Progetto “Accountability”

Anno Accademico 2020/2021

Simone Lucato (matr. 1217322) Elia Scandaletti (matr. 1216751)

Introduzione

L’applicazione si occupa di modellare un sistema di gestione di conti, sia personali sia non. Infatti l’utente, previo accesso tramite login o iscrizione, troverà da un lato i suoi conti personali, dall’altro i gruppi a cui appartiene, entrambi ordinati per ultima attività. I conti personali comprendono registri in cui annotare entrate e uscite. In entrambi i casi è possibile aggiungere movimenti di denaro, segnando descrizione, importo ed eventuali note. I gruppi sono formati da un insieme di utenti e possono dare origine alle casse comuni, per la gestione di denaro all’interno di un gruppo di persone. Qui si possono effettuare raccolte fondi (per raccogliere una somma di denaro da ogni membro della cassa comune) o spese comuni (di interesse comune, con parametri aggiuntivi per spese personali o aggiunte di tasca propria alla spesa), oltre a tenere monitorate il bilancio di ogni individuo, il punto chiave dell’applicazione.

Vincoli tecnici

L’applicazione è stata sviluppata e testata nel seguente ambiente:

- Qt Creator 4.5.2 basato su Qt 5.9.5
- Sistema operativo: Ubuntu 18.04
- Compilatore: gcc 7.5.0

Suddivisione del lavoro e conteggio ore

Il progetto è stato sviluppato con Elia Scandaletti (matr. 1216751). Dopo una breve parte di analisi preliminare del problema (in quanto l’idea del progetto è sorta da necessità comuni avute l’anno scorso), ci siamo suddivisi il lavoro nel seguente modo: io mi sono occupato della GUI e del Controller mentre Elia ha gestito la parte del Model. La stima personale delle ore è impegate è la seguente:

- Analisi preliminare del problema: 2 ore
- Progettazione GUI: 11 ore
- Apprendimento libreria Qt: 12 ore
- Codifica GUI: 59 ore
- Debugging: 13 ore
- Testing: 6 ore

Il totale è di circa 103 ore di lavoro. Il superamento delle 50 ore prestabilite è stato dovuto principalmente a due fattori: aver sottovalutato la difficoltà del progetto per relativa somiglianza tra molte schermate della GUI e la scarsa attenzione nelle prime ore di codifica a questioni fondamentali nelle applicazioni di Qt (parenting e resizing).

Gerarchie principali

Gerarchie nella vista

1. InnerPage

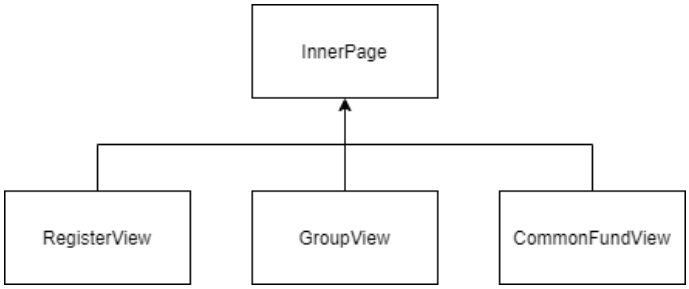

Figura 1: Gerarchia di InnerPage

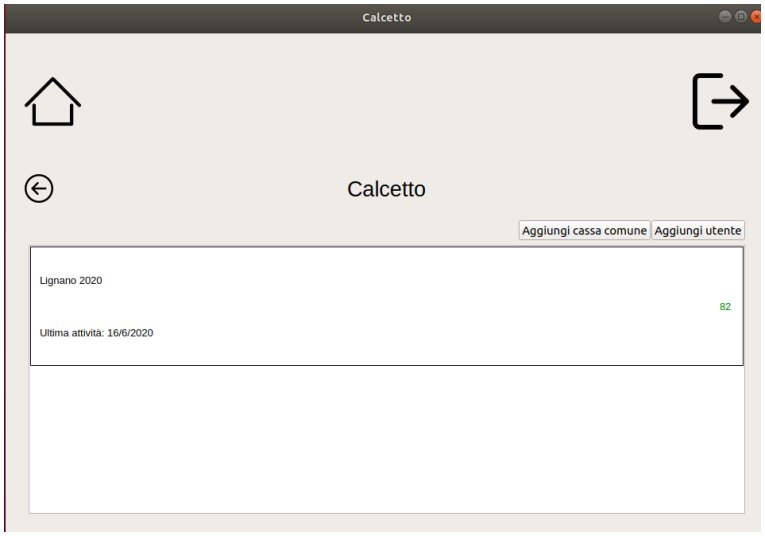

Figura 2: Group

`InnerPage` è la classe base che rappresenta la struttura base delle schermate successive alla prima: `QPushButtonAR` per tornare alla pagina precedente, `QLabel` che indica dove mi trovo e bilancio se previsto, `QHBoxLayout` con bottoni diversi a seconda della classe e `CustomListWidget` riempita da `Box` diversi a seconda dei casi. La classe base nel costruttore dispone i widget nel modo corretto e le classi figlie si occupano solo di inserire i relativi contenuti con appositi setter della classe genitore.
`RegisterView` si occupa della schermata di gestione conti personali, con apposito bottone per aggiungere movimenti.
`GroupView` si occupa della schermata di gestione di un gruppo: qui è possibile aggiungere utenti al gruppo, aggiungere casse comuni o vedere maggiori dettagli di una di queste.
`CommonFundView` si occupa di mostrare tutti i dettagli relativi ad una cassa comune. Sarà possibile aggiungere utenti, raccolte fondi (raccolta di una somma determinata da ogni persona che va a incrementare il bilancio complessivo), spese comuni (a tutte le persone presenti nella cassa comune in quel momento) ed entrate comuni (come ad esempio la restituzione di una cauzione). È inoltre possibile vedere il riassunto della situazione dei crediti di ogni individuo nella cassa comune.

2. ItemBox

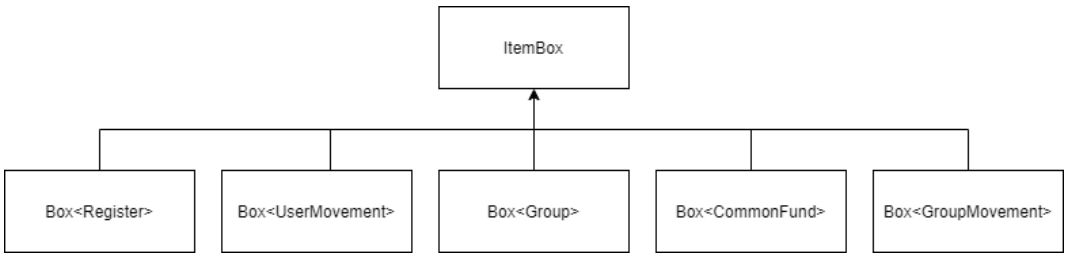

Figura 3: Gerarchia di ItemBox

`ItemBox` rappresenta la classe base dei Box interni a `CustomListWidget`, una classe derivata da `QListWidget` (più dettagli riguardo questa scelta nella sezione delle "Scelte progettuali"). Analogamente a `InnerPage`, anche in questo caso la classe base si occupa solo di creare i `QWidget` necessari alle classi figlie e di disporli in maniera corretta nei `QBoxLayout`. Ciascuna classe figlia di occupa di settare in maniera appropriata i QLabel della classe genitore.
Le differenze sono minime ma comunque significative: `Box<Register>`, `Box<Group>` e `Box<CommonFund>` hanno marcata la data di ultima attività ma mentre Registri e Casse Comuni hanno anche il bilancio attuale mostrato, i Gruppi no (come mostrato in Figura 4). Inoltre `Box<UserMovement>` e `Box<GroupMovement>` mostrano le medesime informazioni ma non sarebbe stato possibile costruirli dalla stessa classe comune AbstractMovement in quanto, per l'appunto, astratta.

3. Classi derivate da QWidget

C'è stata la necessità di creare due classi derivanti da `QWidget`: `QErrLabel` e `QPushButtonAR`. Il primo rappresenta tutti i `QLabel` in cui mostro messaggi di errore. Di conseguenza ho modificato lo stile dei `QLabel` standard cambiando il colore in rosso con `setStyleSheet`, cambiando il font con `setFont` e ho impostato di default il fatto che non fossero visibili con `setVisible`. Il secondo è utilizzato in tutti i bottoni in cui è presente solo l'icona ed era necessario fare il resizing della stessa, in quanto questo non avviene di default. Perciò ho fatto l'override di `paintEvent` per permettere questo.

Gerarchie nel modello

1. Gerarchia dei conti

La classe `AbstractAccount` è la classe base astratta degli account. Presenta alcuni campi caratteristici e relativi getter. Viene specializzata in `Register` e `CommonFund` per la rispettiva gestione di conti personali e casse comuni.

2. Gerarchia dei movimenti

La classe `AbstractMovement` rappresenta la classe base astratta dei movimenti. Contiene informazioni caratteristiche come l'importo, la descrizione, le note, il momento in cui è stato inserito. Viene specializzata in `UserMovement` che aggiunge il registro di appartenenza e `AbstractGroupMovement` che aggiunge la cassa comune di appartenenza e una `std::map<User, double>` per monitorare come variano i crediti degli utenti. `AbstractGroupMovement` si specializza ulteriormente in `Fundraise` e `CommonExpense`. La prima rappresenta una raccolta fondi in cui ogni utente aggiunge una somma al bilancio della cassa comune, la seconda rappresenta una spesa comune a tutti i membri della cassa comune.

Elenco delle chiamate polimorfe

La maggior parte dei metodi virtuali presenti all'interno del modello è dato dalle interfacce. Per questa ragione si riportano quindi solo i metodi virtuali significativi:

1. `std::time_t AbstractAccount::getLastActivity() const;` ritorna l'ultima attività svolta sul conto, movimento o la sua creazione
2. `double MovementInterface::calculateBalanceChange() const;` calcola la variazione del bilancio di un conto
3. `std::map<User, double> AbstractGroupMovement::calculateUserCreditChange() const;` calcola la variazione del credito individuale per il movimento corrente
4. `AbstractGroupMovement::AbstractGroupMovement *clone() const;` fa la copia profonda del movimento
5. `void DBMS::load(Model *m, std::string f) const;` carica il Model* m leggendo il file di salvataggio col nome f
6. `void DBMS::save(const Model *m, std::string f) const;` salva il Model* m sul file f

Inoltre `DBMS::Visitor` implementa numerosi metodi virtuali puri che consentono al `Visitor` di visitare specifiche classi non astratte.

Gestione I/O

Le funzionalità di input/output sono state implementate per salvare i dati di un utente in modo da poter tenere effettivamente una contabilità personale. Il formato che è stato scelto per memorizzare i dati nel file è JSON, inoltre sono state usate di supporto alcune classi offerte da Qt tra cui QFile, QJsonDocument, QJsonObject e QJsonArray.
Nel file di salvataggio vengono memorizzate tutte le possibili informazioni utili, tra cui:

- L'elenco di utenti presenti nel modello
- L'elenco di gruppi presenti nel modello
- L'elenco di casse comuni presenti nel modello
- I movimenti presenti nei registri e nelle casse comuni
- Tutti i dettagli dei movimenti (tra cui importo, descrizione, note, tempo di inserimento...)

Descrizione GUI

L'idea di base è avere una classe `View` che si occupa della gestione delle schermate e della comunicazione tra le classi interne e il `Controller`. Questa classe ha al suo interno i puntatori alle classi interne e all'occorrenza cambia la schermata (generalmente nascondendone una e mostrando quella di interesse). Nella `View` c'è un `QVBoxLayout` che come primo item ha un `Header` con i bottoni per uscire e tornare alla home page (quest'ultimo dopo aver effettuato il login). In questo modo la comunicazione è lineare ed efficace: le classi interne emettono segnali, raccolti dalla classe `View` e gestiti in slot interni o collegati direttamente al `Controller`.
Una soluzione alternativa poteva essere sfruttare il polimorfismo e creare un'unica classe genitore per ogni classe interna e tenere nella classe `View` un unico puntatore alla classe base e modificare il suo tipo dinamico di volta in volta. In questo modo, la comunicazione diventerebbe più complessa perché Qt non supporta l'uso del tipo dinamico nell'emissione di segnali e slot.

Login

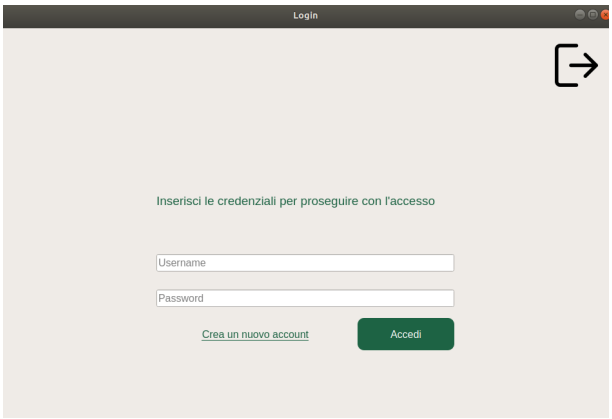

Figura 4: Login

La finestra di `Login` presenta una struttura abbastanza semplice: `QVBoxLayout` che contiene tutti i widget necessari al login, `QLabel` per la descrizione della finestra, `QErrLabel` per gli errori, `QLineEdit` per gli input da utente di username e password per l'accesso e due `QPushButton`, uno per l'accesso e uno per consentire l'iscrizione di un nuovo utente qualora non avesse ancora le credenziali.
Sono state apportate piccole modifiche di stile, come il cambio di font con `setFont`, l'inserimento di placeholders nei `QLineEdit` con `setPlaceHolderText` e modifiche a livello di stile nei `QPushButton` con `setStyleSheet`. Particolarmente interessante è l'istruzione `setStyleSheet("QPushButton{border-style:outset;}")` per dare l'illusione della rimozione del bordo dei `QPushButton`. Altro particolare è l'uso della proprietà `retainSizeWhenHidden` di `QErrLabel` per fare in modo che anche da nascosti tengano la loro dimensione originale, al fine di evitare spostamenti poco piacevoli di widget quando compaiono errori.
Al click di `inscriptionButton` viene emesso il segnale `showInscriptionBox`, raccolto dalla `View`, che costruisce la classe `Inscription` e la mostra come pop-up indipendente dalla vista precedente. Ho preferito creare una classe a sé stante invece di creare la finestra dallo slot di `Login` in quanto dal punto di vista logico iscrizione e accesso sono differenti, pur portando al medesimo risultato (l'accesso alla home page). Qui una finestra richiede l'inserimento di tutte le informazioni necessarie alla registrazione (nome, cognome, username e password). In caso di username già utilizzato o di campi lasciati vuoti compariranno appositi errori. Al click di `sendButton` invece vengono fatti diversi controlli: immediatamente si controlla vi sono campi lasciati bianchi, in caso compare un messaggio di errore e non si lascia proseguire; altrimenti vengono inviate le credenziali alla `View` e poi passate al `Controller` dove ricevono due ulteriori controlli: l'utente deve esistere in memoria con quelle credenziali e la password deve corrispondere. Se così non è appaiono due messaggi di errore distinti.

HomePage

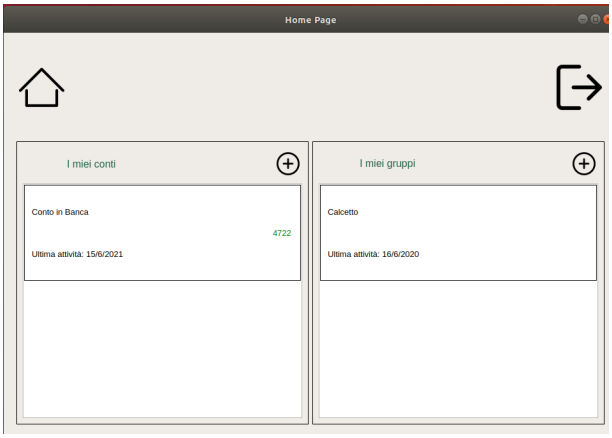

Figura 5: HomePage

Dopo il controllo di liceità dell'utente, si passa alla `HomePage`. Qui troviamo un `QHBoxLayout` che contiene due `QFrame`, uno per i registri e uno per i gruppi: la costruzione è la medesima. In alto un `QHBoxLayout` contiene una `QLabel` di descrizione e un `QPushButtonAR` per aggiungere un registro o un gruppo. In basso un `CustomListWidget` contiene tutti i `Box` con registri e gruppi.
Al click del pulsante di aggiunta compare un `QFrame`. Nel caso dei registri viene chiesto il nome dello stesso ed eventuale bilancio già presente all'interno. Nel caso dei gruppi, si chiede di inserire il nome del gruppo, una descrizione (opzionale) e l'elenco di persone da inserire tramite un apposito widget, `UserFilterList`, con possibilità di filtrare per nome (maggiori informazioni nelle "Scelte progettuali").
Cliccando un box qualsiasi, sarà possibile risalire al contenuto del `Box` tramite l'istruzione, che verrà approfondita nella sezione apposita delle "Scelte progettuali":
`static_cast<CustomListWidget::Box<Register>*>(item->listWidget()->itemWidget(item))->getItem();`
In questo modo posso passare come parametro di un segnale il rispettivo `Register*` o `Group*` per costruire la vista corrispondente.

RegisterView

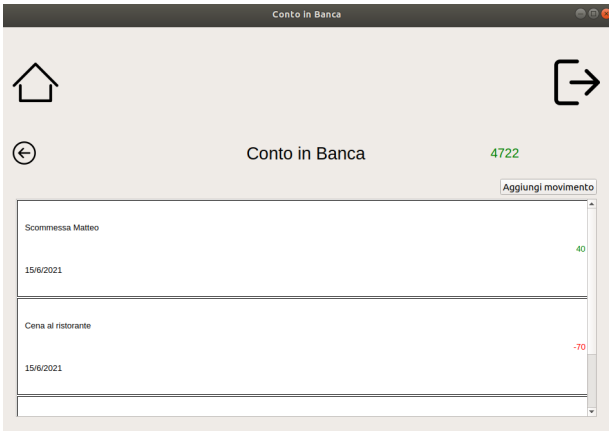

Figura 6: RegisterView

La struttura è quella di `InnerPage`. `CustomListWidget` costruisce una lista di `Box<UserMovement>` di cui mostra descrizione, data e importo di ogni movimento.
Vi è un `QPushButton` che permette di aggiungere movimenti. Per fare ciò è necessario un importo e una descrizione. È facoltativo aggiungere una nota, visualizzabile grazie a `setToolTip` e la data del movimento (di default non vi sono note e la data si presume quella di inserimento).

GroupView

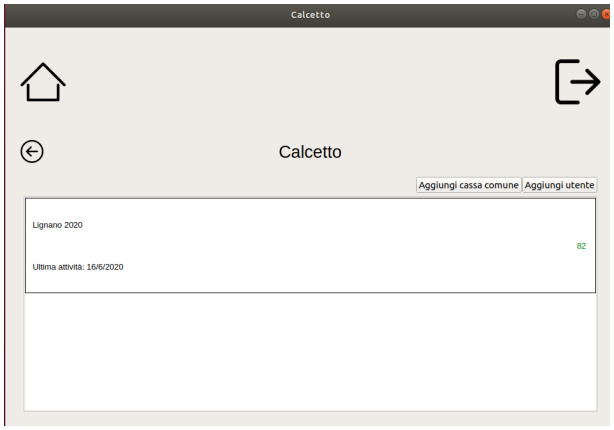

Figura 7: GroupView

La struttura è ancora quella di `InnerPage`. `CustomListWidget` stavolta costruisce una lista di `Box<CommonFund>` di cui mostra nome, data di ultima attività e bilancio corrente.
È possibile aggiungere utenti al gruppo tramite l'apposito `QPushButton` che usa `UserFilterList`. È inoltre possibile aggiungere una cassa comune tramite un altro `QPushButton`, della quale si chiede nome, eventuale bilancio già presente e relative persone iscritte.

CommonFundView

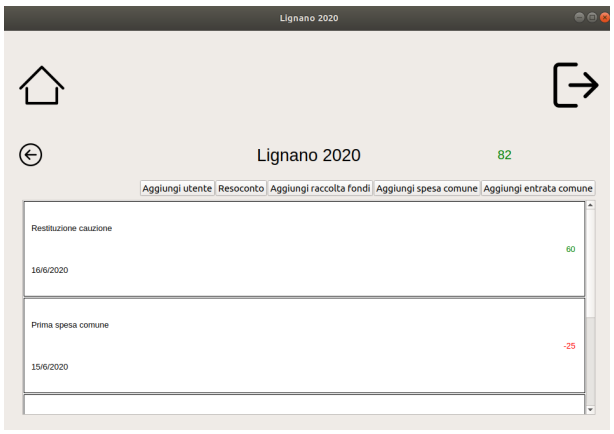

Figura 8: CommonFundView

Resta la struttura tipica di `InnerPage`. `CustomListWidget` costruisce la lista di `Box<GroupMovement>` che si hanno nella cassa comune, dei quali si mostra descrizione, data, importo ed eventuali note. Essendo la parte più importante del progetto, vi sono numerose funzionalità offerte dai `QPushButton`.
È possibile aggiungere utenti alla cassa comune, esattamente come nei gruppi, e visualizzare il recap dei crediti di ogni utente fino al dato momento, realizzato con un `QGridLayout` che ha nelle colonne gli utenti e nelle righe i movimenti con i rispettivi cambio di crediti degli utenti.
È inoltre possibile aggiungere tre tipi di movimenti:

- Raccolte fondi: si inserisce l'importo raccolto per persona nell'apposito `QLineEdit` validato da `QDoubleValidator` con relativa descrizione, note opzionali e data della raccolta, ove difforme. In caso vi siano utenti che non hanno dato l'importo pieno, è possibile inserire l'importo dato nell'`UserInput` (maggiori informazioni nella sezione "Scelte progettuali").

- Spese comuni: si inserisce l'importo della spesa (il valore dello scontrino), descrizione, note e relativa data.
  Qui è possibile che degli utenti abbiano speso soldi a titolo personale e quindi non rientrante nella spesa comune: in questo caso si aggiorna solo il bilancio del singolo individuo.
  È inoltre possibile che i soldi dati non fossero a sufficienza e che quindi un utente abbia dovuto anticipare dei soldi di tasca sua: anche in questo caso si aggiorna il bilancio della singola persona, ma si modifica anche il bilancio del movimento e della cassa comune.

- Entrate comuni: sono entrate comuni e indifferenziate a ogni persona appartenente alla cassa comune (ad esempio la restituzione di una cauzione). È richiesto l'importo complessivo ricevuto, la descrizione del movimento, eventuali note e data del movimento.

Descrizione Modello

Il punto più importante nella realizzazione del modello è stata la gestione di una memoria condivisa. Il progetto infatti rende necessario questa implementazione perché diversi oggetti, quando modificati, portano a cambiamenti anche al di fuori dell'utente considerato. Ad esempio, decido di creare una cassa comune perché vado in vacanza con una persona, anche quella persona dovrà essere a conoscenza dell'avvenuta creazione della cassa comune. Da qui la necessità di gestire la memoria in modo condiviso. Per fare ciò è stato necessario l'uso di `SharedPtr` (l'equivalente di `std::shared_ptr`). Perciò, in tutti i casi in cui è richiesta una informazione condivisa non identificativa, si è utilizzato lo `SharedPtr` o il `CustomPointer` (un puntatore che può essere facilmente convertito da `SharedPtr` a `WeakPtr` e viceversa).
Un altro punto da notare nella descrizione del modello è l'uso particolarmente elaborato dei costruttori. Questo può essere visto in un'ottica abbastanza semplice: minimizzare in numero di eccezioni, gestendo internamente richieste non ammissibili. In questo modo si crea un oggetto "non valido", convertibile a `false` per gestire richieste che violerebbero la coerenza dei dati, come creare un utente che esiste già o richiederne uno che non esiste.

Alcune scelte progettuali

Uso delle lambda functions

Una delle funzionalità offerte da Qt 5, oltre al cambio di firma della funzione di `connect`, che permette la connessione l'invio di segnali e slot da parte di oggetti templatizzati, offre anche la possibilità di collegare un segnale a una lambda function. L'uso principale che ne ho fatto è stato quello di emettere un ulteriore segnale, evitando l'appesantimento della classe con uno slot.
Perché appesantimento? Perché, oltre agli slot in sé, se li avessi ultilizzati intensamente, molte variabili dovevano essere inserite nella parte privata della classe e non potevano essere dichiarate e implementate localmente nel costruttore perché ne avrei avuto bisogno all'esterno dello stesso.
Aggiungo un esempio particolarmente significativo: `CommonFundView`. Qui ho necessità di 5 slot, ciascuno corrispondente ai 5 segnali di `clicked` di altrettanti `QPushButton` presenti. Gli slot emettono segnali contenenti fino a 7 parametri, parametri che sarebbero dovuti essere inseriti nella parte privata della classe. Inoltre nei 5 slot usati, vengono impiegate 9 lambda functions che corrisponderebbero ad altrettanti slot, qualora avessi voluto implementarli in questo modo. Di conseguenza, le classi sarebbero risultate più appesantite in questo modo.

Templatizzazione di ItemBox

`QListWidget` è un widget ideale alla rappresentazione degli elementi della lista che avevo in mente: di facile implementazione e intuitivo per l'utente. Ma ogni volta che era necessario usarlo l'inizializzazione era la medesima e cambiava solo il tipo del `Box` inserito. Per ovviare a questo problema è stato creato `CustomListWidget`, un widget derivato da `QListWidget`. Questa classe derivata ha un costruttore templatizzato a due parametri, in modo da poter costruire una lista di `Box<Group>` partendo da una `olist<Subscription>` e una lista di `Box<CommonFund>` partendo da una `olist<Partecipation>`.
Questa classe presenta anche un metodo di `refresh` templatizzato, in modo da aggiornare i dati in tempo reale, in caso di alterazioni alle liste dovute ad aggiunte dell'utente.
Infine, ma non per importanza, la classe interna `Box` ha un metodo, `getItem()` che ritorna l'oggetto contenuto all'interno del `Box`. Questo metodo è fondamentale per poter passare da una schermata all'altra. Ad esempio per passare dalla `HomePage` a `RegisterView` è necessario prendere il `QListWidgetItem*` ritornato dal segnale `itemClicked` e da questo ricavare l'oggetto contenuto con: `static_cast<CustomListWidget::Box<Register>*>(item->listWidget()->itemWidget(item))->getItem();`
Il metodo `listWidget()` ritorna la lista di appartenenza del `QListWidgetItem* item`. Il metodo `itemWidget(item)` ritorna il `QWidget*` nella posizione indicata da `item`. Lo `static_cast` è safe in quanto in quella posizione per costruzione sono presenti `CustomListWidget::Box<Register>*` e `getItem()` ritorna l'oggetto templatizzato (in questo caso `Register*`).

UserFilterList

`UserFilterList` è un widget che permette, data una `olist<User>`, di filtrare gli utenti inseriti per username tramite un `QLineEdit`. Per fare ciò l'username degli utenti è astrattamente inserito in uno `QStandardItemModel*` sotto forma di `QStandardItem*`. Il filtraggio è reso dalla classe `QSortFilterProxyModel` che fa da proxy tra il "source model", la classe `QStandardItemModel` e la `QListView` che mostra effettivamente gli username degli utenti. Infatti collegando il segnale `textChanged` del `QLineEdit` dell'input, è possibile invocare `setFilterWildcard()` del proxy passandogli come parametro proprio il testo scritto, ottenendo come effetto proprio quello di filtro di input.

UserInput

`UserInput` è un widget usato per inserire i singoli importi degli utenti all'interno di spese comuni e raccolte fondi. È composto da un `QVBoxLayout` che contiene degli `QHBoxLayout` con al suo interno una `QLabel` con lo username dell'utente e un `QLineEdit` per inserire eventuali importi, validati da un `QDoubleValidator`.

Istruzioni di compilazione

Per compilare il programma nella macchina virtuale fornita:

1. Estrarre i file dalla cartella zip
2. Aprire il terminale nella cartella unzippata
3. Eseguire il comando `qmake`
4. Eseguire il comando `make`
5. Eseguire il comando `./accountability`

Viene fornito il file save.json per testare l'applicazione. Le credenziali di accesso sono:
Username: `francesco.ranzato`
Password: `ProgettoPao`
È bene non eseguire il comando `qmake -project`, piuttosto si usi il file `accountability.pro` fornito. Altrimenti, se non viene rimossa -O2 dalle flag di compilazione, il programma va in crash.

Variazioni rispetto alla consegna precedente

Eliminazione dello SharedPtr

Nella precedente versione, `SharedPtr` (che riproduce lo `std::shared_ptr`) e `CustomPointer` erano incaricati di gestire la memoria in modo condiviso. Ora le classi che richiedono la gestione condivisa della memoria (User, Group, CommonFund, Register e Partecipation) hanno un campo `T::Data` (con `T` che indica una delle classi sopra elencate) che contiene le informazioni condivise e un puntatore `T::Data*` interno alla classe stessa. Alla creazione dell'oggetto, queste informazioni vengono inserite in opportune `olist` del modello, che ora gestisce completamente la memoria.

Introduzione del DeepPtr

Per la gestione della memoria in modo profondo viene utilizzato il template di classe `DeepPtr` (che riproduce lo `std::unique_ptr`). Le differenze sono: l'utilizzo del costruttore di copia, che usa il metodo polimorfo `clone()` e l'`operator<(const T&)` per l'ordinamento che si basa sul valore dell'oggetto puntato. In questo modo il `DeepPtr` viene usato in combinazione con la `olist` per gestire gli `AbstractGroupMovement`, cioé i movimenti di una cassa comune, in modo profondo.
Nella versione precedente, questo risultato era ottenuto con un puntatore ad `AbstractGroupMovement` nella classe `GroupMovement`, non più esistente.