NETB380 Programming practice

Variant 5: Expense Manager

Kristiyan Stanislavov F74234
Stanislav Atanasov F60191
Georgi Ovcharov F79137

08 February 2021
Sofia, Bulgaria

Program description

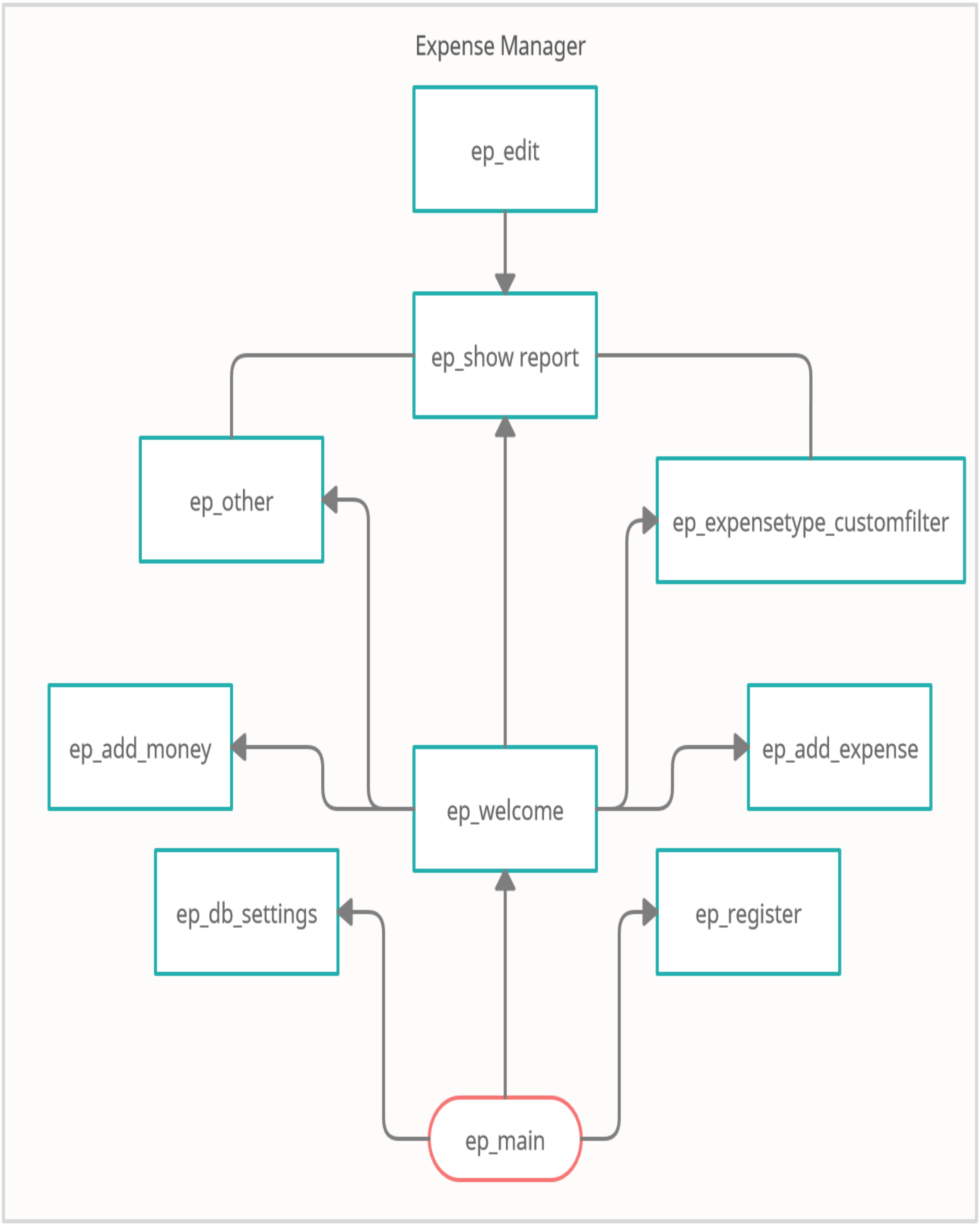
Using C++/Qt implement a program that helps in creation of expense reports and expense statistics for managing daily/monthly/yearly expenses. The program must be able to take as input the persons expenses and to create and arrange the different reports.

The program must have the following features:

1. GUI that allows:
    1. The user to put his expenses.
    2. The user to assign his query.
        1. For e.g.:
            1. Create report for monthly expenses for 2020.
            2. Create report for daily expenses at "02.11.2020".
2. Create daily/monthly/yearly reports based on the user query and the available data into the database.
3. Full database support for the data needed to create the reports based on the persons expenses. DBMS must be PostgreSQL.

Content

Application map



Overview of the solution

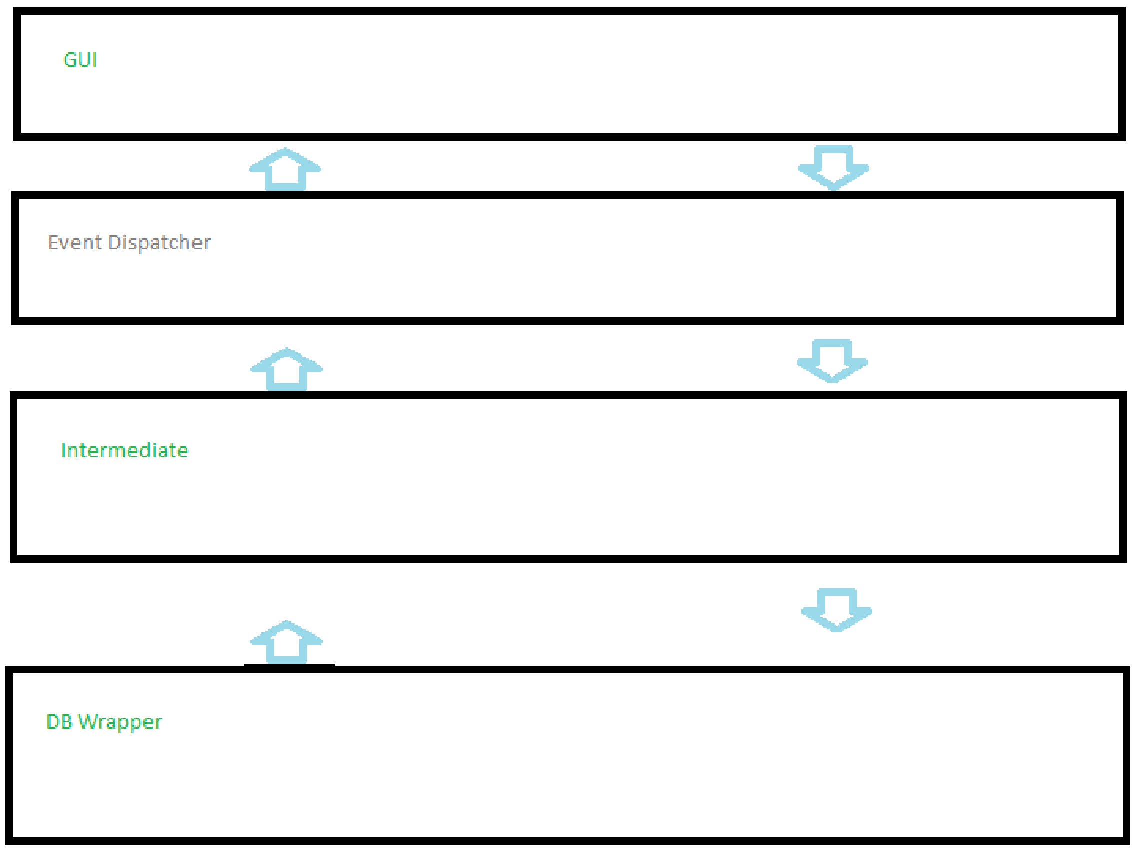
Our approach is to divide the software into three layers. This will allow us to have good interoperability between different databases and keep the same logic. There are three main layers: GUI, Intermediate and DB Wrapper. All of the layers are containing breakdown with different classes.

GUI layer is consisted of all graphical user interface objects such as windows, buttons and all user visible part of the program.

Intermediate layer will handle requests to the database from GUI objects. It will also contain the logic necessary to process user provided data to the relevant input data for the different DB API's.

DB Wrapper will provide generalized API's to the DB. Connection to the database will be handled by this layer.

Between Intermediate and GUI layers we have one abstract layer that will be called Event Dispatcher. This layer will be used to make connection between GUI objects and their actions and the Intermediate layer that will forward or receive data from DB to achieve the desired action. Furthermore, this layer will take advantage of the integrate signal-slot system of Qt framework.



GUI Layer

This layer will consist of every possible graphical user interface object or as we have said earlier, it will contain all visible part of the program for the user. It will provide the user with the ability to interact with the program by either forwarding inputs or by displaying outputs to the user. In addition to that, every GUI specific logic will be handled by its class. This covers all input validations, all data population depending on the widgets inside and all data display logic. This will allow us to easily add, remove or modify specific functionality.

Every GUI object that needs data from DB will inherit the base class "**EP_BaseClass_GUI_ReportMain**". This will allow us to interact with the Intermediate layer without direct connection. Event dispatcher will provide signals definitions that will be connected to the specific GUI object slots if needed.

All of the different GUI objects are appearing or closing depending on the user desire/action.

For e.g the program always starts by showing ep_main class object and from this point the user can go to:

- DB connection settings window
- Registration window
- Simply log-in into program by providing Log-in data and pressing Log-in button.

Into this layer we have the following classes:

- ep_add_expense
- ep_add_money
- ep_custom_menu
- ep_customlabel
- ep_db_settings
- ep_edit
- ep_expensetype_customfilter
- ep_other
- ep_register
- ep_show_report
- ep_welcome
- ep_main

Class ep_main

This class is representing the main window of the program. It is designed by the Qt Creator. It has the following properties:

- Layout: Vertical
- Default size: 345x333

- Icon: money icon
- Title: Expanse Manager

This window is the first thing the users sees. The first thing that catches the user's attention is the red colour of the DB Configuration button. When launching the application for the first time, it is necessary to connect it to the database. By clicking the button, a new window appears with the DB settings.

Ep_main has a conviniently placed log in group box for the users to sign in. Each field has placeholders aiding to the intuitivity of the program. Clicking the button checks if any of the fields are empty. If they are, a messagebox appears stating the situation. In case they are not, it proceedes to check if the username and password match any of the registered users in the database. A successful log in leads the user to the "welcome" window.The password LineEdit box has the echoMode set to password, hiding the actual input on the screen to prevent password theft.

In case there is no created account yet or a new user wants to use the program with a different account there is the Create a new account button.

Class ep_db_settings

This class is used to connect the application to the database. It has the followng properties:

- Layout: Vertical
- Default size: 251x425
- Icon: money icon
- Title: DB Settings

Similar to the ep_main window, this one asks for credentials for the user to provide.

Firstly, it is necessary to provide a host address, username, password and name of the database. The password field is hidden like in ep_main. After successfully completing the fields and pushing the connect button, the user is greeted with a messagebox infroming that the connection to the database is completed.

After that, the tables from the database need to be deployed in the application by clicking the Deploy tables button.

In case a user wants to have the information from the database deleted, they need to press the Drop tables button.

Class ep_register

The class functions as a way to create new accounts. It has the followng properties:

- Layout: Vertical
- Default size: 256x326
- Icon: money icon
- Title: Register form

The ep_register window is a simple window with all its elements grouped in a single groupbox. The credentials needed in order to create a new account are a username, a password, repetition of the password to ensure the user has not miswritten the password the first time and an e-mail address.

The application has input validation that checks every LineEdit box and displays a messagebox if there are any incorrect inputs and shows the errors. In the username section it checks whether the field is empty or a user with the same username already exists. A check of both passwords is done as well and if they do not match, respectively a message is shown.

Class ep_welcome

Ep_welcome is the core of the application. From there the user can check everything the program is capable of. It has the followng properties:

- Layout: Grid
- Default size: 844x558
- Icon: money icon
- Title: Expense Manager

The first thing the user sees in this window is a big greeting message with their username. Down below is shown the current balance of the user, which can be either positive or negative.

Under that there are two buttons that lead to two other windows. The add money window is used to increase the balance of the current user, while the add expense has the purpose of adding deploying information of an expense the user adds.

The expenses are divided into two ways – by date and by type. The type is decided and given by the user when they create the expense in add expense.
On a horizontal line, the user can decide to search by date by the following criteria:

- Today – today's date
- This week – last 7 days
- This month – last 30 days
- Last year – last 365 days
- All time – all expenses made whenever
- Custom – pops up a window to select a custom interval of time

The type selection is pretty similar. It consist of five main types – Transport, Food, Clothes, Utility, Bank and Other. The Other type is manually typed by the user in the type field in add expense.

On the top of the window are conviniently placed today's date and the current time for easy use.

Class ep_add_money

Ep_add_money has the task of increasing the user's balance. It has the followng properties:

- Layout: Horizontal
- Default size: 584x54
- Icon: money icon
- Title: Add Money

Consists of a doublespinbox that asks for a positive number, a dateEdit to specify the date of increasing the sum of the balance and a push button to confirm the changes. Alignment of the sum added is to the right to suit more aesthetically pleasing looks. The locale is changed to C, since the dot on the numpad could change to a comma depending on the OS language the user has chosen. This eliminates the annoyance of having a comma in the place of a decimal point.

Class ep_add_expense

Ep_add_expense has the task of placing an expense in the database. It has the followng properties:

- Layout: Grid
- Default size: 602x166
- Icon: money icon
- Title: Add Expense

Ep_add_expense has the following fields:

- Name of expense – what the expense is about
- Type of expense – choose one of the five main types or add a new type manually
- Price – positive floating point number, aligned right, changed locale for floating point issue
- Date – set on current date by default
- Description – a short description of the expense can be added

Class ep_show_report

Ep_show_report outputs the user's query request. It has the followng properties:

- Layout: Vertical
- Default size: 720x441
- Icon: money icon
- Title: Type of report: *(depending on the query)

Whenever a button requesting a query from the user to show something ther user wants, the ep_show_report window pops up.

In the top part it shows the user's username and the current date and time, helping the user to understand the report better.

Below it is the report itself. Depending on what the user's request for sorting the expenses had been, all the queries meeting the criteria are shown in a table.

Class ep_edit

Ep_edit lets the user edit information on an expense. It has the followng properties:

- Layout: Vertical
- Default size: 602x168
- Icon: money icon
- Title: Edit expense

By right-clicking on an expense in the ep_show report window, the user can activate the edit function. The window is similar to ep_add_expense with the only difference it edits a previously added expense, instead of adding a new one. The push button here says save instead of add.

Class ep_expensetype_customfilter

Ep_expensetype_customfilter lets the user choose a custom type they want to have a report on that does not figure in the ep_welcome window (a manually added user type) . It has the followng properties:

- Layout: Grid
- Default size: 242x109
- Icon: money icon
- Title: Select type

Consist of a dropdown combobox with all the existing types. When a type is selected, the user needs to press the apply button and the ep_show_report appears with the custom filter query.

Class ep_other

Ep_other lets the user choose a custom date they want to have a report on. It has the followng properties:

- Layout: Vertical
- Default size: 239x168
- Icon: money icon
- Title: Select date

This window has two datetimeedits, the first of which is the one which sets the beginning of the search and the second one sets the end time. The two fields have a centered alignment. On clicking the search button the ep_show_report appears with the custom filter query.

Event Dispatcher

Event dispatcher is consisted of only one class that will be used for its signal definitions. These signals are part of the Qt framework. They will be connected to different slots from either the GUI layer or the Intermediate layer. These signals will be emitted only when specific requests need to be done. For example, database connection request when user presses connect button into the DB Settings window.

Into the program execution there will be only one event dispatcher object. This object will be passed as pointer to every new GUI object and will be used by the main object of the intermediate layer. This will permit us to use the integrate signal-slot system of Qt framework.

Every signal is well commented and also the name is self-explanatory.

Intermediate

Intermediate layer is responsible to process GUI objects request and provide data to DB and vice versa.

Here all logic necessary to complete the request is made. For e.g. DB API for adding expense requires that the expense group id that is available into the table of expense groups and user provides only a name of the group. We also keep all additional logic for post-processing here. Intermediate layer is also responsible to extract the data from DB when user request specific report to be generated. Both intermediate layer and GUI layer are inheriting base class "**EP_BaseClass_GUI_ReportMain**" which is allowing us to have connection to the same object of type Event Dispatcher and User Data.

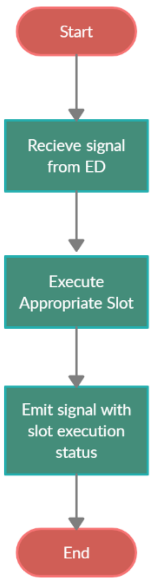
Intermediate layer is consisted of two classes:

- EP_Report_Main
- EP_User_Data

Class EP_Report_Main

This class is consisted mainly of public slots that are connected to Event Dispatcher signals. These signals will be emitted by the GUI objects and will be processed by the EP_Report_Main object. We are having only one EP_Report_Main object as it is sufficient to process the requests. Furthermore, we are allowing the integrated signal-slot system of Qt framework to take care about concurrency of the signals-slots executions. This system also allows us to transmit specific information from the signal to slot which is giving us the possibility to gather the data provided by the user and forward it to lower layers.

Find below chart of every slot execution of EP_Report_Main.



This kind of architecture is allowing us to work independently of the current state of the program and perform specific action only on request.

We have the following slots in EP_Report_Main:

- Main_UI_Slot
  - EP_ReportMain_GetUserLogInStatus();
- Register_UI_Slots
  - EP_ReportMain_GetUserDataRegisterStatus();
- DB_Settings Slots
  - EP_ReportMain_DeployTableInCurrentDB();
  - EP_ReportMain_DropTableInCurrentDB();
- Welcome screen slots.
  - EP_ReportMain_ProcessReport(EP_Report_Types TypeOfReport, QList<QString> dataToProcess);
  - EP_ReportMain_Update_activeUserData();
  - EP_ReportMain_Update_activeUserExpGroups(int TypeOfReq);
- Edit window
  - EP_ReportMain_UpdateExpense(QList<QString> rowData);
- Add expense window
  - EP_ReportMain_AddExpense(QString nameOfExpense, QString typeOfExpense, QString amountOfExpense, QString descriptionOfExpense, QDateTime date, int ExpType);
- Internal slots
  - EP_ReportMain_OpenDBConnection(int idOfRequest);

All of them are providing different functionalities for the different GUI objects.

Class EP_User_Data

This class is consisted of several members that are holding specific user data. All of the members have setters and getters. They are consisted of:

- Registration details
- Log-in details
- Current user data from DB table “user_accounts”
- Current user available expense groups from DB table “expense_groups”

There are some corner cases when we are skipping ED layer and GUI objects are directly populating data into the EP_User_Data object. In these cases, we need to have user information before specific request to DB is made from EP_Report_Main. Same as EP_Report_Main we have only one object of type EP_User_Data that is created into the main part of the program. EP_User_Data is not inheriting EP_BaseCLass_GUI_ReportMain as it is one of the objects that this base class is providing access to.

DB Wrapper

DB Wrapper class is intended to facilitate the communication between the DBMS and the application.

It implements basic DB connectivity, direct writing to and reading from the database, as well as user authentication and level management.

The idea behind the DB structure was to provide better all around ACL (access control layer), intended for multi-user interactions and implementing such things as expense and income groups, different currencies and account types.

The DB Wrapper was developed as core implementation layer, that needs to provide compatibility and translation with the base DB, which can be used with different endpoints.

Most of the responses from this level are either int, bool or QList of QLists of QString – one of he better multi dimensional array implementations in QT.

Structure

void openDB();

This method accepts Qstring parameters – hostname, username, password and dbname.

It's role is to create the initial DB connection and to check DBMS credentials.

void closeDB();

This method is used if we need to close the DBMS connection

int isValid();

This method accepts int parameter user id and is used to check if the given user id is valid and active in the DB.

Returns 0 for valid user and negative value depending on the error that occurred:

void deployTables();

This method is used to deploy the initial DBMS structure creating all required tables and setting some default values.

void dropTables();

This method is used to drop all DBMS tables. Useful for destroying all info from the DB.

bool isDBOpen();

Method used to check if e have a working DBMS connection.

int registerUser();

Method accepting QString parameters username, password, email and used to register new user.

Returns 0 for OK registration and different int depending on the error

int loginUser();

Method that accepts parameters of type QString – username and password that is used to authenticate the user.

Returns user id integer and value bellow 0 if there is error.

int addAcountType();

Method that accepts QStrings type and description and adds new account of that type.

Returns 0 for OK and bellow for error.

QList<QList<QString>> getAccountTypes();

Method that returns QList of QLists of QString of all account types in the DB.

int updateUserAccount();

Method used to update the user account – accepts int parameters aid for account ID to be update, new account type ID and new currency id, as well as QStrings – new name and new description.

Returns 0 for ok and bellow for error.

int addCurency();

Method for adding new currency, accepting ISO and longname QString parameters for currency ISO and long name.

Returns 0 for ok and bellow for error.

QList<QList<QString>> getCurrencies();

Method to retrieve all currencies in the DB.

Returns QList of QLists of QStrings

int addExpenseGroup();

Add new group for the expenses or income. Accepts int for the user id and QStrings for name of the expense and description.

Returns 0 for ok and bellow for error.

QList<QList<QString>> getExpenseGroups();

Method that returns QList of Qlists of QStrings of all the (int) user id groups for expenses.

int addUserAccount();

Method user to add new user account accepts int for user id and currency id, double for the amount and QString for name and description.

Returns 0 if user account is added OK and bellow if error.

QList<QList<QString>> getUserAccounts();

Returns QList of QLists of QStrings for all user accounts and accepts User ID as parameter.

int addExpense();
Method used for adding new expense in the db. Accepts int for user id, account id, expense group, date and type of the action (0 for expense, 1 for income).

Returns the standard 0 for OK and bellow depending on the error.

QList<QList<QString>> getExpenses();
Method used for getting all user entries in the expense table. Accepts integer for user id, account id, type, group, time and limit, double for amount and QString for different search parameters.

This method allows for different filters to be applied on the request, so we can limit the response.

Returns QList of QLists of QStrings

int updateExpense();
Method used to update the given (int) expense ID. Accepts (int) new account ID, (double) new amount, (int) new group, (QString) new description, (int) new add date and (QString) new name.

Returns 0 for OK and bellow depending on error.