Turkish Aeronautical Association University

Degree in Computer Engineering

Fall 2022-2023

Practical Work of Database Systems Course - CENG301

Work done by:

Cihan Balkır 190444053

Marya Aktaş 200444001

Mert Örnek 190444076

Contents

Introduction and Scenario

In the ‘Database Systems’ course, we are asked to implement a user-friendly interface that performs some operations on a database. This user-friendly application is essentially a database management system. Our implementation is based on a ‘Student Record System’.

In our implementation, there is a database called ‘Students’, and we have 4 tables in this database which are Student, Grades, Courses and Department. With the facilities of this database, in our application, the user can perform the following operations:

- Adding a new student record to the database,
- Showing the student records,
- Showing all grades of all students,
- Showing all courses in the database,
- Calculating the GPA of each student and seeing the ordered GPA’s (with this way, the user can easily see the student that has the highest GPA or the lowest GPA)
- Finding all courses that a student takes,
- Deleting a student record from the database,
- Remove the whole database

We used Python and MySQL for implementing this program. For better understanding the tables, we used MySQL Workbench. We used the modules:

- **Tkinter** for implementing the graphical user interface
- **MySQL connector** for connecting the databases

Tables

The tables we used in Students database are **Student**, **Grades**, **Courses** and **Department**. Department and Courses tables are static tables that only include unchangeable values. The table description of this database is:

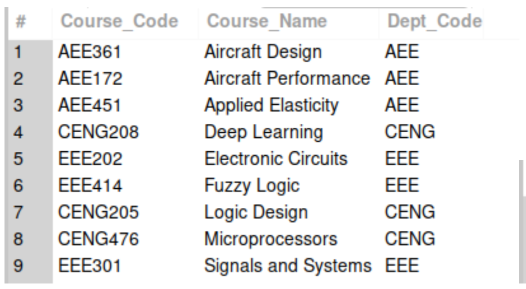
Student (<u>ID</u>, FName, LName)

Grades (st_id, Course_Code, grade, Dept_Code)

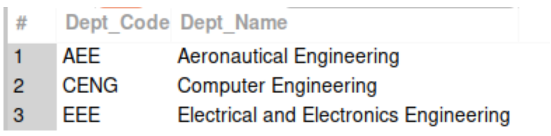
Courses (<u>Course_Code</u>, Course_Name, <u>Dept_Code</u>)

Department (<u>Dept_Code</u>, Dept_Name)

The **Department** table is shown in Figure 1 and the **Courses** table is shown in Figure 2. **Department** table has single-valued primary key which is the Dept_Code and **Courses** table has multi-valued primary key which the values are comes from **Course_Code** and **Dept_Code** attributes.



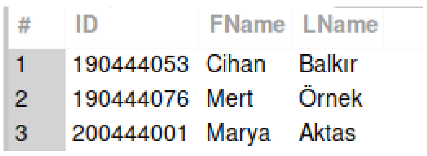
#	Course_Code	Course_Name	Dept_Code
1	AEE361	Aircraft Design	AEE
2	AEE172	Aircraft Performance	AEE
3	AEE451	Applied Elasticity	AEE
4	CENG208	Deep Learning	CENG
5	EEE202	Electronic Circuits	EEE
6	EEE414	Fuzzy Logic	EEE
7	CENG205	Logic Design	CENG
8	CENG476	Microprocessors	CENG
9	EEE301	Signals and Systems	EEE



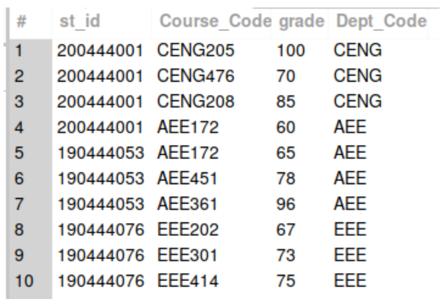
#	Dept_Code	Dept_Name
1	AEE	Aeronautical Engineering
2	CENG	Computer Engineering
3	EEE	Electrical and Electronics Engineering

(Figure 1) (Figure 2)

The **Student** table has a single-valued primary key which is the **ID** attribute. An example of this table is shown below in Figure 3. The **Grades** table has no primary keys. This table has 3 foreign keys which are **st_id**, **Course_Code** and **Dept_Code.** The details will be presented later in this report. A sample of rows of **Grades** table is shown in Figure 4.



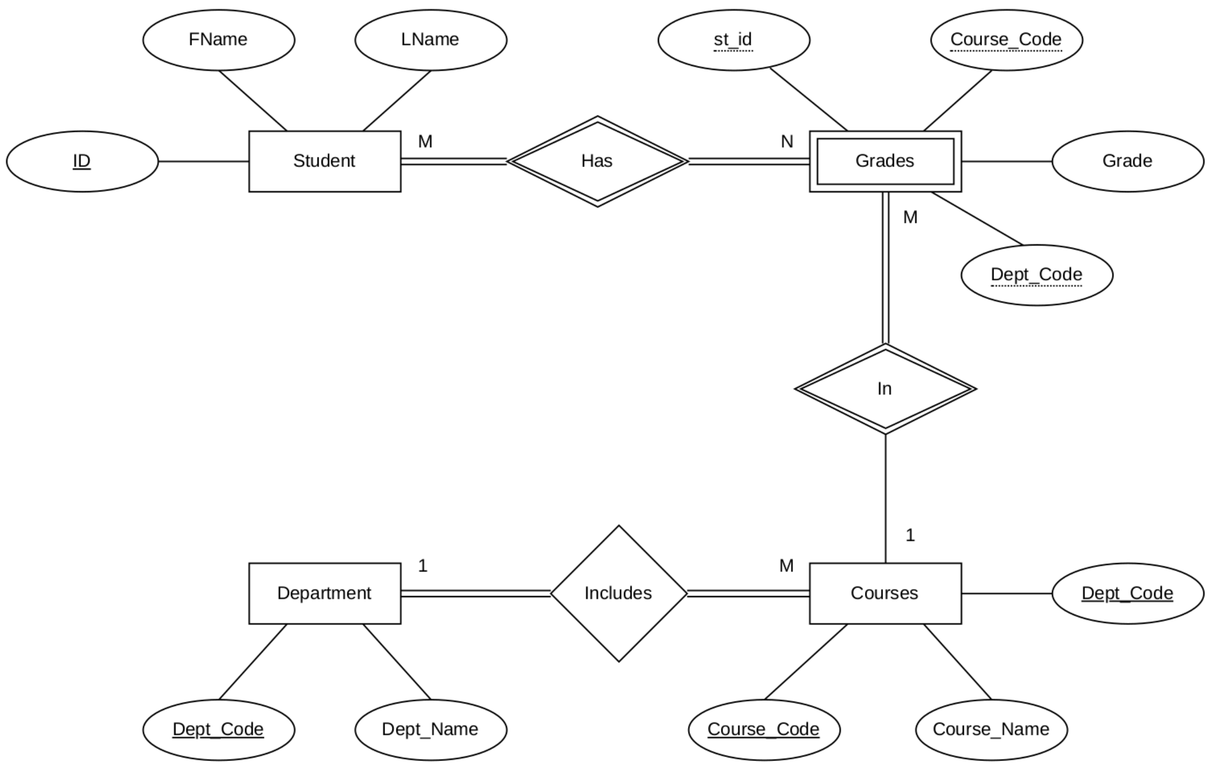
#	ID	FName	LName
1	190444053	Cihan	Balkır
2	190444076	Mert	Örnek
3	200444001	Marya	Aktas



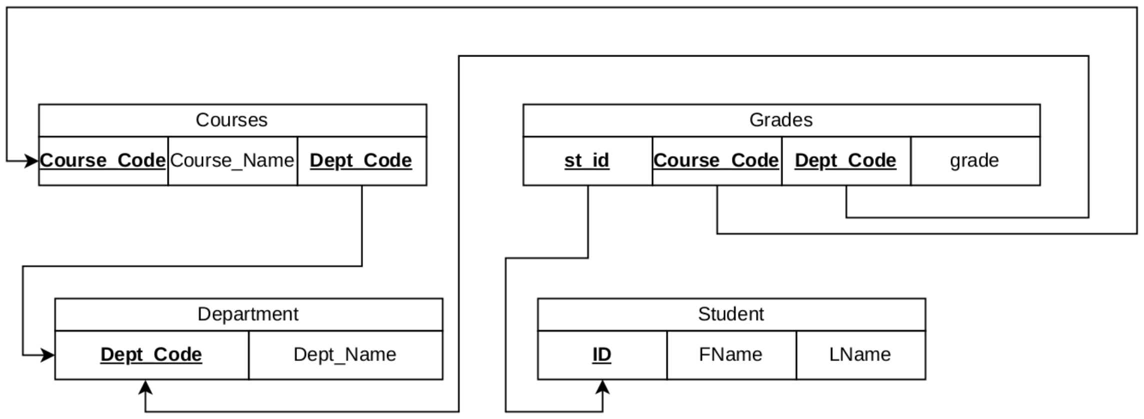
#	st_id	Course_Code	grade	Dept_Code
1	200444001	CENG205	100	CENG
2	200444001	CENG476	70	CENG
3	200444001	CENG208	85	CENG
4	200444001	AEE172	60	AEE
5	190444053	AEE172	65	AEE
6	190444053	AEE451	78	AEE
7	190444053	AEE361	96	AEE
8	190444076	EEE202	67	EEE
9	190444076	EEE301	73	EEE
10	190444076	EEE414	75	EEE

(Figure 3) (Figure 4)

ER Diagram and Mapping



(Figure 5)



(Figure 6)

The ER diagram of our database is shown above in Figure 5. And the mapping of these relations is shown in Figure 6.

According to the ER diagram and the mapping;

Student is a table of the system and it;

- Has *FName* and *LName* as **attributes**,
- Has *ID* and as **primary keys,**

❖ All students must choose at least one grade. (**Total participation**)
❖ All students must have at least one grade based on his/her course. (**Total participation**)

Grades is another table of the system and it;

- Has *Grade* as **attribute**,
- Has *st_id* as **foreign key** that references ID in **Student** table.
- Has *(Course_Code, Dept_Code)* as **foreign key** that references to *(Course_Code, Dept_Code)* in **Courses** table.

❖ All grades must be taken by at least one student. (**Total participation**)
❖ All grades must be related to exactly one course. (**Total participation**)

Department is another table of the system and it;

- Has *Dept_Code* as **primary key**,
- Has *Dept_Name* as **UNIQUE attribute**.

❖ Each department needs to have at least one course. (**Total participation**)
❖ Each course belongs to exactly one department. (**Total participation**)

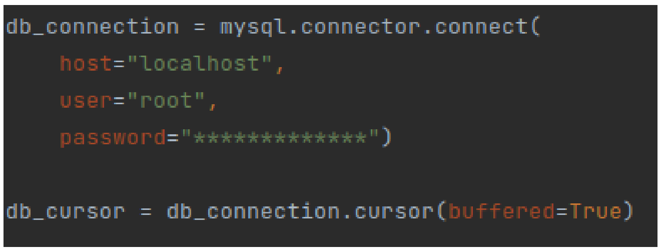
Courses is the last table of the system and it;

- Has (*Course_Code, Dept_Code)* as **primary key**,
- Has *Dept_Code* as **foreign key** that references to *Dept_Code* in **Department** table,
- Has *Course_Name* as **UNIQUE attribute**.

❖ All courses must be related to exactly one department. (**Total participation**)
❖ There is no necessity for every course must be at least one student enrolled. (**Partial participation**)

Main Python Code

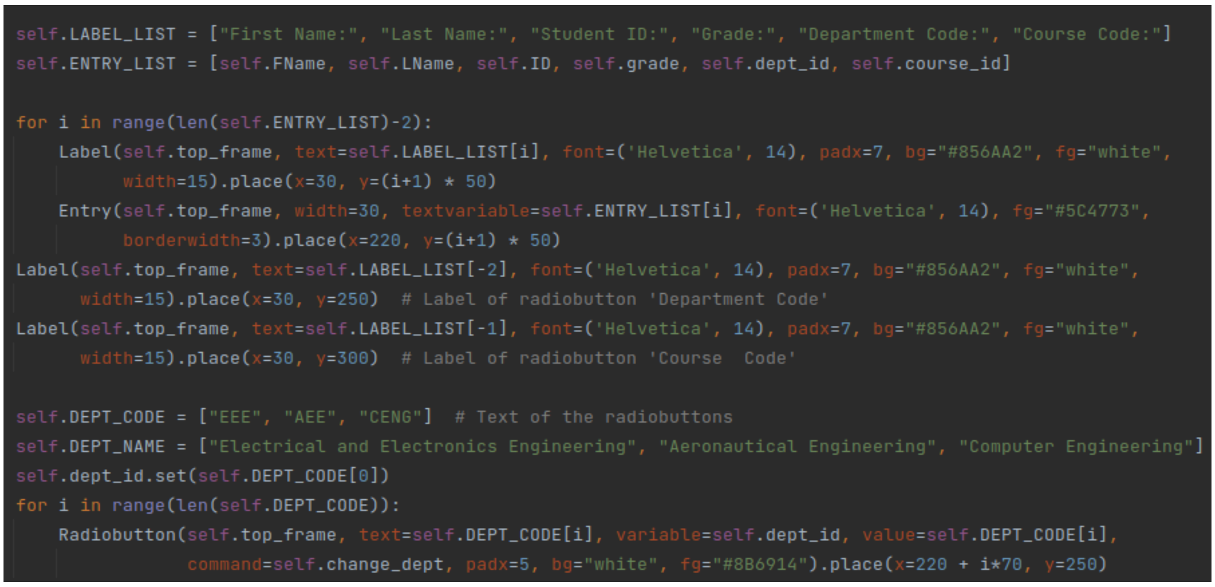
We first connected to the database as shown in Figure 7. After that, we created a class called **StudentManage** that inherits **Tk** for preparing the GUI environment. Using object-oriented programming in this project leads the methods to communicate easier with each other. Using **cursor()** method returns a MySQLCursor() class object which is essentially a pointer to the database. Setting ‘True’ to **buffered** option of the cursor makes the cursor fetches all rows from the server after an operation is executed.



```
db_connection = mysql.connector.connect(
    host="localhost",
    user="root",
    password="*************")

db_cursor = db_connection.cursor(buffered=True)
```

(Figure 7)

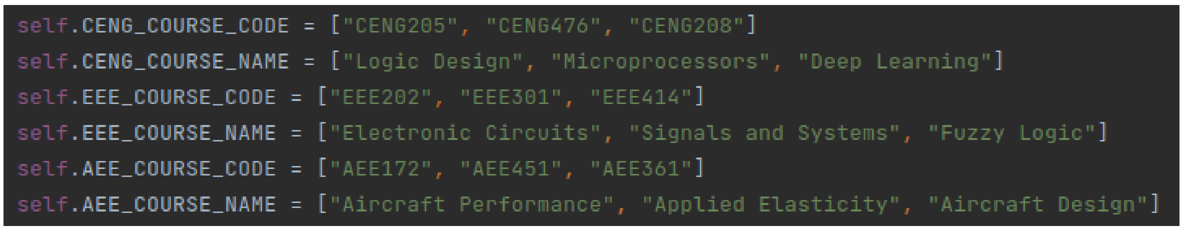
We first created two main frames that are **top_frame** and **bottom_frame**. **top_frame** is for storing the labels, the entries and radio buttons for adding a new student record in the database and the buttons that the user can choose to perform some operations in the **__init__** function. For creating labels, entries and buttons, we used loops to simplify the code. We created some empty string variables that can be used to retrieve the entries and gathered them in a list called **self.ENTRY_LIST** and we also created a list called **self.LABEL_LIST** for creating the labels and the entries in the for loop by retrieving them by their index. This technique is very useful because when we wanted to create another entry and label, we could simply just modify these lists and for loop will manage the rest. This technique provides maintenance. Creating the entries, labels and the radio buttons are shown in Figure 8. We created radio buttons instead of entries because there are static departments and courses. That means, the user can choose one of the departments and then choose one course that belongs to this department. We did not include the last two labels in the **self.LABEL_LIST** is not included in the for loop. Because if we did, the program also would have put entries next to labels of course code and department code but we wanted radio buttons instead.



```
self.LABEL_LIST = ["First Name:", "Last Name:", "Student ID:", "Grade:", "Department Code:", "Course Code:"]
self.ENTRY_LIST = [self.FName, self.LName, self.ID, self.grade, self.dept_id, self.course_id]

for i in range(len(self.ENTRY_LIST)-2):
    Label(self.top_frame, text=self.LABEL_LIST[i], font=('Helvetica', 14), padx=7, bg="#856AA2", fg="white",
          width=15).place(x=30, y=(i+1) * 50)
    Entry(self.top_frame, width=30, textvariable=self.ENTRY_LIST[i], font=('Helvetica', 14), fg="#5C4773",
          borderwidth=3).place(x=220, y=(i+1) * 50)
Label(self.top_frame, text=self.LABEL_LIST[-2], font=('Helvetica', 14), padx=7, bg="#856AA2", fg="white",
      width=15).place(x=30, y=250)  # Label of radiobutton 'Department Code'
Label(self.top_frame, text=self.LABEL_LIST[-1], font=('Helvetica', 14), padx=7, bg="#856AA2", fg="white",
      width=15).place(x=30, y=300)  # Label of radiobutton 'Course  Code'

self.DEPT_CODE = ["EEE", "AEE", "CENG"]  # Text of the radiobuttons
self.DEPT_NAME = ["Electrical and Electronics Engineering", "Aeronautical Engineering", "Computer Engineering"]
self.dept_id.set(self.DEPT_CODE[0])
for i in range(len(self.DEPT_CODE)):
    Radiobutton(self.top_frame, text=self.DEPT_CODE[i], variable=self.dept_id, value=self.DEPT_CODE[i],
                command=self.change_dept, padx=5, bg="white", fg="#8B6914").place(x=220 + i*70, y=250)
```

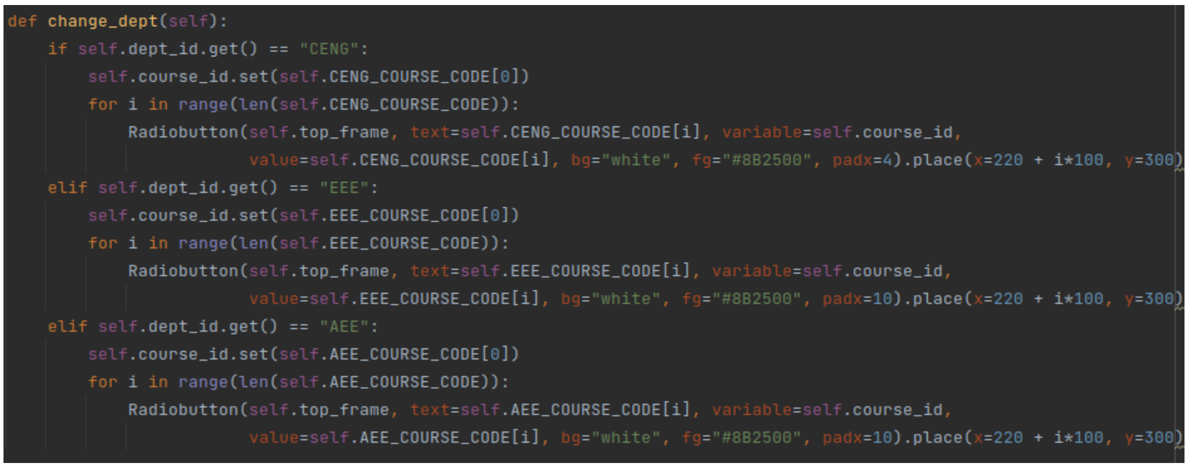
(Figure 8)

The **self.change_dept** method is for displaying the courses of the selected department as shown in Figure 10. We again used the technique of creating the radio buttons for courses. In Figure 9, we can see the lists of courses with their names. They will be useful for also inserting the values of the **Courses** table in **create_tables** method. Using radio buttons that are created depending on the department ensures the department of the selected course is true.



```
self.CENG_COURSE_CODE = ["CENG205", "CENG476", "CENG208"]
self.CENG_COURSE_NAME = ["Logic Design", "Microprocessors", "Deep Learning"]
self.EEE_COURSE_CODE = ["EEE202", "EEE301", "EEE414"]
self.EEE_COURSE_NAME = ["Electronic Circuits", "Signals and Systems", "Fuzzy Logic"]
self.AEE_COURSE_CODE = ["AEE172", "AEE451", "AEE361"]
self.AEE_COURSE_NAME = ["Aircraft Performance", "Applied Elasticity", "Aircraft Design"]
```

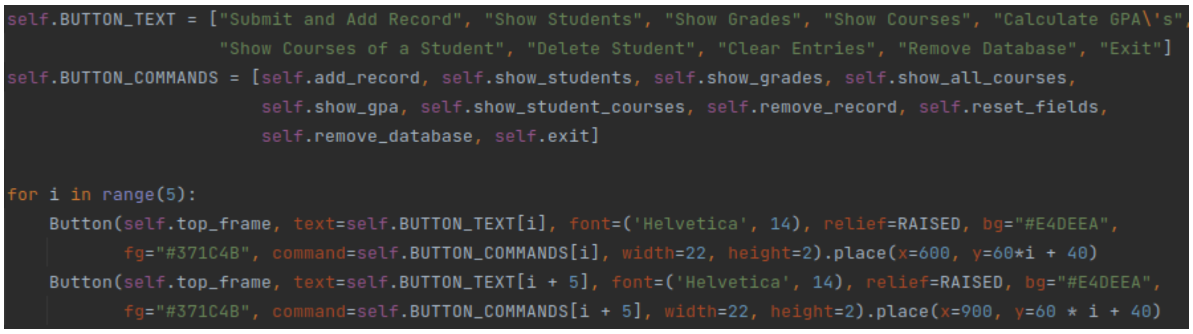
(Figure 9)



```
def change_dept(self):
    if self.dept_id.get() == "CENG":
        self.course_id.set(self.CENG_COURSE_CODE[0])
        for i in range(len(self.CENG_COURSE_CODE)):
            Radiobutton(self.top_frame, text=self.CENG_COURSE_CODE[i], variable=self.course_id,
                        value=self.CENG_COURSE_CODE[i], bg="white", fg="#8B2500", padx=4).place(x=220 + i*100, y=300)
    elif self.dept_id.get() == "EEE":
        self.course_id.set(self.EEE_COURSE_CODE[0])
        for i in range(len(self.EEE_COURSE_CODE)):
            Radiobutton(self.top_frame, text=self.EEE_COURSE_CODE[i], variable=self.course_id,
                        value=self.EEE_COURSE_CODE[i], bg="white", fg="#8B2500", padx=10).place(x=220 + i*100, y=300)
    elif self.dept_id.get() == "AEE":
        self.course_id.set(self.AEE_COURSE_CODE[0])
        for i in range(len(self.AEE_COURSE_CODE)):
            Radiobutton(self.top_frame, text=self.AEE_COURSE_CODE[i], variable=self.course_id,
                        value=self.AEE_COURSE_CODE[i], bg="white", fg="#8B2500", padx=10).place(x=220 + i*100, y=300)
```

(Figure 10)

For placing the buttons, we again followed the same steps as creating the labels and entries as shown in Figure 11. We created a list called **self.BUTTON_TEXT** that has the texts of the buttons and a list called **self.BUTTON_COMMANDS** that has the methods that will be assigned to ‘command’ option of Button() function.



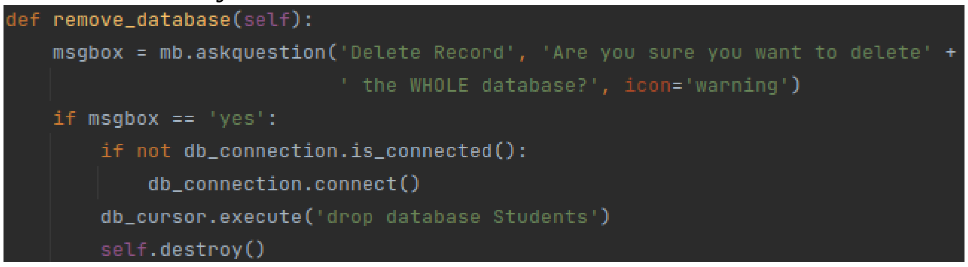
```
self.BUTTON_TEXT = ["Submit and Add Record", "Show Students", "Show Grades", "Show Courses", "Calculate GPA\'s",
                    "Show Courses of a Student", "Delete Student", "Clear Entries", "Remove Database", "Exit"]
self.BUTTON_COMMANDS = [self.add_record, self.show_students, self.show_grades, self.show_all_courses,
                        self.show_gpa, self.show_student_courses, self.remove_record, self.reset_fields,
                        self.remove_database, self.exit]

for i in range(5):
    Button(self.top_frame, text=self.BUTTON_TEXT[i], font=('Helvetica', 14), relief=RAISED, bg="#E4DEEA",
           fg="#371C4B", command=self.BUTTON_COMMANDS[i], width=22, height=2).place(x=600, y=60*i + 40)
    Button(self.top_frame, text=self.BUTTON_TEXT[i + 5], font=('Helvetica', 14), relief=RAISED, bg="#E4DEEA",
           fg="#371C4B", command=self.BUTTON_COMMANDS[i + 5], width=22, height=2).place(x=900, y=60 * i + 40)
```

(Figure 11)

The methods of the **StudentManage** class are **change_dept, reset_fields, exit, remove_database, create_tables, add_records, show_grades, show_students, show_gpa, show_student_courses, show_all_courses, remove_record, place_records, place_grades, place_gpa** and **place_courses.**

The **reset_fields** method simply resets all entries in the **self.top_frame** by setting zero-length string to text variable of all entries. The **exit** method destroys the window and terminates the code after asking the user whether she is sure or not.

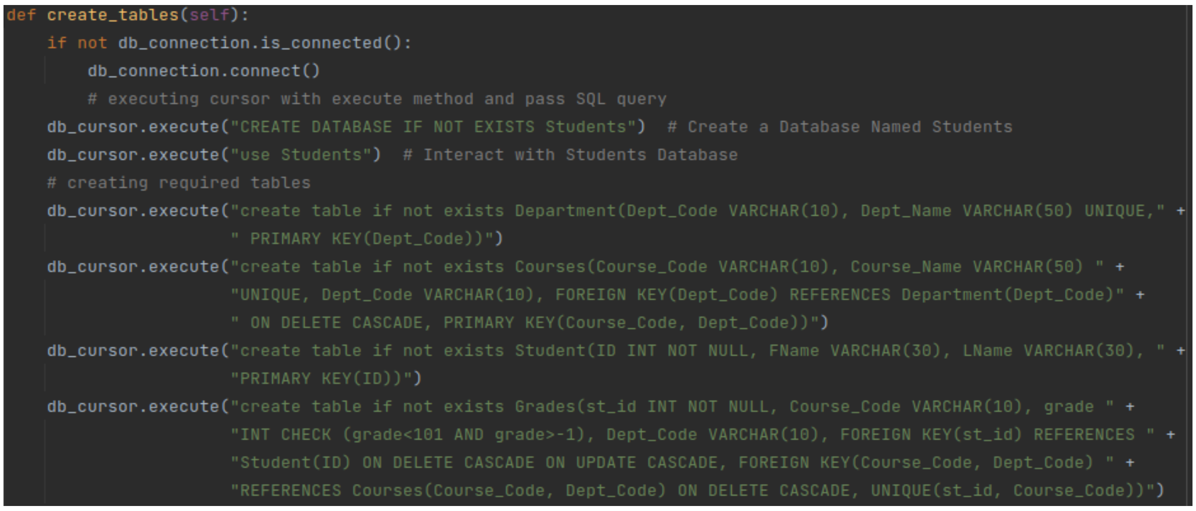
The **remove_database** method deletes the **Student** database from the databases by executing ‘drop database’ command as shown in Figure 12. After that, the window will be destroyed, and the code will be terminated.



```
def remove_database(self):
    msgbox = mb.askquestion('Delete Record', 'Are you sure you want to delete' +
                            ' the WHOLE database?', icon='warning')
    if msgbox == 'yes':
        if not db_connection.is_connected():
            db_connection.connect()
        db_cursor.execute('drop database Students')
        self.destroy()
```

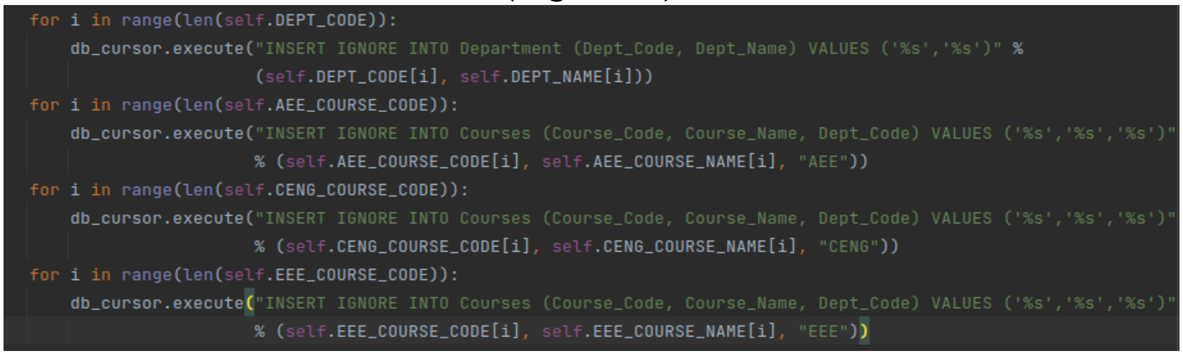
(Figure 12)

The **create_tables** method creates the tables and their relations if the tables don’t exist as shown in Figure 13. The method also uses the lists in Figure 9 to insert the data into the Courses and Department tables as shown in Figure 14. We used INSERT IGNORE INTO instead of INSERT INTO for inserting the data, because since this **create_table** method will be executed in **__init__** function, when we want to execute the code two or more times without removing database, the program raises an error caused by inserting a duplicate of primary keys. With INSERT IGNORE INTO command, a row won’t be inserted if it results in an error, but the statement also won’t generate an error. We used **commit()** function to commit permanently all these changes to the database. The foreign keys use CASCADE on deletion or updating. They will do the same as their references.



```python
def create_tables(self):
    if not db_connection.is_connected():
        db_connection.connect()
        # executing cursor with execute method and pass SQL query
    db_cursor.execute("CREATE DATABASE IF NOT EXISTS Students")  # Create a Database Named Students
    db_cursor.execute("use Students")  # Interact with Students Database
    # creating required tables
    db_cursor.execute("create table if not exists Department(Dept_Code VARCHAR(10), Dept_Name VARCHAR(50) UNIQUE," +
                      " PRIMARY KEY(Dept_Code))")
    db_cursor.execute("create table if not exists Courses(Course_Code VARCHAR(10), Course_Name VARCHAR(50) " +
                      "UNIQUE, Dept_Code VARCHAR(10), FOREIGN KEY(Dept_Code) REFERENCES Department(Dept_Code)" +
                      " ON DELETE CASCADE, PRIMARY KEY(Course_Code, Dept_Code))")
    db_cursor.execute("create table if not exists Student(ID INT NOT NULL, FName VARCHAR(30), LName VARCHAR(30), " +
                      "PRIMARY KEY(ID))")
    db_cursor.execute("create table if not exists Grades(st_id INT NOT NULL, Course_Code VARCHAR(10), grade " +
                      "INT CHECK (grade<101 AND grade>-1), Dept_Code VARCHAR(10), FOREIGN KEY(st_id) REFERENCES " +
                      "Student(ID) ON DELETE CASCADE ON UPDATE CASCADE, FOREIGN KEY(Course_Code, Dept_Code) " +
                      "REFERENCES Courses(Course_Code, Dept_Code) ON DELETE CASCADE, UNIQUE(st_id, Course_Code))")
```

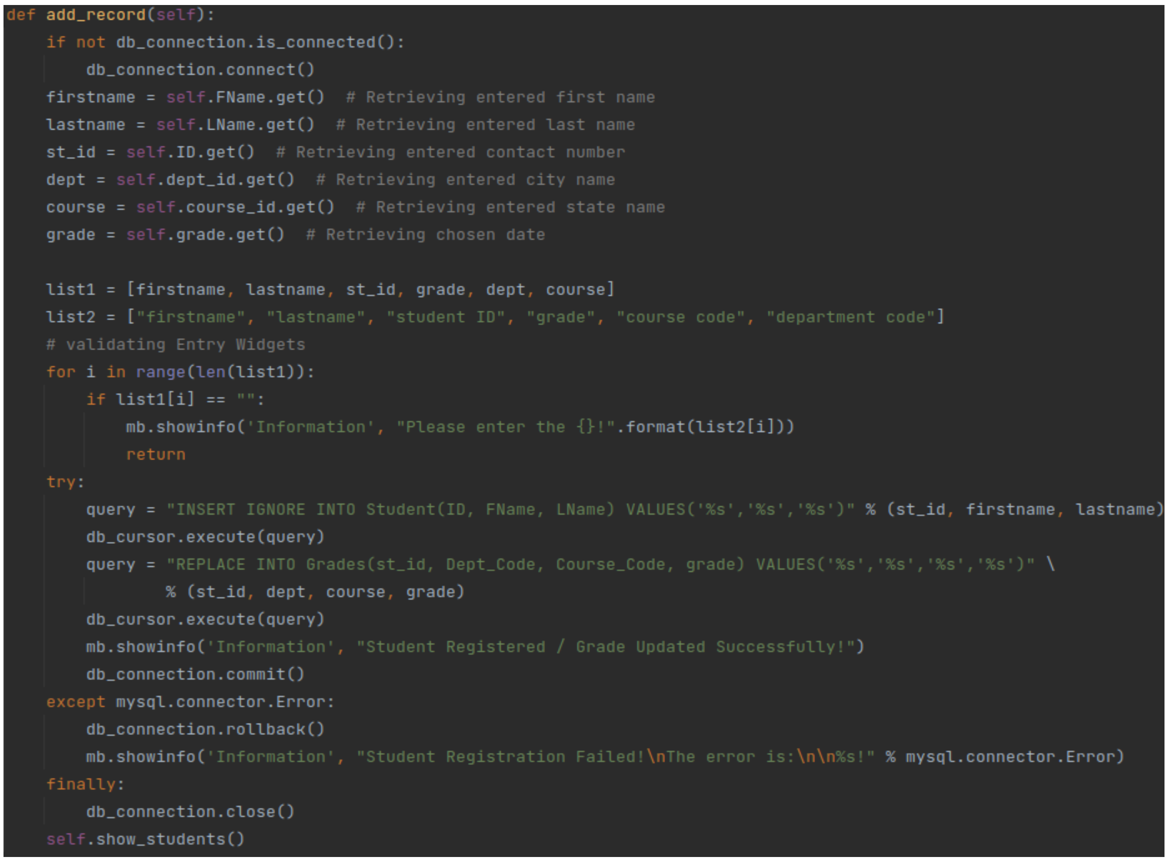
(Figure 13)



```python
for i in range(len(self.DEPT_CODE)):
    db_cursor.execute("INSERT IGNORE INTO Department (Dept_Code, Dept_Name) VALUES ('%s','%s')" %
                      (self.DEPT_CODE[i], self.DEPT_NAME[i]))
for i in range(len(self.AEE_COURSE_CODE)):
    db_cursor.execute("INSERT IGNORE INTO Courses (Course_Code, Course_Name, Dept_Code) VALUES ('%s','%s','%s')"
                      % (self.AEE_COURSE_CODE[i], self.AEE_COURSE_NAME[i], "AEE"))
for i in range(len(self.CENG_COURSE_CODE)):
    db_cursor.execute("INSERT IGNORE INTO Courses (Course_Code, Course_Name, Dept_Code) VALUES ('%s','%s','%s')"
                      % (self.CENG_COURSE_CODE[i], self.CENG_COURSE_NAME[i], "CENG"))
for i in range(len(self.EEE_COURSE_CODE)):
    db_cursor.execute("INSERT IGNORE INTO Courses (Course_Code, Course_Name, Dept_Code) VALUES ('%s','%s','%s')"
                      % (self.EEE_COURSE_CODE[i], self.EEE_COURSE_NAME[i], "EEE"))
```

(Figure 14)

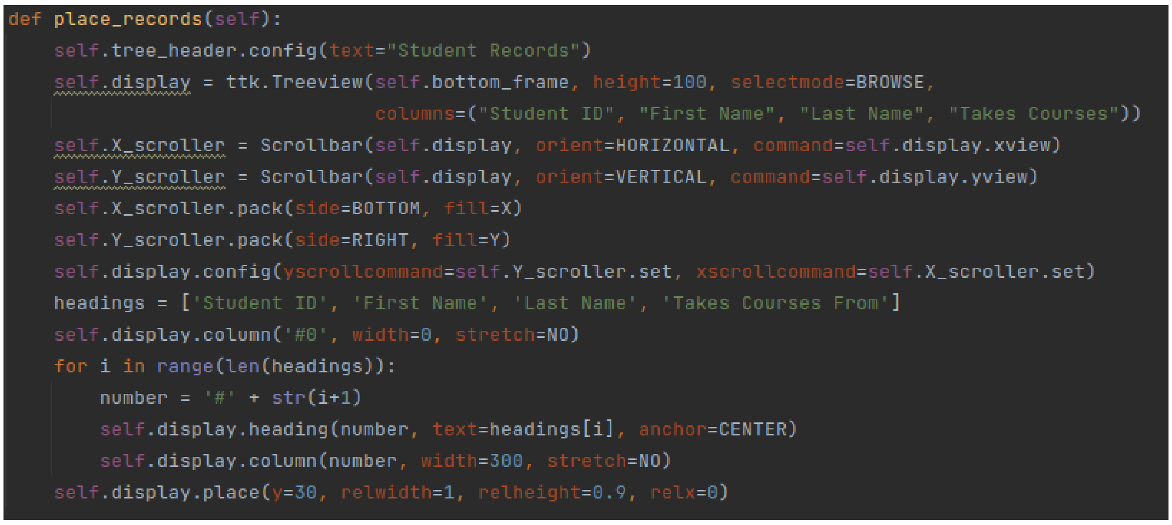
The **add_record** method allows the user to add new student data into the database as shown in Figure 15. First, if the database connection is not provided, the program makes sure that it is connected. Then, the program retrieves the entries of the user, then gathers them in a list to make the code simpler. If the user does not enter all entries, there will be a new pop-up window that warns the user. After the user enters all the entries, a new student will be added to the database. REPLACE allows us to insert a row if it doesn’t exist and update the row according to the primary keys if it already exists. We used REPLACE in inserting the data to Grades table, because the user also can update the grade. If everything goes perfectly the code commits the changes, if not**, rollback()** function takes back the changes on the database. We used IGNORE while inserting values to Student. It is because, since ID and Dept_Code are primary keys, we should ignore the errors and not insert the row in this case. For other errors such as ID is not an integer, the second query will generate an error for them while the first query doesn’t. This ensures there is no problem with other errors. After making these changes, the code commits the changes. If there is an error raised, then the code takes back the changes. After all these operations, the program will display all student records with **show_students**.



```
def add_record(self):
    if not db_connection.is_connected():
        db_connection.connect()
    firstname = self.FName.get()  # Retrieving entered first name
    lastname = self.LName.get()  # Retrieving entered last name
    st_id = self.ID.get()  # Retrieving entered contact number
    dept = self.dept_id.get()  # Retrieving entered city name
    course = self.course_id.get()  # Retrieving entered state name
    grade = self.grade.get()  # Retrieving chosen date

    list1 = [firstname, lastname, st_id, grade, dept, course]
    list2 = ["firstname", "lastname", "student ID", "grade", "course code", "department code"]
    # validating Entry Widgets
    for i in range(len(list1)):
        if list1[i] == "":
            mb.showinfo('Information', "Please enter the {}!".format(list2[i]))
            return
    try:
        query = "INSERT IGNORE INTO Student(ID, FName, LName) VALUES('%s','%s','%s')" % (st_id, firstname, lastname)
        db_cursor.execute(query)
        query = "REPLACE INTO Grades(st_id, Dept_Code, Course_Code, grade) VALUES('%s','%s','%s','%s')" \
                % (st_id, dept, course, grade)
        db_cursor.execute(query)
        mb.showinfo('Information', "Student Registered / Grade Updated Successfully!")
        db_connection.commit()
    except mysql.connector.Error:
        db_connection.rollback()
        mb.showinfo('Information', "Student Registration Failed!\nThe error is:\n\n%s!" % mysql.connector.Error)
    finally:
        db_connection.close()
    self.show_students()
```

(Figure 15)

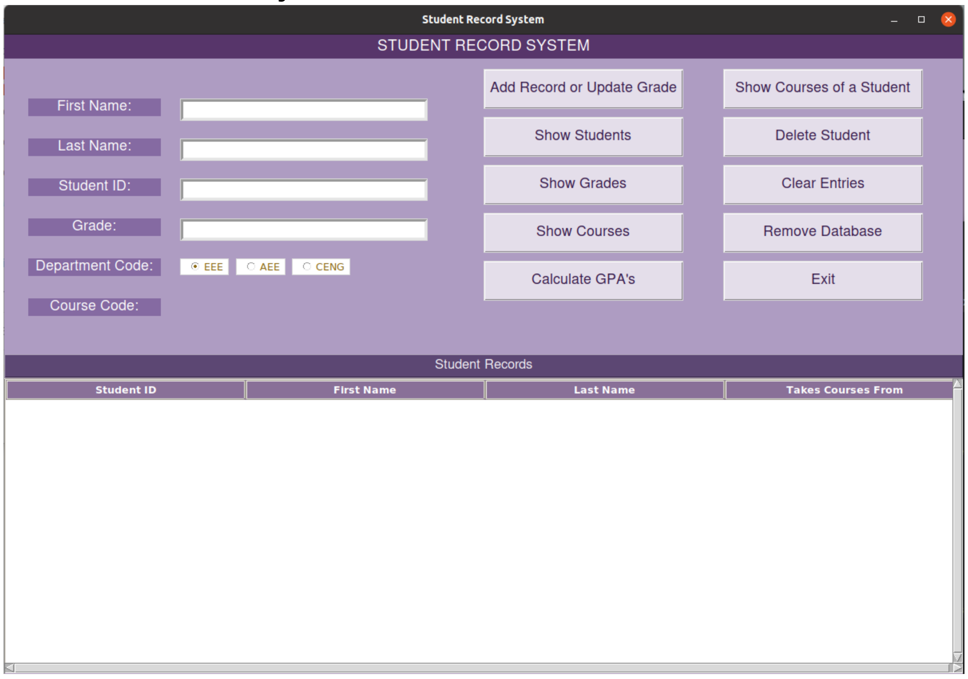
For showing the attributes, we used Treeview widget for creating a tabular structure. For arranging the features of this treeview for displaying the **Student** table we created a method called **place_records** as shown in Figure 16. **self.tree_header** is the label that shows the title of the information which is placed in **__init__** function. We also created three other placing methods which are **place_grades, place_gpa, place_courses**. The only differences between these methods are the title, column numbers, column names, and column widths. The **place_records** method shows the columns: Student ID, First Name, Last Name, Takes Courses From. The column ‘Takes Courses From’ indicates the department code that the student takes courses from.



```python
def place_records(self):
    self.tree_header.config(text="Student Records")
    self.display = ttk.Treeview(self.bottom_frame, height=100, selectmode=BROWSE,
                                columns=("Student ID", "First Name", "Last Name", "Takes Courses"))
    self.X_scroller = Scrollbar(self.display, orient=HORIZONTAL, command=self.display.xview)
    self.Y_scroller = Scrollbar(self.display, orient=VERTICAL, command=self.display.yview)
    self.X_scroller.pack(side=BOTTOM, fill=X)
    self.Y_scroller.pack(side=RIGHT, fill=Y)
    self.display.config(yscrollcommand=self.Y_scroller.set, xscrollcommand=self.X_scroller.set)
    headings = ['Student ID', 'First Name', 'Last Name', 'Takes Courses From']
    self.display.column('#0', width=0, stretch=NO)
    for i in range(len(headings)):
        number = '#' + str(i+1)
        self.display.heading(number, text=headings[i], anchor=CENTER)
        self.display.column(number, width=300, stretch=NO)
    self.display.place(y=30, relwidth=1, relheight=0.9, relx=0)
```

(Figure 16)

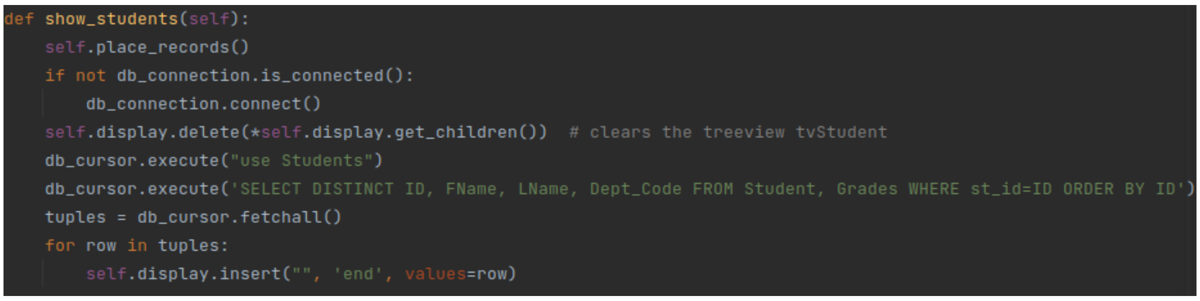
After placing every Tk component and widget, the interface that shows the student records without any data looks like:



(Figure 17)

After creating and placing the Treeview's according to which tabular data we want to show on the screen, we created 5 methods for retrieving the data from the tables using MySQL queries, and we placed the data on these Treeview's. For showing the student records on the screen, the **show_students** method is shown in Figure 18. The method places the Treeview (which is denoted by **self.display** in our code) by calling **place_records** method. And if the database connection is not provided, then the program connects to the database. After that, the program removes all column entries in the Treeview. Then, it retrieves the rows 'ID', 'FName', 'LName', 'Dept_Code' from the **Student** and **Grades** tables, the rows are ordered by the students' ID in ascending order. **fetchall()**

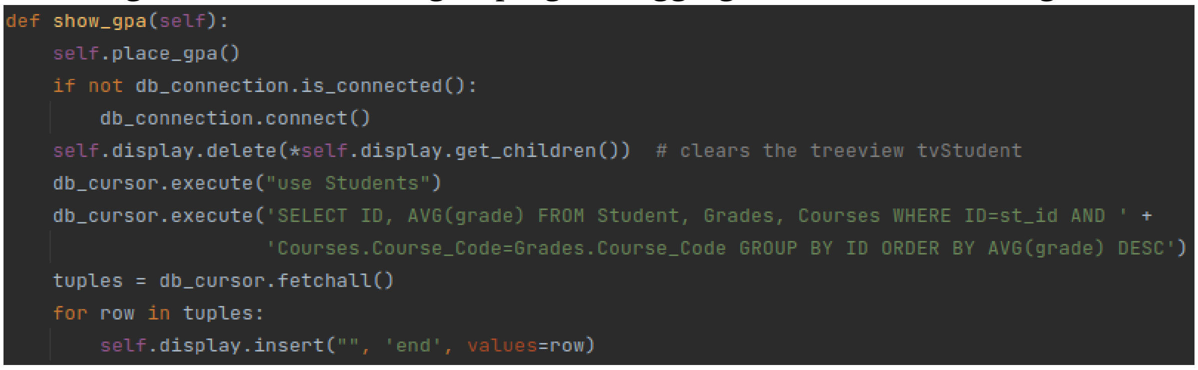
function takes all remaining tuples from the last executed statement by using the cursor. It basically returns an array that has the tuples. With this **fetchall**() function, it inserts these rows into the Treeview.



```python
def show_students(self):
    self.place_records()
    if not db_connection.is_connected():
        db_connection.connect()
    self.display.delete(*self.display.get_children())  # clears the treeview tvStudent
    db_cursor.execute("use Students")
    db_cursor.execute('SELECT DISTINCT ID, FName, LName, Dept_Code FROM Student, Grades WHERE st_id=ID ORDER BY ID')
    tuples = db_cursor.fetchall()
    for row in tuples:
        self.display.insert("", 'end', values=row)
```

(Figure 18)

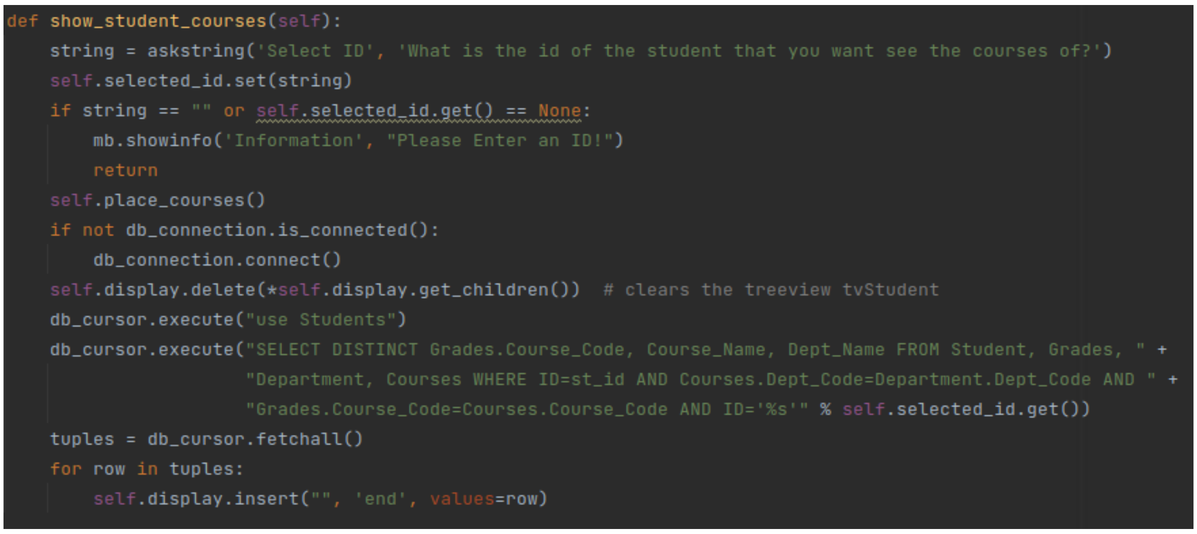
The **show_gpa** method shows the average **grades** for each student. For retrieving this data, we used grouping and aggregation as shown in Figure 19.



```python
def show_gpa(self):
    self.place_gpa()
    if not db_connection.is_connected():
        db_connection.connect()
    self.display.delete(*self.display.get_children())  # clears the treeview tvStudent
    db_cursor.execute("use Students")
    db_cursor.execute('SELECT ID, AVG(grade) FROM Student, Grades, Courses WHERE ID=st_id AND ' +
                      'Courses.Course_Code=Grades.Course_Code GROUP BY ID ORDER BY AVG(grade) DESC')
    tuples = db_cursor.fetchall()
    for row in tuples:
        self.display.insert("", 'end', values=row)
```

(Figure 19)

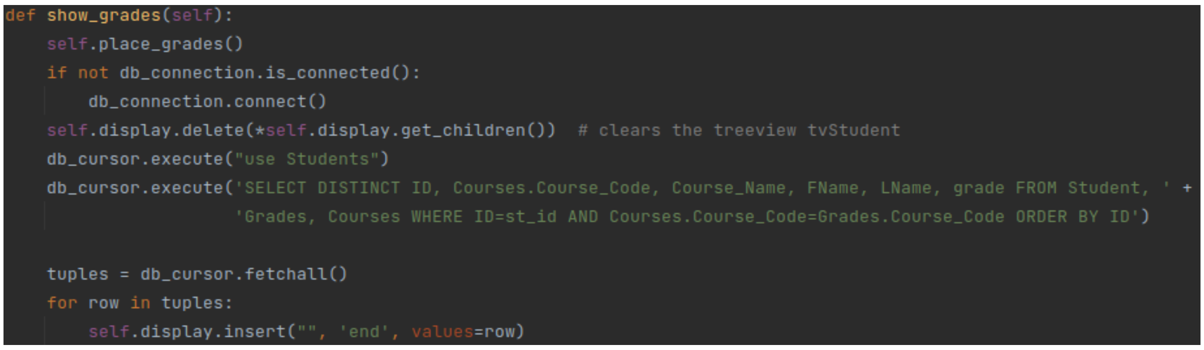
The program also displays the courses of a specific student that the ID of this student is entered by the user by using **show_student_courses** as shown in Figure 20. **self.selected_id** is a string variable that is defined in the **__init__** function. It is used to retrieve the entered ID. If it is an empty string or its type is None, then the program will give a warning saying that the user should enter a value. Then the program places the Treeview to show the columns: Course Code, Course Name and the Department Name by using **place_courses()**. If the connection is not provided, the program connects to the database. And it deletes all rows in the previously defined Treeview's rows. The program retrieves the tuples using the SQL query that is shown. It first looks at the foreign and primary key for generating the relation. Then it looks at the students where ID equals to the entered ID. After all, it inserts the values into Treeview.



```python
def show_student_courses(self):
    string = askstring('Select ID', 'What is the id of the student that you want see the courses of?')
    self.selected_id.set(string)
    if string == "" or self.selected_id.get() == None:
        mb.showinfo('Information', "Please Enter an ID!")
        return
    self.place_courses()
    if not db_connection.is_connected():
        db_connection.connect()
    self.display.delete(*self.display.get_children())  # clears the treeview tvStudent
    db_cursor.execute("use Students")
    db_cursor.execute("SELECT DISTINCT Grades.Course_Code, Course_Name, Dept_Name FROM Student, Grades, " +
                      "Department, Courses WHERE ID=st_id AND Courses.Dept_Code=Department.Dept_Code AND " +
                      "Grades.Course_Code=Courses.Course_Code AND ID='%s'" % self.selected_id.get())
    tuples = db_cursor.fetchall()
    for row in tuples:
        self.display.insert("", 'end', values=row)
```

(Figure 20)

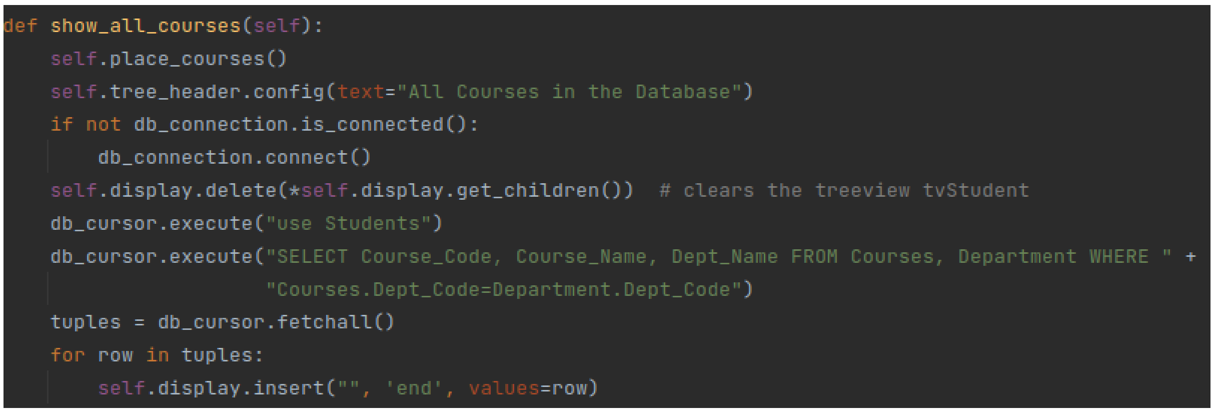
The user can see all grade records also. The **show_grades** method shows the grades of all courses of all students as shown in Figure 21. It first places the Treeview with columns: Student ID, Course Code, Course Name, First Name, Last Name, Grade. Then it deletes the rows from the previous Treeview. We used DISTINCT because some rows could be same while creating this 'join' of tables and we don't want to show duplicates. After retrieving the data, it will be inserted into the Treeview.



```python
def show_grades(self):
    self.place_grades()
    if not db_connection.is_connected():
        db_connection.connect()
    self.display.delete(*self.display.get_children())  # clears the treeview tvStudent
    db_cursor.execute("use Students")
    db_cursor.execute('SELECT DISTINCT ID, Courses.Course_Code, Course_Name, FName, LName, grade FROM Student, ' +
                      'Grades, Courses WHERE ID=st_id AND Courses.Course_Code=Grades.Course_Code ORDER BY ID')

    tuples = db_cursor.fetchall()
    for row in tuples:
        self.display.insert("", 'end', values=row)
```

(Figure 21)

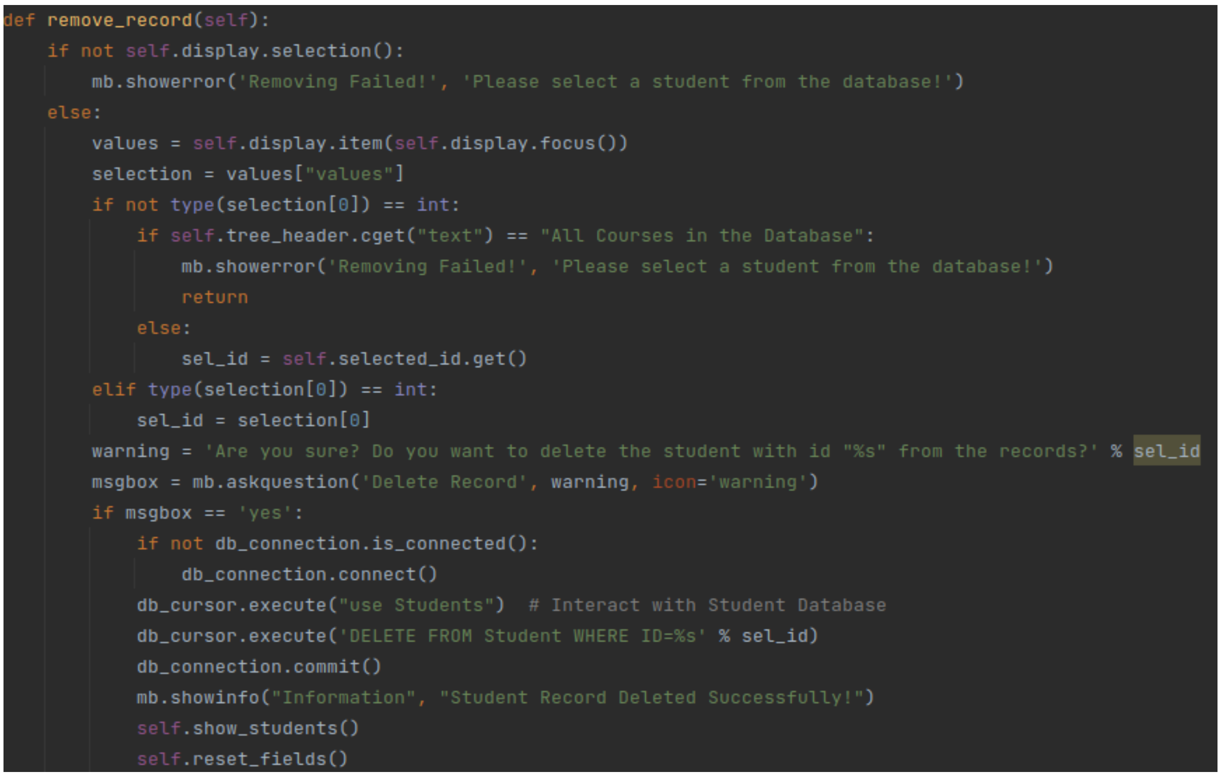
The program also provides a facility for showing all courses. Like **show_student_courses**, it places the columns: Course Code, Course Name and the Department that this course belongs to. Then it retrieves these attributes and inserts them into Treeview.



```
def show_all_courses(self):
    self.place_courses()
    self.tree_header.config(text="All Courses in the Database")
    if not db_connection.is_connected():
        db_connection.connect()
    self.display.delete(*self.display.get_children())  # clears the treeview tvStudent
    db_cursor.execute("use Students")
    db_cursor.execute("SELECT Course_Code, Course_Name, Dept_Name FROM Courses, Department WHERE " +
                      "Courses.Dept_Code=Department.Dept_Code")
    tuples = db_cursor.fetchall()
    for row in tuples:
        self.display.insert("", 'end', values=row)
```

(Figure 22)

The user can also delete student records from the database by **remove_record** method as shown in Figure 23. Treeview widget has a function called **selection()** that allows us to retrieve the row that is clicked by the user from the Treeview. If there is no selected row, the program will give a warning. After that, to check if the Treeview's first column is the student ID or not, I looked at the type of the first column. All Treeviews that are created in **place** methods, have the first column either Student ID or Course Code. If it is an integer this means it is student ID because there is no other first column with type integer. If it is not an integer, this means the method **place_courses** is used. In this case **sel_id** (which is the selected student ID for deleting) will be the value of the first column. There are two **show** methods that use **place_courses** which are **show_all_courses** or **show_student_courses**. In **show_all_courses**, there are no student attributes shown. So, we cannot select a student to delete in this method. So, if the title **(self.tree_header)** is "All Courses in the Database", the program gives a warning and shows the Student table after that. If the **show_student_courses** is executed, then there won't be the student ID in the Treeview but instead we will have one student selected. So, the user can choose one of the courses of selected student and this selected student will be deleted from the database. For doing this we assigned the value of the **self.selected_id** (for showing the courses of this selected student) to **sel_id** (for deleting the student). After all these operations, the student will be deleted from the **Student** table using DELETE FROM. Since all foreign keys that references Student ID are set to CASCADE for deleting and updating, when we delete this ID from Student, all other tables -that reference it- will be automatically deleted also. After all these operations, the code will clear the entries and show the Student table.

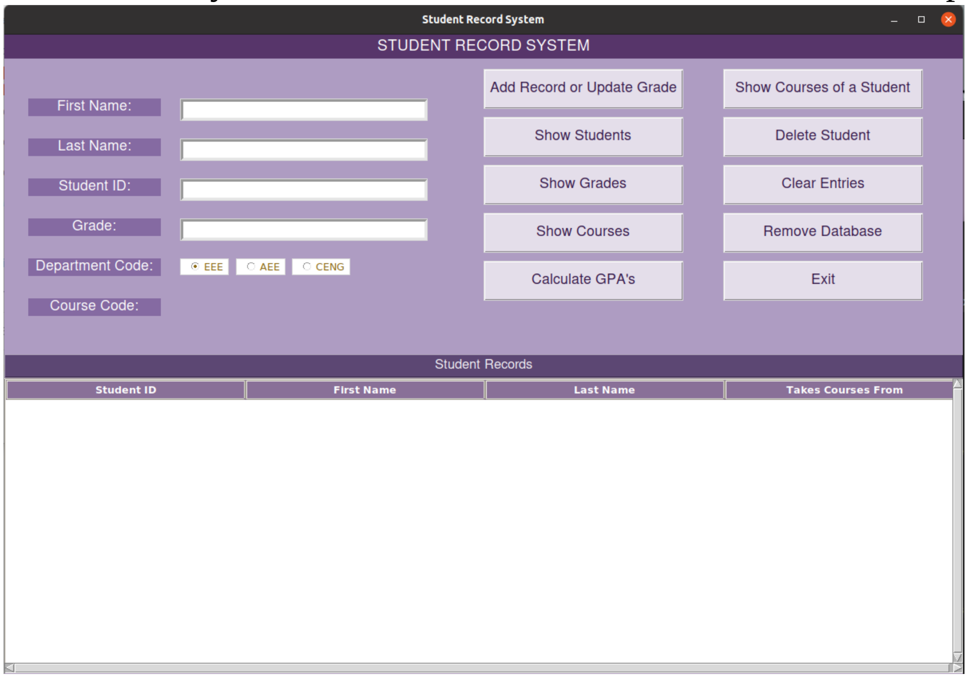


```
def remove_record(self):
    if not self.display.selection():
        mb.showerror('Removing Failed!', 'Please select a student from the database!')
    else:
        values = self.display.item(self.display.focus())
        selection = values["values"]
        if not type(selection[0]) == int:
            if self.tree_header.cget("text") == "All Courses in the Database":
                mb.showerror('Removing Failed!', 'Please select a student from the database!')
                return
            else:
                sel_id = self.selected_id.get()
        elif type(selection[0]) == int:
            sel_id = selection[0]
        warning = 'Are you sure? Do you want to delete the student with id "%s" from the records?' % sel_id
        msgbox = mb.askquestion('Delete Record', warning, icon='warning')
        if msgbox == 'yes':
            if not db_connection.is_connected():
                db_connection.connect()
            db_cursor.execute("use Students")  # Interact with Student Database
            db_cursor.execute('DELETE FROM Student WHERE ID=%s' % sel_id)
            db_connection.commit()
            mb.showinfo("Information", "Student Record Deleted Successfully!")
            self.show_students()
            self.reset_fields()
```

(Figure 23)

User Manual

The application mainly looks like as follows when we first execute the program.

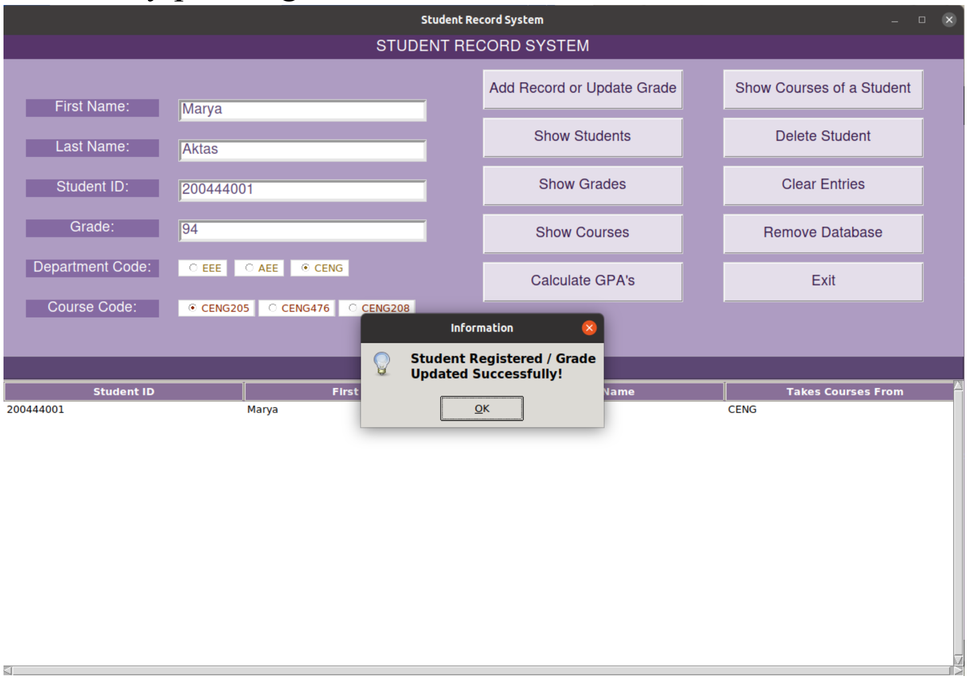


First Name:
Last Name:
Student ID:
Grade:
Department Code: [✓ EEE] AEE CENG
Course Code:

Student ID	First Name	Last Name	Takes Courses From

(Figure 24)

The user can here add a new student to the database by entering values to the entries and choosing department code. After choosing a department code, three other buttons will appear as shown in Figure 25. After choosing a course code the user can choose 'Add Record or Update Grade' button. After clicking 'OK', the user can see all students in the database as shown in Figure 26. The user can clear the entries by pressing the 'Clear Entries' button.

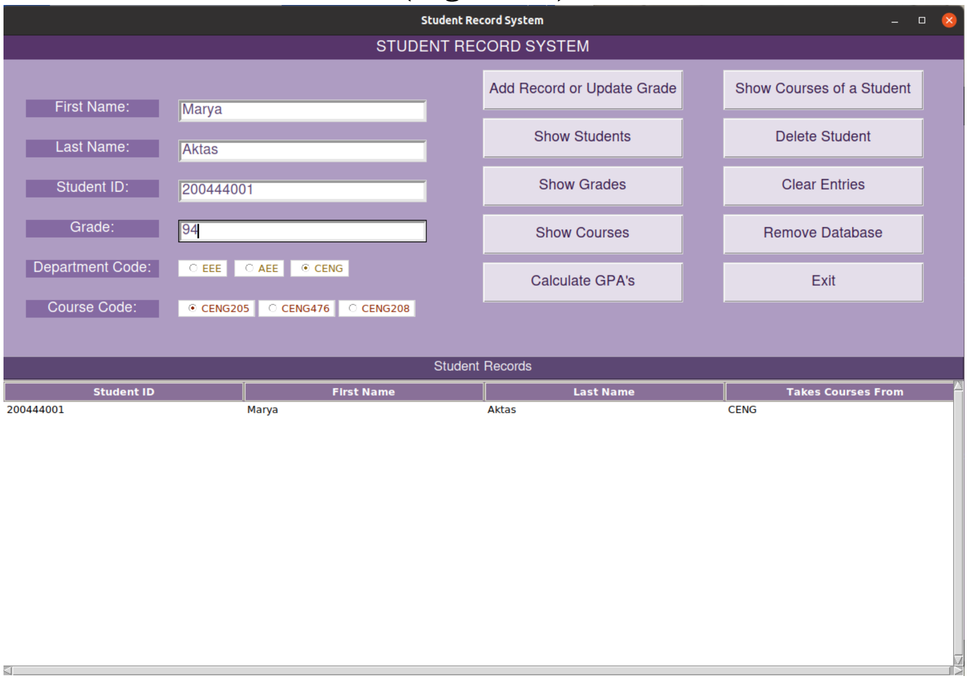


First Name: Marya
Last Name: Aktas
Student ID: 200444001
Grade: 94
Department Code: EEE AEE [✓ CENG]
Course Code: [✓ CENG205] CENG476 CENG208

Student Registered / Grade Updated Successfully!

Student ID	First	Name	Takes Courses From
200444001	Marya		CENG

(Figure 25)

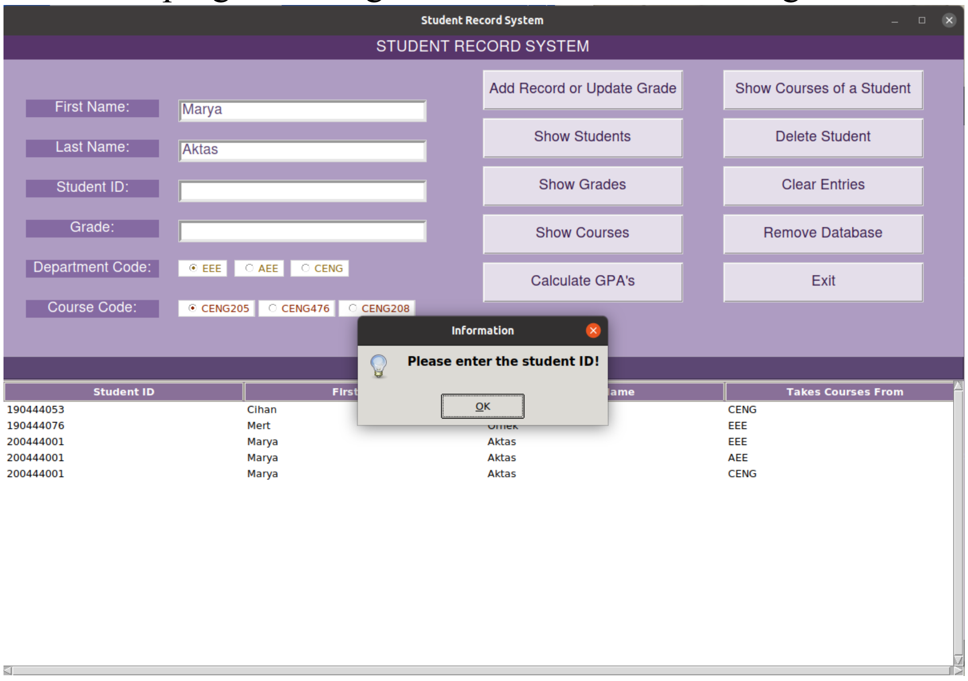


First Name:	Marya
Last Name:	Aktas
Student ID:	200444001
Grade:	94
Department Code:	EEE AEE [✓ CENG]
Course Code:	[✓ CENG205] CENG476 CENG208

Student ID	First Name	Last Name	Takes Courses From
200444001	Marya	Aktas	CENG

(Figure 26)

If the user does not enter one or more attributes and presses the ‘Add Record or Update Grade’ the program will give an error as shown in Figure 27.



First Name:	Marya
Last Name:	Aktas
Student ID:	
Grade:	
Department Code:	[✓ EEE] AEE CENG
Course Code:	[✓ CENG205] CENG476 CENG208

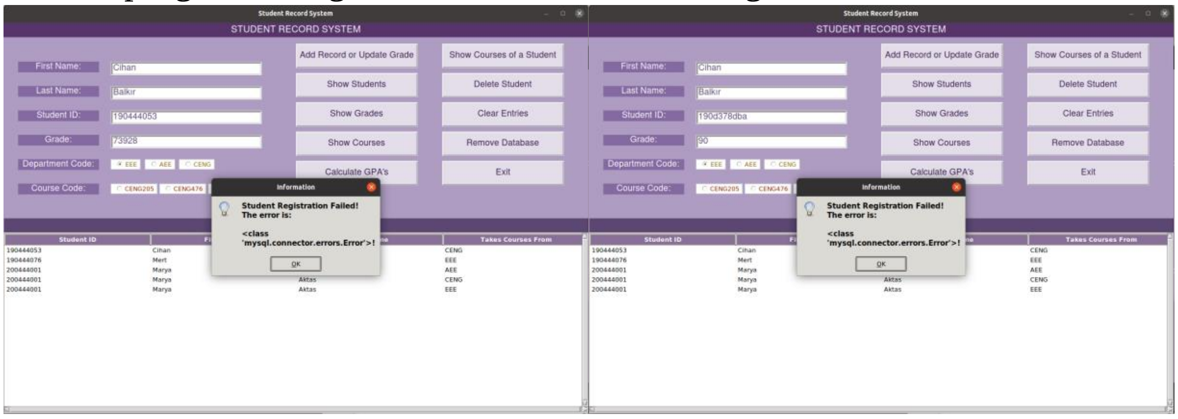
Information

Please enter the student ID!

Student ID	First	ame	Takes Courses From
190444053	Cihan	[illegible]	CENG
190444076	Mert	Omek	EEE
200444001	Marya	Aktas	EEE
200444001	Marya	Aktas	AEE
200444001	Marya	Aktas	CENG

(Figure 27)

If the user tries to insert a non-integer student ID, or a grade not between 0 and 100 the program will give an error as shown in Figure 28.



First Name: Cihan
Last Name: Balkır
Student ID: 190444053
Grade: 73928
Department Code: [✓ EEE] AEE CENG
Course Code: CENG205 CENG476

Student Registration Failed!
The error is:

<class
'mysql.connector.errors.Error'>!

Student ID	Fi	e	Takes Courses From
190444053	Cihan		CENG
190444076	Mert		EEE
200444001	Marya		AEE
200444001	Marya	Aktas	CENG
200444001	Marya	Aktas	EEE

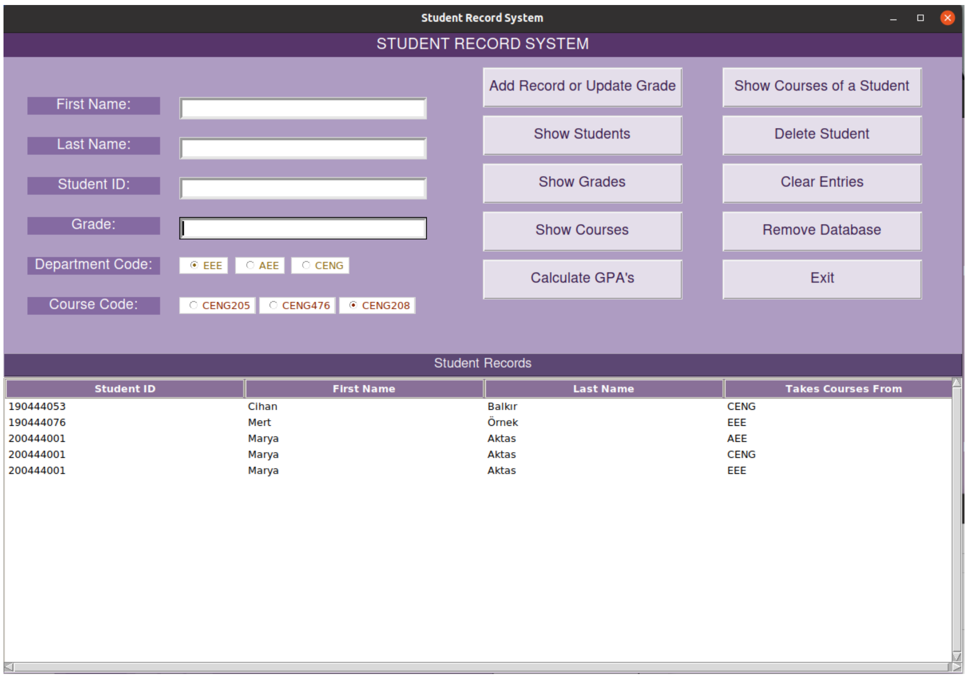
First Name: Cihan
Last Name: Balkır
Student ID: 190d378dba
Grade: 90
Department Code: [✓ EEE] AEE CENG
Course Code: CENG205 CENG476

Student Registration Failed!
The error is:

<class
'mysql.connector.errors.Error'>!

(Figure 28)

The user can choose courses from one or more departments. An example is shown below.

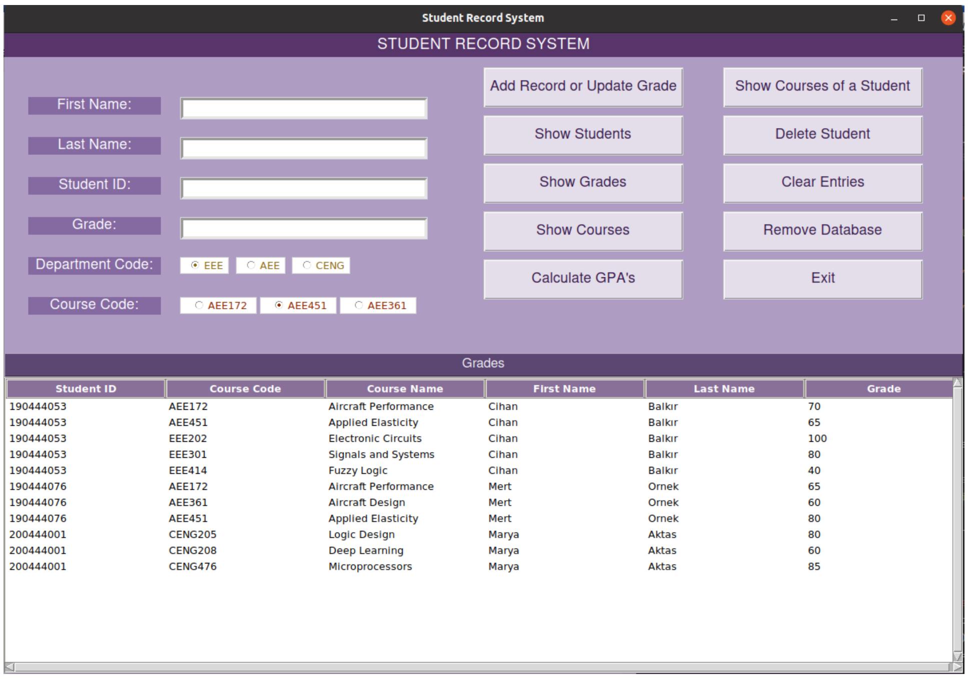


First Name:
Last Name:
Student ID:
Grade:
Department Code: [✓ EEE] AEE CENG
Course Code: CENG205 CENG476 [✓ CENG208]

Student ID	First Name	Last Name	Takes Courses From
190444053	Cihan	Balkır	CENG
190444076	Mert	Örnek	EEE
200444001	Marya	Aktas	AEE
200444001	Marya	Aktas	CENG
200444001	Marya	Aktas	EEE

(Figure 29)

The user can see the grades of all courses of all students by clicking 'Show Grades' as shown below.

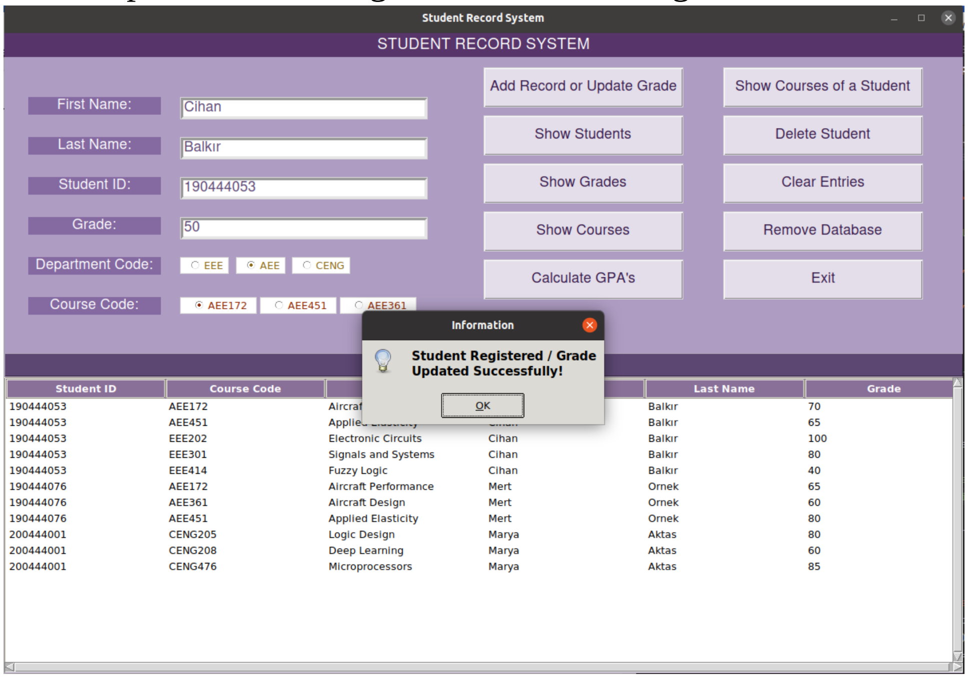


First Name:	
Last Name:	
Student ID:	
Grade:	
Department Code:	[✓ EEE] AEE CENG
Course Code:	AEE172 [✓ AEE451] AEE361

Student ID	Course Code	Course Name	First Name	Last Name	Grade
190444053	AEE172	Aircraft Performance	Cihan	Balkır	70
190444053	AEE451	Applied Elasticity	Cihan	Balkır	65
190444053	EEE202	Electronic Circuits	Cihan	Balkır	100
190444053	EEE301	Signals and Systems	Cihan	Balkır	80
190444053	EEE414	Fuzzy Logic	Cihan	Balkır	40
190444076	AEE172	Aircraft Performance	Mert	Ornek	65
190444076	AEE361	Aircraft Design	Mert	Ornek	60
190444076	AEE451	Applied Elasticity	Mert	Ornek	80
200444001	CENG205	Logic Design	Marya	Aktas	80
200444001	CENG208	Deep Learning	Marya	Aktas	60
200444001	CENG476	Microprocessors	Marya	Aktas	85

(Figure 30)

If the user enters a tuple with already existing ID and course code, the program will update the grade in this case. An example is shown below in Figure 31. In this case we are expecting the grade of 'AEE172' will be updated as 50 when it is 70 in the first place. The final grade is shown in Figure 32.

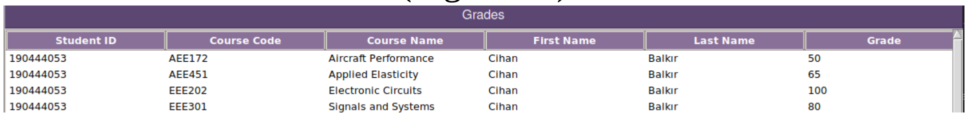


First Name:	Cihan
Last Name:	Balkır
Student ID:	190444053
Grade:	50
Department Code:	EEE [✓ AEE] CENG
Course Code:	[✓ AEE172] AEE451 AEE361

Student Registered / Grade Updated Successfully!

Student ID	Course Code			Last Name	Grade
190444053	AEE172	Aircraf		Balkır	70
190444053	AEE451	Applie		Balkır	65
190444053	EEE202	Electronic Circuits	Cihan	Balkır	100
190444053	EEE301	Signals and Systems	Cihan	Balkır	80
190444053	EEE414	Fuzzy Logic	Cihan	Balkır	40
190444076	AEE172	Aircraft Performance	Mert	Ornek	65
190444076	AEE361	Aircraft Design	Mert	Ornek	60
190444076	AEE451	Applied Elasticity	Mert	Ornek	80
200444001	CENG205	Logic Design	Marya	Aktas	80
200444001	CENG208	Deep Learning	Marya	Aktas	60
200444001	CENG476	Microprocessors	Marya	Aktas	85

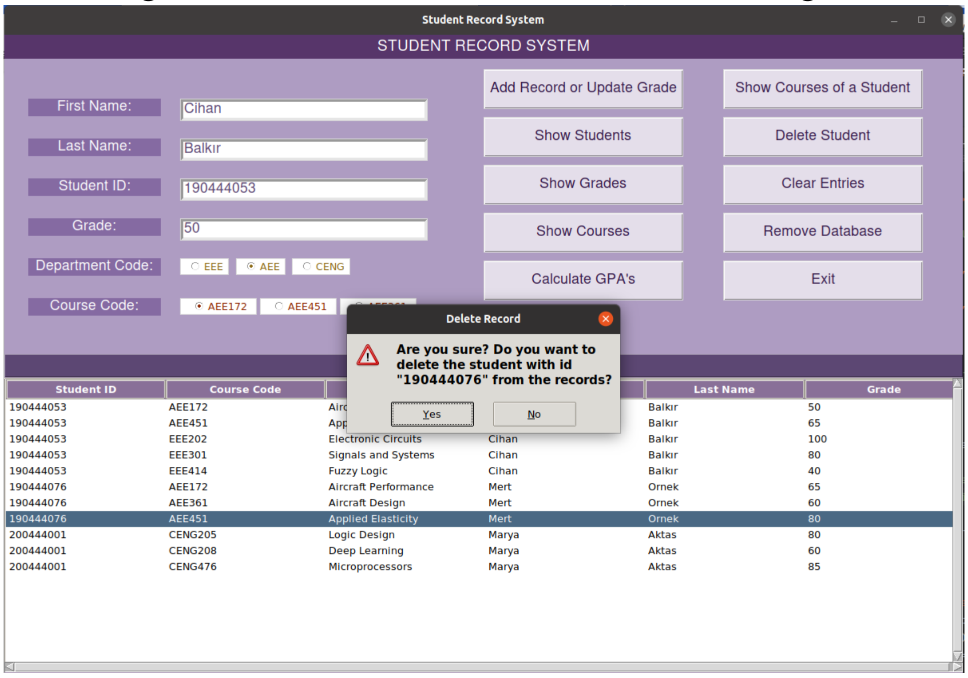
(Figure 31)



Student ID	Course Code	Course Name	First Name	Last Name	Grade
190444053	AEE172	Aircraft Performance	Cihan	Balkır	50
190444053	AEE451	Applied Elasticity	Cihan	Balkır	65
190444053	EEE202	Electronic Circuits	Cihan	Balkır	100
190444053	EEE301	Signals and Systems	Cihan	Balkır	80

(Figure 32)

The user can also delete a student record from this table by choosing one of the rows then clicking the ‘Delete Student’ button as shown in Figure 33.



First Name: Cihan
Last Name: Balkır
Student ID: 190444053
Grade: 50
Department Code: EEE [✓ AEE] CENG
Course Code: [✓ AEE172] AEE451

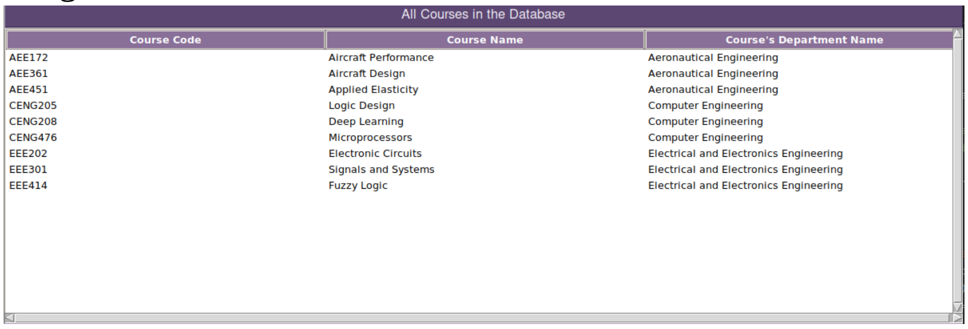
Delete Record

Are you sure? Do you want to delete the student with id "190444076" from the records?

Student ID	Course Code			Last Name	Grade
190444053	AEE172	Airc		Balkır	50
190444053	AEE451	App		Balkır	65
190444053	EEE202	Electronic Circuits	Cihan	Balkır	100
190444053	EEE301	Signals and Systems	Cihan	Balkır	80
190444053	EEE414	Fuzzy Logic	Cihan	Balkır	40
190444076	AEE172	Aircraft Performance	Mert	Ornek	65
190444076	AEE361	Aircraft Design	Mert	Ornek	60
190444076	AEE451	Applied Elasticity	Mert	Ornek	80
200444001	CENG205	Logic Design	Marya	Aktas	80
200444001	CENG208	Deep Learning	Marya	Aktas	60
200444001	CENG476	Microprocessors	Marya	Aktas	85

(Figure 33)

The user can also see the courses in the database by clicking ‘Show Courses’ as shown in Figure 34.

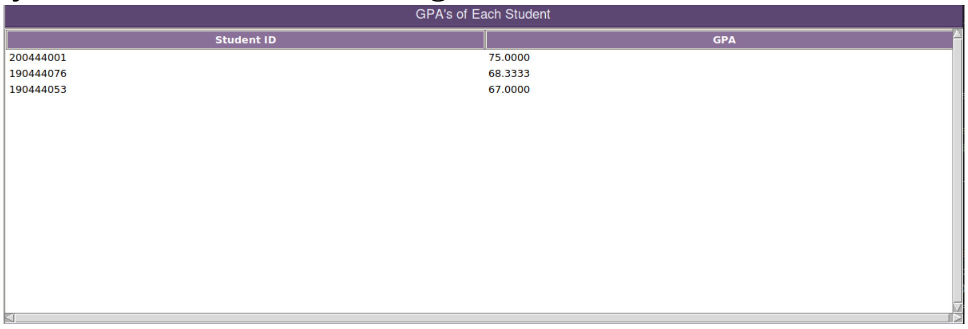


All Courses in the Database

Course Code	Course Name	Course's Department Name
AEE172	Aircraft Performance	Aeronautical Engineering
AEE361	Aircraft Design	Aeronautical Engineering
AEE451	Applied Elasticity	Aeronautical Engineering
CENG205	Logic Design	Computer Engineering
CENG208	Deep Learning	Computer Engineering
CENG476	Microprocessors	Computer Engineering
EEE202	Electronic Circuits	Electrical and Electronics Engineering
EEE301	Signals and Systems	Electrical and Electronics Engineering
EEE414	Fuzzy Logic	Electrical and Electronics Engineering

(Figure 34)

The user can see the GPAs of all students by clicking ‘Calculate GPA’s’ button as shown in Figure 35. Since the values are shown in descending order, the user can easily see the student with the highest GPA or the lowest GPA.

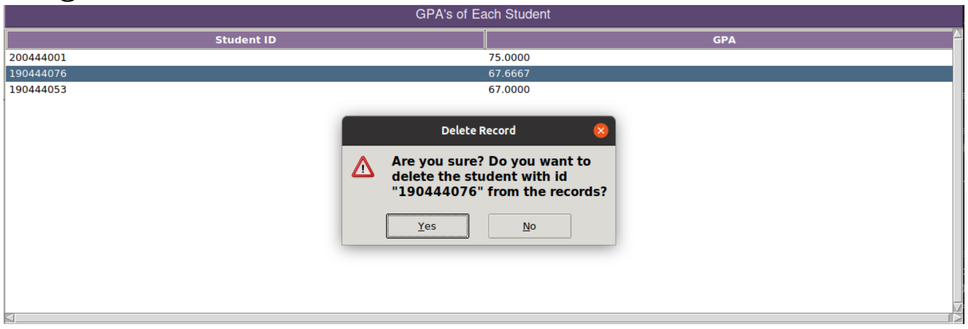


GPA's of Each Student

Student ID	GPA
200444001	75.0000
190444076	68.3333
190444053	67.0000

(Figure 35)

The user can also delete a student record from ‘GPA’s of Each Student’ table by clicking one of the rows and then ‘Delete Student’ button. One instance is shown in Figure 35.



GPA's of Each Student

Student ID	GPA
200444001	75.0000
190444076	67.6667
190444053	67.0000

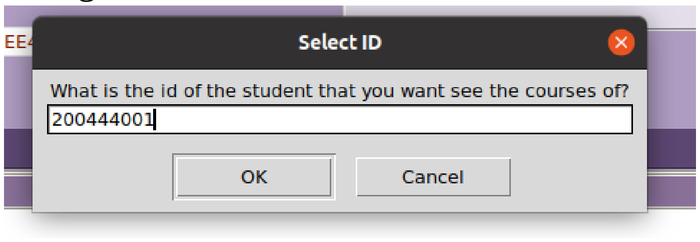
Delete Record

Are you sure? Do you want to delete the student with id "190444076" from the records?

(Figure 35)

The user can return to the beginning page where she can see the student records by clicking ‘Show Students’ and can perform a deletion after selecting a row and clicking ‘Delete Student’ button.

If the user wants to the courses of a specific student takes, she can click ‘Show Courses of a Student’ button. After clicking it, a window will appear for asking the ID of this student as shown in Figure 36. After clicking ‘OK’, the courses will be shown as in Figure 37.

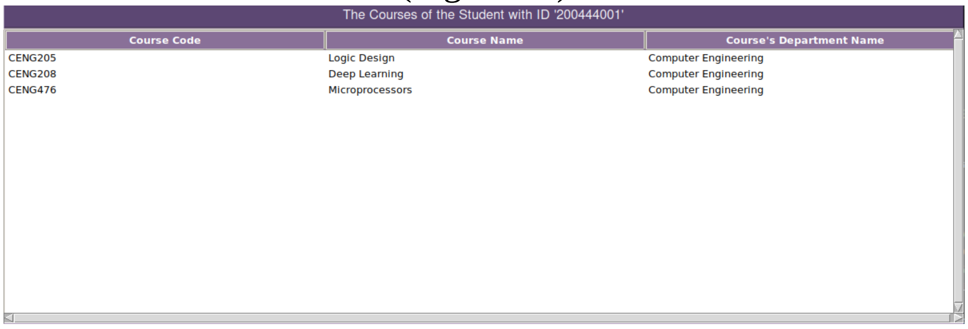


Select ID

What is the id of the student that you want see the courses of?

200444001

(Figure 36)

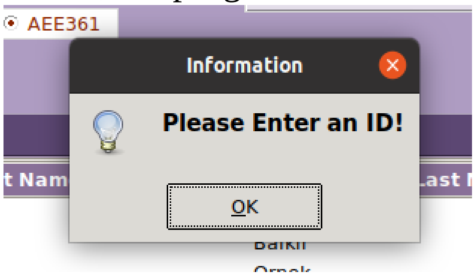


The Courses of the Student with ID '200444001'

Course Code	Course Name	Course's Department Name
CENG205	Logic Design	Computer Engineering
CENG208	Deep Learning	Computer Engineering
CENG476	Microprocessors	Computer Engineering

(Figure 37)

If the user does not enter an ID, the program will raise an error as follows.

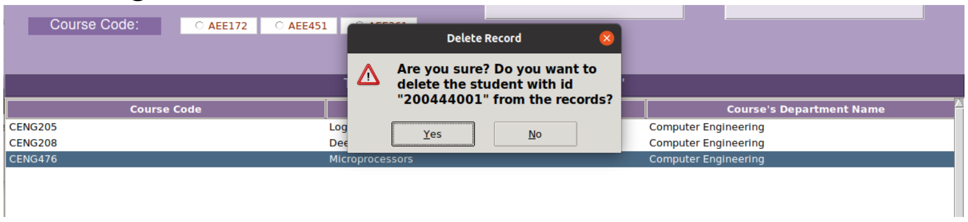


Information

Please Enter an ID!

(Figure 38)

The user can also delete this selected student in this table by selecting one of the rows and clicking 'Delete Student' button.

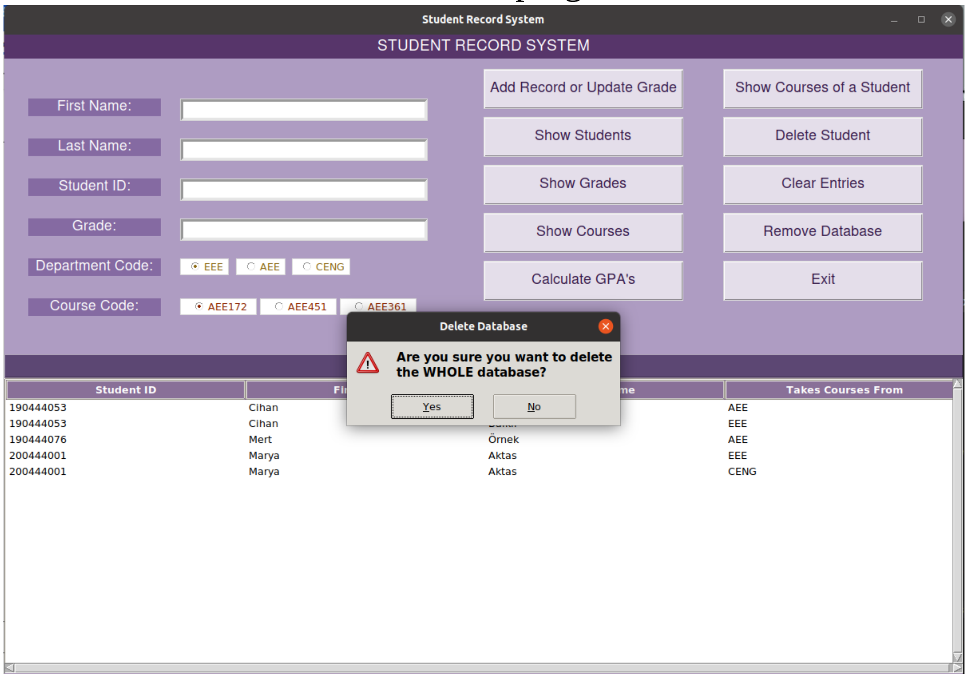


Are you sure? Do you want to delete the student with id "200444001" from the records?

Course Code		Course's Department Name
CENG205	Log	Computer Engineering
CENG208	Dee	Computer Engineering
CENG476	Microprocessors	Computer Engineering

(Figure 39)

The user can delete the database by clicking 'Remove Database' button whenever she wants also as shown in Figure 40. After clicking 'Yes', the Students database will be deleted, and the program will be terminated.



Are you sure you want to delete the WHOLE database?

Student ID	Fi	me	Takes Courses From
190444053	Cihan		AEE
190444053	Cihan		EEE
190444076	Mert	Örnek	AEE
200444001	Marya	Aktas	EEE
200444001	Marya	Aktas	CENG

(Figure 40)

The user can exit the code whenever she wants by clicking 'Exit' button.

Conclusion

In this program we only have static values of departments and courses, but the developer can change these values or add values with small changes. Since we tried to think about every possibility, we think that this application can be useful for a real student record system for teachers.

While implementing this program, we learned how to be a team, how we can assign tasks between each other. We also had the chance to practice more with SQL queries. After learning all these theoretical parts, using them with real life examples and problems has contributed a lot to better understanding of these topics. Not only learning the queries but also learning how we can connect different environments i.e., MySQL and Python, broadened our perspective and knowledge on programming.