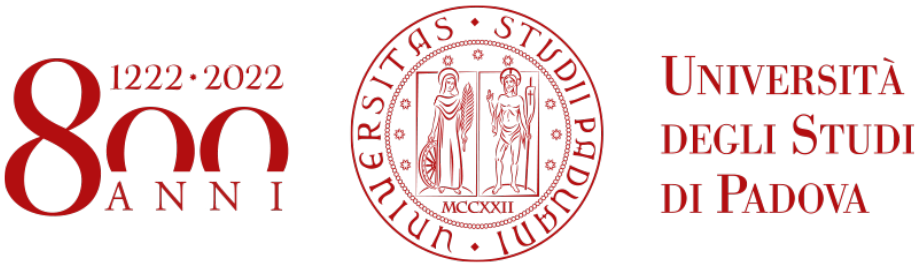


Progetto PaO
QMM - QtMoneyManager

Elia Pasquali 1225412
Claudio Giaretta 1225419

A.A. 2021/2022

1 Introduzione

Il programma QMM si propone come un gestore finanziario personale. Raccogliendo tutte le proprie entrate ed uscite, permette di catalogarle in varie categorie, in modo da ottenere in seguito dei grafici da queste.

2 Descrizione programma

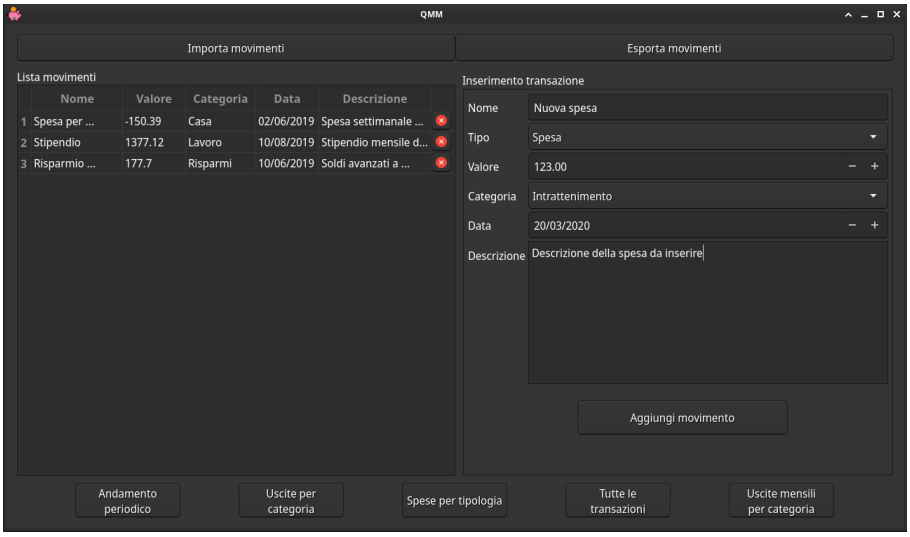
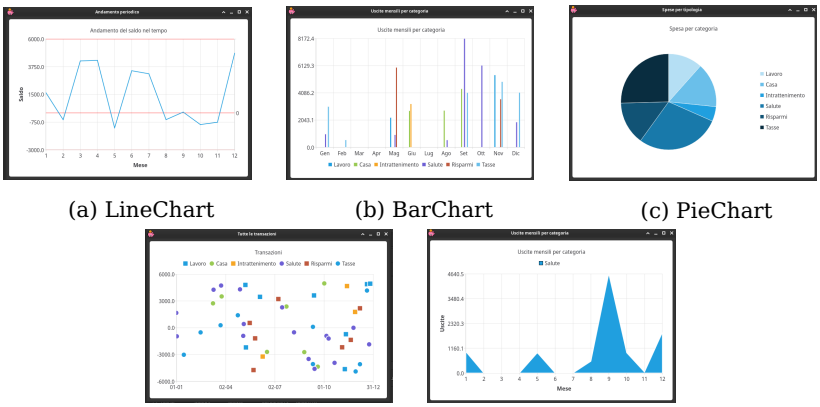


Figure 1: Home del programma

Il gestore QMM si apre sulla pagina principale che permette di importare la proprio lista di movimenti oppure inserirli manualmente. Ogni movimento che viene inserito è caratterizzato da un *nome* ed un *valore* obbligatori, un *tipo* che indica se è un'entrata o una spesa, una *categoria* a scelta tra *Lavoro, Casa, Intrattenimento, Salute, Risparmi* e *Tasse,* una *data* e un campo aggiuntivo per una *descrizione*. In presenza movimenti poi è possibile andare a creare dei grafici tramite i bottoni presenti nel basso. Per crearne uno viene richiesto l'anno di interesse per selezionare quali transazioni inserire.

3 Progettazione

Il programma è stato sviluppato in C++ e con il framework grafico Qt. Si è utilizzato il design pattern *Model-View-Controller* per la progettazione



(a) LineChart (b) BarChart (c) PieChart

(d) ScatterChart (e) AreaChart

Figure 2: Grafici offerti dal programma

della GUI. È stata creata una gerarchia di classi che sfruttano i concetti di *polimorfismo* e altre caratteristiche dell'*Object-Oriented Programming*, ad esempio una particolare attenzione ad estensibilità e manutenibilità del codice. Per la gestione dell'I/O dei dati è stato scelto il formato JSON.

3.1 Gerarchia di tipi

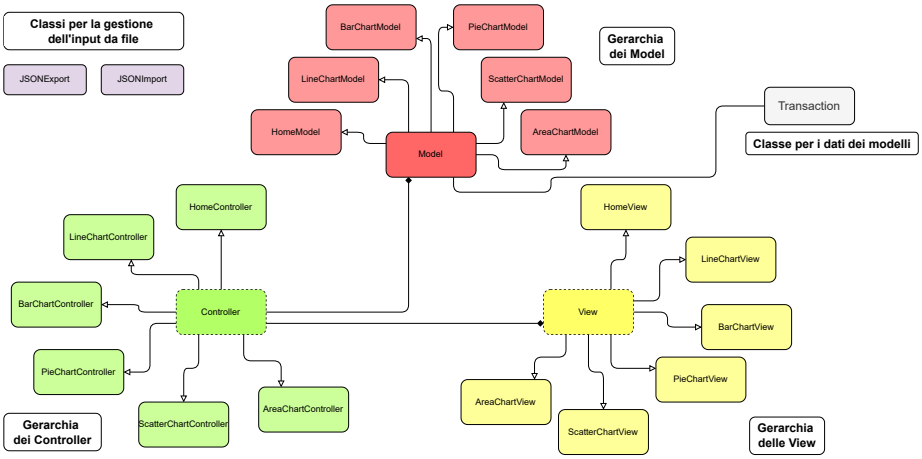


Figure 3: Grafico della classi

Per seguire l'approccio MVC sono state definite tre classi base per i `Model`, le `View` e i `Controller`. Di queste, Controller e View sono astratte

per la presenza di metodi virtuali puri.

La classe che si occupa di gestire il tipo del dominio gestito dal programma è `Transaction`, che contiene i dati del movimento e metodi di supporto come getter e setter.

Sono presenti inoltre due classi di supporto per la gestione dei file con i dati, `JSON_import` e `JSON_export` che sfruttano gli strumenti offerti dalla libreria Qt per interagire con i file JSON.

3.1.1 Model

La classe base Model contiene al suo interno la lista delle transazioni e funzioni di supporto comuni e utilizzabili da tutte le sue derivate.

La classe contiene metodi get e set della lista, rappresentata come vettore delle transazioni. Altri sono per l'inserimento di una nuova transazione tramite GUI e due di supporto per calcolare il range di anni delle transazioni presenti nel modello, insieme ad una funzione che permette di verificare l'assenza di uscite.

Viene definito il distruttore virtuale nella base.

Le classi derivate contengono un costruttore di copia che, sfruttando il polimorfismo, permette di costruirle a partire dalla lista di transazioni presenti nel modello del padre.

3.1.2 View

La classe View è definita come derivata da QWidget. Questo perché andrà a definire le schermate della nostra GUI.

Sono stati definiti un costruttore con valori di default e il distruttore virtuale ed alcuni metodi di supporto come `setTitle` e `setSize` per questioni grafiche.

Un altro metodo definito nella base utilizzabile da tutte le derivate è `errorMessage` per informare l'utente di errori tramite delle finestre di dialogo.

Nella View è presente il campo `year` che si riferisce, ad esempio, all'anno gestito dal grafico. Questa è inserita all'interno di View in ottica di estensibilità del programma con la possibile aggiunta di altre tipologie di grafico o altri strumenti per la gestione dei movimenti.

Viene definito il metodo virtuale puro privato `connectWidgets`, che serve a definire una regola da rispettare per tutti i figli, cioè implementare una funzione che si occupi di collegare tutti i segnali dei Widget interattivi della View con i relativi slots. All'interno del progetto attualmente i grafici non hanno funzionalità dinamiche oltre al mostrare il grafico, ma questa scelta nella base segue l'*Open/Close Principle* di SOLID.

La classe View ridefinisce il metodo `closeEvent` di `QWidget`, che insieme al segnale `closeView` si occupa di eliminare la View alla richiesta

di chiusura.

La View fa affidamento sul meccanismo di distruzione di Qt, che sfrutta il legame tra padre e figlio dei suoi oggetti eliminando tutti i figli al momento della distruzione del padre.

3.1.3 Controller

La classe Controller, derivante sempre da QWidget, si occupa di collegare i Model contenti i dati e la logica del programma alle View che si occupano solo della visualizzazione di questi. Definisce un costruttore e un distruttore virtuale.

La classe è astratta per i due metodi virtuali puri `getView` e `getModel`. Questi in ogni sottoclasse ritornano dei puntatori "corretti" al Model e alla View legata al Controller. Per corretto si intende che viene effettuato uno `static_cast` verso il tipo che ci interessa, dato che abbiamo la certezza che questo sia effettivamente il tipo presente nel Controller avendo il pieno controllo della costruzione.

È presente un metodo virtuale che si occupa di rendere visibile la View che il Controller gestisce, inserito nella base essendo in vista di possibili estensioni.

Legati al meccanismo di segnali e slot di Qt nel Controller sono definiti il metodo virtuale `connectView` che si occupa di unire i suoi slot con i segnali che può ricevere dalla View.

Lo slot definito in Controller è proprio quello che si occupa di ricevere il segnale di chiusura dalla View e gestire la eliminazione di tutti gli elementi.

3.1.4 Gestione I/O

Vista la natura dell'importazione ed esportazione, vista come azione che viene eseguita al bisogno e una frequenza limitata, si è deciso di definire le classi con dei metodi statici, per essere richiamati all'occorrenza senza necessità di instaziazioni ripetute.

Le due classi fanno uso degli strumenti offerti da Qt in per la gestione dei file JSON, come i `QJsonDocument` e i `QJsonArray`.

I controlli di validità fatti su i file che vengono importati sono:

- Verifica se il file è completamente vuoto: se lo è restituisce un errore e blocca l'importazione
- Controllo della lista dei movimenti: se questa è vuota ritorna un avviso senza bloccare il processo.

I file utilizzati dal programma seguono la seguente struttura: nel file JSON deve essere presente un array di nome `transactionsList` e le varie transazioni devono fornire i valori di `name`, `value`, `category`, `day`, `month`, `year`, `type` e `short_desc`.

```
{
  "transactionList": [
    {
      "name" : Nome della transazione (String),
      "value" : Valore della transazione (double),
      "category" : Categoria della transazione (int),
      "day": Giorno della transazione (int),
      "month" : Mese della transazione (int),
      "year" : Anno della transazione (int),
      "type" : Tipologia della transazione (bool),
      "short_desc" : Descrizione della transazione (String)
    }
  ]
}
```

3.1.5 Transaction

La classe `Transaction` si occupa di definire il tipo di riferimento dell'intero progetto. Sfrutta classi di Qt per la gestione di date `QDate` e stringhe `QString`, per facilitare poi l'interazione con i vari widget.

Le categorie, definite a priori, sono strutturate tramite una `enum` per facilitare la caratterizzazione dei movimenti. Per ottenere i nomi delle categorie è stata definita una mappa che ritorna la stringa della categoria corrispondente.

4 Descrizione sviluppo

4.1 Ore di lavoro

Fase del progetto	Ore impiegate
Analisi del dominio	4
Progettazione GUI	6
Studio Qt	6
Sviluppo View	7
Sviluppo Controller	8
Sviluppo Model	8
Sviluppo Gestione I/O	2
Test e Debug	8
Stesura relazione	3
Totale	52

Le 50 ore richieste sono state superate di poco, la causa principale è stata la fase di debug. Durante i test sulla macchina virtuale fornita si è presentato un crash. Questo avveniva alla creazione di un grafico che non presentava problemi sull'ambiente di lavoro personale. Alla fine è risultato essere un *undefined behavior* causato dall'accesso a oggetti in memoria non ancora allocati. La natura di questo problema portava il compilatore a dare messaggi di errore che non avevano alcuna apparente connessione con il problema, rendendo più difficile trovare il punto di rottura.

4.2 Suddivisione del lavoro

Il lavoro non è stato diviso in modo esplicito. La maggior parte dello sviluppo è stata individuale ma con frequenti incontri in aula e laboratori. Questo ha permesso di avere un buon dialogo durante la progettazione e per la risoluzione dei problemi riscontrati.

Principalmente mi sono occupato dello sviluppo delle interfacce e dei rispettivi modelli. Durante la fase di test mi sono occupato di un refactor delle classi per l'I/O e di alcuni controlli sui tipi e i dati ottenuti dalla GUI, per garantire un minimo di robustezza.

Abbiamo svolto la fase di test insieme, in modo da provare il programma su sistemi diversi e per avere una maggiore sicurezza sui risultati ottenuti.

4.3 Ambiente di sviluppo

Il progetto è stato sviluppato con l'IDE QtCreator, i due sistemi di riferimento sono stati

Ambiente personale		Macchina virtuale	
S.O.	Arch Linux	**S.O.**	Ubuntu 18.04.3 LTS
Compilatore	GNU g++ 12.1	**Compilatore**	GNU g++ 7.3
Qt	Qt 5.15.5	**Qt**	Qt 5.9.5

Per facilitare il lavoro di coppia è stato utilizzato Git per il versionamento e per sfruttare i meccanismi di branching. Avendo scelto Github come hosting per il repository remoto abbiamo deciso di sfruttare la CI offerta dalle Github Action.

Tramite un container `ubuntu-18.04` che rispecchia l'ambiente di test, la action scaricava i pacchetti `qt5-default` e `libqt5chart5-dev` per avere le ultime versioni di Qt e QtCharts disponibili, per poi posizionarsi nella cartella corretta ed eseguire in serie `qmake` $\rightarrow$ `make`.

Il successo di una build della CI non garantiva la correttezza del funzionamento del programma, ma almeno quella di corretta compilazione.

5 Istruzioni di compilazione

La macchina virtuale non conteneva la libreria QtCharts necessaria per l'intero progetto, quindi è necessario portarla in uno stato adatto andando ad aggiornare ed installare `qt5-default` e `libqt5charts5-dev`.

Inoltre, il file `pro` generato automaticamente da `qmake -project` non è adatto alla compilazione del progetto. Viene quindi fornito un file che contiene:

- Le specifiche per lo standard del linguaggio da rispettare (`C++11`)
- I flag di compilazione come `-no-pie`, necessaria per rendere il programma eseguibile tramite doppio click su questa versione di Ubuntu.
- Il percorso delle risorse grafiche esterne, ad esempio le icone necessarie al programma.

Il processo di compilazione è quindi `qmake` $\rightarrow$ `make`

Il programma viene consegnato con dei file json generati automaticamente contenenti una serie di transazioni per poter testare il programma.