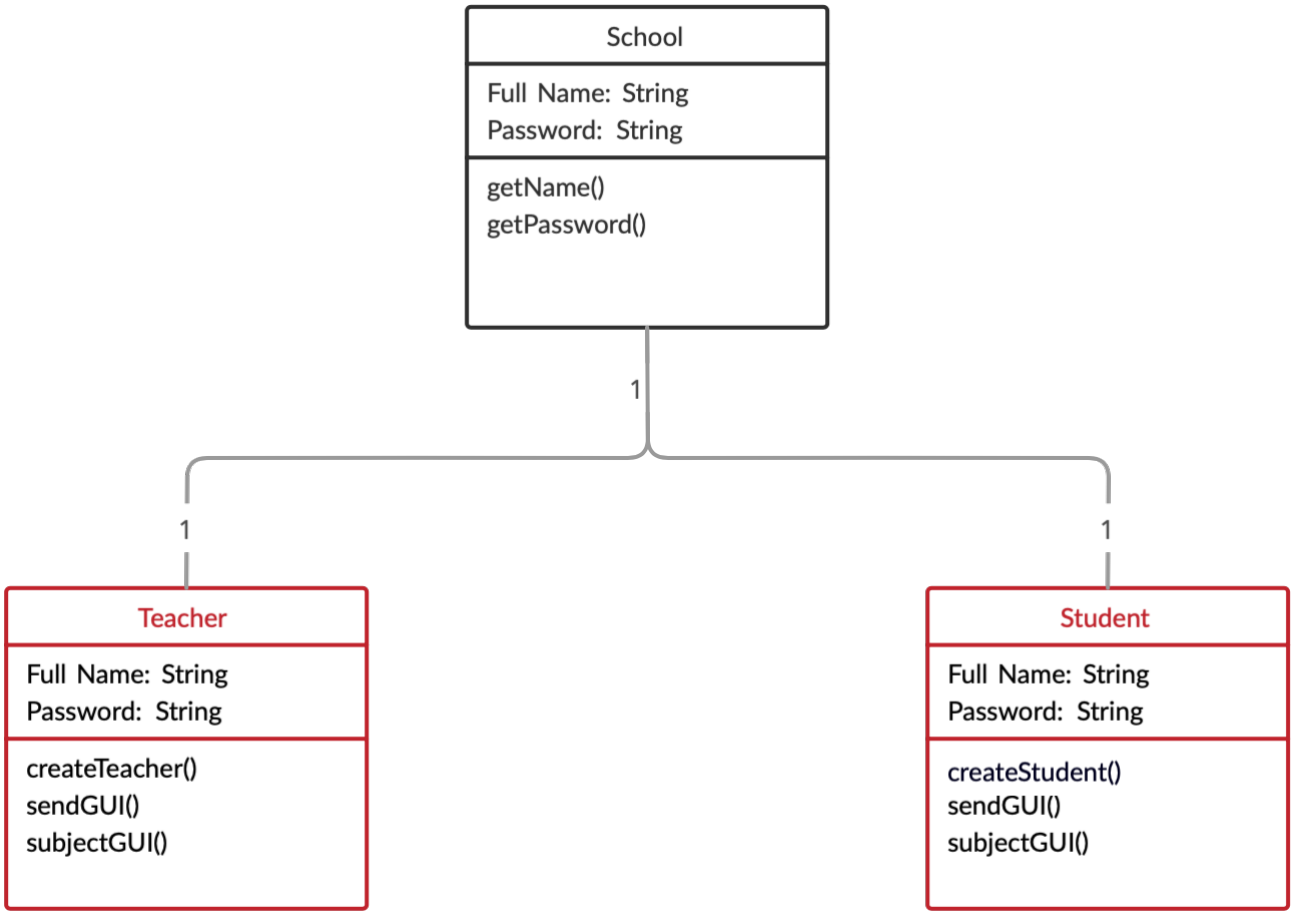


IPSO

INPUT	PROCESS	STORAGE	OUTPUT
- First Name - Last Name - Password - Role - Code - Grade	- Check if the password is valid. - Check if the user is authorised. - Add user to database if they do not exist. - Check that the full-name and password are correct in order for them to login. - Update the subjects available from subjects.txt into database. - Update the student's grade from the Tkinter window into the student database.	- Student Database - Teacher Database - Class Database - Text file	Tkinter window

In my system, data would need to be validated and checked by the database so there is more process compared to the current system. I would be storing student data and teacher data in separate databases so that the databases are differentiated.

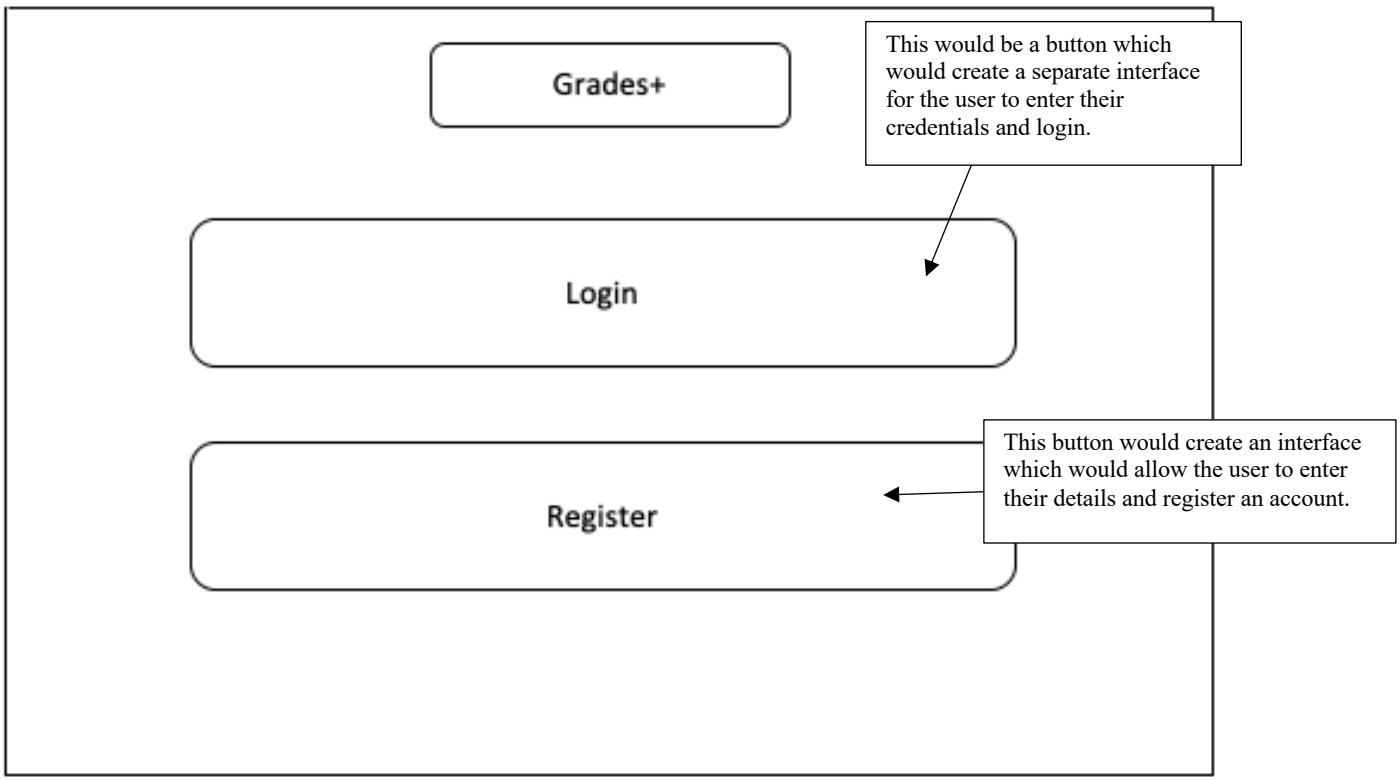
OOP CLASS DESIGN



The classes shown are what I would be using as I intend to write the code in the OOP programming paradigm. The school class would be the base/parent class and teacher/student class would be the subclasses which inherits the methods “getName()” and “getPassword()” from the base class. I’ve modelled the students and teachers as separate objects as they have different methods. The objects would be instantiated once the user has logged in. I have decided to use OOP as it would be easier for me to add functionalities for the separate interfaces.

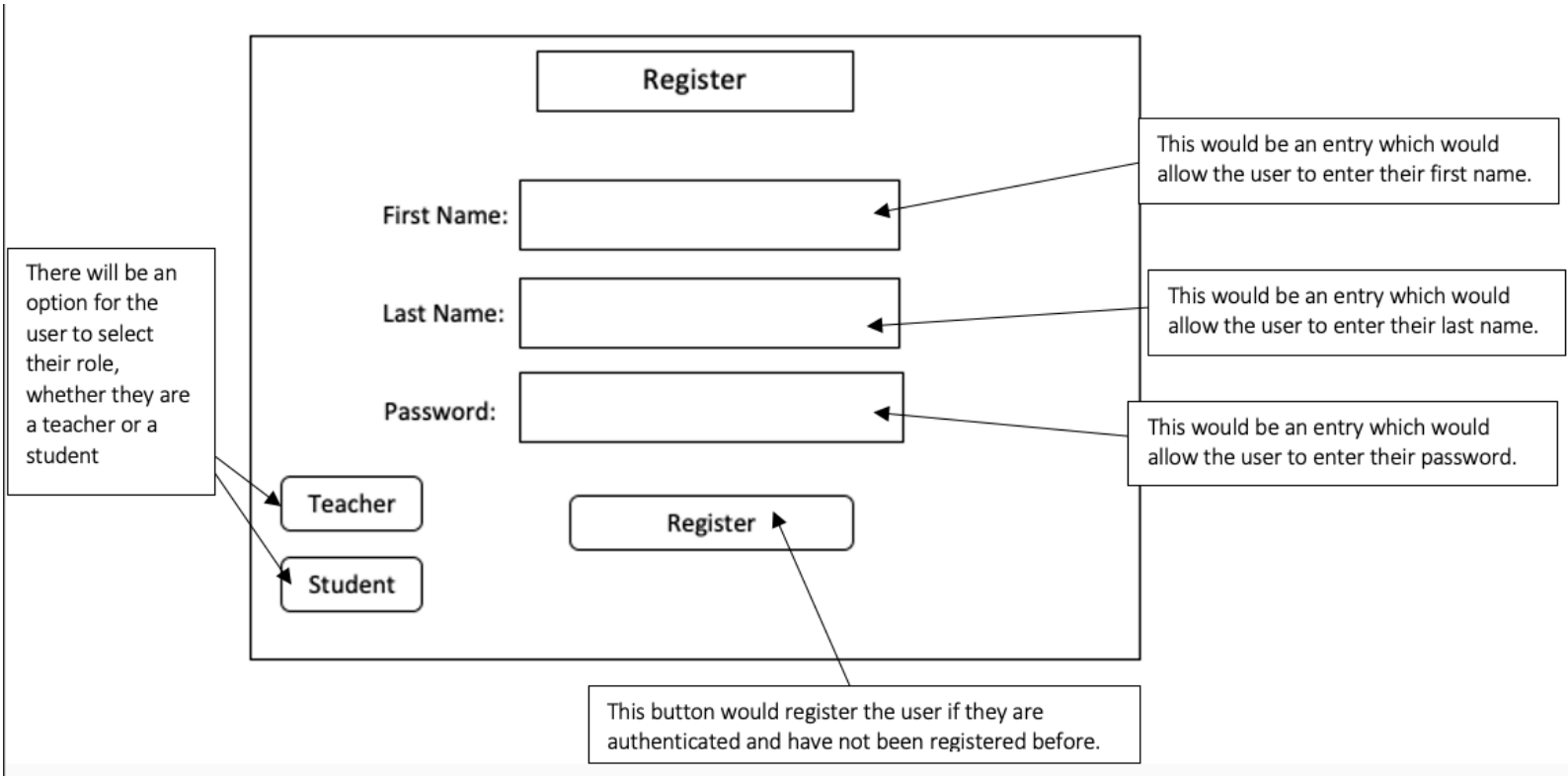
INTERFACE DESIGN

MENU INTERFACE



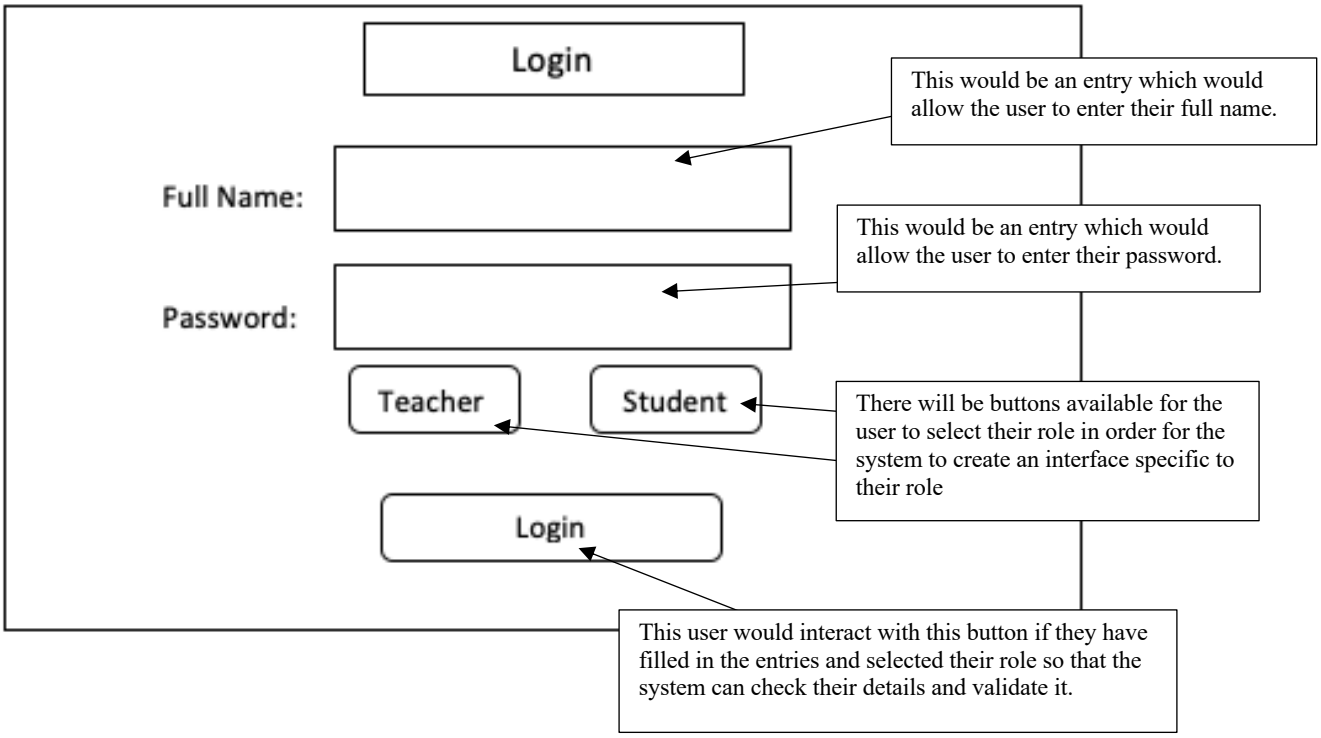
The menu interface is an interface that would appear when the program is first running. It would give the user an option to either login or register in the form of buttons. As illustrated above, I have tried to design my menu interface to be able as simple as possible as if I were to create an extremely complicated interface, it would be hard for the user to understand how the system works.

REGISTER INTERFACE



If the user chose the register option through the menu interface, a register form would appear allowing the user to enter their details and select their role. The register button would run an algorithm to check if the user is authorised and if their password meets the criteria. When the user has registered, their credentials would be stored, and they would be able to login.

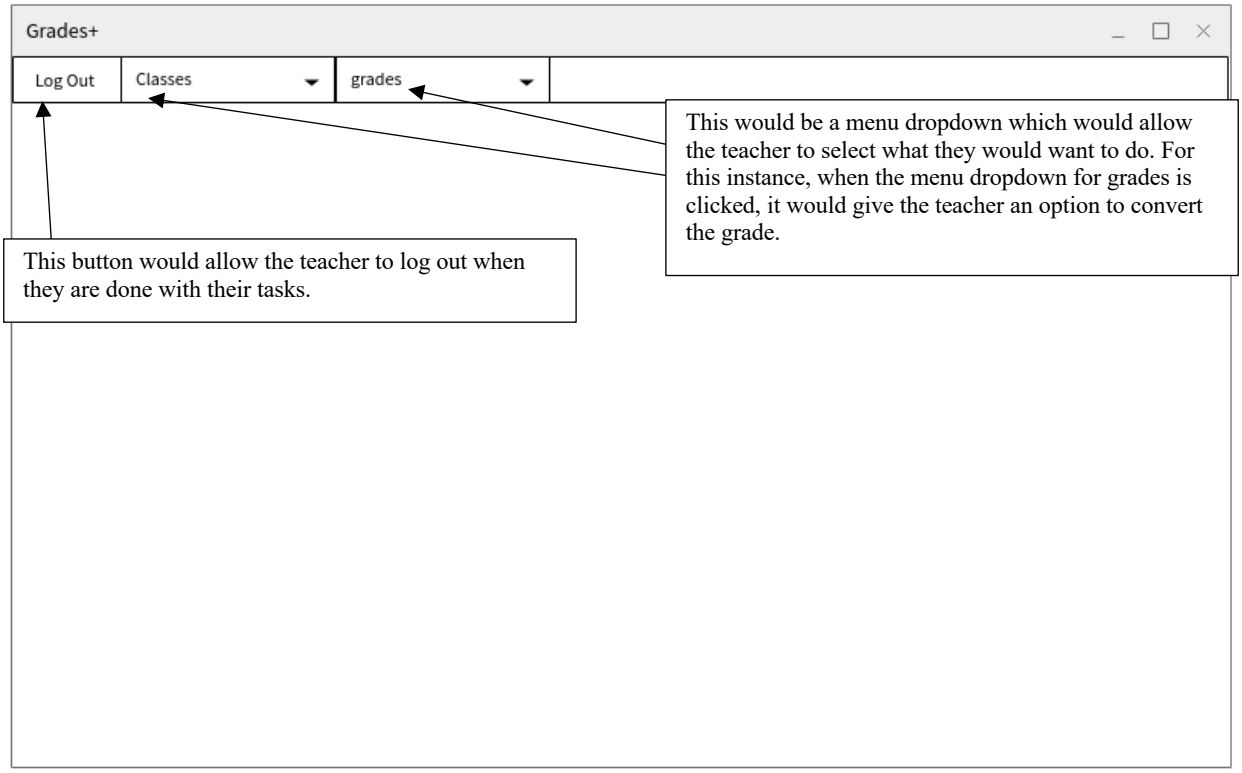
LOGIN INTERFACE



A login form will appear if the user chooses the login option at the menu interface. The role buttons are key to this interface as they would specify what interface to give the user. When the user enters their credentials, the system would verify it and then create a new interface dedicated to their role. This is because the two interfaces, student and teacher, have different functionalities. Once logged in, their interface would appear.

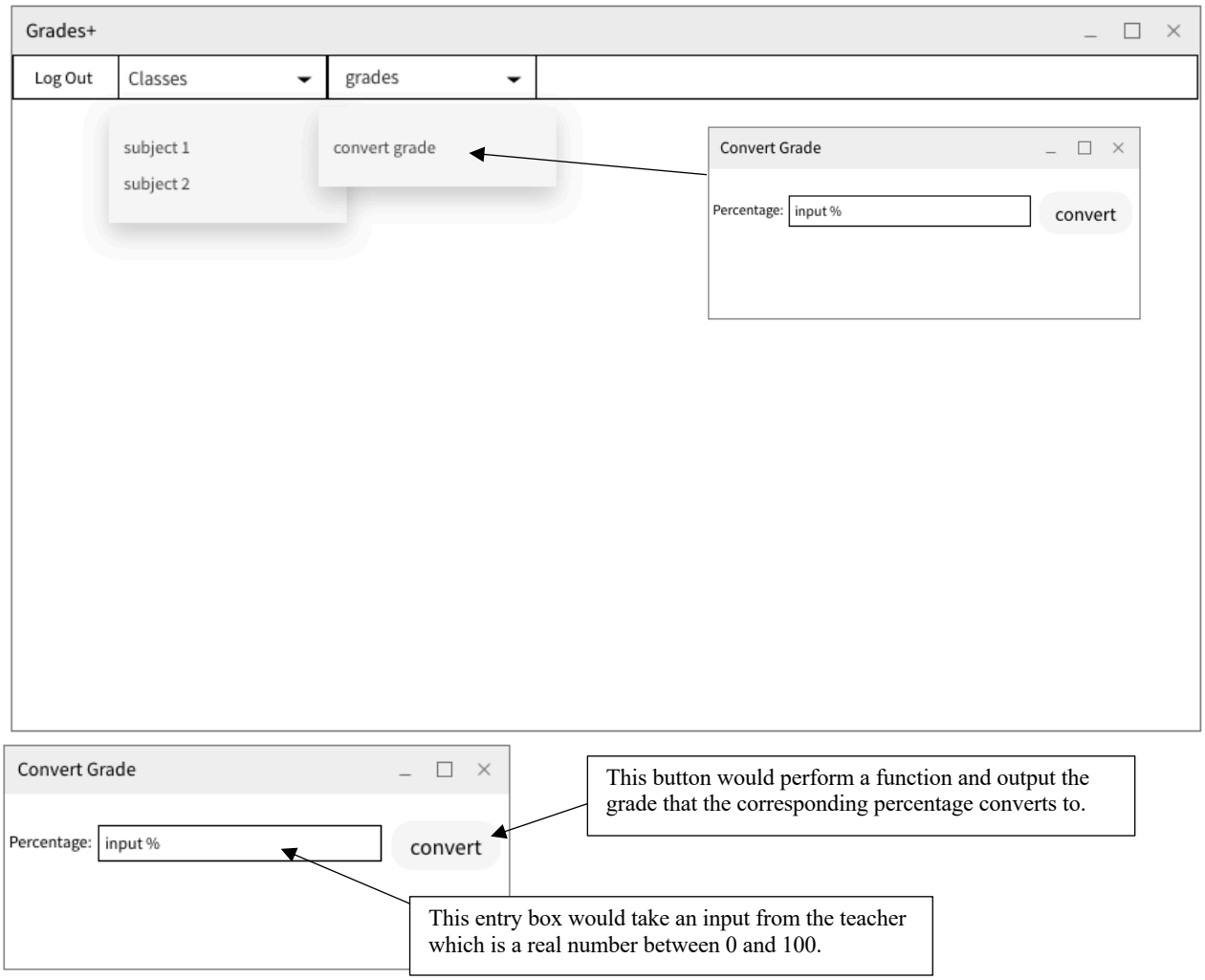
TEACHER GUI

MAIN INTERFACE



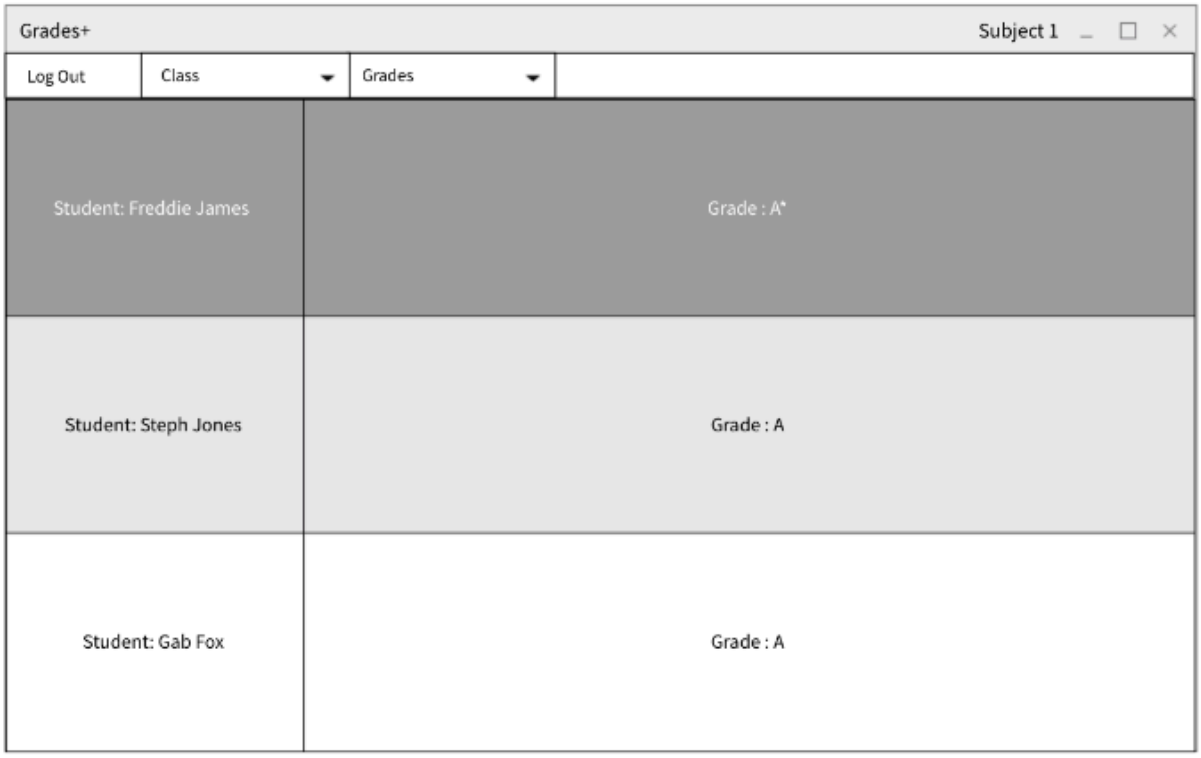
Supposed that the user was authenticated as a teacher, the three main options available are “log out”, “classes” and “grades. They are the functions which the teachers have access to. As they are menu dropdowns, there are sub-functions such as “convert grade”.

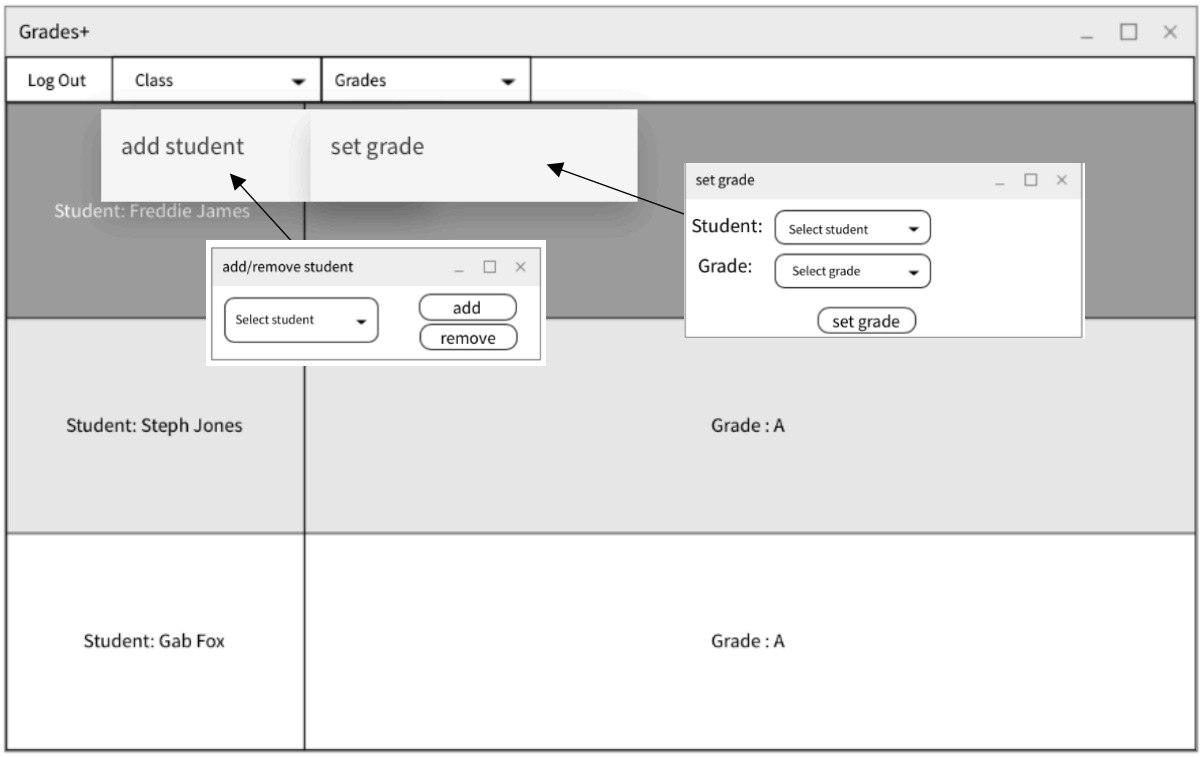
When the menu dropdown(s) is clicked:

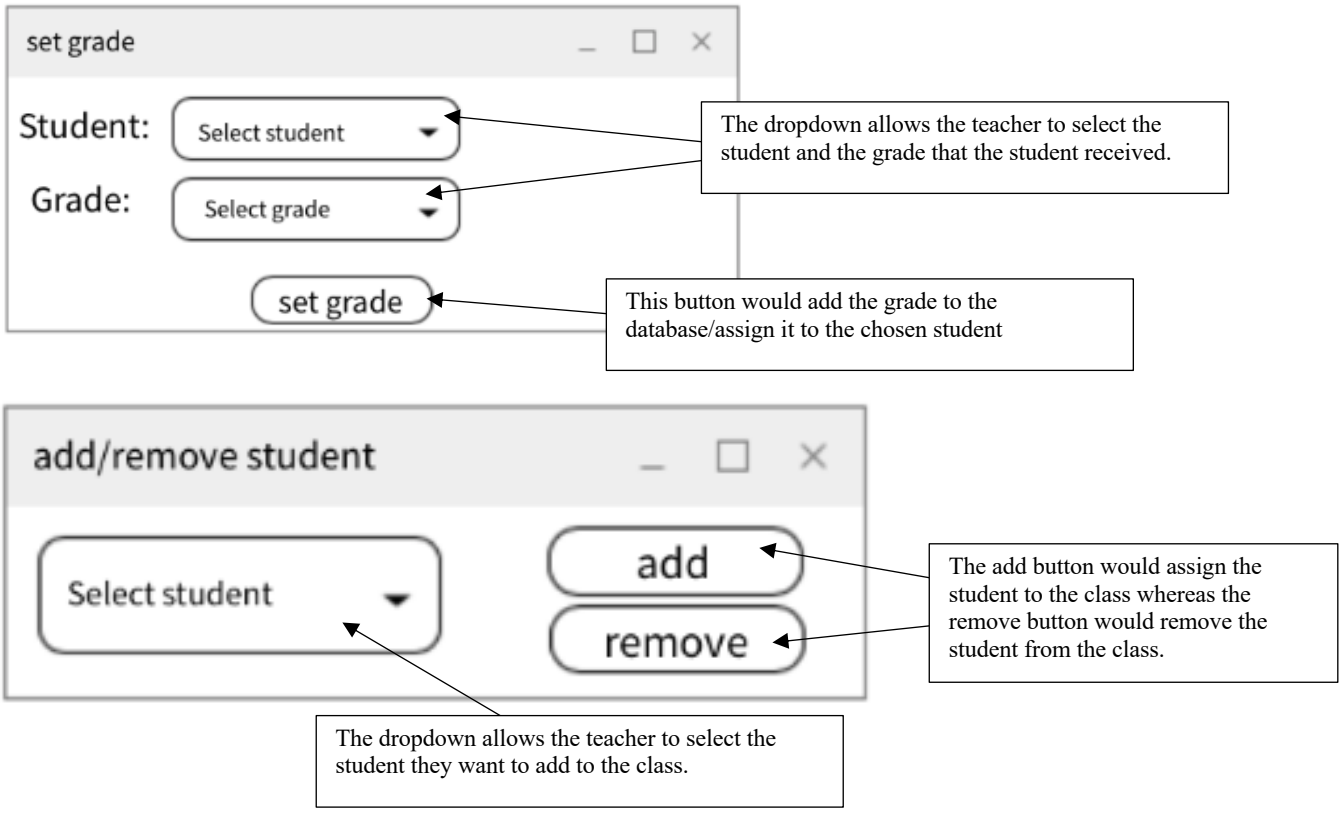


The convert grade function is used if the teacher does not have a grade boundary. It takes an input which would be the percentage the student achieved overall, and using a base grade boundary, the percentage is converted into a grade and then outputted. Under the class's dropdown, the teacher would also be able to add the class(es) that they teach.

CLASS INTERFACE



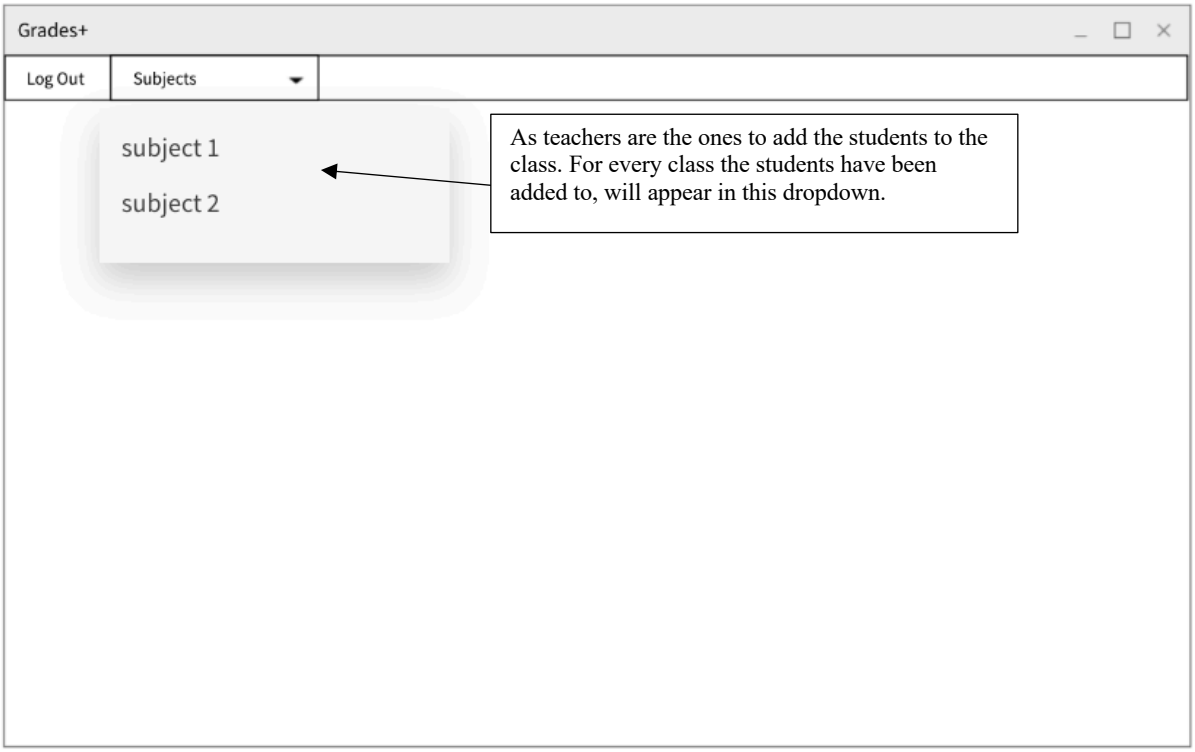




If the user opens the class interface, they would be able to see the grades that was set for the students in their designated class. They are also able to add students to the class and set grades. The designs above show how both functions work.

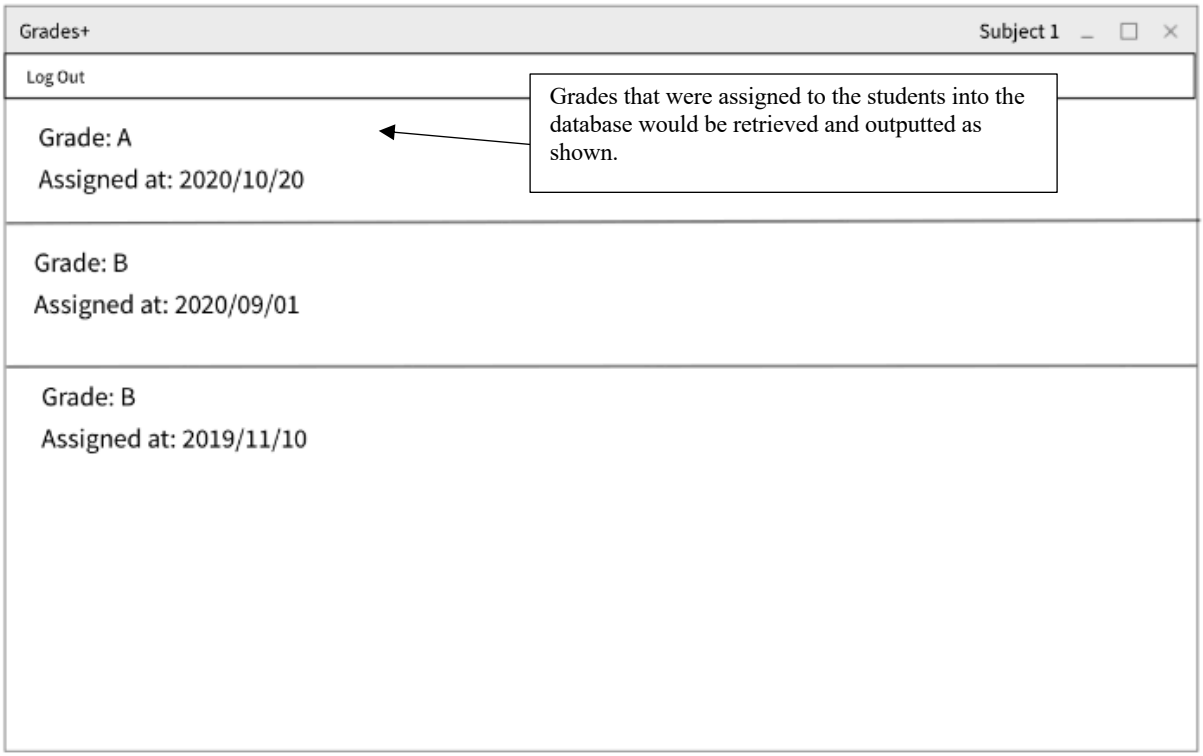
STUDENT INTERFACE

MAIN INTERFACE



The student interface is much simpler than the teacher's interface as the teacher's interface is the backbone for the student interface. If the student is not in any class, they would have to ask the teacher(s) who are in the class to add them to it otherwise they would not be able to access their grades.

CLASS INTERFACE



When the student has selected the class from the dropdown, they would be able to see the grades that were set by the teacher. This would be in order of the date that the teacher assigned the grade so that the student can track how much progress they have made with their exams.

PSEUDOCODE

REGISTER USER

Pseudocode

```
SUB PROCEDURE registerUser
  name ← Input("Name")
  password ← Input("Password")
  role ← Input("Role")

  IF name IS authorised THEN
    IF LENGTH(password) > 5 THEN
      OUTPUT "Password too long"
    ELSE
     IF role ← "teacher" THEN
       OUTPUT "You have been registered as a teacher."
     ELSEIF role ← "student" THEN
       OUTPUT "You have been registered as a student."
  ELSE
    OUTPUT "You are not authorised"
END PROCEDURE
```

Explanation

Before the user can access the functionalities, they must be registered so that the program can recognise them if they were to login. In order to do this, the user needs to input their name, password and role. The role is what distinguishes the user.

LOGIN USER

Pseudocode

```
SUB PROCEDURE loginUser
  name ← Input("Name")
  password ← Input("Password")

  IF name ← storedName THEN
     IF password ← storedPassword THEN
       OUTPUT "You have logged in."
     ELSE
       OUTPUT "You failed to login."
  ELSE
    OUTPUT "You have not registered!"
END PROCEDURE
```

Explanation

After the user has registered, to access the system, they would need to login. The login system requires the user to input their name and password. The system would check if the name is the same as a stored name and then check the passwords. If it is a match then the user would be able to access the system and its functionalities, otherwise, they would not be able to access it.

SET GRADE TO THE STUDENT

Pseudocode

```
SUB PROCEDURE addGrade
  name ← Input("Students Name")
  class ← Input("Class")
  grade ← Input("Grade")

  IF name in class THEN
    OUTPUT "Grade has been added to this student."
  ELSE
    OUTPUT "Student is not in this class!"
END PROCEDURE
```

Explanation

For the teacher to set a grade to the student, they would have to input the student's name, class and the grade that they achieved. The system would check if the student is in the class and if they are, the grade would be saved in the system for the student to access later on.

RETRIEVE GRADE FROM THE SYSTEM

Pseudocode

```
SUB PROCEDURE getGrade
  student ← Input("Name")
  class ← Input("Class")
  IF name in class THEN
    OUTPUT "Your grades are:"

    FOR grade IN system.grades(student, class)
      OUTPUT grade
    NEXT grade

  ELSE
    OUTPUT "You are not in this class!"
END PROCEDURE
```

Explanation

After a teacher has set the grades to the system, the students would be able to see it. Through the student's interface, when the student opens the interface for their subject, the grades that were set would appear.

CONVERT GRADE

Pseudocode

```
SUB PROCEDURE convertGrade
  percentage ← Input("Percentage")
  IF percentage > 80 THEN
    OUTPUT "A*"
  ELSEIF percentage ≥ 70 THEN
    OUTPUT "A"
  ELSEIF percentage ≥ 50 THEN
    OUTPUT "B"
  ELSEIF percentage ≥ 40 THEN
    OUTPUT "C"
  ELSEIF percentage ≥ 30 THEN
    OUTPUT "D"
  ELSE
    OUTPUT "U"
  ENDIF
END PROCEDURE
```

Explanation

If a teacher did not have a fixed grade boundary, the system has one in-built which approximates the grade according to the percentage. The algorithm follows a check whether the grade falls into the range and then outputs the grade.

ADD/REMOVE STUDENT FROM A CLASS

Pseudocode

```
student ← Input("student")
class ← Input("class")
decision ← Input("Do you want to add or remove a student?")
IF teacher in class THEN
  IF decision ← "add" THEN
    IF student in class THEN
      OUTPUT "Student is already in this class."
    ELSE
      OUTPUT "Student has been added to this class."
  ELSEIF decision ← "remove" THEN
    IF student in class THEN
      OUTPUT "Student was removed this class."
    ELSE
      OUTPUT "Student is not in this class."
ELSE
  OUTPUT "You are not in this class."
ENDIF
```

Explanation

An important role that the teachers play in the new system is that they would have to manually add the students to their corresponding class(es). If this does not happen then teachers would be incapable of assigning grades to the students.

TECHNICAL SOLUTION

FILES

The files my system would be using for its functionalities are as shown above. The "subjects.txt" file exists to store the names of all the subjects that are being taught in the 6th form. The first time the program is running, all of the subjects from the text file are added to the database. Then, every time the program is run after the first time, it would check if there are any new subjects in the file and then add it to the database. The two files, "validstudents.txt" and "validteachers.txt", are essential to check if the user registering is a member of the 6th form. Each student that is valid would be stored in this text file by the head of 6th form. The authentication system would read the file and check if the user exists in that list. The "main.py" and "School.py" is where the code is. The user would run the system through "main.py".

DATABASE

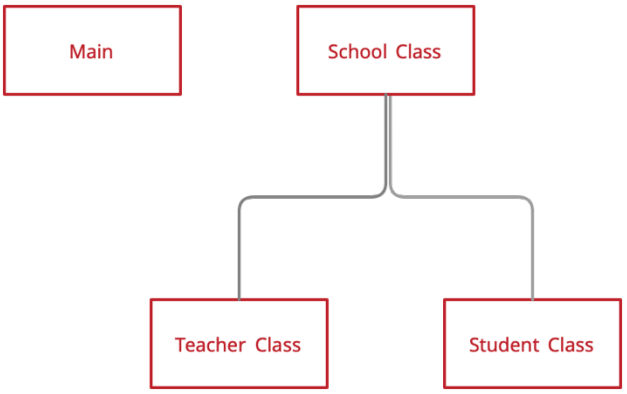
I have decided to use databases to store the user's data and student's grades. In order to use it, I would be using PostgreSQL to create the main database. Then in my code file, it would have SQL queries which would create the table (if it doesn't exist) when the program is first running. I would be using PSequel, which is a GUI client for PostgreSQL, to create queries and be able to see the tables and the data that is stored in it. It would also allow me to show that the program can insert data and correctly retrieve the data that is stored.

Creation of database:

```
postgres=# create database school;
CREATE DATABASE
postgres=#
```

(OOP) CLASSES

The following OOP class design shows the classes I am going to be using in my code and the relationship between it. As shown, the base class School would have two sub-classes teacher and student which would inherit the functions from the school class. The main class would import the sub-classes from the school class in order to instantiate the objects.



SCHOOL CLASS

IMPORT MODULES

```
#import modules which would be used for the system
from tkinter import * #used for gui
from tkinter import ttk
from tkinter import messagebox
from argon2 import PasswordHasher #used to hash the password
import time
import psycopg2 #used for the database
```

SCHOOL CLASS (BASE)

```
class School:
    def __init__(self, userFullName, password):
        #this would store the user's properties so that it can be used throughout the program.
        self.userFullName = userFullName.lower().strip()
        self.password = password

    #this function would return the attribute "userFullName"
    def getName(self):
        return self.userFullName

    #this function would return the attribute "password"
    def getPassword(self):
        return self.password
```

The __init()__ function is used to instantiate the teacher/student objects. The parameters of the _init()__ function such as the user's name and password can be retrieved using the getName() and getPassword() functions.

MESSAGEBOX PROCEDURE

```
#this procedure would create a messagebox when required in my system
def messageBox(self, textLabel, messageType):
    if messageType == "error":
        #this is an error box which would be created when the input is error.
        messagebox.showerror("Grades+", "Error: " + textLabel)
    elif messageType == "info":
        #this creates an info box
        messagebox.showinfo("Grades+", textLabel)
```

The message box function is designed to take two parameters, the text that the program would output and the type of message box, then it would create a messagebox using Tkinter's GUI functions. I would be using this function to make the user aware of anything, for example, an errorbox would appear if the user inputs an invalid password.

ISPASSWORDVALID FUNCTION

```
#this function checks if the password is valid
def isPasswordValid(self, password):
    if not password: #checks if the string is empty (stirng-falsy)
        #this would show errorbox as the entry is invalid
        self.messageBox("Enter a password.", "error")

    #this condition checks if the password's length is less than 5
    elif len(password) < 5:
        #as the password is smaller than 5 characters, it would throw an errorbox.
        self.messageBox("Password must be atleast 5 characters long.", "error")
        return False
    #checks if the password's length is more than 10
    elif len(password) > 10:
        self.messageBox("Password cannot be longer than 10 characters. ", "error")
        return False
    else:
        return True #if password is valid, the function would return true
```

The isPasswordValid() function takes a parameter which would be the password and check if it is valid through a small algorithm. The password would have to be between 5-10 characters long for it to be valid. If the password does not meet the requirements, an error would be thrown at the user.

TEACHER CLASS

CREATETEACHER FUNCTION

```
#this class is a subclass which inherits the methods of it's parent class which is School.
class Teacher(School):
    #this function would follow an algorithm in order to add the teacher to the database
    def createTeacher(self, window):
        #this creates a connection to the database
        connection = psycopg2.connect(host = "localhost", user = "postgres", password = "postgres", port = "5432", database = "school")
        cursor = connection.cursor()

        #Here I am opening and reading the lines of the valid teachers textfiles to check if the corresponding teacher is valid.
        teachersFile = open("validteachers.txt", "r")
        validTeachers = teachersFile.readlines()
        validated = False
        #looping through the file to see if the teacher is in it
        for teacher in validTeachers:
            if self.getName() == teacher.strip().lower():
                validated = True
                #this sql statement is used to check if the teacher exists in the database
                cursor.execute("SELECT teacher_name FROM teachers WHERE teacher_name = %s;", (self.getName(),))
                if cursor.fetchone() is not None:
                    #teacher exists in the database since value is not null
                    self.messageBox("Already registered.", "error")
                else:
                    #the teacher is valid
                    #the system would checks if password is valid
                    if self.isPasswordValid(self.getPassword()) == True:
                        #as the teacher does not exist; the sql statement would add the teacher to the database
                        hashedPassword = PasswordHasher().hash(self.getPassword())
                        cursor.execute("INSERT INTO teachers (teacher_id, teacher_name, teacher_password) VALUES (DEFAULT, %s, %s);", (self.getName(), hashedPassword))
                        connection.commit()
                        self.messageBox("Success! You have been registered.", "info")
                        window.after(3000, window.destroy)

        #if value of validated did not change then that means the student is not valid
        if validated == False:
            self.messageBox("You are not a validated teacher.", "error")

        connection.close()
```

The Teacher class is a sub-class of the School class and provides the main functionalities of my system for the users who are classed as a Teacher. The createTeacher() function works with the registerUser() function to add the user to the database. If the user is validated, a teacher object would be instantiated, and it would follow an algorithm to check if the user already exists in the database so that they could insert their data in the database if not.

SENDGUI FUNCTION

```
#this function is the core as it provides the main interface for the teacher.
def sendGUI(self):
    connection = psycopg2.connect(host = "localhost", user = "postgres", password = "postgres", port = "5432", database = "school")
    cursor = connection.cursor()

    #this creates the window for the interface
    gui = Tk()
    gui.title("Grades+")
    gui.geometry("1500x800")
    menubar = Menu(gui)

    #this procedure would stop the program from running
    def logOut():
        exit(0)

    #this function is for the class interface
    def subjectGUI(subject):
        #this would create the class interface
        subjectGUI = Toplevel(gui)
        subjectGUI.title(subject)
        menubar = Menu(gui)

        #the treeview method is being used to show the teachers the grades that were assigned to the teacher
        gradesTable = ttk.Treeview(subjectGUI, columns = ("Name", "Grade", "Date assigned"), show = "headings")
        gradesTable.heading("Name", text = "Name")
        gradesTable.heading("Grade", text = "Grade")
        gradesTable.heading("Date assigned", text = "Date assigned")
        gradesTable.grid()

        #this sql statement retrieves the subject id for the subject arguement that was passed into the function
        cursor.execute("SELECT subject_id FROM subjects WHERE subject = %s;", (subject,))
        subjectid = cursor.fetchone()

        #this parameterised sql statement is used to retrieve the student's name grade and the date that the grade was assigned on.
        cursor.execute("SELECT students.student_name, grades.grade, grades.date FROM students INNER JOIN grades ON grades.student_id = students.student_id WHERE grades.subject_id = %s;", (subjectid,))

        data = cursor.fetchall()
        for i in range(len(data)):
            #the student's data is being added onto the treeview
            gradesTable.insert("", "end", values = (data[i][0], data[i][1], data[i][2]))
```

The sendGUI() function is called when the teacher logs onto their account. This function has other functions nested in it as they are called whilst the teacher uses the system. The logOut() procedure would be called when the teacher selects the dropdown option to sign out of the app as they have finished using its functionalities. The subjectGUI() function would be called when the teacher has chosen the subject that they would be checking/making changes in. Once the teacher has chosen the subject, this calls the subjectGUI() function which creates the interface for the subject. It also shows the student's grades that were previously stored in the form of a tree view.

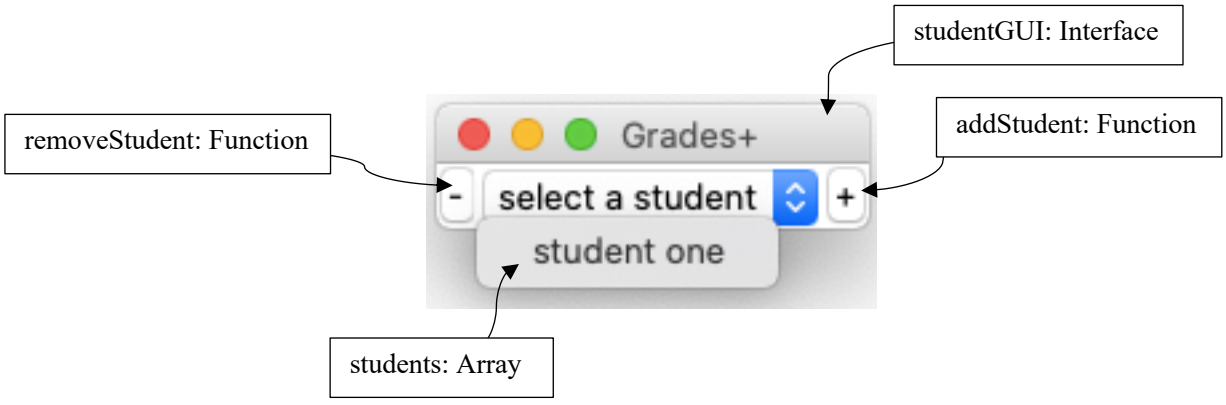
STUDENTGUI FUNCTION

The StudentGUI() function is nested in the subjectGUI() function as the functionalities exist in a menu dropdown which is in the subject's interface.

addStudent Function

```
#this function is designated to adding students to a class
def addStudent():
    cursor.execute("SELECT subject_id FROM subjects WHERE subject = %s;", (subject,))
    subjectid = cursor.fetchone()

    #this sql statement retrieves all students that are in the class
    cursor.execute("SELECT student_name FROM studentinclass WHERE subject_id = %s AND student_name = %s;", (subjectid, s.get(),))
    inClass = False

    if cursor.fetchone() is not None:
        #the student is in class so it throws an error to the teacher
        self.messageBox("This student is already in this class.", "error")
        inClass = True

    #if the student is not in the class, then they would be added to the class and
    if inClass == False:
        if s.get() != "select a student":
            self.messageBox(s.get() + " was added to this class.", "info")

            cursor.execute("SELECT subject_id FROM subjects WHERE subject = %s;", (subject,))
            subjectid = cursor.fetchone()
            #this sql statement retrieves the id that is associated with the chosen student (needed so that the student can be added to the database)
            cursor.execute("SELECT student_id FROM students WHERE student_name = %s;", (s.get(),))
            studentid = cursor.fetchone()
            #this sql statement inserts the student to the class
            cursor.execute("INSERT INTO studentinclass (subject_id, student_id, student_name) VALUES (%s, %s, %s);", (subjectid, studentid, s.get(),))
            connection.commit()
```

This function is used by teachers to add a student to the class they chose in the main interface. The addStudent() function is called if the teacher decides to choose the "+" button. The system would check if the student is already in the class to make sure that no duplicate users are being stored in the same class.



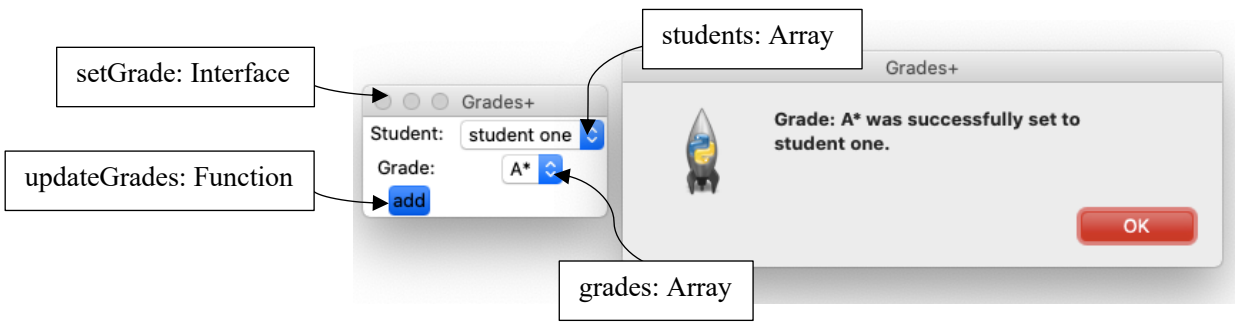
removeStudent Function

```
#this function is called when the teacher wants to remove a student from a class
def removeStudent():
    cursor.execute("SELECT subject_id FROM subjects WHERE subject = %s;", (subject,))
    subjectid = cursor.fetchone()
    #this sql statement retrieves every student in the chosen class
    cursor.execute("SELECT student_name FROM studentinclass WHERE subject_id = %s AND student_name = %s;", (subjectid, s.get(),))
    inClass = False

    if cursor.fetchone() is not None:
        #the student was found
        #this sql statement would remove the chosen student from the class in the database
        cursor.execute("DELETE FROM studentinclass WHERE student_name = %s AND subject_id = %s;", (s.get(), subjectid,))
        connection.commit()
        self.messageBox(s.get() + " is no longer in this class.", "info")
        inClass = True

    #if the student was not in the class, it would throw an error to the teacher.
    if inClass == False: #student is not in the class
        if s.get() != "select a student":
            self.messageBox("This student is not in this class.", "error")

#these buttons are what the teacher clicks in order to add or remove the student.
Button(studentGUI, text = "-", command = removeStudent).grid(column = 0, row = 1)
Button(studentGUI, text = "+", command = addStudent).grid(column = 2, row = 1)
```

The removeStudent() function would be called when the teacher wants to remove a student from the chosen class. The system would first check if the student is in that class in order to remove them. The user would receive an errorbox if the student is not in the class. The removeStudent() function will be called when the user clicks on the “-“ button.

```
# this function is one of the main functionalities for the teacher
# student's grades are assigned through this function
def setGrade(subject):
    setgrades = Toplevel(gui)

    cursor.execute("SELECT subject_id FROM subjects WHERE subject = %s;", (subject,))
    subjectid = cursor.fetchone()
    #this sql statement retrieves the students who are in the class
    cursor.execute("SELECT student_name FROM studentinclass WHERE subject_id = %s;", (subjectid,))

    names = cursor.fetchall()
    students = []
    classEmpty = False
    #i am looping through the retrieved data so that they can be added onto an array and hence be presented on a dropdown menu
    if names is None or not names:
        classEmpty = True
    else:
        #this gets every students name in the database
        for s in names:
            for student in s:
                #the names are added onto the array
                students.append(student)
    #if there is no-one in the class, they would not be able to assign the grades so they must add students to the class
    if classEmpty == True:
        Label(setgrades, text = "This class currently has no students. Please add students using the menu.").grid()
    else: #if there are students in the class
        #this creates a dropdown so the teacher can select a teacher
        s = StringVar(setgrades)
        s.set("select a student")
        #a dropdown for the teacher to select a student.
        Label(setgrades, text = "Student: ").grid(column = 0,row = 1)
        OptionMenu(setgrades, s, *students).grid(column = 1,row = 1)

        grade = StringVar(setgrades)
        grade.set("select a grade")
        grades = ["A*","A","B","C","D","E","U"]
        #the grades U-A* are put as a dropdown so the teacher can assign it to the student
        Label(setgrades, text = "Grade: ").grid(column = 0,row = 3)
        OptionMenu(setgrades, grade, *grades).grid(column = 1 ,row = 3)

        #this functions assigns the grade to the student
        def updateGrades():
            if grade.get() != "select a grade" and s.get() != "select a student":
                cursor.execute("SELECT student_id FROM students WHERE student_name = %s;", (s.get(),))
                studentid = cursor.fetchone()
                date = time.strftime("%Y-%m-%d")
                #the grade is assigned to the student in the database
                cursor.execute("INSERT INTO grades (grade_id, subject_id, student_id, grade, date) VALUES (DEFAULT, %s, %s, %s, %s);", (subjectid, studentid, grade.get(), date))
                connection.commit()
                self.messageBox("Grade: " + grade.get() + " was successfully set to " + s.get() + ".", "info")
        #the "add" button is what the teacher clicks on to set the grade.
        Button(setgrades, text = "add", command = updateGrades).grid()
        connection.commit()
```

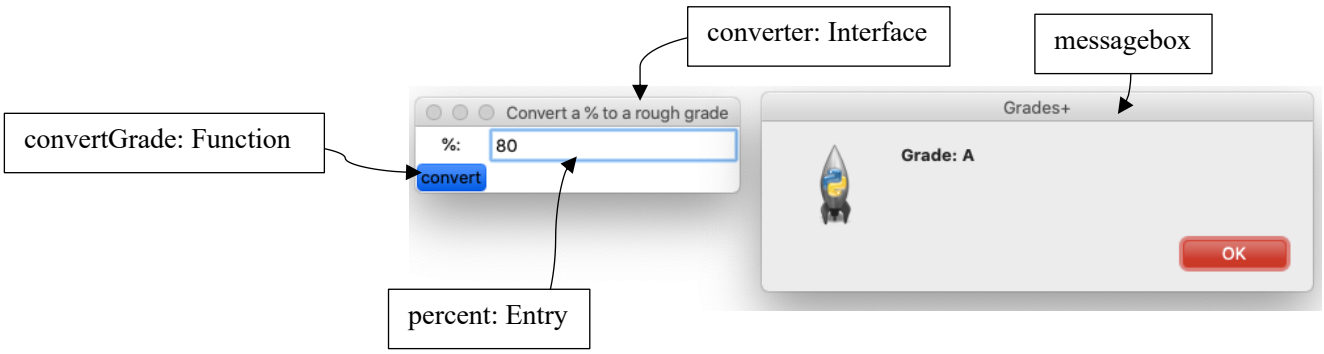
This function is important to my system as it is what the system is based around. When the teacher has finished marking and has graded the student's exam papers, they would use the setGrade() interface to assign the grade. The updateGrades() function is called when the teacher clicks a button when they have selected the student and grade.



covertGrade Function

```
#this function creates the interface for the teacher to convert a % to a grade
def grade():
    global converter
    converter = Toplevel(gui)
    converter.title("Convert a % to a rough grade")
    #this function is used to convert percentages to a grade
    def convertGrade():
        percentage = percent.get()
        #checks if the percentage is greater than 100
        if percentage > 100:
            self.messageBox("Percentage cannot be higher than 100.", "error")
        elif percentage >= 82: #around 82% is an A*
            self.messageBox("Grade: A*", "info")
        elif percentage >= 70:  #around 70% is an A
            self.messageBox("Grade: A", "info")
        elif percentage >= 57:  #around 57% is an B
            self.messageBox("Grade: B", "info")
        elif percentage >= 45:  #around 45% is an C
            self.messageBox("Grade: C", "info")
        elif percentage >= 30:  #around 30% is an D
            self.messageBox("Grade: D", "info")
        elif percentage >= 20:  #around 20% is an E
            self.messageBox("Grade: E", "info")
        elif 0 > percentage:
            self.messageBox("% must be a number between 0 and 100.", "error")
        else:
            self.messageBox("Grade: U", "info")

    percent = IntVar(converter)
    Label(converter, text = "%:").grid(column = 0,row = 1)
    Entry(converter, textvariable = percent).grid(column = 1,row = 1)
    #the "convert" button calls the convertGrade function and converts the %
    Button(converter, text = "convert", command = convertGrade).grid()
```

The convertGrade() function creates an interface from which the user can input a number from 0-100 and then it would output a grade that is based on a custom grade boundary.

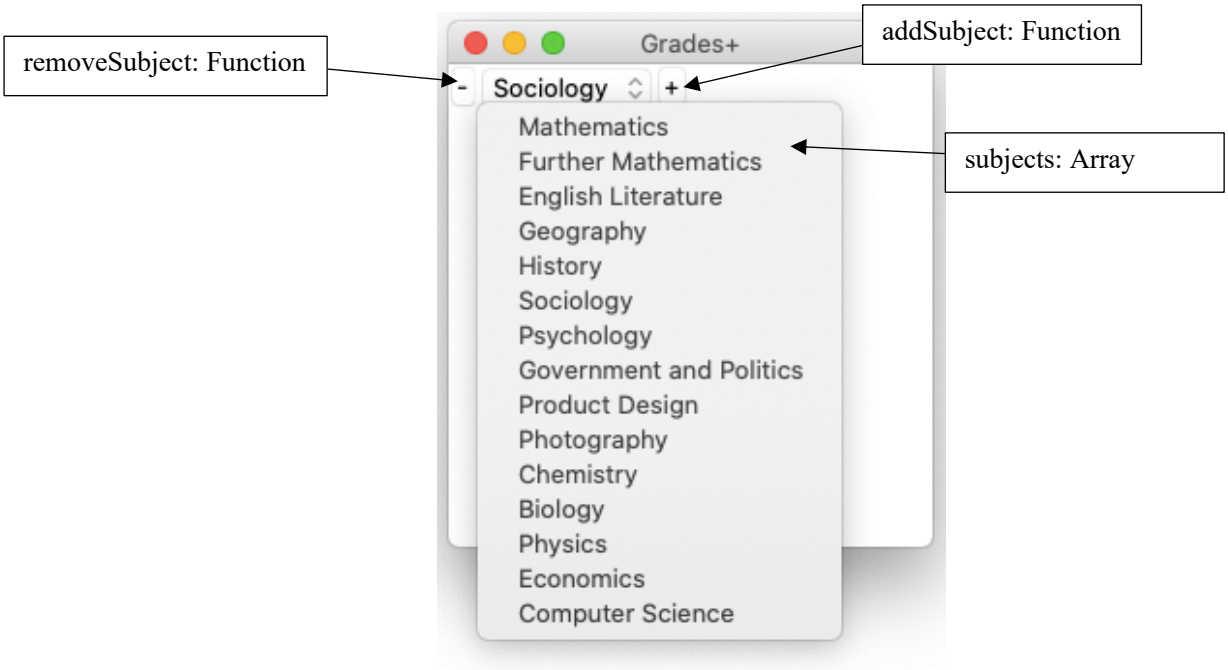


Subjects Function

addSubject Function

```
#this function is used by the teacher to add a class that they teach
def addSubject():
    cursor.execute("SELECT subject_id FROM subjects WHERE subject = %s;", (s.get(),))
    subjectid = cursor.fetchone()
    #this retrieves the teachers that are in the chosen class (s.get()).
    cursor.execute("SELECT teacher_name FROM teacherinclass WHERE subject_id = %s AND teacher_name = %s;", (subjectid, self.getName(),))
    inClass = False
    if cursor.fetchone() is not None:
        #the teacher is already in the class so an error is thrown
        self.messageBox("You are already in this class.", "error")
        inClass = True

    #if the teacher is not in the class, the system would add it to the database
    if inClass == False:
        if s.get() != "select a subject":
            #add the teacher to the class/subject array in both teachers table and subjects table
            self.messageBox("You are now in this class.", "info")
            cursor.execute("SELECT teacher_id FROM teachers WHERE teacher_name = %s;", (self.getName(),))
            teacherid = cursor.fetchone()
            #this sql statements inserts the teacher into the class's table.
            cursor.execute("INSERT INTO teacherinclass (subject_id, teacher_id, teacher_name) VALUES (%s, %s, %s);", (subjectid, teacherid, self.getName(),))
            connection.commit()
            classMenu.add_command(label = s.get(), command = lambda subject = s.get(): subjectGUI(subject))
```

This function would be called if the teacher was to add a class that they teach. The system would first check if the teacher is already in that class in the database and if they aren't they would be added to the TeacherInClass table for that subject in the database.



removeSubject Function

```
#this function would be called if the teacher wanted to leave a class
def removeSubject():
    cursor.execute("SELECT subject_id FROM subjects WHERE subject = %s;", (s.get(),))
    subjectid = cursor.fetchone()

    cursor.execute("SELECT teacher_name FROM teacherinclass WHERE subject_id = %s AND teacher_name = %s;", (subjectid, self.getName(),))

    inClass = False
    if cursor.fetchone() is not None:
        #if the teacher was found, the system  would remove teacher from the class in the database
        cursor.execute("SELECT teacher_id FROM teachers WHERE teacher_name = %s;", (self.getName(),))
        teacherid = cursor.fetchone()

        cursor.execute("SELECT subject_id FROM subjects WHERE subject = %s;", (s.get(),))
        subjectid = cursor.fetchone()
        #the DELETE sql statement gets rid of the entry where the teacher is in the class.
        cursor.execute("DELETE FROM teacherinclass WHERE teacher_id = %s AND subject_id = %s;", (teacherid, subjectid,))
        connection.commit()

        #this removes the subject from the menu
        classMenu.delete(s.get())

        self.messageBox("You are no longer in this class.", "info")
        inClass = True

    #However, if the teacher is not in the class, an error would be called.
    if inClass == False:
        self.messageBox("You are not in this class.", "error")
```

The removeSubject() function will be called if the teacher was to move from teaching a subject to another. The system would check if the teacher is in the class through the database and then would decide what to do.

STUDENT CLASS

CREATESTUDENT FUNCTION

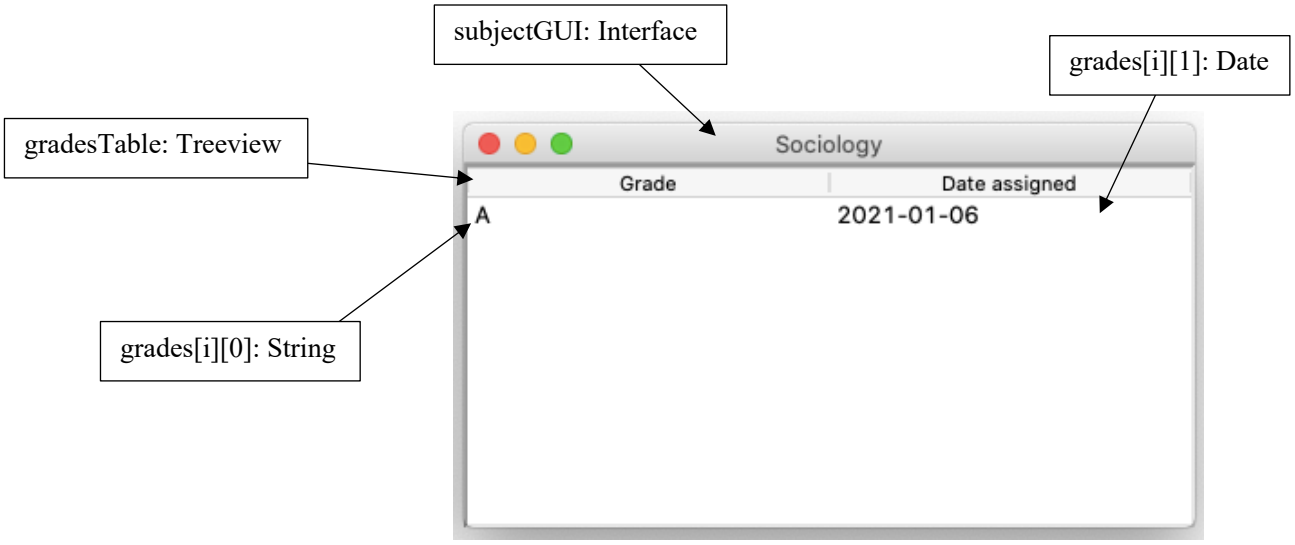
```
#the student class is used to instantiate students as objects. Like the teacher class, this class also inherits the methods of the School class.
class Student(School):
    def createStudent(self, window):
        connection = psycopg2.connect(host = "localhost", user = "postgres", password = "postgres", port = "5432", database = "school")
        cursor = connection.cursor()

        #the following checks if the student is in the validated students textfile
        studentsFile = open("validstudents.txt", "r")
        validStudents = studentsFile.readlines()
        validated = False
        #loops through each line in the textfile
        for student in validStudents:
            #if the line is the same as the students name then it means that they are validated
            if self.getName() == student.strip().lower():
                validated = True
                #as the student is valid, the system would check if they are already in the database
                cursor.execute("SELECT student_name FROM students WHERE student_name = %s;", (self.getName(),))
                if cursor.fetchone() is not None:
                    #student exists in database since value is not null
                    self.messageBox("Already registered.", "error")
                else:
                    #as the student does not exist, the system would check if the password is valid
                    if self.isPasswordValid(self.getPassword()) == True:
                        #if it is, then the student is added to the database
                        hashedPassword = PasswordHasher().hash(self.getPassword())
                        cursor.execute("INSERT INTO students (student_id, student_name, student_password) VALUES (DEFAULT , %s , %s);", (self.getName(), hashedPassword))
                        connection.commit()
                        connection.close()
                        self.messageBox("Success! You have been registered.", "info")
                        window.after(3000, window.destroy)
        #if value of the Boolean validated did not change then that means the student is not valid
        if validated == False:
            self.messageBox("You are not a validated student.", "error")

        connection.close()
```

The system for registering a student is the same as it is for the teacher. The system would first check if the user is valid and then check if they exist in the database. The user would be added to the database if their password is valid and if the user does not already exist in the database, otherwise an errorbox would be thrown at the user.

SUBJECTGUI FUNCTION (GRADES INTERFACE)

```
#when this function is called, it shows the student their grades for the chosen subject
def subjectGUI(subject):
    subjectGUI = Toplevel(gui)
    subjectGUI.title(subject)

    cursor.execute("SELECT subject_id FROM subjects WHERE subject = %s;", (subject,))
    subjectid = cursor.fetchone()

    cursor.execute("SELECT student_id FROM students WHERE student_name = %s;", (self.getName(),))
    studentid = cursor.fetchone()

    #this sql statement retrieves the date and grade that were assigned to the student for the class
    cursor.execute("SELECT grade, date FROM grades WHERE student_id = %s AND subject_id = %s;", (studentid, subjectid,))
    grades = cursor.fetchall()
    #to represent the grades, i am using treeview.
    gradesTable = ttk.Treeview(subjectGUI, columns = ("Grade", "Date assigned"), show = "headings")
    gradesTable.heading("Grade", text = "Grade")
    gradesTable.heading("Date assigned", text = "Date assigned")
    gradesTable.grid()

    for i in range(len(grades)):
        gradesTable.insert("", "end", values = (grades[i][0], grades[i][1]))
```

This function is the core of the student's interface as the student's grades (that were assigned) would be shown on a tree-view table. The student would pick the class from a dropdown that they would want to check their grades from and then a new interface would be created with the results. The subject-class would only exist in the student's dropdown menu if the teacher has added the student to the class, otherwise, the student would not be able to check their assigned results.



MAIN.PY

IMPORTS

```
#these are the imports i would be using to create my system
from tkinter import * #used for gui
from School import *
from argon2 import PasswordHasher #used to hash the password
import psycopg2 #used for the database
import os
```

The following imports would be used to create the new grades system.

ROLE AUTHENTICATE FUNCTION

```
#verify if the code is valid for a teacher/student
def roleAuthenticate(code):
    #codes are used to verify the user; the return values are used to indicate the role the user has.
    if code == "STARK2021":
        return "Student"
    elif code == "TEARK2021":
        return "Teacher"
    #if the code entry is blank
    elif not code:
        return True
    else:
        return "Invalid code"
```

This function is required in order to verify the user's role. During authentication, the user would enter a code. This code is checked through an algorithm if it matches the verified code in order for the user to be validated. The codes would be supplied by the head of year to ensure that only the 6th form students are the ones registering. There would be other precautions to avoid external users from having access to the system.

MAIN FUNCTION

FILES

```
#this creates a file to track the validated students
def createStudentsFile():
    if os.path.exists("validstudents.txt"): #this would check if the file exists
        pass
    else:
        #opens the file/creates it if it does not exist and writes the data into it.
        studentFile = open("validstudents.txt", "w")
        studentFile.write("#enter names of students that are valid\n")   #the default messages are for admin to customise the textfile.
        studentFile.write("#format: firstname lastname (e.g. John Appleseed)")
        studentFile.close()

#this creates a file to track the validated teachers
def createTeachersFile():
    if os.path.exists("validteachers.txt"):
        pass
    else:
        #opens the file/creates it if it does not exist and writes the data into it.
        teacherFile = open("validteachers.txt", "w")
        teacherFile.write("#enter names of teachers that are valid\n")
        teacherFile.write("#format: firstname lastname (e.g. John Appleseed)")
        teacherFile.close()

def createSubjectsFile():
    if os.path.exists("subjects.txt"):
        pass
    else:
        #this creates the file to input subjects that the school teaches. (by default there are the subjects that the 6th form teaches)
        subjectsFile = open("subjects.txt", "w")
        subjectsFile.write("#enter the subjects below:")
        subjectsFile.close()

#this function checks for any new subjects in the subjects files and updates it
def updateSubjectClass():
    #creating a connection to the database
    connection = psycopg2.connect(host = "localhost", user = "postgres", password = "postgres", port = "5432", database = "school")
    cursor = connection.cursor()

    #opens the subject textfile and gets every line in it.
    subjectsFile = open("subjects.txt", "r")
    subjects = subjectsFile.readlines()

    for subject in subjects:  #loops through all lines in subjects
        if subject.strip() == "#enter the subjects below:":
            pass #disregards the line
        else:
            #this sql statement retrieves the subject from subjects. I did this so that the system can check if the subject already exists or not.
            cursor.execute("SELECT subject FROM subjects WHERE subject = %s;", (subject.strip(),))

            if cursor.fetchone() is None:
                #this would mean that thesubject does not exist and hence, adds the subject to the database
                #the sql statement creates a new row by inserting the subject into the subjects table.
                cursor.execute("INSERT INTO subjects (subject) VALUES (%s);", (subject.strip(),))
                connection.commit()

    subjectsFile.close()
    connection.close()

createStudentsFile()
createTeachersFile()
createSubjectsFile()
```

The createStudentsFile(), createTeachersFile() and createSubjectsFile() functions are mainly required for the first time the program is running. This is because the file is created during the first run as the files tend to not exist then. These files are used by my end-user to input data that would be required for the authenticated system such as the student's name that attend the school. For the subjects file, if my end-client wanted to add a new subject, the system would create a new table for it as shown in the updateSubjectClass() function.

DATABASE

```
databaseStatus = False
#I am using exception handling to check whether the database is running or not.
try:
    #this would create a connection to the database
    psycopg2.connect(host = "localhost", user = "postgres", password = "postgres", port = "5432", database = "school")
    databaseStatus = True
except:
    #this would mean that an error was called/the database is offline
    print("Database isn't running right now.")
```

Here, I am using exception handling to connect to the database and check if it is online. When the database is offline, it would call an error so, by catching the error, it would just tell the user that the database is not running. The program would continue running if the database is online.

```
#if the database is running, then i would create the data-tables and the menu interface for the user
if databaseStatus == True:
    global auth
    connection = psycopg2.connect(host = "localhost", user = "postgres", password = "postgres", port = "5432", database = "school")
    cursor = connection.cursor()

    #this creates the students table if it doesnt exist
    cursor.execute("""CREATE TABLE IF NOT EXISTS students (
                student_id SERIAL PRIMARY KEY,
                student_name VARCHAR(150),
                student_password VARCHAR(1000)
                );""")

    #this creates the teachers table if it doesn't exist
    cursor.execute("""CREATE TABLE IF NOT EXISTS teachers (
                teacher_id SERIAL PRIMARY KEY,
                teacher_name VARCHAR(150),
                teacher_password VARCHAR(1000)
                );""")

    #this creates the table for the subjects if it doesn't exist
    cursor.execute("""CREATE TABLE IF NOT EXISTS subjects (
                subject_id SERIAL PRIMARY KEY,
                subject VARCHAR(50)
                );""")

    #this creates the table for TeachersInClass if it doesn't exist
    cursor.execute("""CREATE TABLE IF NOT EXISTS teacherinclass (
                subject_id INTEGER,
                teacher_id INTEGER,
                FOREIGN KEY(subject_id) REFERENCES subjects(subject_id),
                FOREIGN KEY(teacher_id) REFERENCES teachers(teacher_id),
                teacher_name VARCHAR(150)
                );""")
```

```
#this creates the table for StudentInClass if it doesn't exist
cursor.execute("""CREATE TABLE IF NOT EXISTS studentinclass (
            subject_id INTEGER,
            student_id INTEGER,
            FOREIGN KEY(subject_id) REFERENCES subjects(subject_id),
            FOREIGN KEY(student_id) REFERENCES students(student_id),
            student_name VARCHAR(150)
            );""")

#this creates the table for the grades if it doesn't exist
cursor.execute("""CREATE TABLE IF NOT EXISTS grades (
                grade_id SERIAL PRIMARY KEY,
                subject_id INTEGER,
                student_id INTEGER,
                FOREIGN KEY(student_id) REFERENCES students(student_id),
                FOREIGN KEY(subject_id) REFERENCES subjects(subject_id),
                grade VARCHAR(2),
                date DATE
            );""")

connection.commit()
connection.close()
updateSubjectClass()
```

If the database is running then the program would connect to the database in order to set it up. The following queries check if the tables already exist, and if they don't they would create the tables individually. The tables are in accordance with the normalisation I did in my design in order to ensure that the data is being stored more efficiently.

AUTHENTICATION SYSTEM

REGISTER

```
#this is the function for the registration interface
def register():
    global register
    #this create a new interface for the user to register
    register = Toplevel(auth)
    register.title("Registration")

    #create stringvar's for username and password so that the stored data can be collected
    #the strinvar data will be stored in the database for the student/teacher.
    global userFirstName
    global userLastName
    global password
    global code
    global role

    userFirstName = StringVar(register)
    userLastName = StringVar(register)
    password = StringVar(register)
    code = StringVar(register)
    role = StringVar(register)
    #this sets the default option for the roles dropdown.
    role.set("pick a role")

    Label(register, text = "First name:").grid(column = 0, row = 1)
    #this create an entry so that data can be inputted for the first name.
    Entry(register, textvariable = userFirstName).grid(column=1, row=1)

    Label(register, text = "Last name:").grid(column = 0, row = 2)
    #this create an entry so that data can be inputted for the last name.
    Entry(register, textvariable = userLastName).grid(column = 1, row = 2)

    Label(register, text = "Password:").grid(column = 0, row = 3)
    #this create a label and an entry so that data can be inputted for the password.
    #show = '*' will change the characters into an asterix to hide the password.
    Entry(register, textvariable = password, show = "*").grid(column = 1, row = 3)

    Label(register, text = "Code:").grid(column = 0, row = 4)
    #this create an entry so that data can be inputted for the code.
    Entry(register, textvariable = code).grid(column = 1, row = 4)

    #a dropdown for the user to choose if they are a student or a teacher.
    OptionMenu(register, role, "Student", "Teacher").grid(sticky = E)
    #this create empty label for spacing purposes.
    Label(register,text = "").grid()
    #this create button to register the user. the function linked to the button will be called when the button is clicked.
    Button(register, text = "Register", command = registerUser).grid()
```

The register function creates a window so that the user can input their information and create an account if they don't have one. The user would have to input their name, password and the code which would be given to them. The code would make sure that the user is registering for the correct role.

```
#this function is used to register a new user into the database
def registerUser():
    #check if the user is a valid student/teacher
    authenticate = roleAuthenticate(code.get())
    userFullName = userFirstName.get().strip() + " " + userLastName.get().strip()

    #check if the user's role and code match
    if(authenticate ==  "Student" and role.get() == "Student"):
        #if it smatches, it creates a student object
        studentClass = Student(userFullName, password.get())
        studentClass.createStudent(register)
    elif(authenticate == "Teacher" and role.get() == "Teacher"):
        #this would instantiate a teacher object
        teacherClass = Teacher(userFullName, password.get())
        teacherClass.createTeacher(register)

    elif(authenticate == "Invalid code"):
        auth.destroy() #this deletes the window
    elif(role.get() == "pick a role"):
        #creates an error box because the user has not picked a role
        messagebox.showerror("Grades+", "Error: Pick a role")
    elif(authenticate == True):
        #error box is created as the entry value for the code is empty
        messagebox.showerror("Grades+", "Error: Enter a code.")
    else:
        auth.destroy()
```

When the user fills in their details in the register window, after pressing the button, it would run the registerUser() function which is dedicated to creating the object for the user and storing their data into the database. The registerUser() function first checks the user's role and then it uses the Student or Teacher class to instantiate an object with the data from the register window. The createStudent()/createTeacher() function is the function in the class which connects to the database and inserts the user's data into it.

LOGIN

```
def login():
    global login
    global fullname
    global password
    global role

    #this create a new interface for the user to login
    login = Toplevel(auth)
    login.title("Login")

    #the following stringvar's are used to collect the data being inputted.
    fullname = StringVar(login)
    password = StringVar(login)
    role = StringVar(login)
    role.set("pick a role")

    Label(login, text = "Full name:").grid(column = 0, row = 1)
    #this create an entry so that data can be inputted for the full-name.
    Entry(login, textvariable = fullname).grid(column = 1, row = 1)

    Label(login, text = "Password:").grid(column = 0, row = 2)

    #this create an entry so that data can be inputted for the password.
    Entry(login, textvariable = password, show = "*").grid(column = 1, row = 2)

    #this would give the user a dropdown from which they can pick their role.
    OptionMenu(login, role, "Student", "Teacher").grid(sticky = E)

    #this create button to register the user and if validated the main interface would be created.
    Button(login, text = "Login", command = loginUser).grid()
```

The login function would be called if the user chooses to login. They would have to input their full name and password. In order to ensure they are designated the correct interface; they would have the option to select their role. After inputting the data, they would press the login button which calls the loginUser() function.

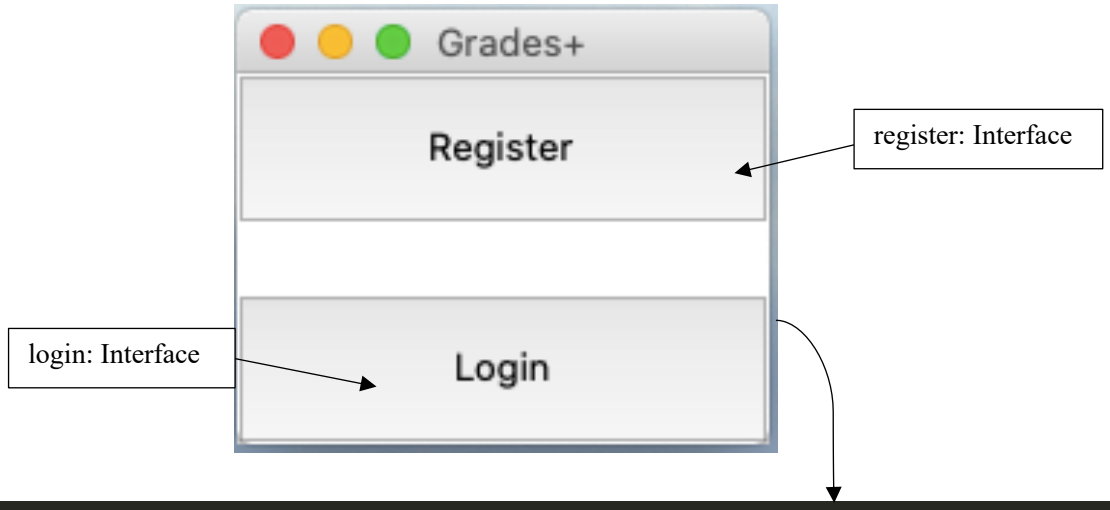
```
def loginUser():
    #this creates a connection to the database
    connection = psycopg2.connect(host = "localhost", user = "postgres", password = "postgres", port = "5432", database = "school")
    cursor = connection.cursor()

    #checks the global variable role as to identify the user's interface.
    if(role.get() == "Student"):
        userFullName =  fullname.get().lower().strip()
        #this sql statement retrieves the attribute "student name" if it exists
        cursor.execute("SELECT student_name FROM students WHERE student_name = %s;", (userFullName,))

        #if the student exists in the database
        if cursor.fetchone() is not None:
            #this sql statement retrieves the password which is associated with the student
            cursor.execute("SELECT student_password FROM students WHERE student_name = %s;", (userFullName,))
            storedPassword = cursor.fetchone()
            authorised = False

            #as an error is called when the password is incorrect, i would catch it using exception handling.
            try:
                #this verifies if the password input is the same as the hashed password in thed database.
                PasswordHasher().verify(storedPassword[0], password.get())
                authorised = True
            except:
                #if theres an error, this would be because the password is incorrect hence i would make an errorbox to show this.
                messagebox.showerror("Grades+", "Error: Incorrect password.")

            if authorised == True:
                login.destroy()
                #this creates student object
                studentClass = Student(userFullName, password.get())
                #this would create the interface for the user
                studentClass.sendGUI()

            connection.close()
        else: #if the student doesn't exist in the database, an errorbox would be called.
            messagebox.showerror("Grades+", "Error: This account does not exist or has not been registered yet.")
    elif(role.get() == "Teacher"):
        userFullName =  fullname.get().lower().strip()
        #this sql statement would check if teacher exists in the database
        cursor.execute("SELECT teacher_name FROM teachers WHERE teacher_name = %s;", (userFullName,))
        if cursor.fetchone() is not None: #if the teacher exists
            cursor.execute("SELECT teacher_password FROM teachers WHERE teacher_name = %s;", (userFullName,))
            storedPassword = cursor.fetchone()

            authorised = False
            try:
                #checks if the input of password is the same as the one in the database.
                PasswordHasher().verify(storedPassword[0], password.get())
                authorised = True
            except:
                messagebox.showerror("Grades+", "Error: Incorrect password.")

            #if the user's password was validated, it would instantiate the teacher object and create the interface.
            if authorised == True:
                login.destroy()
                teacherClass = Teacher(userFullName, password.get())
                teacherClass.sendGUI()

            connection.close()
        else: #if the teacher doesn't exist, an errorbox will be called
            messagebox.showerror("Grades+", "Error: This account does not exist or has not been registered yet.")
```

The loginUser() function allows the system to check if the user exists in the database, and if they do, the interface would be created. This is done through SQL queries and using the data that was input into the login system. Then, the system would retrieve the password associated with the user and checks if the input for the password is the same, and if it is, then the object for the user would be instantiated and an interface would be created.

SCREENSHOTS OF THE SYSTEM

AUTHENTICATION SYSTEM

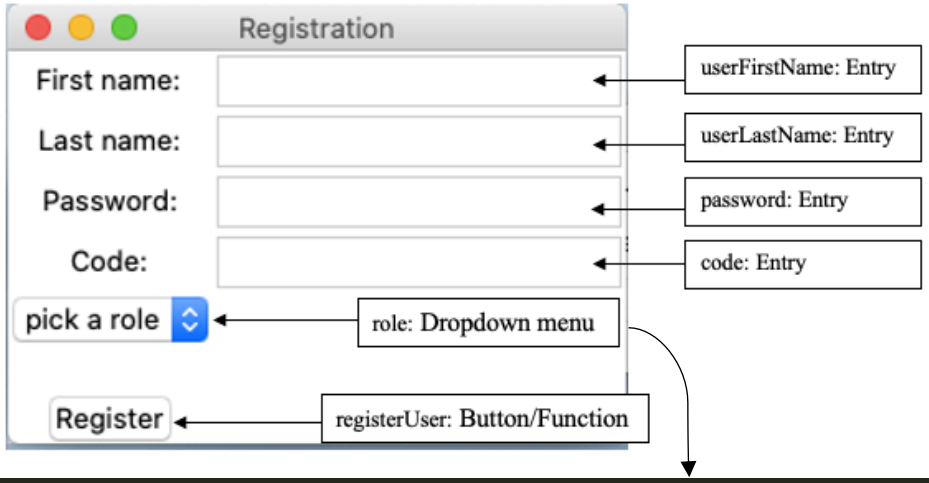
The following screenshots show the interface that first appears when the program is run. The authentication system exists to authorise users through the use of a database. The user would have to decide if they would want to log in (if already registered) or to register an account. This would create a new window that would allow the user to input their information.




```
#this would create the interface for the authentication system
auth = Tk()
auth.title("Grades+")
#this create a button to give the user the option to register. When clicked, the register interface would be created.
Button(text = "Register", height = "3", width = "20", command = register).pack()
Label(text="").pack()
#this create a button to give the user the option to login. When clicked, the login interface would be created.
Button(text = "Login", height = "3", width = "20", command = login).pack()

#this would run the tkinter window
auth.mainloop()
```

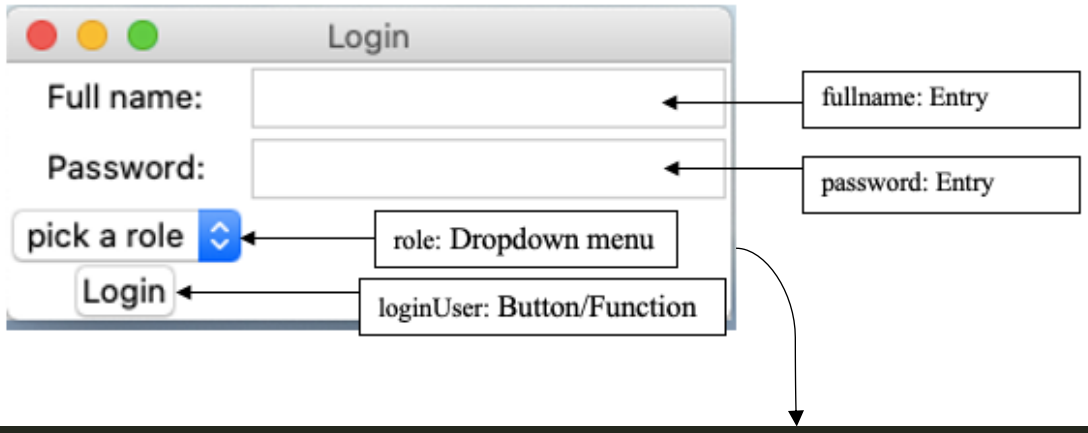
By using Tkinter, I have created two buttons that are connected to their corresponding functions such as the register button will run the register function when the user clicks on it. I have made the authentication simple as it would be easier for new teachers/students to use and those who do not have much experience using a computer.

When the user clicks on the register button, a new window would appear asking for the user's first name, last name, password, code and role that identifies them. These are entries that would instantiate an object for the user and check if they are existing in the database. This would allow the database to ensure that there are no duplicate users. The role dropdown menu allows the user to choose their role which would be either a Teacher or Student. The system would use this to create the interface which is suited to their role after they log in.




```
#this is the function for the registration interface
def register():
    global register
    #this create a new interface for the user to register
    register = Toplevel(auth)
    register.title("Registration")

    #create stringvar's for username and password so that the stored data can be collected
    #the strinvar data will be stored in the database for the student/teacher.
    global userFirstName
    global userLastName
    global password
    global code
    global role

    userFirstName = StringVar(register)
    userLastName = StringVar(register)
    password = StringVar(register)
    code = StringVar(register)
    role = StringVar(register)
    #this sets the default option for the roles dropdown.
    role.set("pick a role")

    Label(register, text = "First name:").grid(column = 0, row = 1)
    #this create an entry so that data can be inputted for the first name.
    Entry(register, textvariable = userFirstName).grid(column=1, row=1)

    Label(register, text = "Last name:").grid(column = 0, row = 2)
    #this create an entry so that data can be inputted for the last name.
    Entry(register, textvariable = userLastName).grid(column = 1, row = 2)

    Label(register, text = "Password:").grid(column = 0, row = 3)
    #this create a label and an entry so that data can be inputted for the password.
    #show = '*' will change the characters into an asterix to hide the password.
    Entry(register, textvariable = password, show = "*").grid(column = 1, row = 3)

    Label(register, text = "Code:").grid(column = 0, row = 4)
    #this create an entry so that data can be inputted for the code.
    Entry(register, textvariable = code).grid(column = 1, row = 4)

    #a dropdown for the user to choose if they are a student or a teacher.
    OptionMenu(register, role, "Student", "Teacher").grid(sticky = E)
    #this create empty label for spacing purposes.
    Label(register,text = "").grid()
    #this create button to register the user. the function linked to the button will be called when the button is clicked.
    Button(register, text = "Register", command = registerUser).grid()
```

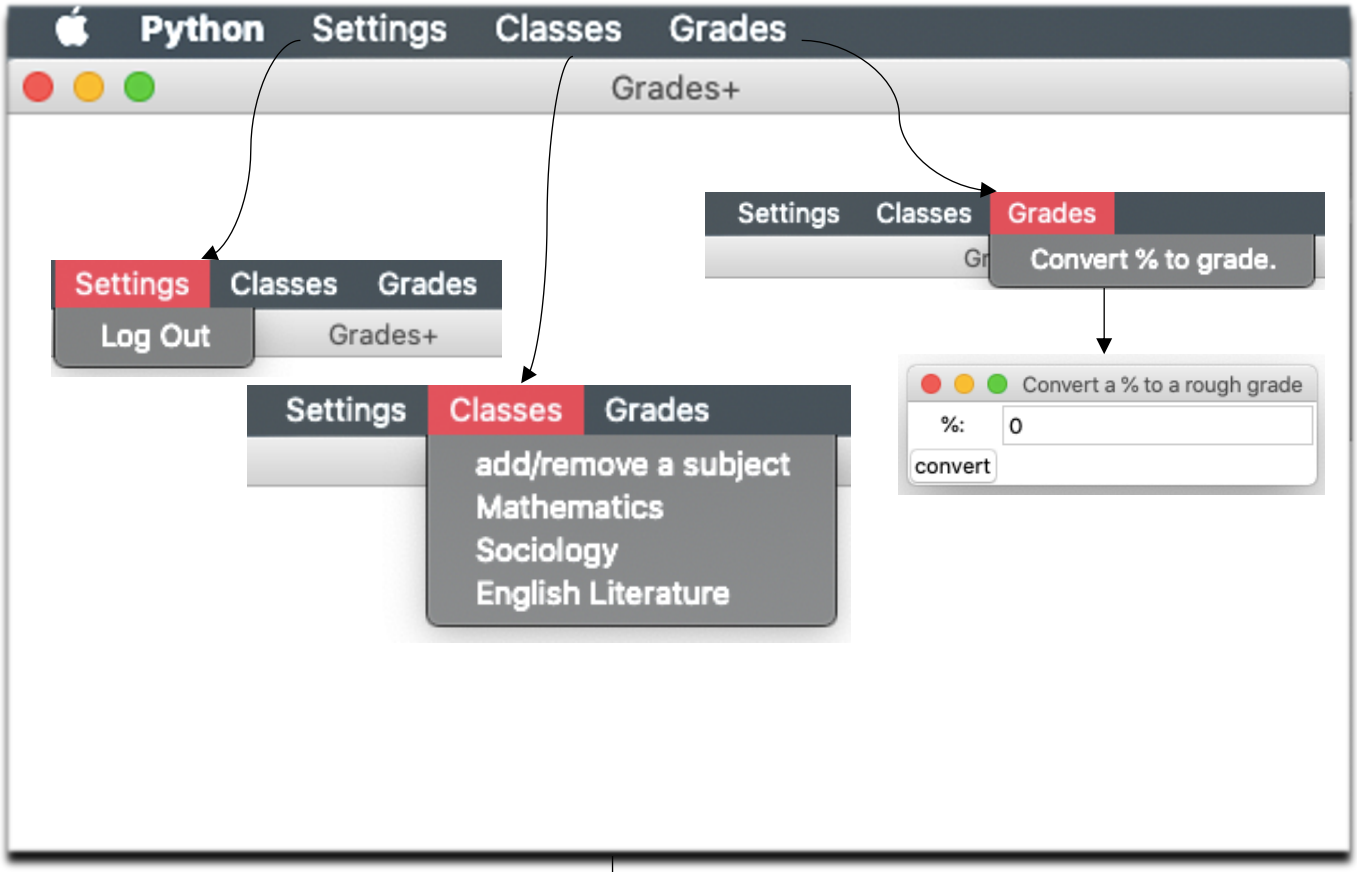
The login interface would appear if the user would want to login into an account that exists. This would require them to enter their full name, password and choose their role. The database would check if there are matching credentials after the login button is clicked. If the user's credentials are correct then the interface would be created for the user, allowing them to use the functionalities.




```
def login():
    global login
    global fullname
    global password
    global role

    #this create a new interface for the user to login
    login = Toplevel(auth)
    login.title("Login")

    #the following stringvar's are used to collect the data being inputted.
    fullname = StringVar(login)
    password = StringVar(login)
    role = StringVar(login)
    role.set("pick a role")

    Label(login, text = "Full name:").grid(column = 0, row = 1)
    #this create an entry so that data can be inputted for the full-name.
    Entry(login, textvariable = fullname).grid(column = 1, row = 1)

    Label(login, text = "Password:").grid(column = 0, row = 2)

    #this create an entry so that data can be inputted for the password.
    Entry(login, textvariable = password, show = "*").grid(column = 1, row = 2)

    #this would give the user a dropdown from which they can pick their role.
    OptionMenu(login, role, "Student", "Teacher").grid(sticky = E)

    #this create button to register the user and if validated the main interface would be created.
    Button(login, text = "Login", command = loginUser).grid()
```

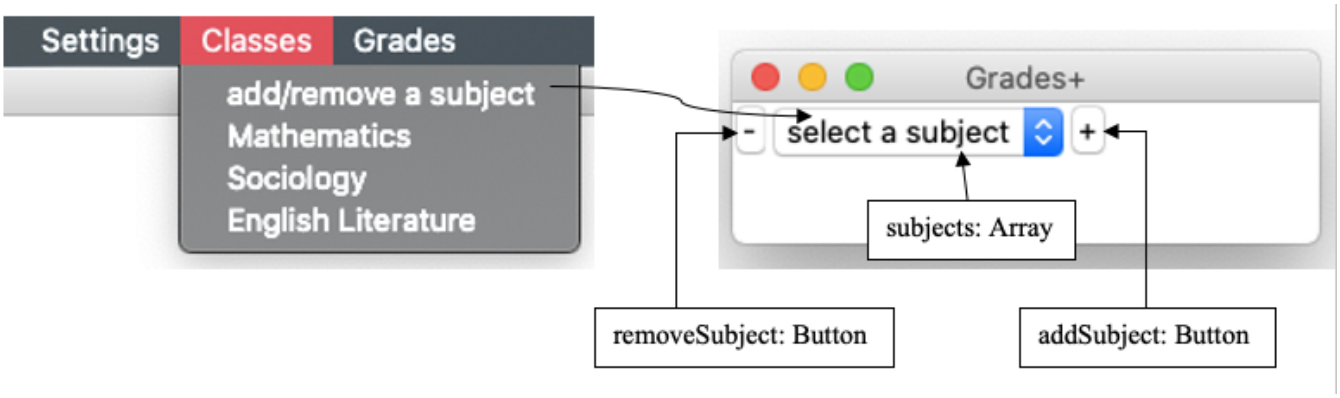
TEACHER INTERFACE

The teacher interface is designed to allow the user to select an option from a menu dropdown and to use the functionalities to add students to a class, convert a % to a grade or to set a grade to a student. As shown in the image below, after a user login as a teacher, there would be three menu options: Settings, Classes and Grades. Settings have a logout function if the user decides to stop using the program. The classes menu is designed to allow teachers to add the classes they teach and go into the class and set the students their corresponding grades. The grades menu has a function that allows the teacher to convert a percentage to a grade.

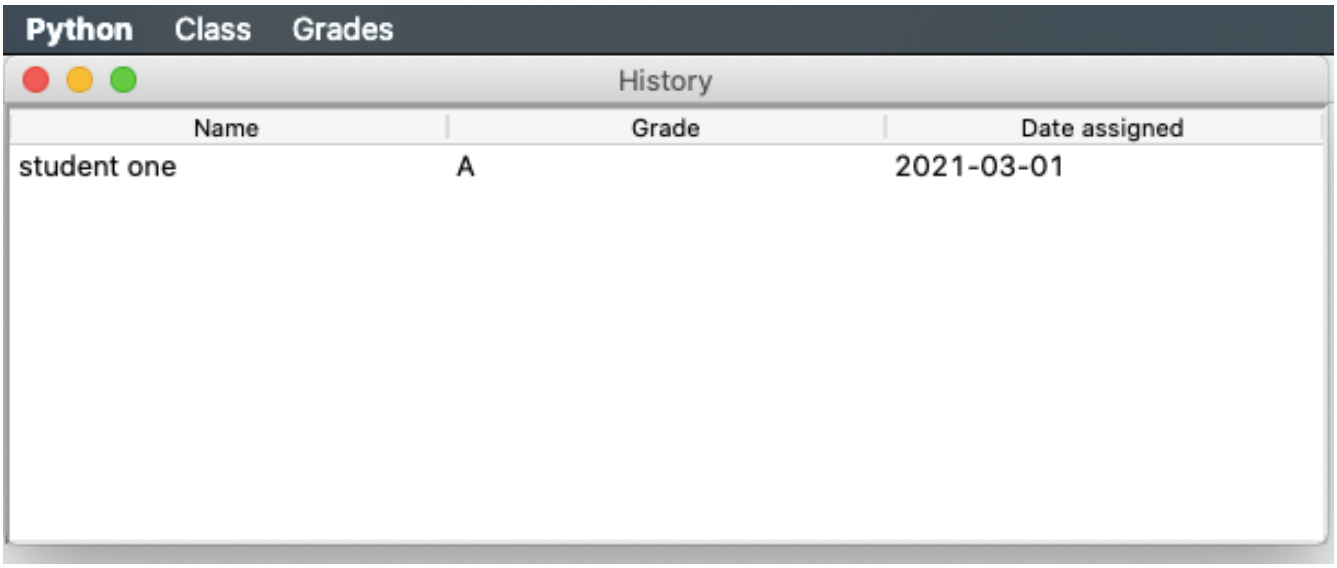



```
#this creates the window for the interface
gui = Tk()
gui.title("Grades+")
gui.geometry("1500x800")
menubar = Menu(gui)
```

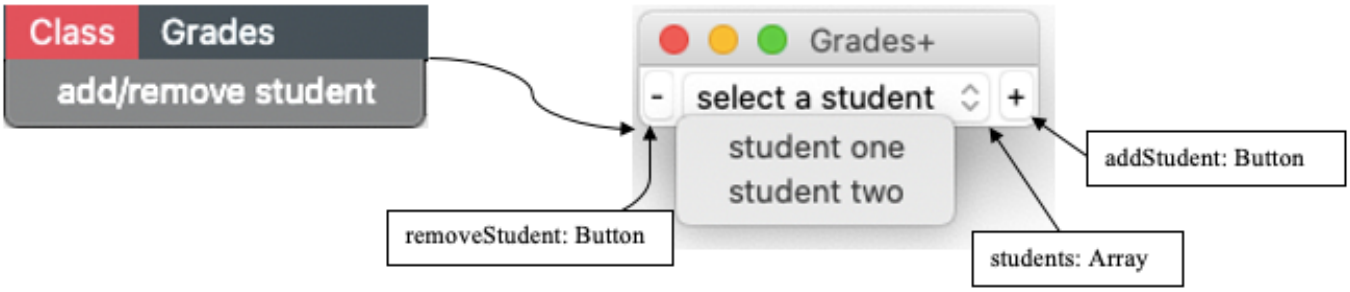
The classes menu allows the teacher to add/remove a subject that they teach. The following screenshot shows that, when you click on the "add/remove a subject" option, a window would appear with a dropdown that has 2 buttons and a dropdown from which the teacher can select the subject. The "-" button calls the removeSubject() function which removes the teacher from the subject. The "+" button calls the addSubject() function which adds the teacher to the subject.



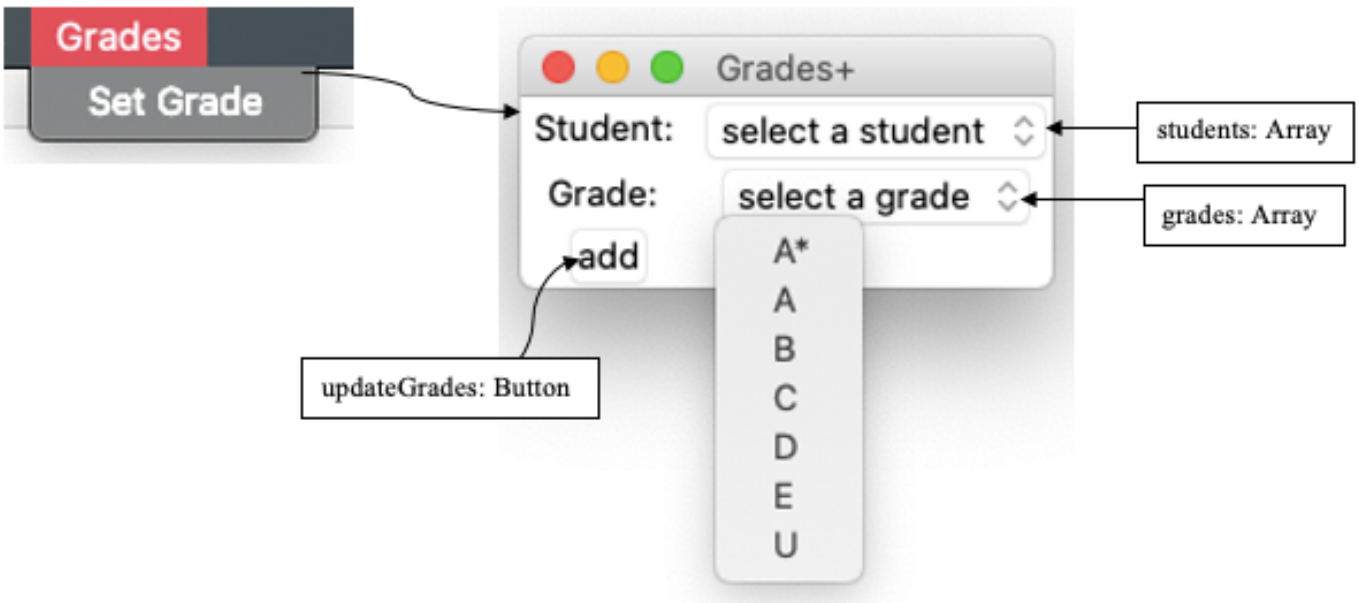
After adding the subject, this would be shown in the menu dropdown. The teacher would have to then pick a subject from the menu dropdown in which they would assign the students their grades. The following screenshot shows the Treeview that shows the students in the class and the grades that they were set. This allows other teachers that teach the subject acknowledge what was set as the grade. This also allows teachers to see who had performed the best and who had performed the worst.



The following screenshot shows how the “add/remove student” works. A window would appear with a dropdown and two buttons. The dropdown would have the students which are registered into the system and the teacher can add them if they are not in the class. The “+” button runs the addStudent() which would check if the user is in the class and if they aren’t they would be added. The “-“ button would run the removeStudent() who would remove the student from the class if they exist in it. The “add/remove student” menu option is important as if the students were not added to the class, the teacher would be unable to set them a grade.

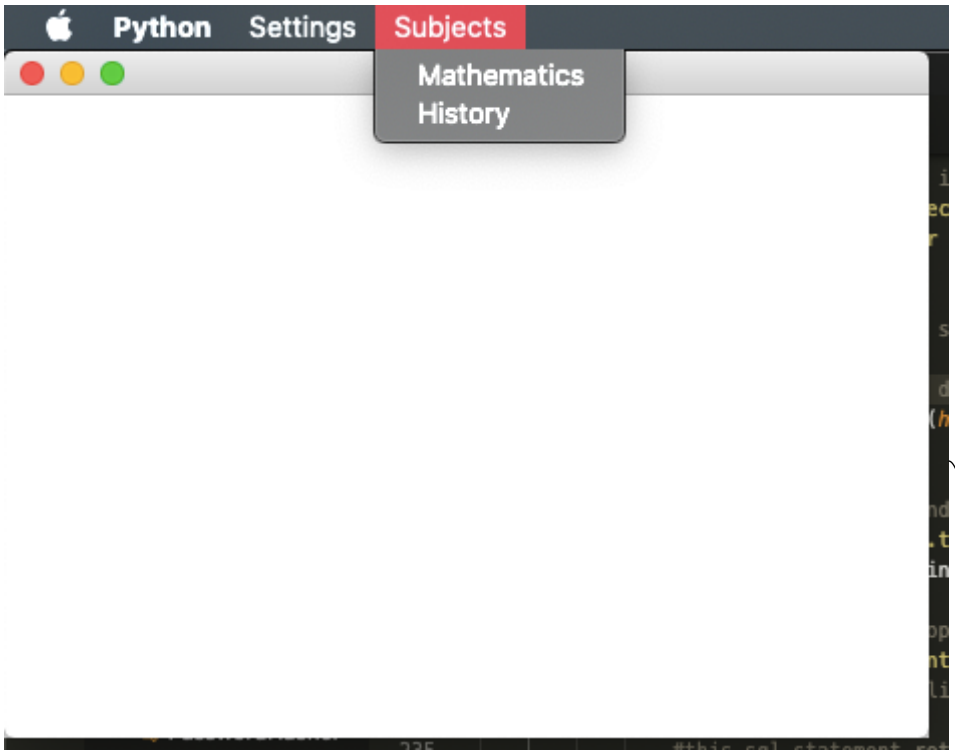


The other function that the teacher can use it the “set grade” menu option. Here the teacher would select the student which is in the class and the grade they achieved on the exam. The “add” button would call the updateGrades() function which would store the data into the database so that the student can access it on their interface.



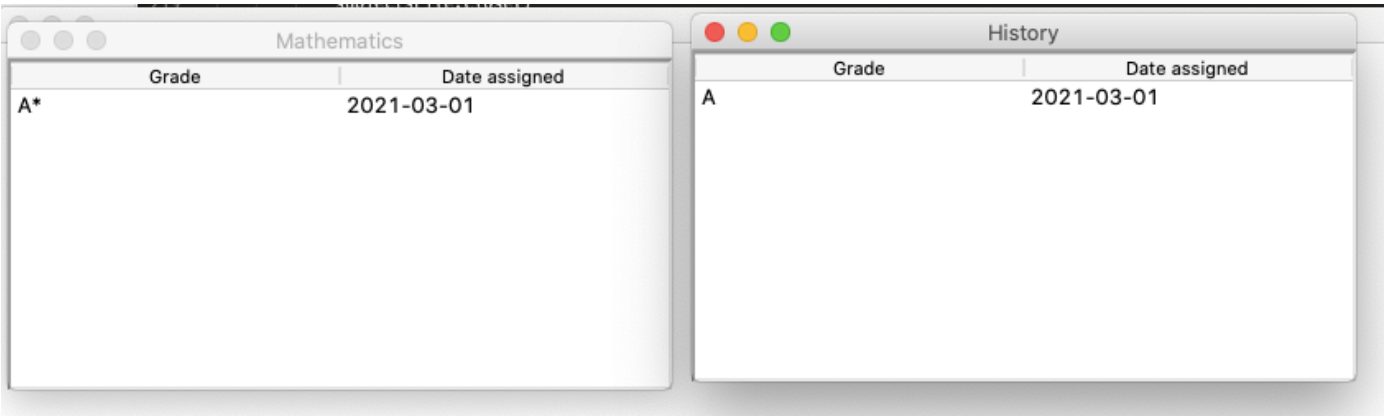
STUDENT INTERFACE

The student interface would allow the students to view the grades that they had been assigned. The following screenshot shows the student interface. In the interface, the subject's menu shows the student the student the classes they were added to. This is done by the teacher using the "add/remove student" function. Once the student selects on the subject, a window would appear showing the grade and the date it was assigned.




```
#when this function is called, it shows the student their grades for the chosen subject
def subjectGUI(subject):
    subjectGUI = Toplevel(gui)
    subjectGUI.title(subject)

    cursor.execute("SELECT subject_id FROM subjects WHERE subject = %s;", (subject,))
    subjectid = cursor.fetchone()

    cursor.execute("SELECT student_id FROM students WHERE student_name = %s;", (self.getName(),))
    studentid = cursor.fetchone()

    #this sql statement retrieves the date and grade that were assigned to the student for the class
    cursor.execute("SELECT grade, date FROM grades WHERE student_id = %s AND subject_id = %s;", (studentid, subjectid,))
    grades = cursor.fetchall()
    #to represent the grades, i am using treeview.
    gradesTable = ttk.Treeview(subjectGUI, columns = ("Grade", "Date assigned"), show = "headings")
    gradesTable.heading("Grade", text = "Grade")
    gradesTable.heading("Date assigned", text = "Date assigned")
    gradesTable.grid()

    for i in range(len(grades)):
        gradesTable.insert("", "end", values = (grades[i][0], grades[i][1]))
```

This is an example of what the window would look like to a student. The Treeview would show all the grades that are relevant to the subject. Here, the student was assigned a A* in Mathematics and an A in History.



In the database, the data can also be seen in the same way where student_id = 1 is the user that is in the mathematics and history class.

grade_id	subject_id	student_id	grade	date
1	1	1	A*	2021-03-01
2	5	1	A	2021-03-01

```
#these are the imports i would be using to create my system
from tkinter import * #used for gui
from School import *
from argon2 import PasswordHasher #used to hash the password
import psycopg2 #used for the database
import os

#verify if the code is valid for a teacher/student
def roleAuthenticate(code):
    #codes are used to verify the user; the return values are used to indicate the role the user has.
    if code == "STARK2021":
        return "Student"
    elif code == "TEARK2021":
        return "Teacher"
    #if the code entry is blank
    elif not code:
        return True
    else:
        return "Invalid code"

#this function is used to register a new user into the database
def registerUser():
    #check if the user is a valid student/teacher
    authenticate = roleAuthenticate(code.get())
    userFullName = userFirstName.get().strip() + " " + userLastName.get().strip()

    #check if the user's role and code match
    if(authenticate ==  "Student" and role.get() == "Student"):
        #if it smatches, it creates a student object
        studentClass = Student(userFullName, password.get())
        studentClass.createStudent(register)
    elif(authenticate == "Teacher" and role.get() == "Teacher"):
        #this would instantiate a teacher object
        teacherClass = Teacher(userFullName, password.get())
        teacherClass.createTeacher(register)

    elif(authenticate == "Invalid code"):
        messagebox.showerror("Grades+", "Error: Enter a valid code.")
    elif(role.get() == "pick a role"):
        #creates an error box because the user has not picked a role
        messagebox.showerror("Grades+", "Error: Pick a role")
    elif(authenticate == True):
        #error box is created as the entry value for the code is empty
        messagebox.showerror("Grades+", "Error: Enter a code.")
    else:
        messagebox.showerror("Grades+", "Error: Enter the correct code.")

def loginUser():
    #this creates a connection to the database
    connection = psycopg2.connect(host = "localhost", user = "postgres", password = "postgres", port = "5432", database =
"school")
    cursor = connection.cursor()

    #checks the global variable role as to identify the user's interface.
    if(role.get() == "Student"):
        userFullName =  fullname.get().lower().strip()
        #this sql statement retrieves the attribute "student name" if it exists
        cursor.execute("SELECT student_name FROM students WHERE student_name = %s;", (userFullName,))

        #if the student exists in the database
        if cursor.fetchone() is not None:
            #this sql statement retrieves the password which is associated with the student
            cursor.execute("SELECT student_password FROM students WHERE student_name = %s;", (userFullName,))
            storedPassword = cursor.fetchone()
            authorised = False
```

```
        #as an error is called when the password is incorrect, i would catch it using exception handling.
        try:
            #this verifies if the password input is the same as the hashed password in thed database.
            PasswordHasher().verify(storedPassword[0], password.get())
            authorised = True
        except:
            #if theres an error, this would be because the password is incorrect hence i would make an errorbox to show this.
            messagebox.showerror("Grades+", "Error: Incorrect password.")

        if authorised == True:
            login.destroy()
            #this creates student object
            studentClass = Student(userFullName, password.get())
            #this would create the interface for the user
            studentClass.sendGUI()

        connection.close()
    else: #if the student doesn't exist in the database, an errorbox would be called.
        messagebox.showerror("Grades+", "Error: This account does not exist or has not been registered yet.")
elif(role.get() == "Teacher"):
    userFullName =  fullname.get().lower().strip()
    #this sql statement would check if teacher exists in the database
    cursor.execute("SELECT teacher_name FROM teachers WHERE teacher_name = %s;", (userFullName,))
    if cursor.fetchone() is not None: #if the teacher exists
        cursor.execute("SELECT teacher_password FROM teachers WHERE teacher_name = %s;", (userFullName,))
        storedPassword = cursor.fetchone()

        authorised = False
        try:
            #checks if the input of password is the same as the one in the database.
            PasswordHasher().verify(storedPassword[0], password.get())
            authorised = True
        except:
            messagebox.showerror("Grades+", "Error: Incorrect password.")

        #if the user's password was validated, it would instantiate the teacher object and create the interface.
        if authorised == True:
            login.destroy()
            teacherClass = Teacher(userFullName, password.get())
            teacherClass.sendGUI()

        connection.close()
    else: #if the teacher doesn't exist, an errorbox will be called
        messagebox.showerror("Grades+", "Error: This account does not exist or has not been registered yet.")

#this is the function for the registration interface
def register():
    global register
    #this create a new interface for the user to register
    register = Toplevel(auth)
    register.title("Registration")

    #create stringvar's for username and password so that the stored data can be collected
    #the strinvar data will be stored in the database for the student/teacher.
    global userFirstName
    global userLastName
    global password
    global code
    global role

    userFirstName = StringVar(register)
    userLastName = StringVar(register)
    password = StringVar(register)
    code = StringVar(register)
    role = StringVar(register)
```

```
    #this sets the default option for the roles dropdown.
    role.set("pick a role")

    Label(register, text = "First name:").grid(column = 0, row = 1)
    #this create an entry so that data can be inputted for the first name.
    Entry(register, textvariable = userFirstName).grid(column=1, row=1)

    Label(register, text = "Last name:").grid(column = 0, row = 2)
    #this create an entry so that data can be inputted for the last name.
    Entry(register, textvariable = userLastName).grid(column = 1, row = 2)

    Label(register, text = "Password:").grid(column = 0, row = 3)
    #this create a label and an entry so that data can be inputted for the password.
    #show = '*' will change the characters into an asterix to hide the password.
    Entry(register, textvariable = password, show = "*").grid(column = 1, row = 3)

    Label(register, text = "Code:").grid(column = 0, row = 4)
    #this create an entry so that data can be inputted for the code.
    Entry(register, textvariable = code).grid(column = 1, row = 4)

    #a dropdown for the user to choose if they are a student or a teacher.
    OptionMenu(register, role, "Student", "Teacher").grid(sticky = E)
    #this create empty label for spacing purposes.
    Label(register,text = "").grid()
    #this create button to register the user. the function linked to the button will be called when the button is clicked.
    Button(register, text = "Register", command = registerUser).grid()

def login():
    global login
    global fullname
    global password
    global role

    #this create a new interface for the user to login
    login = Toplevel(auth)
    login.title("Login")

    #the following stringvar's are used to collect the data being inputted.
    fullname = StringVar(login)
    password = StringVar(login)
    role = StringVar(login)
    role.set("pick a role")

    Label(login, text = "Full name:").grid(column = 0, row = 1)
    #this create an entry so that data can be inputted for the full-name.
    Entry(login, textvariable = fullname).grid(column = 1, row = 1)

    Label(login, text = "Password:").grid(column = 0, row = 2)

    #this create an entry so that data can be inputted for the password.
    Entry(login, textvariable = password, show = "*").grid(column = 1, row = 2)

    #this would give the user a dropdown from which they can pick their role.
    OptionMenu(login, role, "Student", "Teacher").grid(sticky = E)

    #this create button to register the user and if validated the main interface would be created.
    Button(login, text = "Login", command = loginUser).grid()

def main():
    #this creates a file to track the validated students
    def createStudentsFile():
        if os.path.exists("validstudents.txt"): #this would check if the file exists
            pass
        else:
            #opens the file/creates it if it does not exist and writes the data into it.
            studentFile = open("validstudents.txt", "w")
```

```
        studentFile.write("#enter names of students that are valid\n")   #the default messages are for admin to customise the
textfile.
        studentFile.write("#format: firstname lastname (e.g. John Appleseed)")
        studentFile.close()

    #this creates a file to track the validated teachers
    def createTeachersFile():
        if os.path.exists("validteachers.txt"):
            pass
        else:
            #opens the file/creates it if it does not exist and writes the data into it.
            teacherFile = open("validteachers.txt", "w")
            teacherFile.write("#enter names of teachers that are valid\n")
            teacherFile.write("#format: firstname lastname (e.g. John Appleseed)")
            teacherFile.close()

    def createSubjectsFile():
        if os.path.exists("subjects.txt"):
            pass
        else:
            #this creates the file to input subjects that the school teaches. (by default there are the subjects that the 6th form
teaches)
            subjectsFile = open("subjects.txt", "w")
            subjectsFile.write("#enter the subjects below:")
            subjectsFile.close()

    #this function checks for any new subjects in the subjects files and updates it
    def updateSubjectClass():
        #creating a connection to the database
        connection = psycopg2.connect(host = "localhost", user = "postgres", password = "postgres", port = "5432", database =
"school")
        cursor = connection.cursor()

        #opens the subject textfile and gets every line in it.
        subjectsFile = open("subjects.txt", "r")
        subjects = subjectsFile.readlines()

        for subject in subjects:  #loops through all lines in subjects
            if subject.strip() == "#enter the subjects below:":
                pass  #disregards the line
            else:
                #this sql statement retrieves the subject from subjects. I did this so that the system can check if the subject already
exists or not.
                cursor.execute("SELECT subject FROM subjects WHERE subject = %s;", (subject.strip(),))

                if cursor.fetchone() is None:
                    #this would mean that thesubject does not exist and hence, adds the subject to the database
                    #the sql statement creates a new row by inserting the subject into the subjects table.
                    cursor.execute("INSERT INTO subjects (subject) VALUES (%s);", (subject.strip(),))
                    connection.commit()

        subjectsFile.close()
        connection.close()

    createStudentsFile()
    createTeachersFile()
    createSubjectsFile()

    databaseStatus = False
    #I am using exception handling to check whether the database is running or not.
    try:
        #this would create a connection to the database
        psycopg2.connect(host = "localhost", user = "postgres", password = "postgres", port = "5432", database = "school")
        databaseStatus = True
    except:
        #this would mean that an error was called/the database is offline
```

```
        print("Database isn't running right now.")

    #if the database is running, then i would create the data-tables and the menu interface for the user
    if databaseStatus == True:
        global auth
        connection = psycopg2.connect(host = "localhost", user = "postgres", password = "postgres", port = "5432", database =
"school")
        cursor = connection.cursor()

        #this creates the students table if it doesnt exist
        cursor.execute("""CREATE TABLE IF NOT EXISTS students (
                student_id SERIAL PRIMARY KEY,
                student_name VARCHAR(150),
                student_password VARCHAR(1000)
                );""")

        #this creates the teachers table if it doesn't exist
        cursor.execute("""CREATE TABLE IF NOT EXISTS teachers (
                teacher_id SERIAL PRIMARY KEY,
                teacher_name VARCHAR(150),
                teacher_password VARCHAR(1000)
                );""")

        #this creates the table for the subjects if it doesn't exist
        cursor.execute("""CREATE TABLE IF NOT EXISTS subjects (
                subject_id SERIAL PRIMARY KEY,
                subject VARCHAR(50)
                );""")

        #this creates the table for TeachersInClass if it doesn't exist
        cursor.execute("""CREATE TABLE IF NOT EXISTS teacherinclass (
                subject_id INTEGER,
                teacher_id INTEGER,
                FOREIGN KEY(subject_id) REFERENCES subjects(subject_id),
                FOREIGN KEY(teacher_id) REFERENCES teachers(teacher_id),
                teacher_name VARCHAR(150)
                );""")

        #this creates the table for StudentInClass if it doesn't exist
        cursor.execute("""CREATE TABLE IF NOT EXISTS studentinclass (
                subject_id INTEGER,
                student_id INTEGER,
                FOREIGN KEY(subject_id) REFERENCES subjects(subject_id),
                FOREIGN KEY(student_id) REFERENCES students(student_id),
                student_name VARCHAR(150)
                );""")

        #this creates the table for the grades if it doesn't exist
        cursor.execute("""CREATE TABLE IF NOT EXISTS grades (
                    grade_id SERIAL PRIMARY KEY,
                    subject_id INTEGER,
                    student_id INTEGER,
                    FOREIGN KEY(student_id) REFERENCES students(student_id),
                    FOREIGN KEY(subject_id) REFERENCES subjects(subject_id),
                    grade VARCHAR(2),
                    date DATE
                );""")

        connection.commit()
        connection.close()
        updateSubjectClass()

        #this would create the interface for the authentication system
        auth = Tk()
        auth.title("Grades+")
        #this create a button to give the user the option to register. When clicked, the register interface would be created.
        Button(text = "Register", height = "3", width = "20", command = register).pack()
```

```
    Label(text="").pack()
    #this create a button to give the user the option to login. When clicked, the login interface would be created.
    Button(text = "Login", height = "3", width = "20", command = login).pack()

    #this would run the tkinter window
    auth.mainloop()

if __name__ == "__main__":
    main()
```

```
#import modules which would be used for the system
from tkinter import * #used for gui
from tkinter import ttk
from tkinter import messagebox
from argon2 import PasswordHasher #used to hash the password
import time
import psycopg2 #used for the database

#create the base class for the system
class School:
    def __init__(self, userFullName, password):
        #this would store the user's properties so that it can be used throughout the program.
        self.userFullName = userFullName.lower().strip()
        self.password = password

    #this function would return the attribute "userFullName"
    def getName(self):
        return self.userFullName

    #this function would return the attribute "password"
    def getPassword(self):
        return self.password

    #this procedure would create a messagebox when required in my system
    def messageBox(self, textLabel, messageType):
        if messageType == "error":
            #this is an error box which would be created when the input is error.
            messagebox.showerror("Grades+", "Error: " + textLabel)
        elif messageType == "info":
            #this creates an info box
            messagebox.showinfo("Grades+", textLabel)

    #this function checks if the password is valid
    def isPasswordValid(self, password):
        if not password: #checks if the string is empty (stirng-falsy)
            #this would show errorbox as the entry is invalid
            self.messageBox("Enter a password.", "error")

        #this condition checks if the password's length is less than 5
        elif len(password) < 5:
            #as the password is smaller than 5 characters, it would throw an errorbox.
            self.messageBox("Password must be atleast 5 characters long.", "error")
            return False
        #checks if the password's length is more than 10
        elif len(password) > 10:
            self.messageBox("Password cannot be longer than 10 characters. ", "error")
            return False
        else:
            return True #if password is valid, the function would return true

#this class is a subclass which inherits the methods of it's parent class which is School.
class Teacher(School):
    #this function would follow an algorithm in order to add the teacher to the database
    def createTeacher(self, window):
        #this creates a connection to the database
        connection = psycopg2.connect(host = "localhost", user = "postgres", password = "postgres", port = "5432", database =
"school")
        cursor = connection.cursor()

        #Here I am opening and reading the lines of the valid teachers textfiles to check if the corresponding teacher is valid.
        teachersFile = open("validteachers.txt", "r")
        validTeachers = teachersFile.readlines()
        validated = False
        #looping through the file to see if the teacher is in it
        for teacher in validTeachers:
```

```
        if self.getName() == teacher.strip().lower():
            validated = True
            #this sql statement is used to check if the teacher exists in the database
            cursor.execute("SELECT teacher_name FROM teachers WHERE teacher_name = %s;", (self.getName(),))
            if cursor.fetchone() is not None:
                #teacher exists in the database since value is not null
                self.messageBox("Already registered.", "error")
            else:
                #the teacher is valid
                #the system would checks if password is valid
                if self.isPasswordValid(self.getPassword()) == True:
                    #as the teacher does not exist; the sql statement would add the teacher to the database
                    hashedPassword = PasswordHasher().hash(self.getPassword())
                    cursor.execute("INSERT INTO teachers (teacher_id, teacher_name, teacher_password) VALUES (DEFAULT,
%s, %s);", (self.getName(), hashedPassword))
                    connection.commit()
                    self.messageBox("Success! You have been registered.", "info")
                    window.after(3000, window.destroy)

        #if value of validated did not change then that means the student is not valid
        if validated == False:
            self.messageBox("You are not a validated teacher.", "error")

        connection.close()

    #this function is the core as it provides the main interface for the teacher.
    def sendGUI(self):
        connection = psycopg2.connect(host = "localhost", user = "postgres", password = "postgres", port = "5432", database =
"school")
        cursor = connection.cursor()

        #this creates the window for the interface
        gui = Tk()
        gui.title("Grades+")
        gui.geometry("1500x800")
        menubar = Menu(gui)

        #this procedure would stop the program from running
        def logOut():
            exit(0)

        #this function is for the class interface
        def subjectGUI(subject):
            #this would create the class interface
            subjectGUI = Toplevel(gui)
            subjectGUI.title(subject)
            menubar = Menu(gui)

            #the treeview method is being used to show the teachers the grades that were assigned to the teacher
            gradesTable = ttk.Treeview(subjectGUI, columns = ("Name", "Grade", "Date assigned"), show = "headings")
            gradesTable.heading("Name", text = "Name")
            gradesTable.heading("Grade", text = "Grade")
            gradesTable.heading("Date assigned", text = "Date assigned")
            gradesTable.grid()

            #this sql statement retrieves the subject id for the subject arguement that was passed into the function
            cursor.execute("SELECT subject_id FROM subjects WHERE subject = %s;", (subject,))
            subjectid = cursor.fetchone()

            #this parameterised sql statement is used to retrieve the student's name grade and the date that the grade was assigned
on.
            cursor.execute("SELECT students.student_name, grades.grade, grades.date FROM students INNER JOIN grades ON
grades.student_id = students.student_id WHERE grades.subject_id = %s;", (subjectid,))

            data = cursor.fetchall()
            for i in range(len(data)):
```

```
            #the student's data is being added onto the treeview
            gradesTable.insert("", "end", values = (data[i][0], data[i][1], data[i][2]))

        #this interface is used by teachers to add students to the class or remove them from the class
        def studentGUI():
            #this creates the new tab for the interface
            studentGUI = Toplevel(subjectGUI)
            #this sql statement retrieves all the students that are registered in the database
            cursor.execute("SELECT student_name FROM students;")
            names = cursor.fetchall()
            students = []
            classEmpty = False
            #I am looping through the fetched data so that it can be added onto an array.
            if names is None or not names:
                classEmpty = True
            else:
                #this gets every students name from the  result
                for s in names:
                    for student in s:
                        students.append(student)

            #if there is no-one in the class, the teacher would be unable to use the functionalities.
            if classEmpty == True:
                Label(studentGUI, text = "There are no students who have currently registered. You can start adding students into your class(es) once they have been registered").grid()
            else:
                #this creates a dropdown with every student that is on the array/database
                s = StringVar(studentGUI)
                s.set("select a student")
                OptionMenu(studentGUI, s, *students).grid(column = 1, row = 1)

                #this function is designated to adding students to a class
                def addStudent():
                    cursor.execute("SELECT subject_id FROM subjects WHERE subject = %s;", (subject,))
                    subjectid = cursor.fetchone()

                    #this sql statement retrieves all students that are in the class
                    cursor.execute("SELECT student_name FROM studentinclass WHERE subject_id = %s AND student_name = %s;", (subjectid, s.get(),))
                    inClass = False

                    if cursor.fetchone() is not None:
                        #the student is in class so it throws an error to the teacher
                        self.messageBox("This student is already in this class.", "error")
                        inClass = True

                    #if the student is not in the class, then they would be added to the class and
                    if inClass == False:
                        if s.get() != "select a student":
                            self.messageBox(s.get() + " was added to this class.", "info")

                            cursor.execute("SELECT subject_id FROM subjects WHERE subject = %s;", (subject,))
                            subjectid = cursor.fetchone()
                            #this sql statement retrieves the id that is associated with the chosen student (needed so that the student can be added to the database)
                            cursor.execute("SELECT student_id FROM students WHERE student_name = %s;", (s.get(),))
                            studentid = cursor.fetchone()
                            #this sql statement inserts the student to the class
                            cursor.execute("INSERT INTO studentinclass (subject_id, student_id, student_name) VALUES (%s, %s, %s);", (subjectid, studentid, s.get(),))
                            connection.commit()

                #this function is called when the teacher wants to remove a student from a class
                def removeStudent():
                    cursor.execute("SELECT subject_id FROM subjects WHERE subject = %s;", (subject,))
                    subjectid = cursor.fetchone()
```

```
                #this sql statement retrieves every student in the chosen class
                cursor.execute("SELECT student_name FROM studentinclass WHERE subject_id = %s AND student_name =
%s;", (subjectid, s.get(),))
                inClass = False

                if cursor.fetchone() is not None:
                    #the student was found
                    #this sql statement would remove the chosen student from the class in the database
                    cursor.execute("DELETE FROM studentinclass WHERE student_name = %s AND subject_id = %s;",
(s.get(), subjectid,))
                    connection.commit()
                    self.messageBox(s.get() + " is no longer in this class.", "info")
                    inClass = True

                #if the student was not in the class, it would throw an error to the teacher.
                if inClass == False: #student is not in the class
                    if s.get() != "select a student":
                        self.messageBox("This student is not in this class.", "error")

            #these buttons are what the teacher clicks in order to add or remove the student.
            Button(studentGUI, text = "-", command = removeStudent).grid(column = 0, row = 1)
            Button(studentGUI, text = "+", command = addStudent).grid(column = 2, row = 1)

        # this function is one of the main functionalities for the teacher
        # student's grades are assigned through this function
        def setGrade(subject):
            setgrades = Toplevel(gui)

            cursor.execute("SELECT subject_id FROM subjects WHERE subject = %s;", (subject,))
            subjectid = cursor.fetchone()
            #this sql statement retrieves the students who are in the class
            cursor.execute("SELECT student_name FROM studentinclass WHERE subject_id = %s;", (subjectid,))

            names = cursor.fetchall()
            students = []
            classEmpty = False
            #i am looping through the retrieved data so that they can be added onto an array and hence be presented on a
dropdown menu
            if names is None or not names:
                classEmpty = True
            else:
                #this gets every students name in the database
                for s in names:
                    for student in s:
                        #the names are added onto the array
                        students.append(student)

            #if there is no-one in the class, they would not be able to assign the grades so they must add students to the class
            if classEmpty == True:
                Label(setgrades, text = "This class currently has no students. Please add students using the menu.").grid()
            else: #if there are students in the class
                #this creates a dropdown so the teacher can select a teacher
                s = StringVar(setgrades)
                s.set("select a student")
                #a dropdown for the teacher to select a student.
                Label(setgrades, text = "Student: ").grid(column = 0,row = 1)
                OptionMenu(setgrades, s, *students).grid(column = 1,row = 1)

                grade = StringVar(setgrades)
                grade.set("select a grade")
                grades = ["A*","A","B","C","D","E","U"]
                #the grades U-A* are put as a dropdown so the teacher can assign it to the student
                Label(setgrades, text = "Grade: ").grid(column = 0,row = 3)
                OptionMenu(setgrades, grade, *grades).grid(column = 1 ,row = 3)

                #this functions assigns the grade to the student
```

```
            def updateGrades():
                if grade.get() != "select a grade" and s.get() != "select a student":
                    cursor.execute("SELECT student_id FROM students WHERE student_name = %s;", (s.get(),))
                    studentid = cursor.fetchone()
                    date = time.strftime("%Y-%m-%d")
                    #the grade is assigned to the student in the database
                    cursor.execute("INSERT INTO grades (grade_id, subject_id, student_id, grade, date) VALUES (DEFAULT,
%s, %s, %s, %s);", (subjectid, studentid, grade.get(), date))
                    connection.commit()
                    self.messageBox("Grade: " + grade.get() + " was successfully set to " + s.get() + ".", "info")
            #the "add" button is what the teacher clicks on to set the grade.
            Button(setgrades, text = "add", command = updateGrades).grid()
            connection.commit()

        #menu is created so that the teacher can see the different options
        studentMenu = Menu(menubar, tearoff = 0)
        studentMenu.add_command(label = "add/remove student", command = studentGUI)
        menubar.add_cascade(label = "Class", menu = studentMenu)

        gradesMenu = Menu(menubar, tearoff = 0)
        gradesMenu.add_command(label = "Set Grade", command = lambda: setGrade(subject))
        menubar.add_cascade(label = "Grades", menu = gradesMenu)

        subjectGUI.config(menu = menubar)

    #this function creates the interface for the teacher to convert a % to a grade
    def grade():
        global converter
        converter = Toplevel(gui)
        converter.title("Convert a % to a rough grade")
        #this function is used to convert percentages to a grade
        def convertGrade():
            percentage = percent.get()
            #checks if the percentage is greater than 100
            if percentage > 100:
                self.messageBox("Percentage cannot be higher than 100.", "error")
            elif percentage >= 82: #around 82% is an A*
                self.messageBox("Grade: A*", "info")
            elif percentage >= 70:  #around 70% is an A
                self.messageBox("Grade: A", "info")
            elif percentage >= 57:  #around 57% is an B
                self.messageBox("Grade: B", "info")
            elif percentage >= 45:  #around 45% is an C
                self.messageBox("Grade: C", "info")
            elif percentage >= 30:  #around 30% is an D
                self.messageBox("Grade: D", "info")
            elif percentage >= 20:  #around 20% is an E
                self.messageBox("Grade: E", "info")
            elif 0 > percentage:
                self.messageBox("% must be a number between 0 and 100.", "error")
            else:
                self.messageBox("Grade: U", "info")

        percent = IntVar(converter)
        Label(converter, text = "%:").grid(column = 0,row = 1)
        Entry(converter, textvariable = percent).grid(column = 1,row = 1)
        #the "convert" button calls the convertGrade function and converts the %
        Button(converter, text = "convert", command = convertGrade).grid()

    #this function would be used by teachers to add or remove a class
    def subjects():
        #this function is used by the teacher to add a class that they teach
        def addSubject():
            cursor.execute("SELECT subject_id FROM subjects WHERE subject = %s;", (s.get(),))
            subjectid = cursor.fetchone()
            #this retrieves the teachers that are in the chosen class (s.get()).
```

```
            cursor.execute("SELECT teacher_name FROM teacherinclass WHERE subject_id = %s AND teacher_name = %s;",
(subjectid, self.getName(),))
            inClass = False
            if cursor.fetchone() is not None:
                #the teacher is already in the class so an error is thrown
                self.messageBox("You are already in this class.", "error")
                inClass = True

            #if the teacher is not in the class, the system would add it to the database
            if inClass == False:
                if s.get() != "select a subject":
                    #add the teacher to the class/subject array in both teachers table and subjects table
                    self.messageBox("You are now in this class.", "info")
                    cursor.execute("SELECT teacher_id FROM teachers WHERE teacher_name = %s;", (self.getName(),))
                    teacherid = cursor.fetchone()
                    #this sql statements inserts the teacher into the class's table.
                    cursor.execute("INSERT INTO teacherinclass (subject_id, teacher_id, teacher_name) VALUES (%s, %s, %s);",
(subjectid, teacherid, self.getName(),))
                    connection.commit()
                    classMenu.add_command(label = s.get(), command = lambda subject = s.get(): subjectGUI(subject))

        #this function would be called if the teacher wanted to leave a class
        def removeSubject():
            cursor.execute("SELECT subject_id FROM subjects WHERE subject = %s;", (s.get(),))
            subjectid = cursor.fetchone()

            cursor.execute("SELECT teacher_name FROM teacherinclass WHERE subject_id = %s AND teacher_name = %s;",
(subjectid, self.getName(),))

            inClass = False
            if cursor.fetchone() is not None:
                #if the teacher was found, the system  would remove teacher from the class in the database
                cursor.execute("SELECT teacher_id FROM teachers WHERE teacher_name = %s;", (self.getName(),))
                teacherid = cursor.fetchone()

                cursor.execute("SELECT subject_id FROM subjects WHERE subject = %s;", (s.get(),))
                subjectid = cursor.fetchone()
                #the DELETE sql statement gets rid of the entry where the teacher is in the class.
                cursor.execute("DELETE FROM teacherinclass WHERE teacher_id = %s AND subject_id = %s;", (teacherid,
subjectid,))
                connection.commit()

                #this removes the subject from the menu
                classMenu.delete(s.get())

                self.messageBox("You are no longer in this class.", "info")
                inClass = True

            #However, if the teacher is not in the class, an error would be called.
            if inClass == False:
                self.messageBox("You are not in this class.", "error")

        classGUI = Toplevel(gui)
        classGUI.geometry("250x250")

        #the following sql statement retrieves all the subjects avaialable
        cursor.execute("SELECT subject FROM subjects;")
        subject = cursor.fetchall()
        subjects = []

        for s in subject: #get every subject in the database
            subjects.append(s[0])

        s = StringVar(classGUI)
        s.set("select a subject")
```

```
        OptionMenu(classGUI, s, *subjects).grid(column = 1, row = 1)
        #the following buttons would add or remove the teacher from a class when it is clicked
        Button(classGUI, text = "-", command = removeSubject).grid(column = 0, row = 1)
        Button(classGUI, text = "+", command = addSubject).grid(column = 2, row = 1)

    aboutMenu = Menu(menubar, tearoff = 0)
    aboutMenu.add_command(label = "Log Out", command = logOut)
    menubar.add_cascade(label = "Settings", menu = aboutMenu)
    classMenu = Menu(menubar, tearoff = 0)
    classMenu.add_command(label = "add/remove a subject", command = subjects)
    #this iterates through the classes that the teacher is in
    cursor.execute("SELECT subject_id FROM teacherinclass WHERE teacher_name = %s;", (self.getName(),))
    subjectid = cursor.fetchall()
    for subject in subjectid:
        if subject is None or not subject:
            pass
        else:
            #the teacher is in the class so it is added to the dropdown
            for subj in subject:
                cursor.execute("SELECT subject FROM subjects WHERE subject_id = %s;", (subj,))
                classes = cursor.fetchone()
                classMenu.add_command(label = classes[0], command = lambda: subjectGUI(classes[0]))

    menubar.add_cascade(label = "Classes", menu = classMenu)
    gradesMenu = Menu(menubar, tearoff = 0)
    gradesMenu.add_command(label = "Convert % to grade.", command = grade)
    menubar.add_cascade(label = "Grades", menu = gradesMenu)
    gui.config(menu = menubar)
    gui.mainloop()
    connection.close()

#the student class is used to instantiate students as objects. Like the teacher class, this class also inherits the methods of the School class.
class Student(School):
  def createStudent(self, window):
    connection = psycopg2.connect(host = "localhost", user = "postgres", password = "postgres", port = "5432", database = "school")
    cursor = connection.cursor()

    #the following checks if the student is in the validated students textfile
    studentsFile = open("validstudents.txt", "r")
    validStudents = studentsFile.readlines()
    validated = False
    #loops through each line in the textfile
    for student in validStudents:
        #if the line is the same as the students name then it means that they are validated
        if self.getName() == student.strip().lower():
            validated = True
            #as the student is valid, the system would check if they are already in the database
            cursor.execute("SELECT student_name FROM students WHERE student_name = %s;", (self.getName(),))
            if cursor.fetchone() is not None:
                #student exists in database since value is not null
                self.messageBox("Already registered.", "error")
            else:
                #as the student does not exist, the system would check if the password is valid
                if self.isPasswordValid(self.getPassword()) == True:
                    #if it is, then the student is added to the database
                    hashedPassword = PasswordHasher().hash(self.getPassword())
                    cursor.execute("INSERT INTO students (student_id, student_name, student_password) VALUES (DEFAULT , %s , %s);", (self.getName(), hashedPassword))
                    connection.commit()
                    connection.close()
                    self.messageBox("Success! You have been registered.", "info")
                    window.after(3000, window.destroy)
    #if value of the Boolean validated did not change then that means the student is not valid
    if validated == False:
```

```
            self.messageBox("You are not a validated student.", "error")

        connection.close()

    #this provides the main interface for the student.
    def sendGUI(self):
        connection = psycopg2.connect(host = "localhost", user = "postgres", password = "postgres", port = "5432", database =
"school")
        cursor = connection.cursor()
        #creates the tkinter window for the interface
        gui = Tk()
        gui.title("Grades+")
        gui.geometry("1500x800")
        menubar = Menu(gui)

        #this procedure would stop the program from running.
        def logOut():
            exit(0)

        #when this function is called, it shows the student their grades for the chosen subject
        def subjectGUI(subject):
            subjectGUI = Toplevel(gui)
            subjectGUI.title(subject)

            cursor.execute("SELECT subject_id FROM subjects WHERE subject = %s;", (subject,))
            subjectid = cursor.fetchone()

            cursor.execute("SELECT student_id FROM students WHERE student_name = %s;", (self.getName(),))
            studentid = cursor.fetchone()

            #this sql statement retrieves the date and grade that were assigned to the student for the class
            cursor.execute("SELECT grade, date FROM grades WHERE student_id = %s AND subject_id = %s;", (studentid,
subjectid,))
            grades = cursor.fetchall()
            #to represent the grades, i am using treeview.
            gradesTable = ttk.Treeview(subjectGUI, columns = ("Grade", "Date assigned"), show = "headings")
            gradesTable.heading("Grade", text = "Grade")
            gradesTable.heading("Date assigned", text = "Date assigned")
            gradesTable.grid()

            for i in range(len(grades)):
                gradesTable.insert("", "end", values = (grades[i][0], grades[i][1]))

        #the menu widget is used to allow the students to pick what they want to do
        settingsMenu = Menu(menubar, tearoff = 0)
        settingsMenu.add_command(label = "Log Out", command = logOut)
        menubar.add_cascade(label = "Settings", menu = settingsMenu)
        subjectMenu = Menu(menubar, tearoff = 0)

        cursor.execute("SELECT student_id FROM students WHERE student_name = %s;", (self.getName(),))
        student = cursor.fetchone()

        cursor.execute("SELECT subject_id FROM studentinclass WHERE student_id = %s;", (student,))
        subjectid = cursor.fetchall()

        #this would check what classes the student is in
        for subject in subjectid:
            if subject is None or not subject:
                pass
            else: #student was found in a class
                for subj in subject:
                    #the class is added to a dropdown for the student to select if they want to see their grade.
                    cursor.execute("SELECT subject FROM subjects WHERE subject_id = %s;", (subj,))
                    classes = cursor.fetchone()
                    subjectMenu.add_command(label = classes[0], command = lambda subject = classes[0]: subjectGUI(subject))
```

```
    menubar.add_cascade(label = "Subjects", menu = subjectMenu)

    gui.config(menu = menubar)
    gui.mainloop()
    connection.close()
```

TESTING

TESTING PLAN

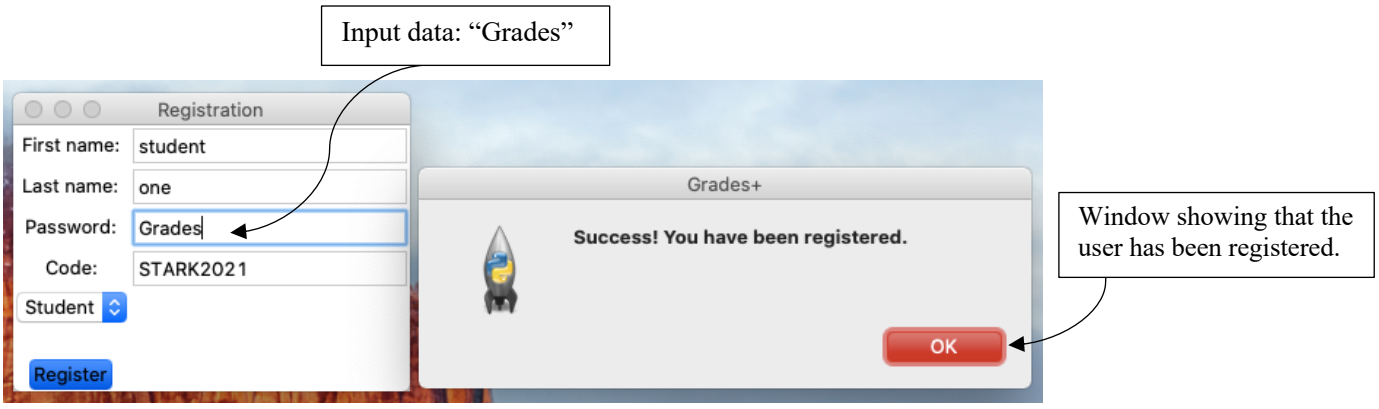
In this section, I am going to test the new system. This would be done by breaking my system into smaller components such as the authentication system (login/register), teacher interface and the student interface. In each of the interfaces, I would test each individual functionality. This would then be recorded in a data and screenshots would be shown to show the outcome. I have chosen to do module testing as it allows me to show how each function works and the type of data which needs to be used. A table would be used to describe the test and the input/output value which would be required for it to work. This allows me to see if I am meeting my system's objectives which is required to fulfil my end-users needs.

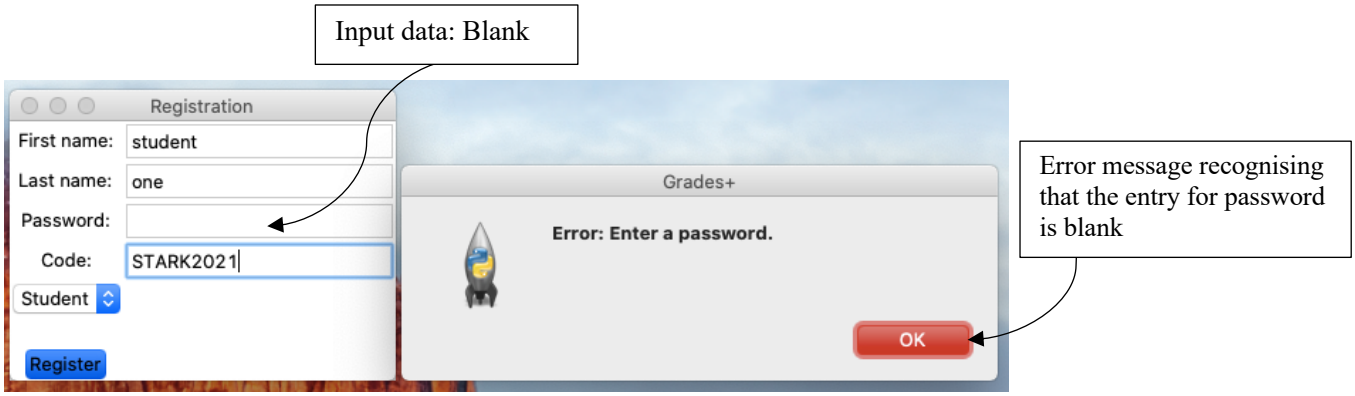
AUTHENTICATION SYSTEM

Test Number	*Description of Test*	*Test Data*	*Expected Result*
1	This test would check if the password that is used during authentication is valid.	A test data I would be using is “Grades” as it is between the 5 < x < 10 range. If the data is not in that range or is blank, this is erroneous data which would result in an error message.	If the password entered is valid, then I would expect no error messages, but this would be dependent on other entries such as role and code. However, if the data is invalid I would expect an error message to appear alerting the user to enter a password which is valid.

This is the actual result:

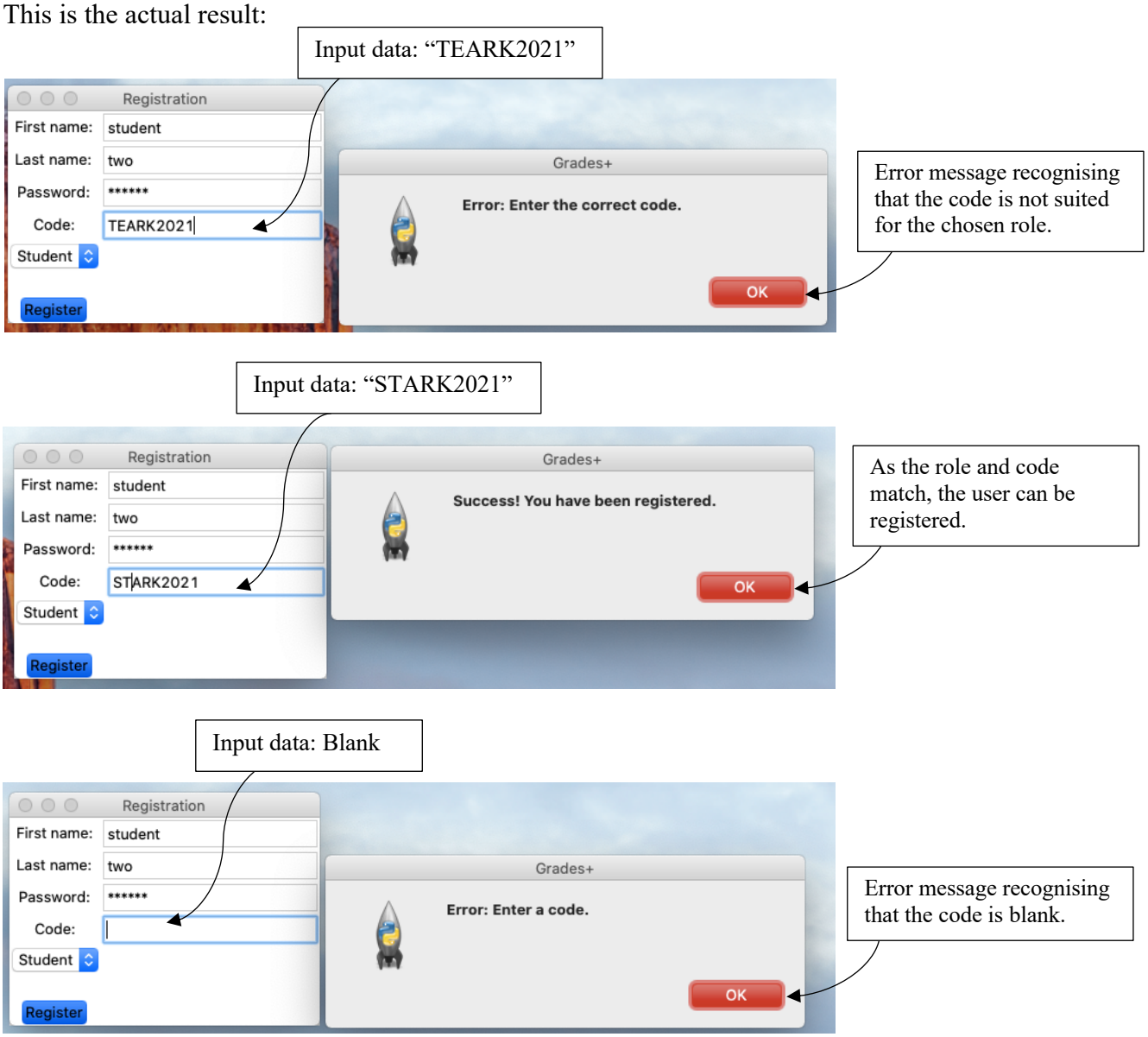
It can be seen that the password would be checked through the isPasswordValid() function and if it is not valid, then there would be an error box which appears.





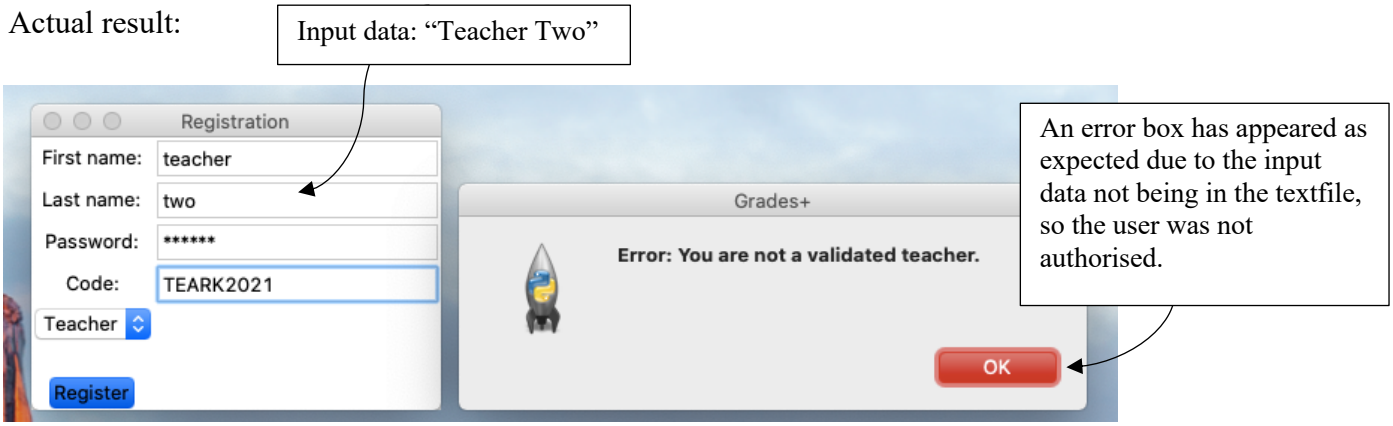
Test Number	*Description of Test*	*Test Data*	*Expected Result*
2	This test would check if the system would recognise the code that is being entered. This is crucial as if the system would allow a student to register as a teacher, they would be able to set their own grades.	The valid codes are: "STARK2021" for a student, "TEARK2021" for a teacher.	If the code entered incorrect for the role that the user chose, then I would expect there to be an error message. I would also expect an error message if there is no code entered.

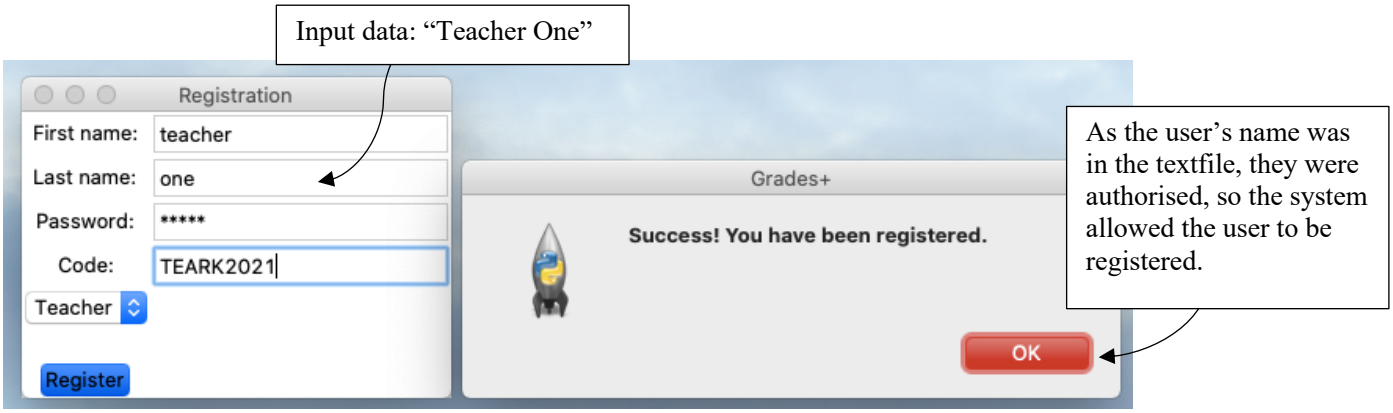
This is the actual result:

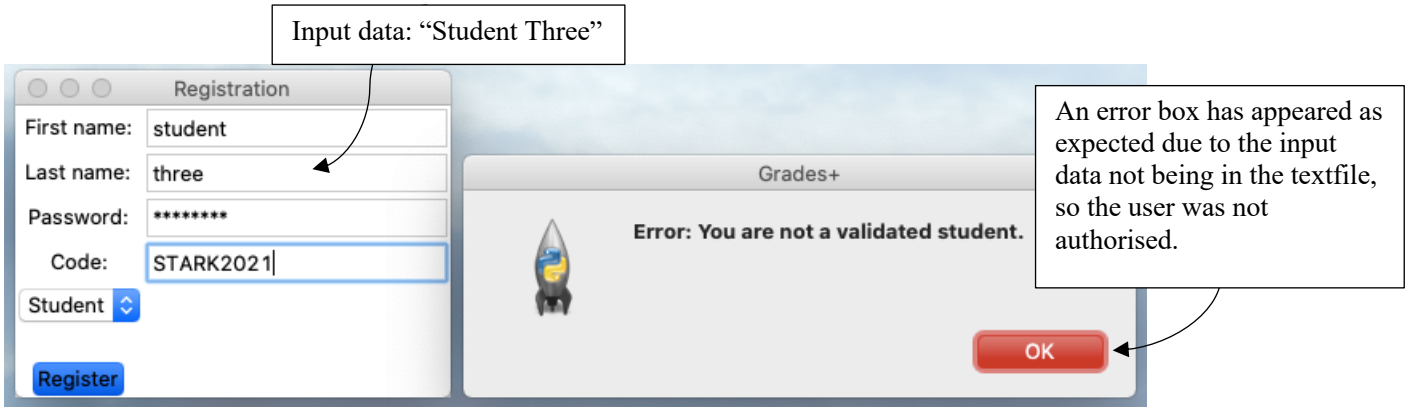


Test Number	*Description of Test*	*Test Data*	*Expected Result*
3	In this test, I would be checking if the user is authenticated by the system. I would also be testing my role dropdown and see if the system can detect if user can register with the role they chose.	As a test data, the student's textfile would have only "student two" in it and "teacher one" in the teacher's textfile.	If the user is not "student two" or "teacher one", I would expect the system to prevent them from registering an account. Otherwise, an error messagebox should appear.

Actual result:



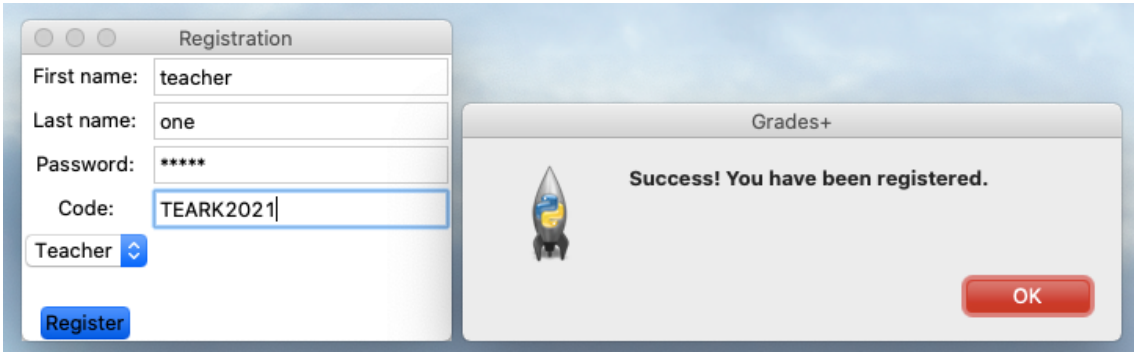




Test Number	*Description of Test*	*Test Data*	*Expected Result*
4	This test is to check if the system would recognise the user by their credentials and log them into the interface.	For this test, I would have an account registered: Full Name: “Teacher One” Password: “Grades”	I would expect the teacher interface to appear if the credentials are correct, however, if the password is misspelt, I would expect an error messagebox. If the name is incorrect, I would expect the system to recognise this and create an errorbox saying that the account does not exist.

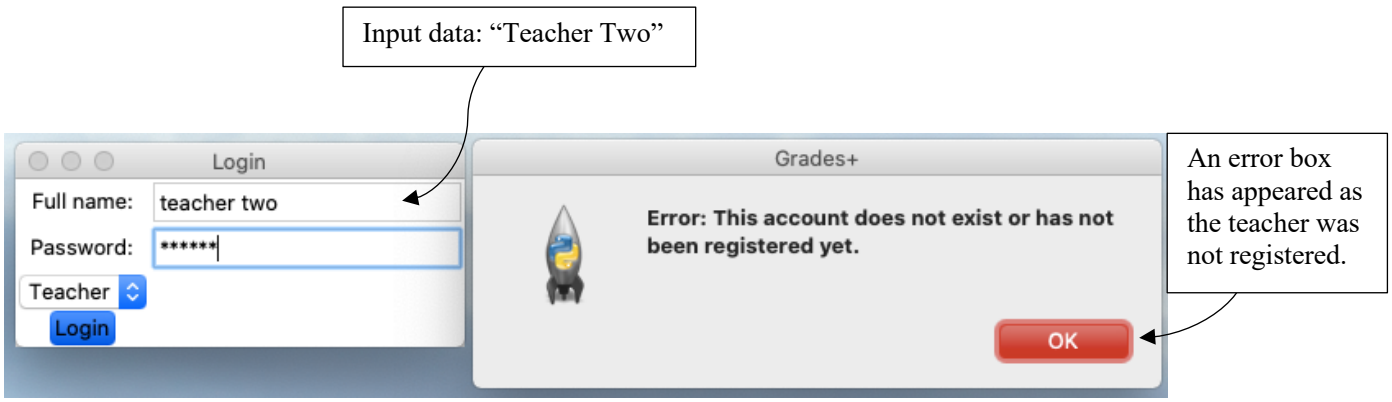
Actual result:

This is the account registration:

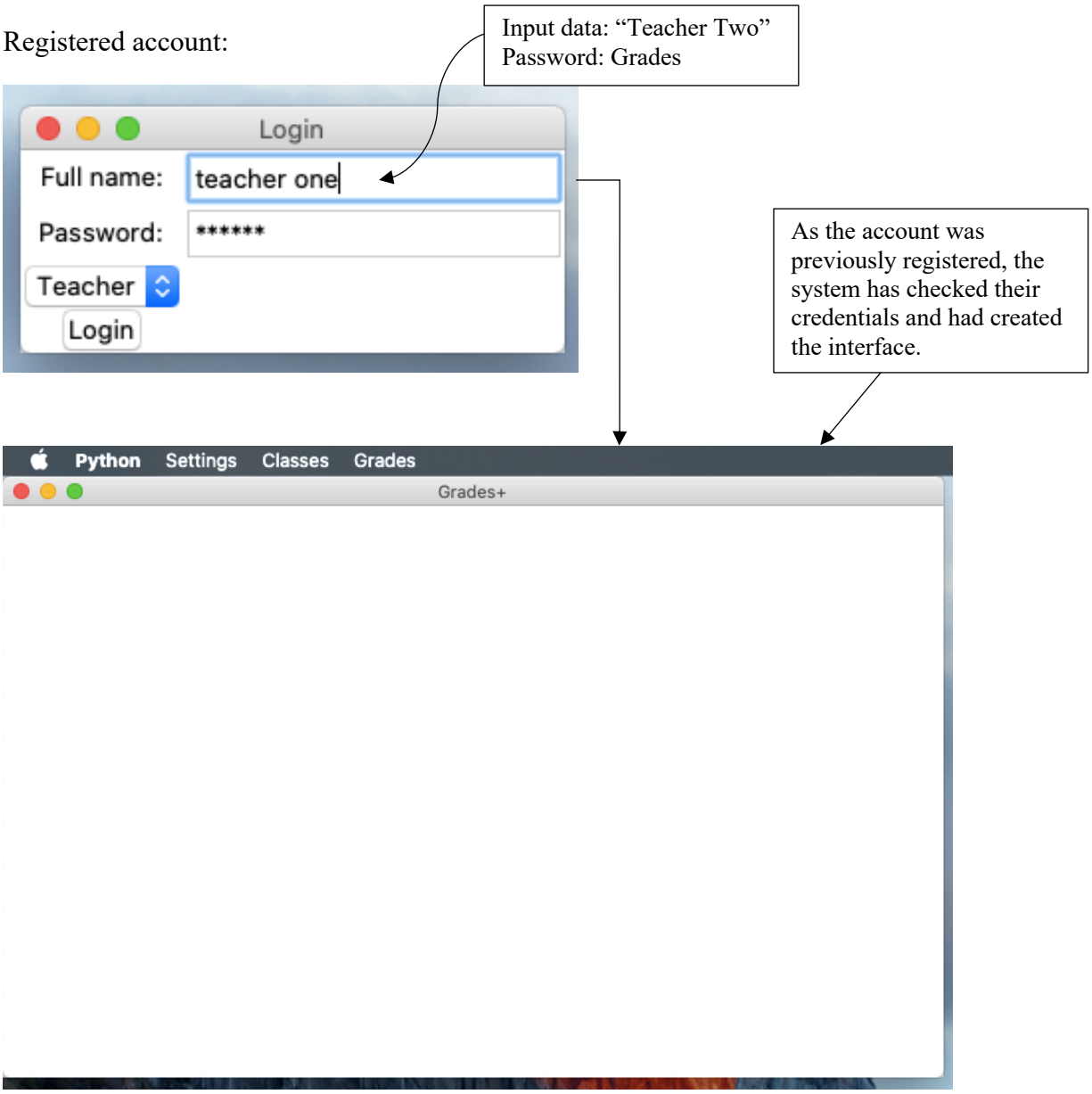


This is the user logging into the account:

Unregistered account:

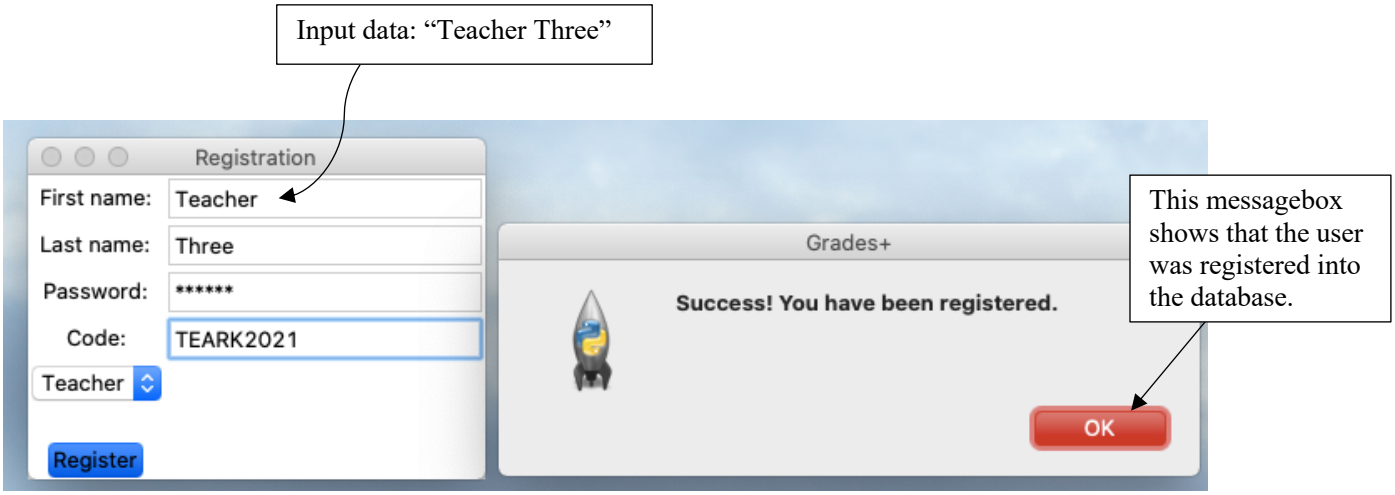


Registered account:

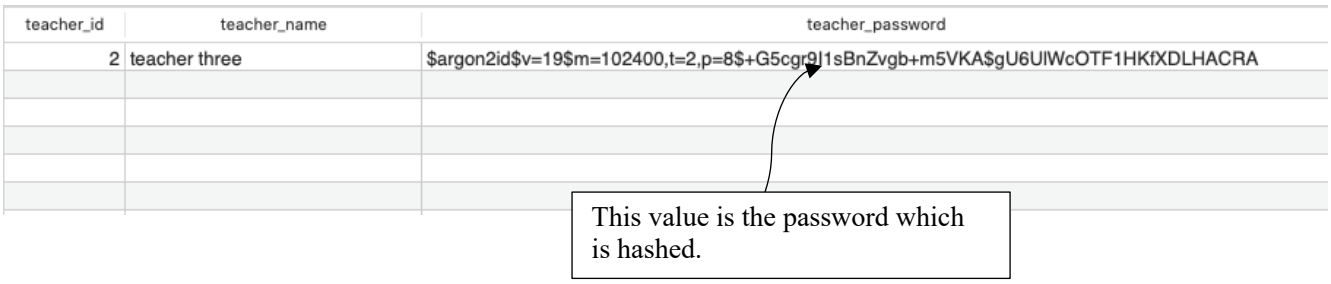


Test Number	*Description of Test*	*Test Data*	*Expected Result*
5	This test is to check if the password that is entered is being hashed and stored into this database.	For this test, I would have an account registered: Full Name: “Teacher Three” Password: “Grades”	In the database, I would expect the password to be hashed so that it is hard for hackers to find a way to get the user’s password. I will show the results using PSequel which is an application that allows me to see what is being stored into the database.

Actual result:



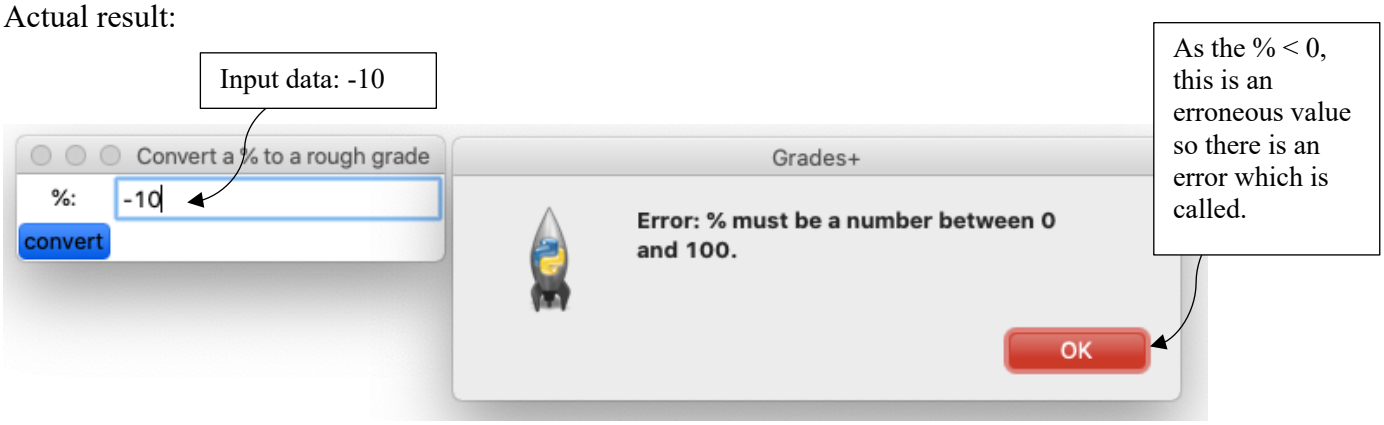
Here, it can be seen that the password that is being stored into the database is hashed using the argon library.

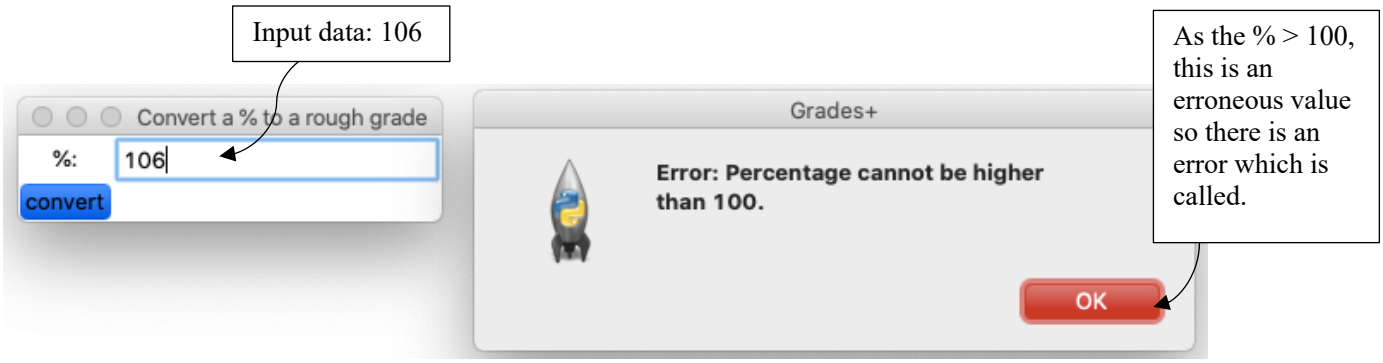


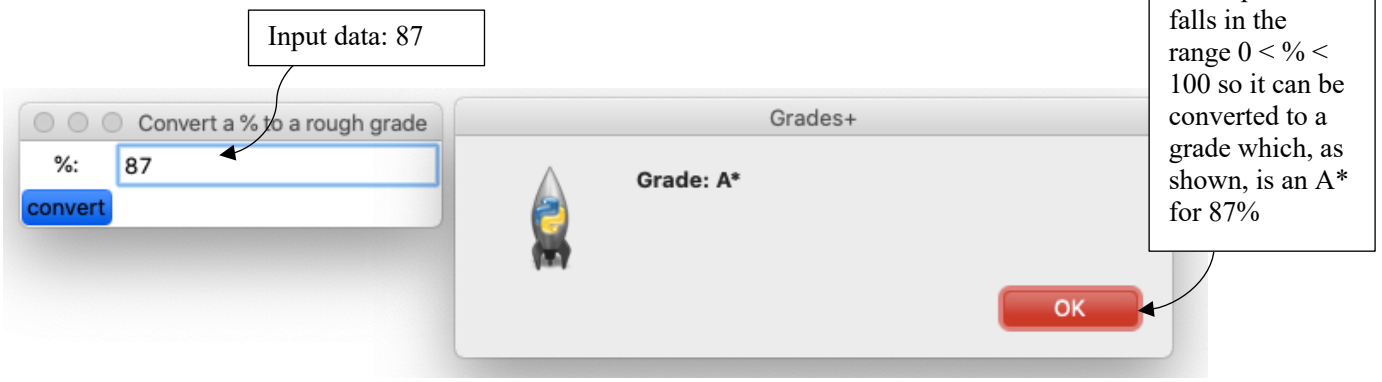
TEACHER INTERFACE

Test Number	*Description of Test*	*Test Data*	*Expected Result*
6	In this test, I would be checking if I can convert a percentage to a grade using the "convert grade" option.	To conduct this test, I would use "-10", "87" and "106" as my values to be entered. The function expects a value between 0 and 100 to be entered so erroneous values which is where % > 100 and % < 0 should throw an error.	For values "-10" and "106" I would expect there to be an error box as they do not fit in the range that is accepted. However, for "87", I would expect there to be an output of "A*".

Actual result:



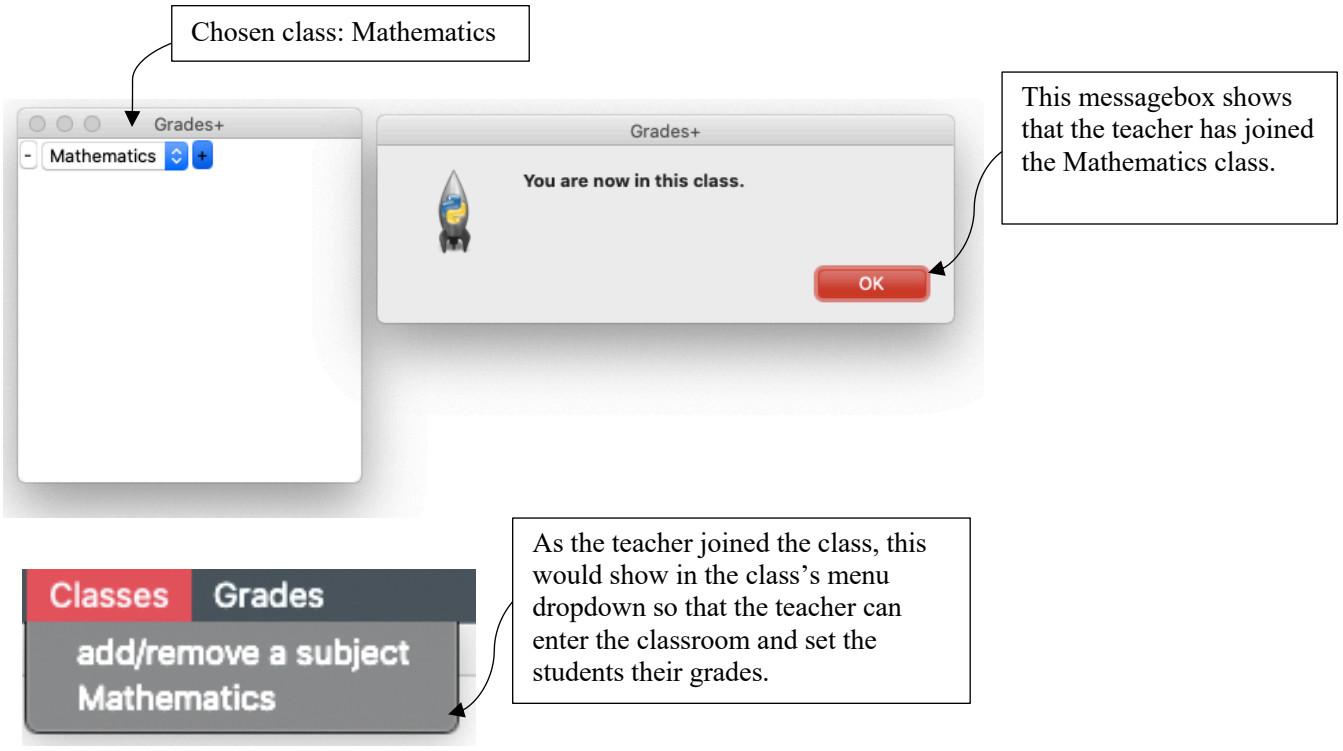




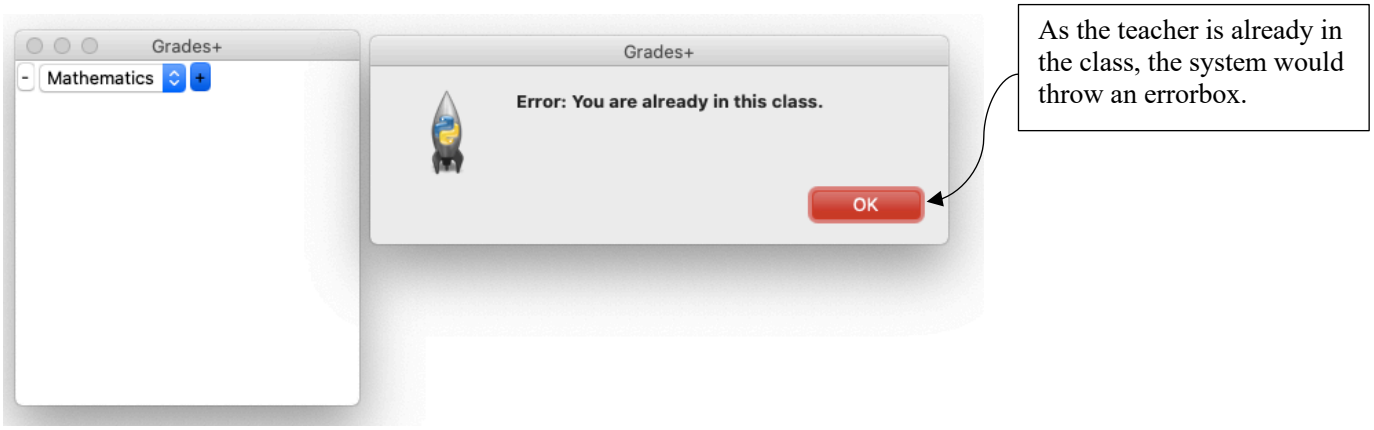
Test Number	*Description of Test*	*Test Data*	*Expected Result*
7	This test is to check if the teacher is able to add and remove a class.	I would add the teacher to the "Mathematics" class and remove them from the class.	When adding the class, I would expect the class to appear on the menu dropdown. If the user was not in the class, I would expect an error saying that the user is not in the class, else, the class would be removed and no longer be available in the menu dropdown.

Actual result:

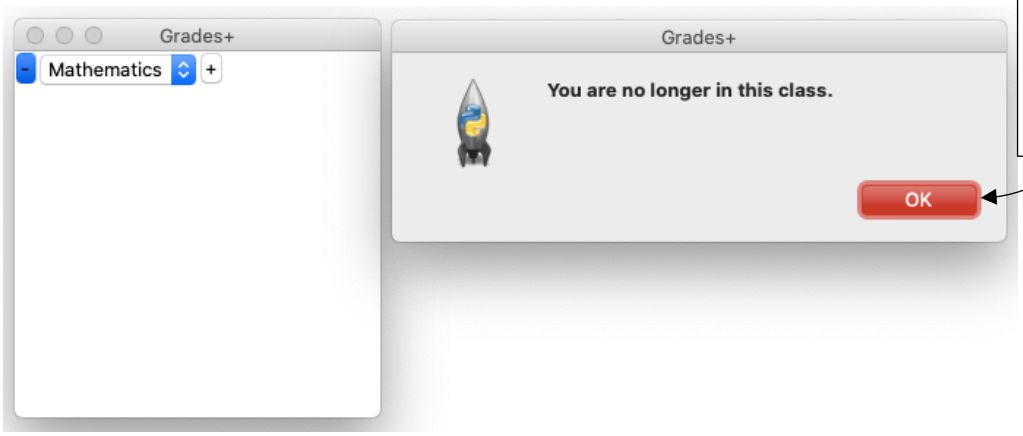
Adding a class:



Joining the same class after adding it:

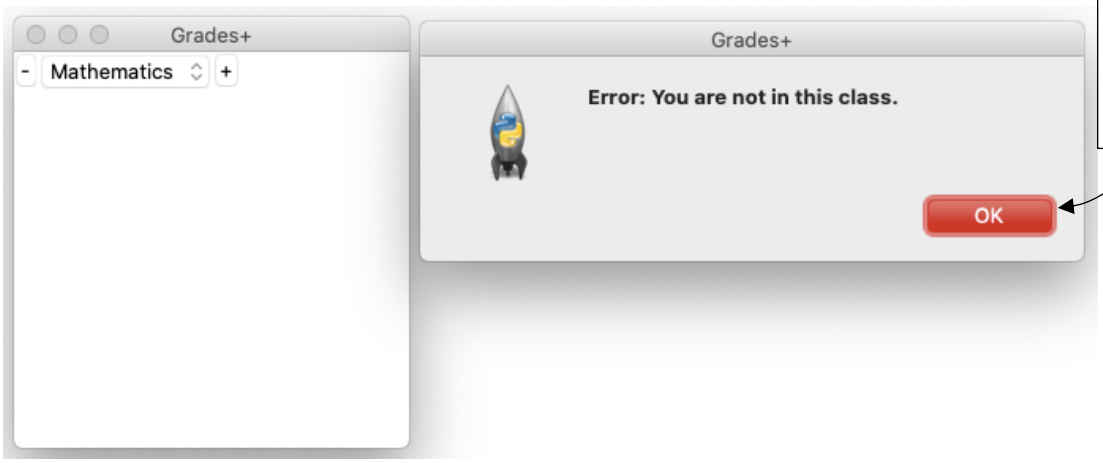


Removing a class:



The system would remove the teacher from the class if they are in it. This would remove the entry in the database. The class will no longer show in the menu dropdown.

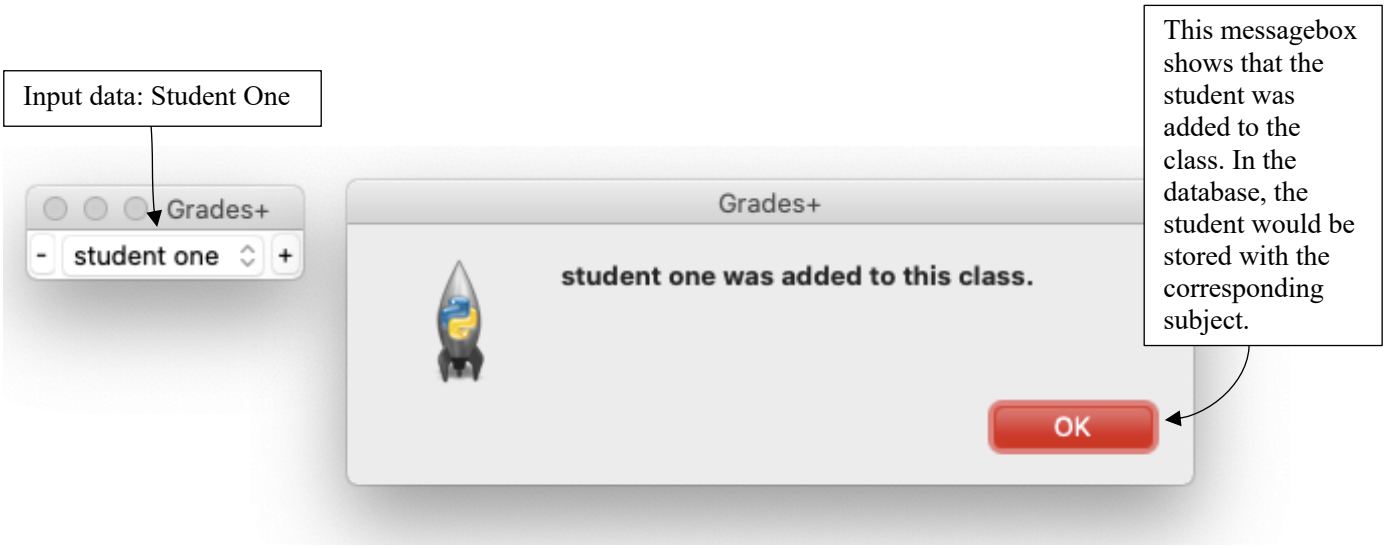
Removing a class when the teacher has already left it/is not in it:



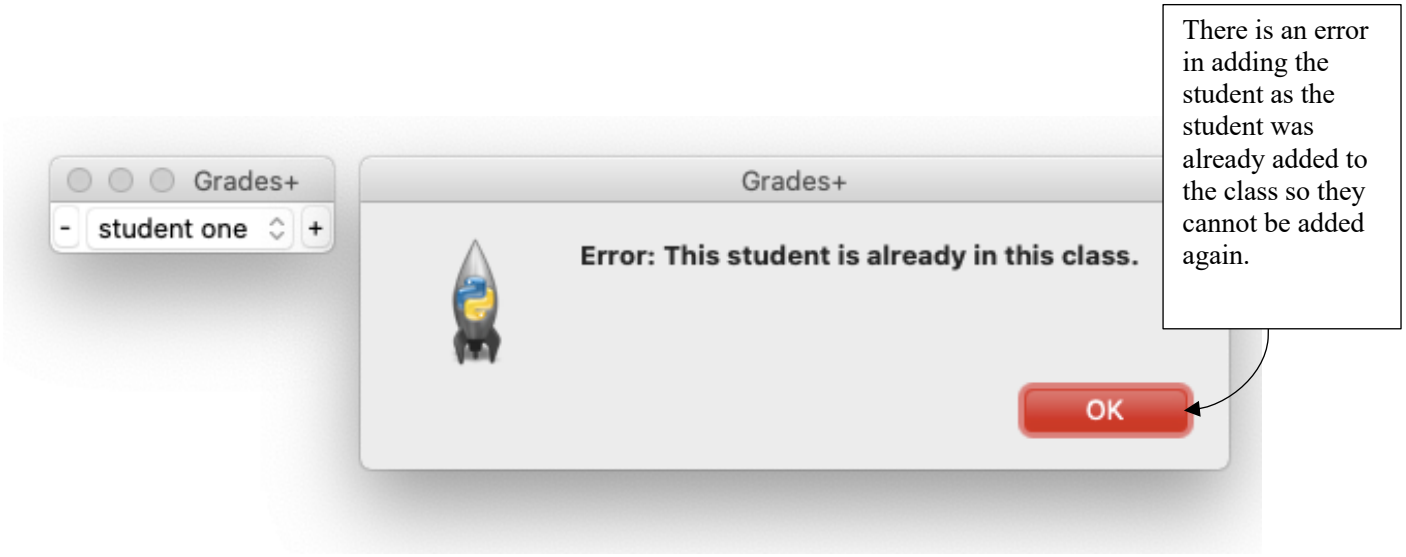
As the teacher has left the class, they are no longer in the database. Hence, when the teacher tries to leave the class after already leaving, there would be an errorbox.

Test Number	*Description of Test*	*Test Data*	*Expected Result*
8	Adding or removing a student from the class.	To test this, I would add a student “student one” to the Maths class I joined in Test 7.	I would expect a messagebox to show that the student has joined. If the student is already in the class, I would expect an errorbox.

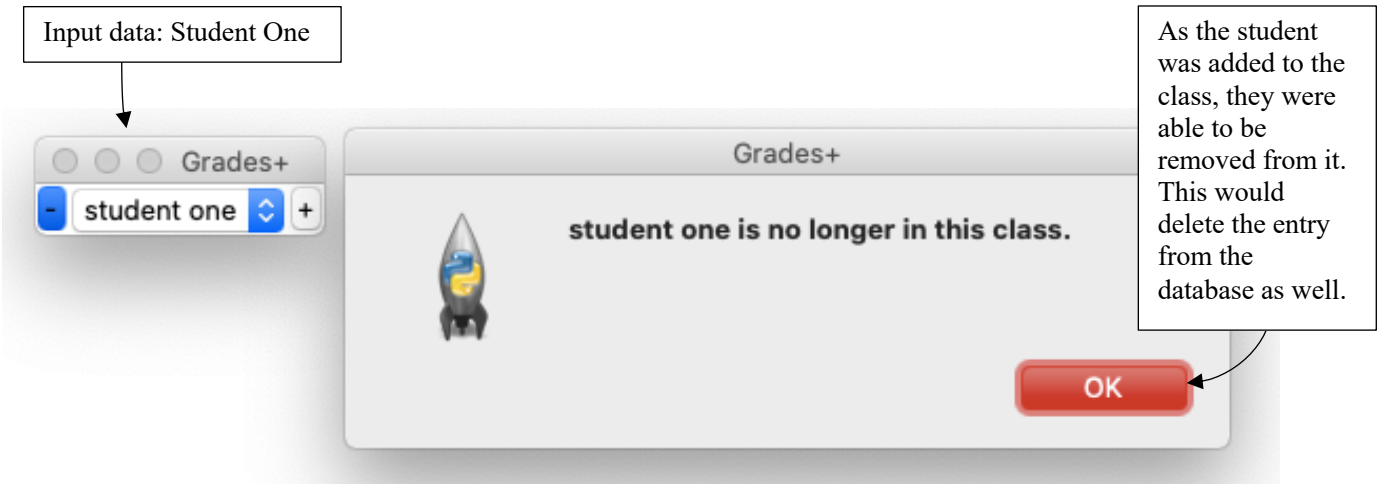
Adding a student to the class:



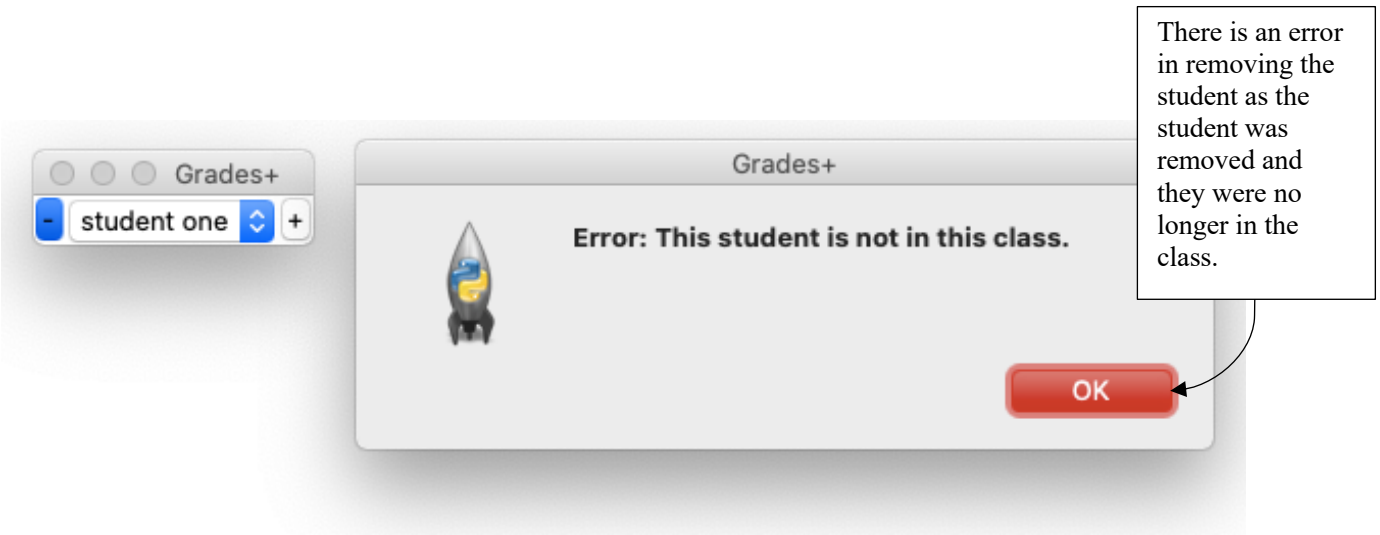
Add student after they have already been added to the class:



Removing a student from the class:

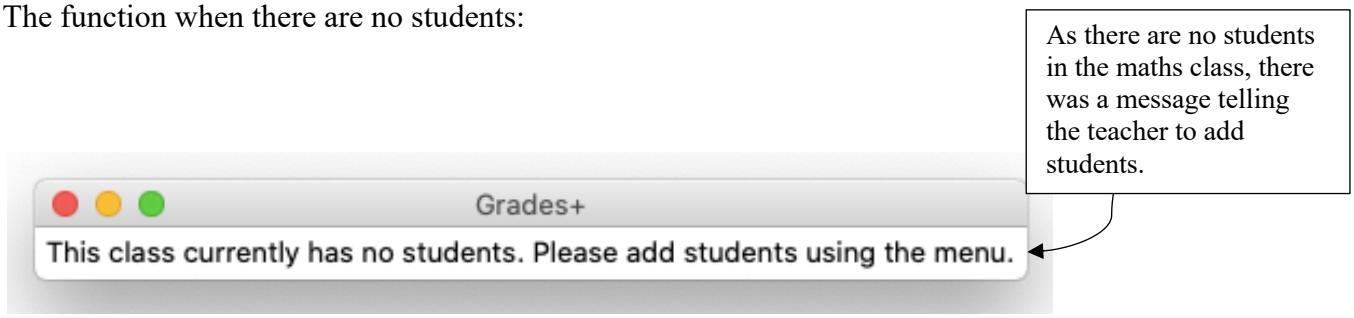


Remove student from the class when they are not in the class:

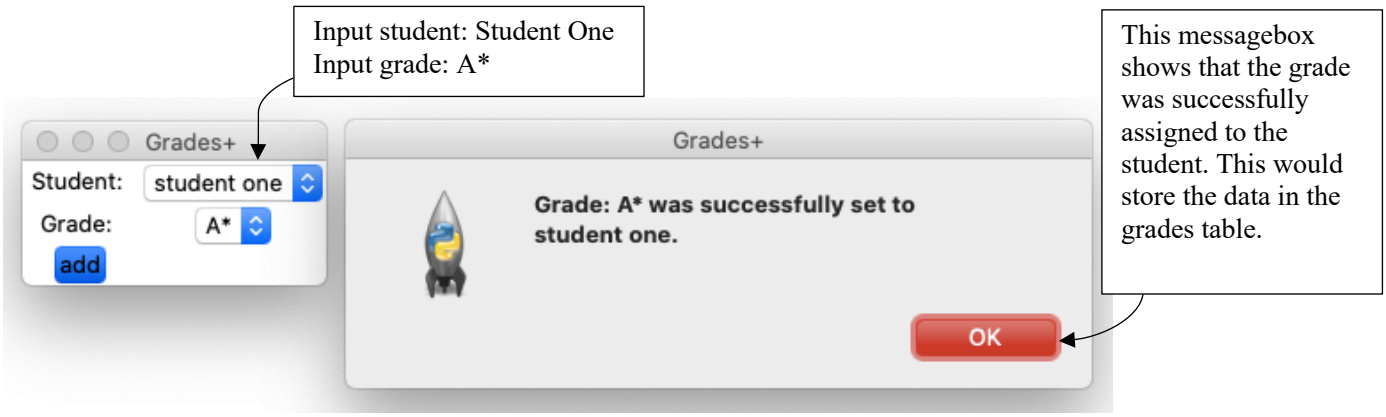


Test Number	*Description of Test*	*Test Data*	*Expected Result*
9	Assignment of grade to the student.	For this test, I would use "student one" and "student two" as I would assign an "A"/"A*".	If there are no students added to the class, I would expect the system to show a message saying that. However, if there are students, there should be a window where the teacher can select the student and grade and assign it. I would expect the grades to be on the Treeview.

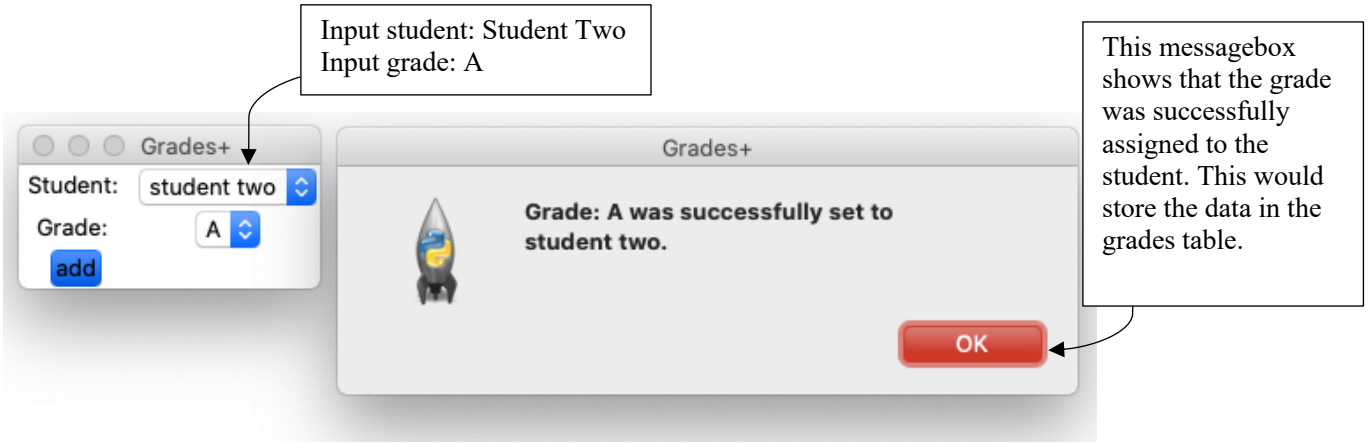
Actual results:
The function when there are no students:



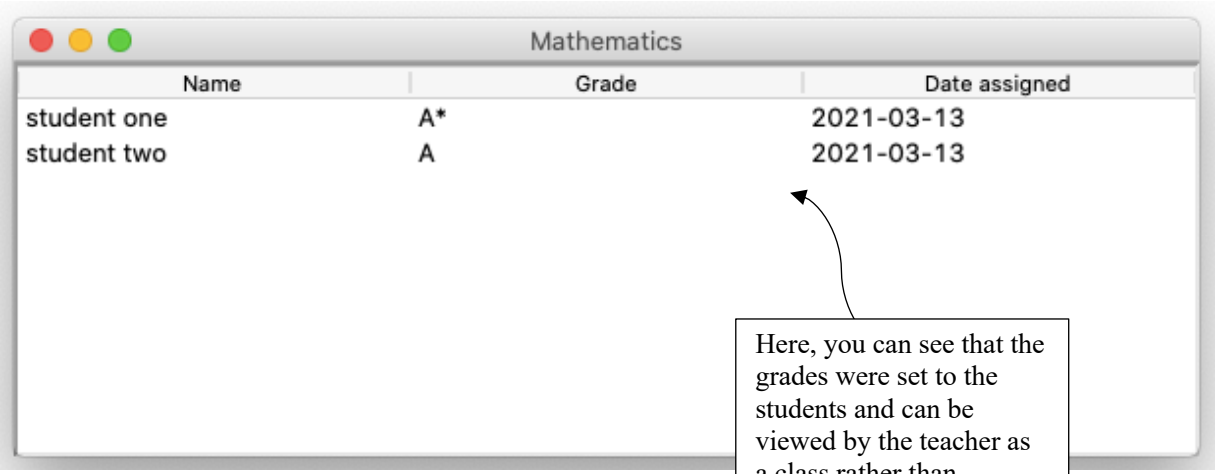
Set "A*" to Student One:



Set "A" to Student Two:



Treeview with the results:

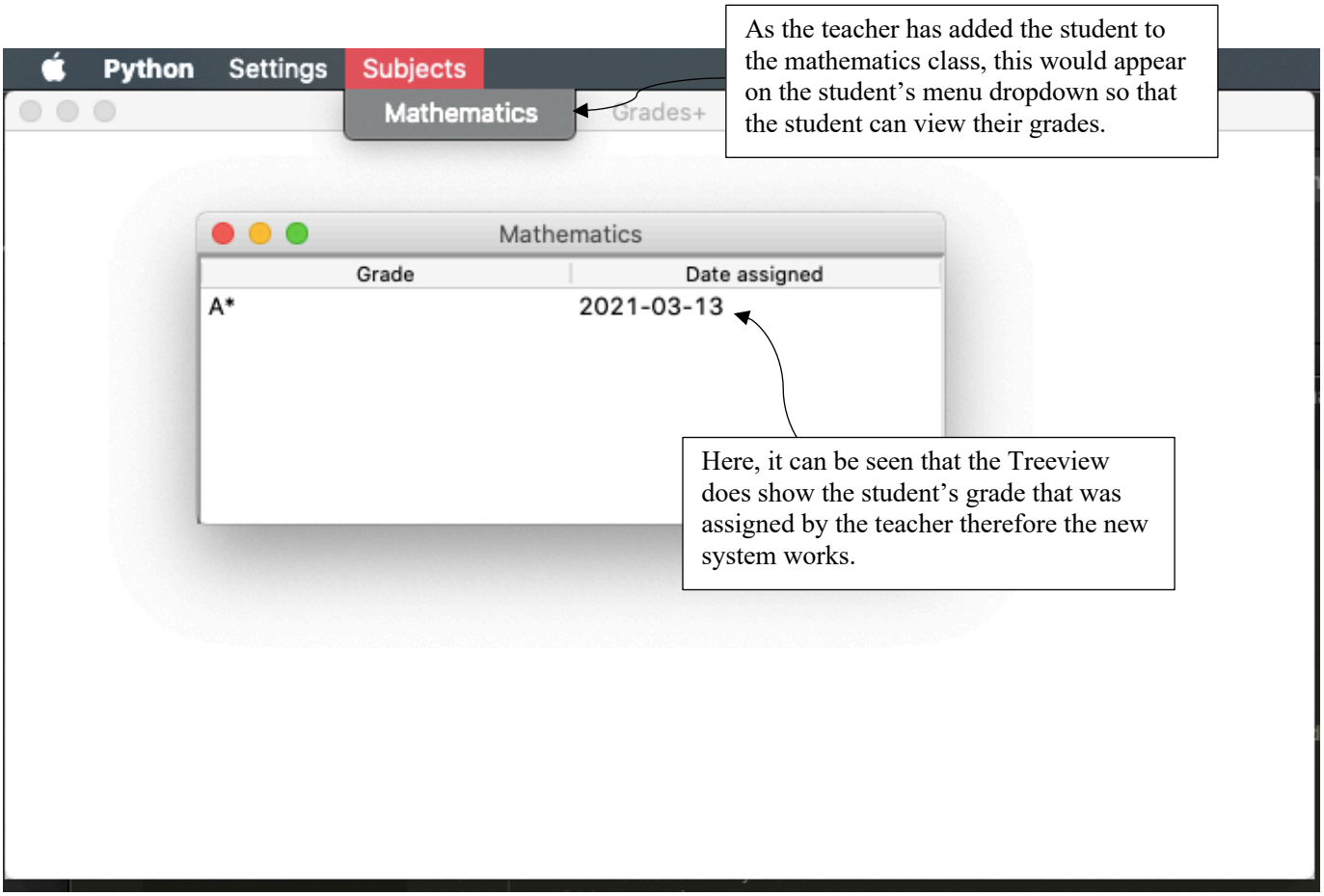


When the teacher assigns the grade to the student, this would be stored in the grades table in the database. As shown, the grades for both students are in this table so that the student can access it from their interface. There is a date on each record so that the student can track their grades over time and can see if they are improving or not for the a-levels they take.

grade_id	subject_id	student_id	grade	date
1	1	1	A*	2021-03-13
2	1	2	A	2021-03-13

STUDENT INTERFACE

Test Number	*Description of Test*	*Test Data*	*Expected Result*
10	Check if the student can view their grades.	I will use "student one" to check if they have been assigned the A* in Mathematics as tested in test 9.	I would expect a Treeview to appear with the grade "A* and the same date as shown in Test 9.



EVALUATION

EVALUATING THE SYSTEM'S OBJECTIVES

No	*Objective*	*Performance Criteria*	*Evaluation*
1	Program only allows authenticated users to register an account and login.	This system would need to have authenticated users using it so that the teacher can recognise their own students and assign them their grades.	**Completely Achieved** The new system would use a text-file to store the authenticated teachers and students. When the user is registering, the system would check if the user exists in it. If they do, the system will check if the code and role match before storing the data into the database. Tests 2 and 4 shows how the system would recognise if the user is authenticated.
1A	There will be an option to choose if you are a student or a teacher.	There needs to be two separate interfaces as the Teacher would have more functionalities than the Student so during authentication, the user should pick what they class as.	**Completely Achieved** Using Tkinter, I was able to create a dropdown so that the user can select their role. This was beneficial to the system as it allowed the data to be stored in the right table. For example, when a student registers an account, their data would be stored in the student's table.
1B	The students will have a code which they will be asked to enter as per to authentication.	The code would be used to check if the user is from the school. There would be other security measures to ensure that only AAA students and teachers can use the new system.	**Completely Achieved** A code would be given to AAA students so that they can register. This was requested in order to prevent unauthorised users from having an account and being able to assign grades to students.
2	Store student and teacher data in a database which would be easy to access for the program so that they can login without any issues.	The data should be securely stored in a database so that the system can compare the entry value with the value stored in the database during authentication.	**Completely Achieved** Using PostgreSQL has allowed me to store data in a relational database. This makes the new system much more efficient as it uses indexing to search for a row rather than having to use text-files to go through each line of code in order to find a specific data.

2A	The system should be able to generate queries from user tasks such as setting the grades for a student.	This objective is so that python can connect to the database and retrieve or store data into it such as the user's information and the grades assigned to students.	**Completely Achieved** In my technical solution, I have shown that the system connects to the database and creates queries so that it can receive data from the database and store data where appropriate. For example, when the user logins, there is a query to check if the password stored is the same as the entry.
2B	Password: make sure there are limitations for these such as min length, max length.	Having a range for the length of the password would make it so that there isn't a lot of memory being used by this.	**Completely Achieved** I have used a function to check if the password which the user enters fits the specific requirements. Test 1 shows how the isPasswordValid() function works which has been made so that I can set these limitations.
3	Hash the passwords so that they are securely stored.	Hashing the passwords are important so that hackers cannot encrypt the password which would endanger the user's security so using hashing allows the password to be securely stored.	**Completely Achieved** Passwords were stored using a hashing library known as argon. Test 5 shows how the hashing works in my system. This ensures that the system is secure so that hackers are unable to trace the original value of the hashed data.
4	The students should be able to view their subjects and the grades that were set. This will be done through a separate user interface.	This is the main functionality of the system as the students should be able to view every grade that the teacher has assigned them in order for them to track and see what subjects they need to focus improving on.	**Completely Achieved** The student interface has shown that the classes which the teacher adds the student to would appear on the menu dropdown. This would allow the student to see the grades assigned for that subject in a Treeview. Test 10 shows how the student is able to view their grades and how the database stores the grade.
5	The teachers should be able to view the students and set grades for students.	This would allow the teacher to assign the grades to a student for the subjects that they teach so that the student can view the grades from their own interface rather than having the teacher to physically give the student their exam papers with the grade.	**Completely Achieved** The new system allows the teacher to view the students grade through a Treeview like the student can. There is a function which creates a window which allows the teacher to set the grade to the student. Test 9 shows how the teacher was able to assign specific grades to the students after adding them to the classroom.

7	There will be an option for teachers to convert a percentage to a rough grade.	If the teacher does not follow a specific grade boundary, having a custom one would allow the teacher to give the student a rough idea of what their percentage would look like as a grade.	**Completely Achieved** The new system was able to have an option where the teacher was able to convert a percentage to a grade. Test 6 shows that entering an erroneous value would cause an error, however, if the value of the % is between 0 and 100, there would be a valid grade that appears. This is done through a custom grade boundary I made which is an average from different exam boards.
8	There will a dropdown for teachers to add the grade to make it easier.	Having a simple GUI allows teachers with limited knowledge about computers to use the new system with ease rather than manually typing the results in a spreadsheet.	**Completely Achieved** The new system uses an array which has the grades from U-A* and it creates a dropdown in a different window where the teacher can select a student and the corresponding grade which would then be stored into the database.

EVALUATING END-USER FEEDBACK

No	*Objective*	*Performance Criteria*	*Evaluation*
1	Program only allows authenticated users to register an account and login.	This system would need to have authenticated users using it so that the teacher can recognise their own students and assign them their grades.	"*The new system's authentication system is secure as tested by me and 2 volunteer teachers. They found the user interface quite interesting due to its uniqueness. Rather than having different GUI tools, I would prefer if this was done in the backend.*" How this outcome could be improved? Ensure that there is an algorithm which authorises the user from their name.
2	The teachers should be able to view the students and set grades for students.	This would allow the teacher to assign the grades to a student for the subjects that they teach so that the student can view the grades from their own interface rather than having the teacher to physically give the student their exam papers with the grade.	"*This feature was very simple to use which I had liked. It allowed me to view whoever was in the class and select the grade from a given range that's available.*"
3	The students should be able to view their subjects and the grades that were set. This will be done through a separate user interface.	This is the main functionality of the system as the students should be able to view every grade that the teacher has assigned them in order for them to track and see what subjects they need to focus improving on.	*"I have tested what the outcome would be once I had set the student a grade. Using John, I was able to see that they were able to view the grade as soon as it was set. This is good as it allows the student to get their grades much faster than they did in the old system. What would be better if the students were to be notified when there is an update"* How this outcome could be improved? Send emails to the user whenever there was an update to their grades.
4	There will be an option for teachers to convert a percentage to a rough grade.	If the teacher does not follow a specific grade boundary, having a custom one would allow the teacher to give the student a rough idea of what their percentage would look like as a grade.	*"I find this feature important as there tends to be custom exams which rely on a random grade boundary so having a feature like this would be very useful to old teachers and new teachers who tests their students quite regularly and want to check the grade that the student is achieving. This had met my expectations."*