Memoeier Documentation and Report

Group: H20

Author: ZHANG Zhong & ZHANG Jiekai

This the the exported PDF file, we recommand you to read the markdown file directly!

And you can visit our github page here.

---

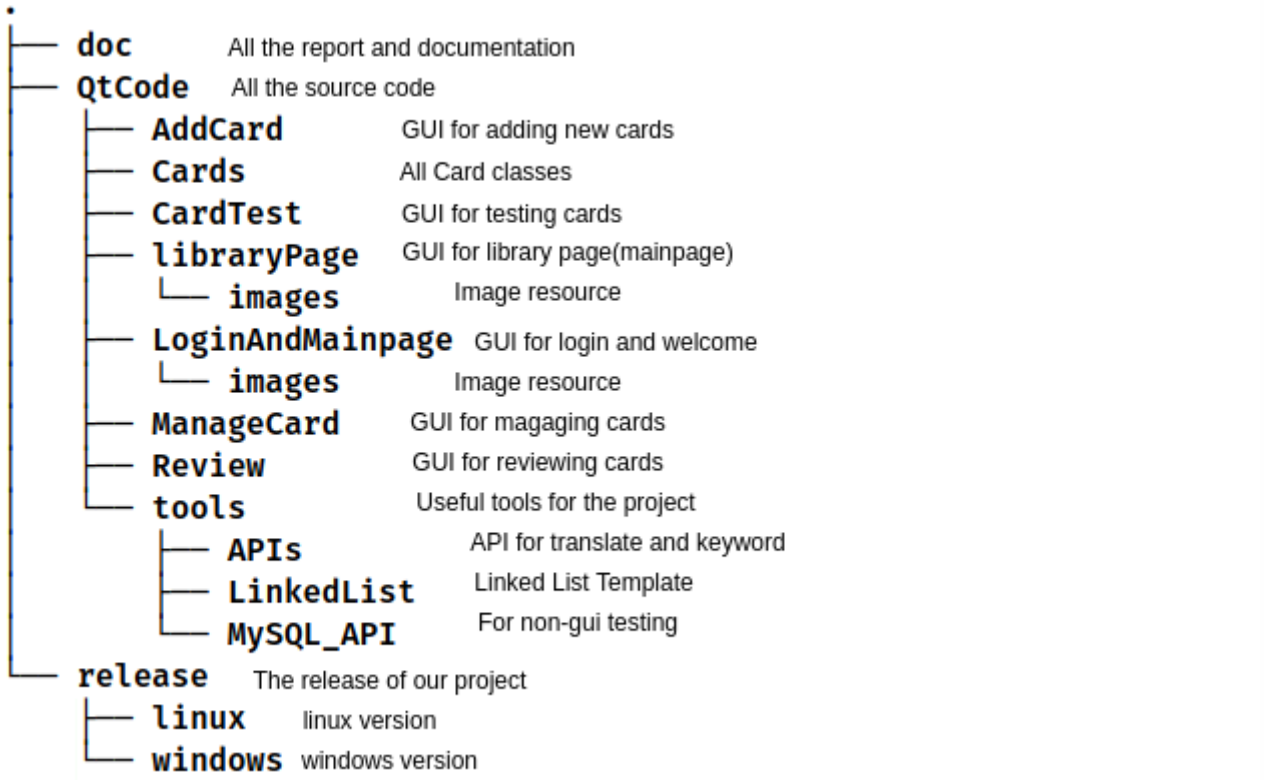
0. Overview

Our Objective

Memorier is a learning aid for you, which can help you remember and review better.

You can store knowledge in the form of **adding memory cards**, and Memorier will arrange regular **review and test** functions for you.
The arrangement of reviewing and testing the cards is according to the **forgetting curve** to strengthen your memory of knowledge. Also, we support **custom review and test**. You can select the **type you want** and the **time period** to create the card set and generate a review or test sequence. In addition, all your cards will be **automatically synchronized** on your cloud account, which is convenient for multi-device switching.

The file struct and usage:



1. Compile and run the program

- **Important Reminder: If you want to share the code, please delete the database info in main.cpp!**
- You can directly use Qt to compile the code.
- Since we are using MySQL database in the code, to run the program, please make sure that you have built the QMYSQL plugin. **There is no integrated support for MySQL plugin in Qt**, you need to download and compile the source code yourself.
- If you would like to know how to compile the MySQL plugin, click HERE for Linux and HERE for Windows
- **If cannot run the compiled program correctly, there is a release version in the "relaese" folder, you can use that one**

2. How to use Memorier

- Log in/Sign up for your account (All your cards will be stored in the cloud, so sign an account is a must)
- Add your card by click "**Add Card**" button on the library page, there are 4 types of cards:
  - Plain text: the most basic card, you can type Problem and Answer to it.
  - Hollow text: you can type Problem and hint to it, you can add some "hollow" in it. At the corresponding window, you can simply select the text using mouse and click "add"; you can also try our auto generate keywords, you can see the button "Show suggest" and "Next suggest", all text clip that match the keyword will be select once you click "apply"; the box showing keywords is editable, so you can type in your own word and click "apply", all text clip that match will be selected.
  - Word: you can type the word you want to remember, the first time you click translation box, the program will automatically translate it for you, you can edit it if you are not satisfied.

> - Multi choices: you can type problem and hint, later you can add up to 5 choices and set any number of them as correct answer.

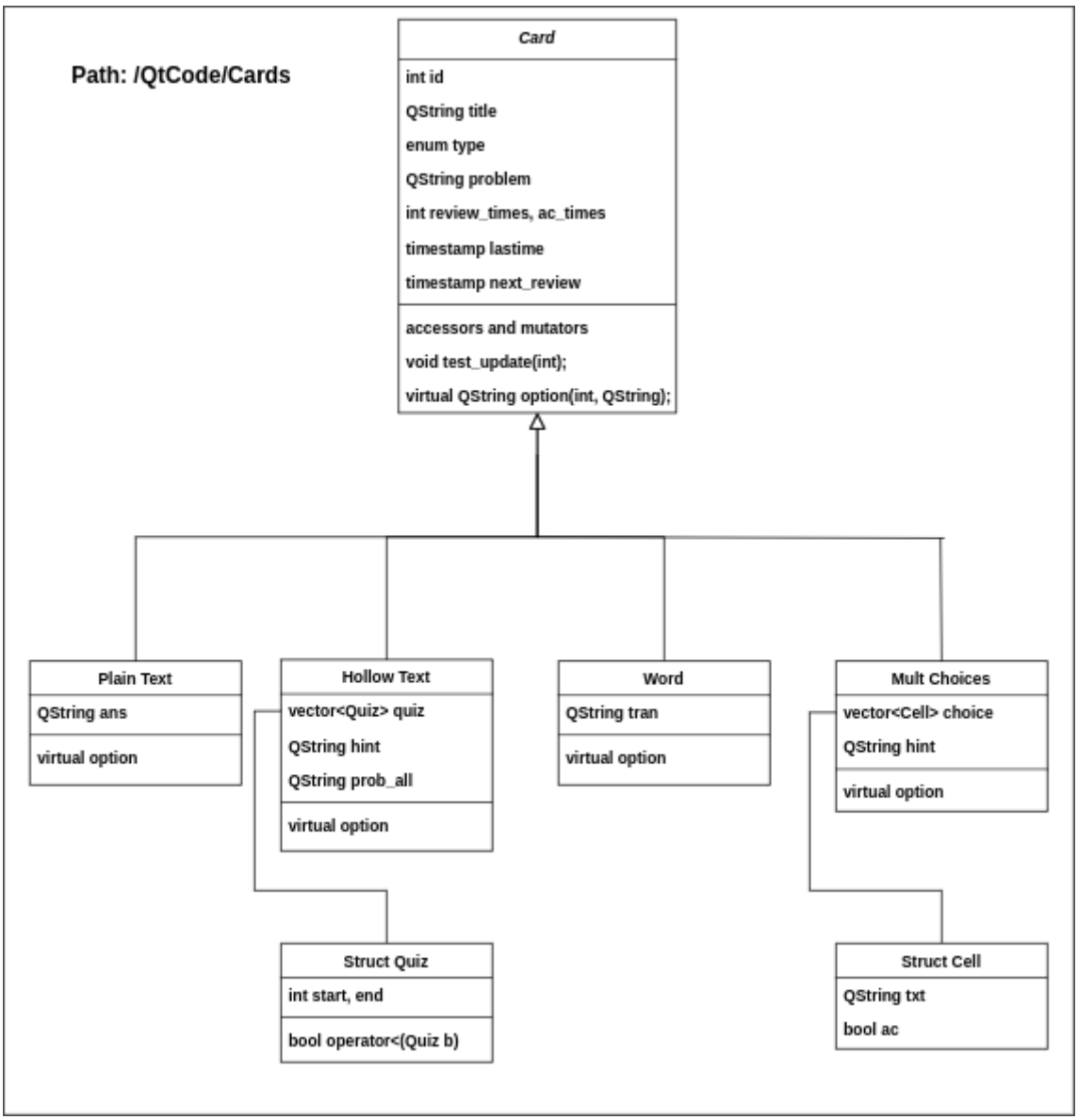
- After adding the card you can "**Review**" them, either by their type or by the time you added it. (In the latter case, we will arrange the next review time according to the forgetting curve)
- The "**Test**" function is similar to review, but you need to answer the question in the card and we will record your score for future Review and Test
- In "**Management**" page you can delete or edit(later) your cards, there is a filter for searching cards.

3. Class structure and functions

In this part, we will go through all the important classes and functions used in our project.

I. Card class and its drived classes

Class diagram:



- Base class: Card
  The **Card** class provide a base class for other cards.
  It contains all the common data members of a class

> **Data members**:
>
> - id: the id in of a card in the database (also a unique identifier)
> - title: the title of a card
> - type: the type of a card (Plain text, Hollow text, Word, Mult choice)
> - problem: the problem body of card
> - review_times, ac_times: time of reviews and times of correctness in the test
> - lastime: last timestamp the card was reviewed
> - next_review: next timestamp the card will be reviewed

> **Member functions**:
>
> - accessors and mutators: gets and sets
> - void test_update(int): update the data members after test
> - virtual QString option(int option, QString data): all the encoding and decoding functions (for storing/reading the data to/from database) new functions should be added here
>   **option id list**:
>   > - 1 - set answer for this card
>   > - 2 - set special data for this card
>   > - 3 - encode to store to database
>   > - 4 - decode for read from database
>   > - 5 - get answer for this card (pair with 1)
>   > - 6 - Butshow text for review
>   > - 7 - extra button for review
>   > - 8 - extra output for review (get full answer)
>   > - 9 - return side-to-side compare version correct answer when test::submit
>   > - 10 - return noting for side-to-side compare when test::submit

- Drived class1: Plain Text
  The **Plain Text** is for storing a card with only problem and answer

> **Data members**:
>
> - ans: storing the answer of the card

> **Member functions**:
>
> - virtual QString option(int option, QString data): same function as in the base class

- Drived class2: Hollow Text
  The **Hollow Text** is for storing a "fill in the blank" card

> **Data members**:
>
> - quiz: a vector with each element storing one of the "blank" of the text, contains the start and end index.
> - hint: some hint of the answer
> - prob_all: encoded problem (for storing in database)

> **Member functions**:

> - virtual QString option(int option, QString data): same function as in the base class

- Drived class3: Word
  The **Word** is a card used to recite words and can also do automatically translation

> **Data members**:
>
> - tran: storing the translation of the word

> **Member functions**:
>
> - virtual QString option(int option, QString data): same function as in the base class

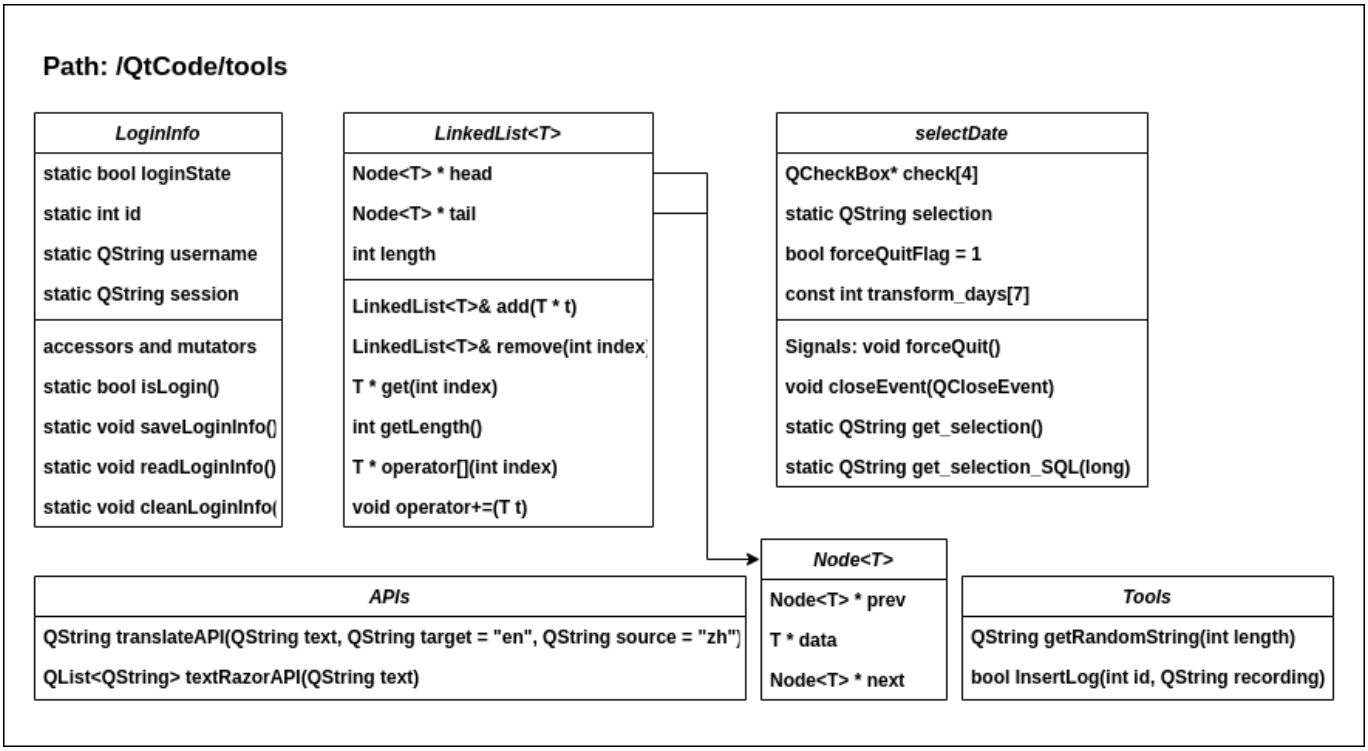
- Drived class4: Mult Choices
  The **Mult Choices** is for storing a card with only problem and answer

> **Data members**:
>
> - choice: a vector with each element storing one of the "Selection" of the problem
> - hint: some hint of the answer

> **Member functions**:
>
> - virtual QString option(int option, QString data): same function as in the base class

II. Tool classes

Class/file diagram:



- **LoginInfo**
  The **LoginInfo** class is used to store and read the login info of the user, it will generate a file .session for storing infomation to implement auto login.

> **Our Login logic**:

> 1. **Sign up**: After the user input the password, we will first **generate a random salt** append to the password, and then use the **hash algorithm to encrypt** it. And sent the encrypted password and its salt to the database.
> 2. **Login**: The program will fetch the **encrypted password and salt** from the database, and hash the password entered by the user append by salt. **If the password after hash is the same, then login success**.
> 3. **When login success**: The program will **generate a session id and its corresponding expire time**, store it in the local ".session" file and also the database.
> 4. **Auto Login**: When the next time user open the login window, the program will detect the local ".session" file and **compare it will the session id in the database**, if the **session is the same and not expired**, user can be auto login.

> **Data members**:
>
> - static bool loginState: a bool varible used to indicate the login state
> - static int id: user's id, used as a nnique identifier
> - static QString username: username
> - static QString session: session id

> **Member functions**:
>
> - accessors and mutators: gets and sets
> - static bool isLogin(): get login state
> - static void saveLoginInfo(): save the login infomation to the local ".session" file
> - static void readLoginInfo(): read the login infomation from the local ".session" file
> - static void cleanLoginInfo(): clean ".session" file (used when session expired or the session id is not correct)

- **LinkedList<T>**
  The **LinkedList<T>** class is a template class of linked list, used to store our various data, such as cards and MySQL query datas

> **Data members**:
>
> - Node<T> * head: the head of the linked list
> - Node<T> * tail: the tail of the linked list
> - int length: the length of the linked list

> **Member functions**:
>
> - LinkedList<T>& add(T * t): add a element to the tail (return reference for implementing cascading call)
> - LinkedList<T>& remove(int index): remove a specified element (return reference for implementing cascading call)
> - T * get(int index): get a element by index
> - int getLength(): get length of the linked list
> - T * operator[](int index): operator overloading, get(index)
> - void operator+=(T t): operator overloading, add(&t)

- **APIs**
  There are two **APIs** we used in our project, one is translate API, another is keyword extraction API.

  > **Functions**:
  >
  > - QString translateAPI(QString text, QString target = "en", QString source = "zh"): Call translation API, will be used when add/edit "word" type cards for auto translation
  > - QList<QString> textRazorAPI(QString text): Call the keyword extraction API, will be used when add/edit a "hollow text" type card for auto selecting keywords

- **tools**
  Some commonly used **tools** in programs

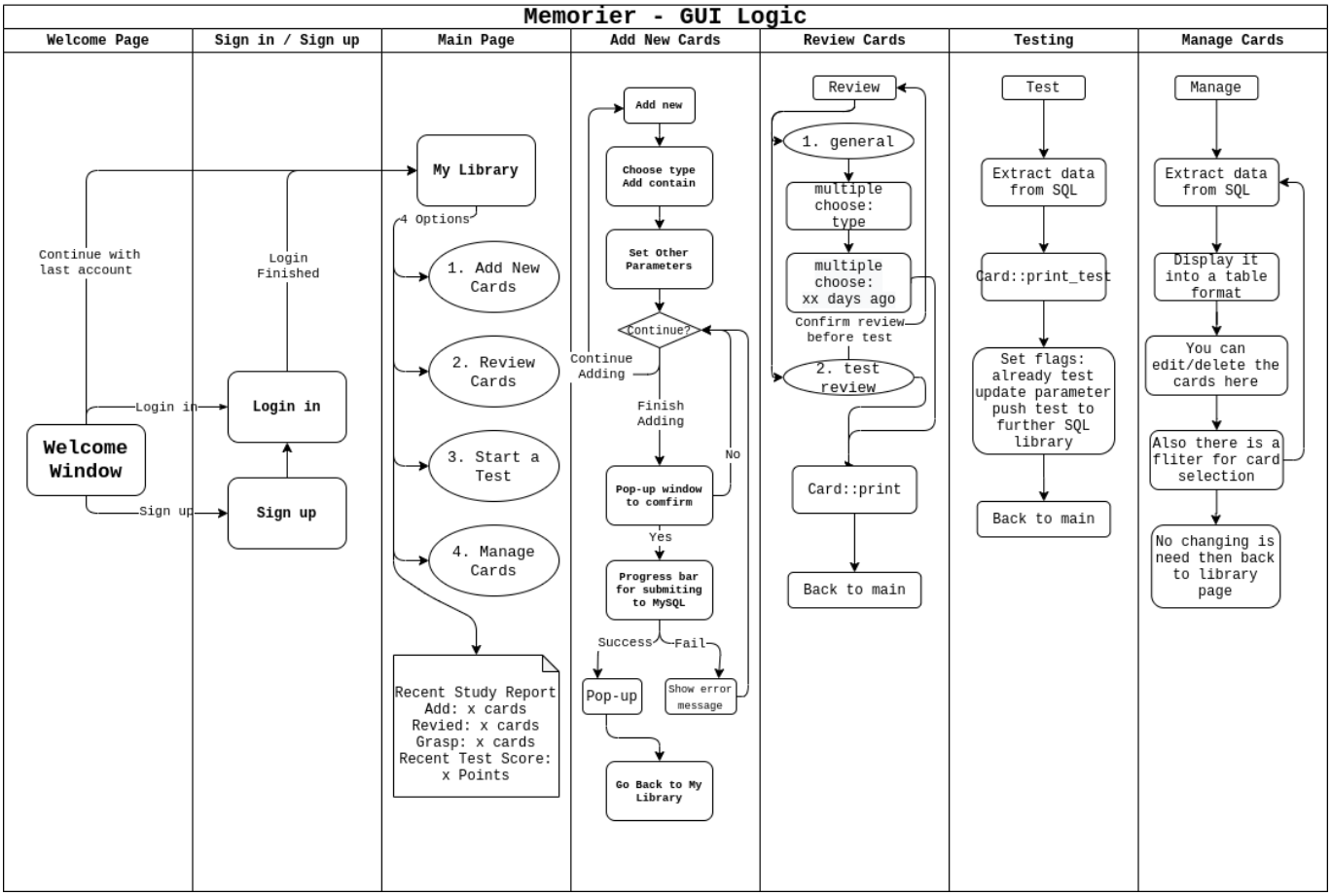
  > **Functions**:
  >
  > - QString getRandomString(int length): generate a random string with certain length, will be used when generate salt and session
  > - bool InsertLog(int id, QString recording): store the use log to the database, like, login, add card...
  >
  >   > we have a sql table for storing users' log, all the login and card activities will be recorded for safety. For detial please refer to 5. SQL data structure.

- **selectdate**
  This Qt Dialog class is used to select cardtype and timeperiod when generating review and test list.

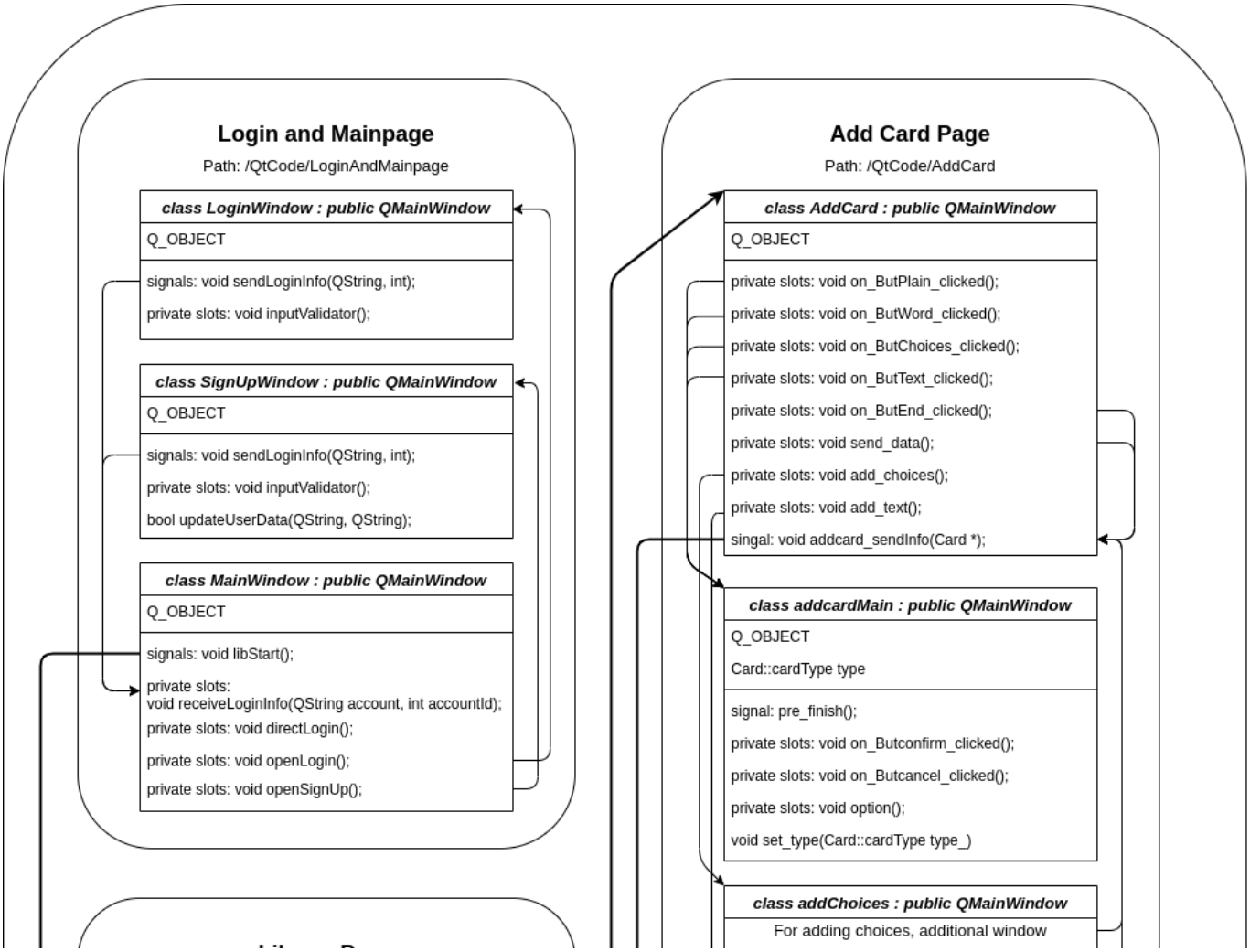
  > **Member functions**:
  >
  > - Butfinsh_clicked(): generate a QString type selection data and stored in "static QString selection"
  > - void closeEvent(QCloseEvent): together with forceQuitFlag to handle with the close event, make sure the current activity will stop if the user do force quit
  > - static QString get_selection_SQL(): translate the selection data to SQL-style QString. It can be attached to SQL query string directly.

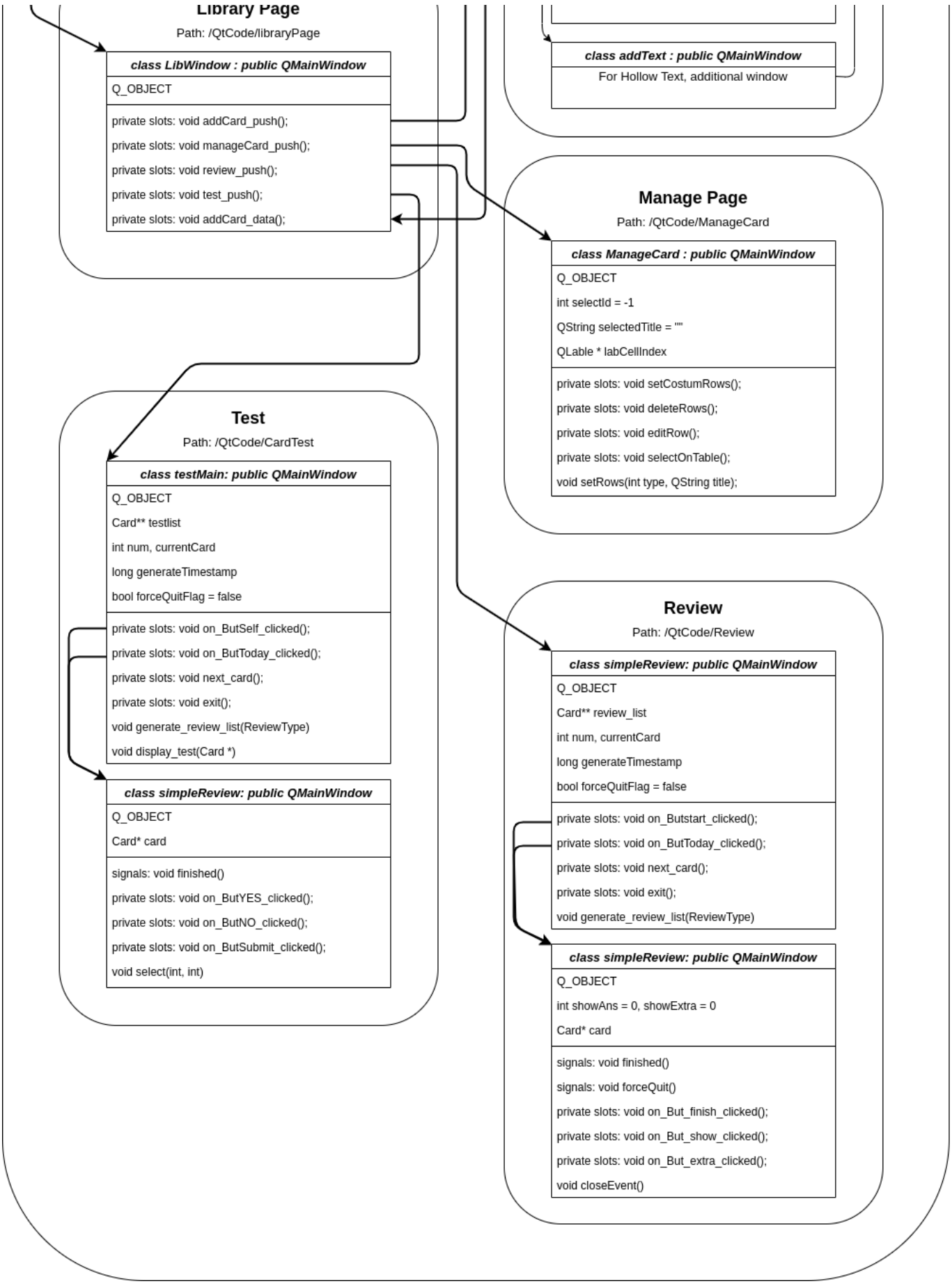
III. GUI(Qt) classes and GUI login

Basic GUI logic:



Class diagram:





The pages we use:

- Login page
  This page is used for login or sign up your account

Before reading the explanation of each function, please make sure you understand our login and sign up strategy:

> **Our Login logic**:
>
> 1. **Sign up**: After the user input the password, we will first **generate a random salt** append to the password, and then use the **hash algorithm to encrypt** it. And sent the encrypted password and its salt to the database.
> 2. **Login**: The program will fetch the **encrypted password and salt** from the database, and hash the password entered by the user append by salt. **If the password after hash is the same, then login success**.
> 3. **When login success**: The program will **generate a session id and its corresponding expire time**, store it in the local ".session" file and also the database.
> 4. **Auto Login**: When the next time user open the login window, the program will detect the local ".session" file and **compare it will the session id in the database**, if the **session is the same and not expired**, user can be auto login.

- **LoginWindow** class:
  Used for login window
  - signals: void sendLoginInfo(QString, int): will be emited after the user login successfully. The signal is received by MainWindow
  - private slots: void inputValidator(): will be triggered by user input, to check whether the input is vaild.
- **SignUpWindow** class:
  Used for sign up an account
  - signals: void sendLoginInfo(QString, int): same as above
  - private slots: void inputValidator(): same as above
  - bool updateUserData(QString, QString): add encrypted information to the database
- **MainWindow** class:
  Welcome page (I know the class name is bad, but I am too lazy to change it XD)
  - signals: void libStart(): start the library(main) page
  - private slots: void receiveLoginInfo(QString account, int accountId): will be triggered by success of login, save the session file to the database and local ".session" file
  - private slots: void directLogin(): handle auto login, will read and check the ".session" file
  - private slots: void openLogin(): triggered by push button, open corresponding window
  - private slots: void openSignUp(): triggered by push button, open corresponding window

- Library page
  This page is used for main page of Menorier, all your operation after login will begin here
  - **LibWindow** class:
    - private slots: void addCard_push(): will be triggered by push button, open add card page
    - private slots: void manageCard_push(): will be triggered by push button, open manage card page

  - private slots: void review_push(): will be triggered by push button, open review page
  - private slots: void test_push(): will be triggered by push button, open testing page
  - private slots: void addCard_data(): will be triggered by adding one valid card, store the card info to database

- Add page

  This page is used for adding new cards

  This was our fisrt page, so a some design are quite naive and were abandoned in other classes. In this page, a global variable "extern Card* __card" is used to store data. The logic flow of this function is quite simple, I will use the number to represent the execution sequence

  - 1. **AddCard** class:

    Choose a card type

    - signals: void addcard_sendInfo(Card*): Return the data
    - private slots: void on_ButPlain_clicked(): open addcardMain and call "set_type(Plain)", connect: "pre_finish()" -> "send_data()"
    - private slots: void on_ButWord_clicked(): open addcardMain and call "set_type(Word)", connect: "pre_finish()" -> "send_data()"
    - private slots: void on_ButChoices_clicked(): open addcardMain and call "set_type(Choices)", connect: "pre_finish()" -> "add_choices()"
    - private slots: void on_ButText_clicked(): open addcardMain and call "set_type(Text", connect: "pre_finish()" -> "add_text()"
    - private slots: void on_ButEnd_clicked(): emit signal and close Window
    - private slots: void add_choices(): open addChoices window, connect: "second_finish()" -> "send_data()"
    - private slots: void add_text(): open addText window, connect: "second_finish()" -> "send_data()"

  - 2. **addcardMain** class:

    Let user input main (or even all) card data

    - void set_type(Card::cardType): Initalize the window accroding to the parameter
    - signals: pre_finish(): mark the end of this window
    - private slots: void on_Butconfirm_clicked(): Transform data into __card, emit signal, close the window
    - private slots: void on_Butcancel_clicked(): Discard all data, close window
    - private slots: void option(): change display in answer QTextEdit accroding to __card->option(2)

  - 3(a). **addChoices** class:

    For Choices card only, let user input multi choice data

    - signals: void second_finish(): mark the end of this window
    - private slots: void on_Butadd_clicked(): set a new line of QTextEdit and QCheckBox visible
    - private slots: void on_Butfinish_clicked(): Transform data into __card, emit signal, close the window

  - 3(b). **addText** class:

    For hollow text card only, let user make some hollow

    - signals: void second_finish(): mark the end of this window

  - private slots: void on_Butfinish_clicked(): Transform data into __card, emit signal, close the window
  - void on_Butadd_clicked(): mark the text under QTextCursor
  - void on_ButAPI_clicked(): call API and generate a list of keywords / switch to next keyword
  - void on_Butauto_clicked(): match the string in API display box and the problem, mark all that matched
  - private: void selection(QTextCursor cur): mark the text under cur

- Review page **This page is used for review**
  - **reviewMain** class:

    Use for choose review by type or by time and generate review list accrodingly
    - enum ReviewType {Today = 0, Ramdom = 1};
    - Card** review_list: storage the review list
    - int num, currentCard: the length of the review list / the card index we are in now
    - void generate_review_list(ReviewType): generate review list accroding to review type
    - private slots: void on_Butstart_clicked(): call "generate_review_list(Ramdom)", open simpleReview for first card and connect: "finished()"->"next_card()"
    - private slots: void on_ButToday_clicked(): call "generate_review_list(Today)", open simpleReview for first card and connect: "finished()"->"next_card()"
    - private slots: void next_card(): update card review-related info, open simpleReview for next card and connect: "finished()"->"next_card()"
    - private slots: void exit(): tackle force exit
  - **simpleReview** class:

    The review window
    - public: void init(Card*): set all box accordingly
    - signals: void finished(): mark the normal end of this window
    - signals: void forceQuit(): mark force quit
    - private: int showAns, showExtra: flag for whether the box is showing ans / extra info
    - private signals: void on_But_show_clicked(): show/hide answer
    - private signals: void on_But_extra_clicked(): show/hide extra info
    - private signals: void on_But_finish_clicked(): emit finished() and close window
- Test page
- **testMain** class:

  Use for choose test by type or by time and generate test list accrodingly
  - enum TestType{Self, Today};
  - Card** testlist: storage the test list
  - int num, currentCard: the length of the test list / the card index we are in now
  - void generate_testlist(TestType): generate test list accroding to test type
  - private slots: void display_test(Card*): open simpleTest and connect: "finished()"->"next_card()"
  - private slots: void on_ButSelf_clicked(): call "generate_testlist(Self)", and call display_test
  - private slots: void on_ButToday_clicked(): call "generate_testlist(Today)", and call display_test

    - private slots: void next_card(): update card test-related info and call display_test
    - private slots: void exit(): tackle force exit

- **simpleTest** class:
  The test window
    - signals: finished(): mark the end of this window
    - private: void select(int,int): mark the text between two indicators
    - private slots: void on_ButYES_clicked(): update card info, close window
    - private slots: void on_ButNO_clicked(): update ui->ButNO condition / uodate card info / close window
    - private slots: void on_ButSubmit_clicked(): call card->option(9/10) to collect and apply side-to-side compare info
- Manage page
  This page is used for manage your existing cards, you can delete/edit them, also there is a fliter for you to select cards

    - **ManageCard** class:

      data member:

      - int selectId = -1: indicting which row is selected
      - QString selectedTitle = "": the title of selected row
      - QLable * labCellIndex: the label for GUI, showing the selection info to user

      member functions:

      - private slots: void setCostumRows(): will be triggered by the filter, display the cards by user's request
      - private slots: void deleteRows(): will be triggered by push button, delete the selected row from database, and refresh the table view
      - private slots: void editRow(): will be triggered by push button, open edit page(in progress)
      - private slots: void test_push(): will be triggered by push button, open testing page
      - private slots: void selectOnTable(): will be triggered by selection on table, change the selectId and selectedTitle, also the label
      - void setRows(int type, QString title): query from the database, get the card data needed to set the table, type and title are filter conditions

4. Brief description of program design

I. Use of OOP constructs and techniques

Storing and using data based on inheritence and function overloading

- A series of classes inherited from the base class (Card) is applied to simplify coding.
- All managing-related data that is useful for all cards are stored in the base class. Core codes can use these data to easily generate card series using different criteria, and it is no need (and should not) to consider what specific type is it.

- All card type-related functions and data are stored in derived classes. Using function overloading and the powerful QString class, one version of core codes is enough to implement all necessary procedures. Specifically, 4 different cards have different “answer type” or even structure to store data, it is tedious to implement the almost-the-same procedure for 4 times in core codes. Therefore, all 4 classes are designed to follow a same protocol to package its data and after sending it to core codes, only one unpackage procedure is needed to apply and data in different types will be transformed into a unified format and can display easily.
- Through this design, it is very easy to reuse the code or doing further development. More on Part III code reusability

Dispersed and well-packed coding

- As it may cause low efficiency and very likely to go into chaos if two developers kept editing the same file or same function repeatedly, or make the file very large and cause confusion, we tried to disperse the coding mission into small parts and finish them one by one. It turns out that this greatly reduced workload of the project.
- In considering the two developers’ expertise in different areas, it is decided that all code will be well packed. The other developer can use a pre-set function as a port to easily gain all data who needs, and should send data in a specific format to a . It increases working efficiency and make it easier to reuse the code and do further development.

II. Use of Data Structures

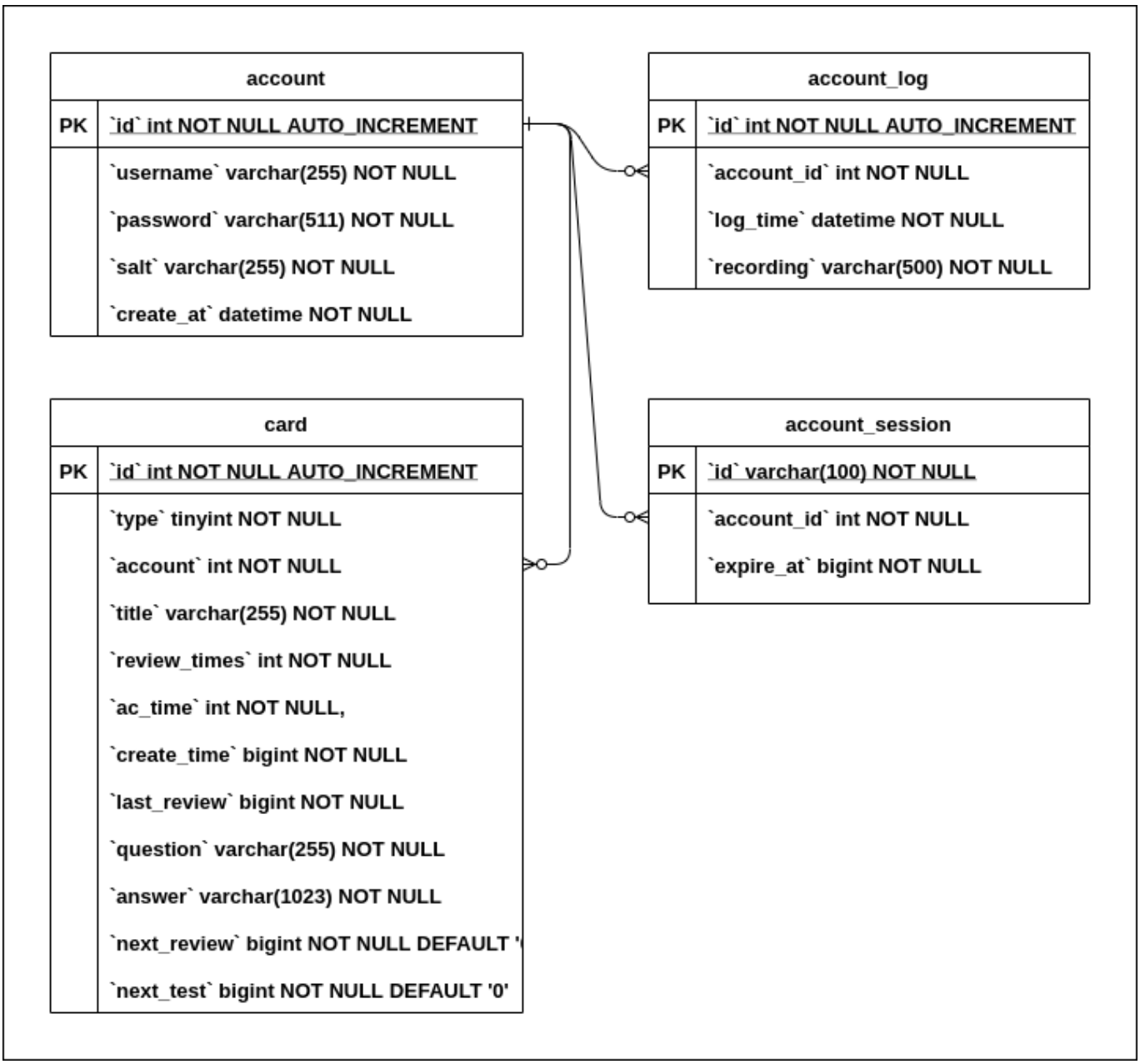
- **Vector**: in card classes to store additionl card info
- **SQL**: more on part 5;
- **Dynamic (pointer) arrays**: to save card lists
- **Linked list**: more on part 3.II
- **splay tree (abandoned)**: in the origin plan, we will implement a user-frendly lightweight class that support dynamic insert, remove, query, sort, merge, etc. using splay tree. only a little non-OOP style test code was produced.

III. Code Reusability

- The code is very easy to reuse / further develop
- To create new card type, it should be derived from Card class or Plain class, and add more feature in "virtual function option()", almost no need to change other core codes.
- To create new UI window, you can add ui programs in AddCard/Review/CardTest and call them through each "main program", very little edit to core codes is enough

5. SQL data structure

SQL structure diagram:



The SQL database structure is exported to the "MySQL_Config" folder, you can check it there

- **account**: store the account infomation, the password here is encrypted. And the id is unique identifier of the user.
- **account_log**: store the log of every user's activities, including login, add, delete, edit, review, test.
- **account_session**: store the session id and its expire timestamp, used for auto login and security.
- **card**: store all the card info and its attributes here.

6. Future plan

Yes, This Project Will Continue Even After the Project Period

- Beautify the UI and add cutscenes or more animation
- Add image uploading function in cloud server
- Add setting page (User can config their review time and rules)
- Ported to mobile phone program, Optimize the touch screen user experience
- Add label feature to classify cards, let users label their cards for easy sorting and review