Mini Project 1

By: Andrew Rooney, Anthony Ma, Mustafa Khairullah

CMPUT 291

1. General Overview and User Guide

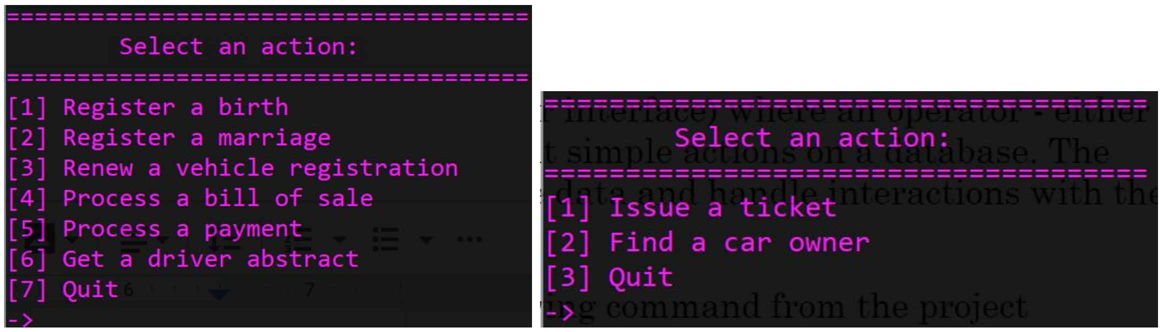
Our application provides a TUI (terminal user interface) where an operator - either a registry agent, or traffic officer can carry out simple actions on a database. The application uses python, and SQLite3 to store data and handle interactions with the data.

The program is started by invoking the following command from the project directory,

```
./start
```

Code Snippet ###: Begin Program

Note that the start command relies on there being exactly one database (".db") file of any name in the project directory. A login screen will appear once started. Enter your credentials here to be greeted by one of two menus; one for registry agents, and one for traffic officers - depending on user type stored in the database for the given credentials. Menus are as shown,



```
=====================================
       Select an action:
=====================================
[1] Register a birth
[2] Register a marriage
[3] Renew a vehicle registration
[4] Process a bill of sale
[5] Process a payment
[6] Get a driver abstract
[7] Quit
->
```

```
====================================
       Select an action:
====================================
[1] Issue a ticket
[2] Find a car owner
[3] Quit
->
```

Figure ###: Agent Menu (left) and Officer Menu (right)

From here, you can choose the desired action to begin augmenting the database. Each option has a menu that will guide you through the usage - simply follow the prompts to enter/change the desired data.

At any point, pressing (Ctrl+C) will take you back to the sign-in screen, and (Ctrl+Z), or selecting the "Quit" option from the menu will exit the program.

2. Design Details

The application file structure has a single level with 4 functional components,

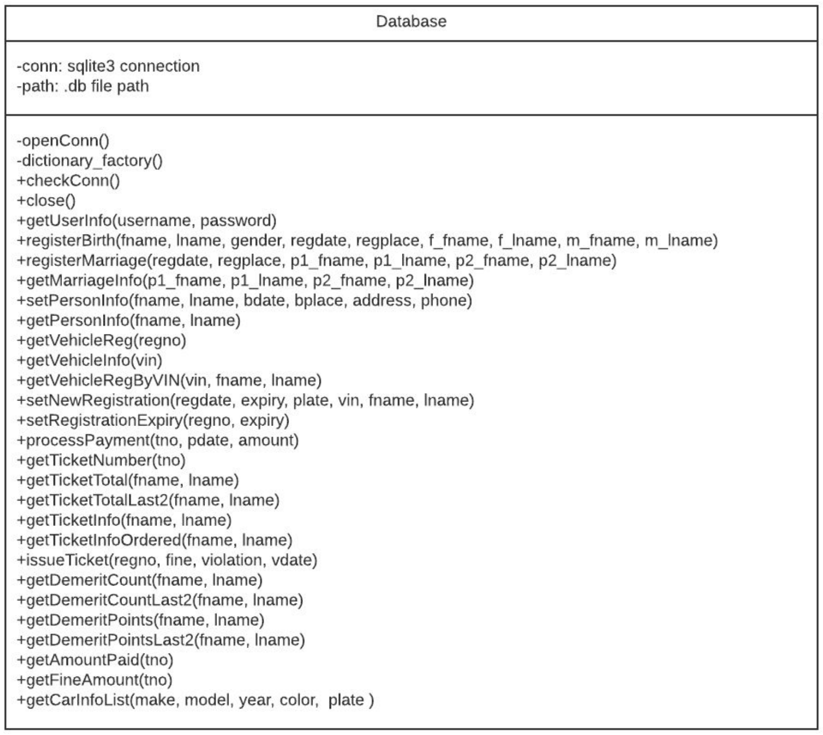
start: A bash script for beginning the application

miniProj.py: Python application to be run as the main function - this file provides the user interactions including menus, and other logic flow with some error checking

transactions.py: Python file containing the 'Database' class which handles the interactions with the database with some error checking

<database>.db: Database file containing data conforming to the schema given in eClass.

The ‘back end’ of the application - the portion of the code that is responsible for handling the connection to, and transactions with the database is found in *transactions.py* and contains the following class structure that is used in *miniProj.py* to provide user functionality,



```
Database

-conn: sqlite3 connection
-path: .db file path

-openConn()
-dictionary_factory()
+checkConn()
+close()
+getUserInfo(username, password)
+registerBirth(fname, lname, gender, regdate, regplace, f_fname, f_lname, m_fname, m_lname)
+registerMarriage(regdate, regplace, p1_fname, p1_lname, p2_fname, p2_lname)
+getMarriageInfo(p1_fname, p1_lname, p2_fname, p2_lname)
+setPersonInfo(fname, lname, bdate, bplace, address, phone)
+getPersonInfo(fname, lname)
+getVehicleReg(regno)
+getVehicleInfo(vin)
+getVehicleRegByVIN(vin, fname, lname)
+setNewRegistration(regdate, expiry, plate, vin, fname, lname)
+setRegistrationExpiry(regno, expiry)
+processPayment(tno, pdate, amount)
+getTicketNumber(tno)
+getTicketTotal(fname, lname)
+getTicketTotalLast2(fname, lname)
+getTicketInfo(fname, lname)
+getTicketInfoOrdered(fname, lname)
+issueTicket(regno, fine, violation, vdate)
+getDemeritCount(fname, lname)
+getDemeritCountLast2(fname, lname)
+getDemeritPoints(fname, lname)
+getDemeritPointsLast2(fname, lname)
+getAmountPaid(tno)
+getFineAmount(tno)
+getCarInfoList(make, model, year, color,  plate )
```

Figure ###: Database UML Diagram

The functions in the *transactions.py* are designed to provide a convenient way for developers to expand the functionality of the application without worrying about the details of the SQL connections. The method names of the Database class are also designed to be very descriptive about what they’re doing. This design allowed us to develop the logic of the application separately from the SQL queries; In this file, all string matching queries use the “COLLATE NOCASE” clause to remove case sensitivity from matches, and all queries are constructed using Python’s DB-API’s ‘?’ operator for parameter substitution, which protects the queries from malicious input[1] (such as injection attacks). Regex is also done on passwords to limit the characters to alphanumeric characters, and some symbols.

It’s worth noting that the method used for assigning unique keys when inserting into tables is to increment the maximum existing key - which always produces a new, unique key. This can be shown by a formal proof by contradiction. The essence of the proof says that if incrementing the maximum key were to not return a unique key, then there was a key greater than the maximum key (contradiction).

The user interface of the application has two main menus that dispatch the user to different components of the application’s logic, the registry agent menu and traffic officer’s menu. The registry agent’s functions include those that add new registrations into the database, update

[1] https://docs.python.org/dev/library/sqlite3.html

them, allow the user to process the payments, or give the user a report of a said chosen entry. The traffic officer's functions include issuing tickets that update the database, and also proving the user to search up an entry in the database by providing certain attributes.

All functions in the main logic of the program (in miniProj.py) were created by following exactly the specifications on eClass, and then implementing the SQL portion in *transactions.py*. This design let us work independently, and saved us time.

3. Testing

Each component of the specification was tested with the primary intention of conforming to the desired logic flow, and appropriate responses to bad input, and other edge cases. The components were tested against these criteria,

- With normal input, does the component do as is described in eClass?
- Does the component reject or handle bad input, such as
    - Input that is already in the DB (for functions to update the table)
    - Reject bad input values (i.e. numerical, or date format) and accept null input where appropriate
    - Cases in which the specifications on eClass require the input to be rejected
- Are all searches in the component are case insensitive?
- Is displayed text correct, and well formatted?
- Are SQL injection attacks countered?
- Are the elements inserted into the database correctly?

The team tested each function as they were created. When a new method was created in transactions.py It was tested by calling the new method in the main function, and reviewing the database before and after changes to ensure that any changes made by the transaction functions are correct, and that any values queried from the database are the desired ones. The DB Browser for SQLite application was used to make life easier when comparing our application's output with database values.

When an item from the eClass list of questions was completed, it was unit tested by the creator following (generally) the specifications aforementioned. Then the tested code was pushed to GitHub, and the group members ran their own tests on all possible edge cases they could think of, making improvements as needed.

Values from Ryan Kang's data set from Assignment 2 were used to populate the database for testing the application.

4. Group Work

We used GitHub for this project, and each partner pushed their changes and work on project into the GitHub repository after a work session, we informed each other of the changes we made via git commit messages and Discord.

Andrew's contributions to the project include designing the menus, and the registerBirth, issueTicket, and processBOS function alongside the necessary functions that acted on the database (row factory, and connection handlers in the Database class). Andrew also came up with many of the overarching design choices on the project. Andrew spent about 8 hours working on the project.

Anthony’s contributions to the project include the registerMarriage, processPayment, and getDriverAbstract functions alongside the necessary functions that acted on the database. Anthony spent about 6 hours working on the project.

Mustafa’s contributions to the project include the renewRegistrations and getCarOwner function alongside the necessary functions that acted on the database. Mustafa spent about 6 hours working on the project.

All three members tested their own functions and applied the criteria specified in the testing function above. If a member decided that a function that acts on the database object was necessary for the function they were working on, they added so at their own discretion.

The breakdown of the work and contributions made to the project are a general breakdown and are not completely accurate, as all 3 members helped and coordinated with the other 2’s work, supplementing their own understanding of the project. Each member shared various ideas and possible improvements to both the program and the design frequently, alongside useful resources found online.

Much of the project was completed when we met up in person, which allowed for the most convenient and efficient level of coordination. The rest of the project was completed on each member’s own time and in group voice calls between the members.