Sales & Statistics

PROGETTO DI PROGRAMMAZIONE AD OGGETTI

Anno Accademico 2021/2022

Relazione di: Michael Amista' (2009116)

In coppia con: Marco Brigo (2011886)

1.Introduzione

In primo luogo è stato deciso come orientare il progetto verso un caso di utilizzo reale e, dopo diverse analisi degli ipotetici impieghi, è stato scelto di realizzare un programma che permettesse di **registrare e analizzare dei dati relativi alle vendite, suddivise in trimestri, di un qualunque negozio nel corso dell'anno attuale**.

Le vendite sono registrate in forma tabellare, attraverso un dataset ben definito, che permette di analizzare gli incassi o le quantità di articoli venduti mediante dei grafici, i quali consentono di osservare diverse situazioni utili a determinare l'andamento delle vendite. Tali tendenze si possono rivelare utili al fine di **capire quali articoli sono maggiormente graditi dai clienti o piuttosto proficui a livello di incassi**.

Per rendere disponibile tutto ciò è stata realizzata un'interfaccia utente con il compito di rendere facilmente fruibili all'utente tali operazioni.

L'utente può registrare gli articoli venduti attraverso il dataset messo a disposizione e visualizzarne le relative statistiche di vendita, sotto forma di diverse tipologie di rappresentazione grafica.

È inoltre possibile salvare e aprire la forma tabellare tramite ***file.json***.

2. Descrizione del programma e funzionalità offerte

Sales & Statistics è un programma, come detto in precedenza, orientato verso l'analisi e lo studio delle vendite di un'attività commerciale, come un negozio.

L'intestazione sottostante rappresenta un generale articolo venduto dal negozio che utilizza l'applicativo, rappresentato dalla relativa classe "Piece", con nome "Name" del marchio "Brand", venduto al prezzo "Price" e il cui profitto totale è uguale a "Total profit"; di tale articolo sono stati venduti n "Sold pieces".

Intestazione del dataset

Name	Brand	Price	Sold pieces	Total profit	January-March	April-June	July-September	October-December

Le colonne "January-March", "April-June", "July-September" e "October-December" rappresentano le quantità dell'articolo "Name" del brand "Brand" vendute nel relativo trimestre, la cui somma è pari al totale dei pezzi venduti "Sold pieces" di quell'articolo.

Il profitto totale di un dato articolo, "Total profit", è dato dal prodotto "Sold pieces" * "Price".

"Total profit" e "Sold pieces" sono aggiornati in tempo reale in base alle quantità trimestrali inserite e al prezzo del relativo articolo; l'aggiornamento è gestito da un QTimer.

Il programma consente di registrare le vendite di ciascun articolo in forma tabellare, da utilizzare poi come input per i risultati prodotti dai grafici. Le rappresentazioni grafiche offerte sono diverse, ognuna volta a visualizzare un andamento o una statistica diversa.

Il programma consente inoltre di **salvare e** di **prelevare la tabella da file in formato JSON**.

La GUI è caratterizzata da due aree principali: lo spazio di **inserimento dei dati in forma tabellare** (a sinistra); e l'area di **visualizzazione dei grafici** (a destra).

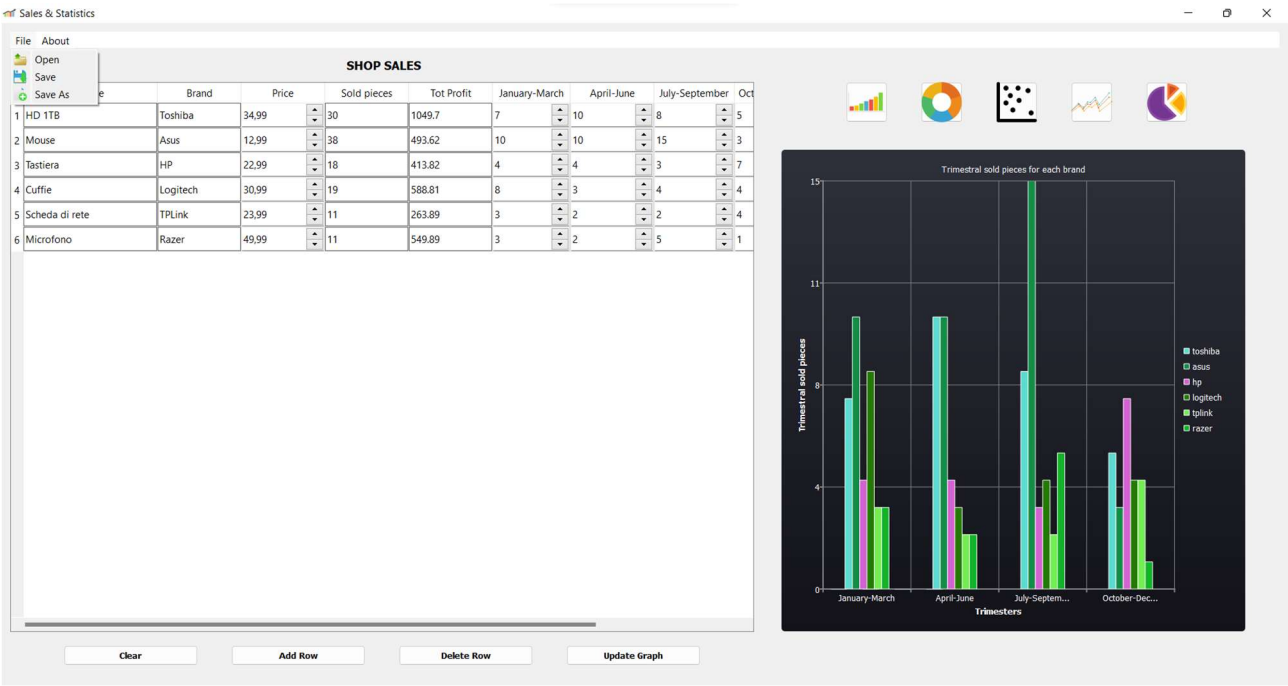
Al primo avvio del programma viene visualizzata a sinistra la tabella di inserimento e a destra un grafico standard, il line chart, privo di dati come la tabella visualizzata.

La tabella è costituita da un record le cui celle sono degli adeguati widget, utili alla tipologia di dati che le colonne devono rappresentare.

I dati vengono mantenuti aggiornati sia nella vista sia nel modello grazie ad un QTimer, collocato nel costruttore del controller, che ha il compito di mantenere aggiornata sia la tabella della vista sia il dataset del modello, ovvero il PiecesCollector (posto come attributo nella classe Model).

Tipi delle colonne del dataset

Colonna	QType
Name	QLineEdit
Brand	QLineEdit
Price	QDoubleSpinBox
Sold pieces	QLineEdit
Total profit	QLineEdit
January-March	QSpinBox
April-June	QSpinBox
July-September	QSpinBox
October-December	QSpinBox
Selected	QCheckBox



L'interfaccia utente offre inoltre diverse funzionalità, tra cui un *QMenu* che raccoglie tutte le principali *QAction* tipiche dei programmi che prelevano e salvano su file, come le funzioni di **Open**, **Save**, **Save As**.
Tali funzionalità sfruttano il formato file **JSON**.

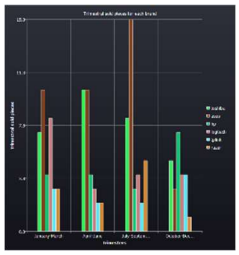
Nell'area di rappresentazione tabellare troviamo alcuni *QPushButton* che forniscono diverse azioni, tra cui:

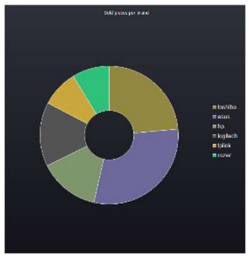
- **Clear**, consente di ripulire completamente la tabella dai dati presenti in essa.
- **Add Row**, permette di aggiungere un nuovo record, inteso come un nuovo articolo venduto, alla tabella.
- **Delete Row**, consente di eliminare una o più righe selezionate tramite la selezione del QCheckBox corrispondente alla o alle righe da eliminare e che è presente nell'ultima colonna di ogni record.
- **Update Graph**, permette di ***aggiornare il grafico attualmente mostrato nell'area destra del programma***, dedicata alla visualizzazione dei grafici, andando ad inserire, nell'andamento mostrato, i nuovi dati immessi nella tabella dopo la creazione del relativo grafico presente a video.

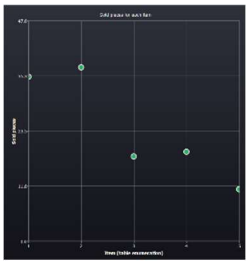
È inoltre possibile notare, nell'area dedicata alla visualizzazione dei grafici, altri cinque *QPushButton* dedicati a creare e visualizzare il relativo grafico mostrato in figura; è inoltre possibile spostare il cursore del mouse sopra uno di questi QPushButton per vedere il grafico a cui si dedicano. Si possono trovare cinque grafici diversi, tra cui: bar chart (1), donut chart (2), point chart (3), line chart (4) e pie chart (5).

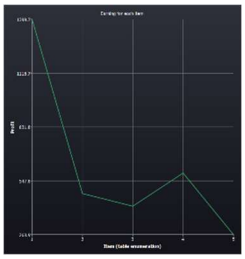
Il *QPushButton* relativo ai grafici permette di crearli e, una volta creati, se la tabella (a sinistra) viene aggiornata e si desidera vedere i nuovi dati nello stesso grafico attualmente mostrato nell'area di visualizzazione dei grafici, si deve utilizzare il *QPushButton* "Update graph", dato che si è deciso di bloccare la costruzione di uno stesso grafico visualizzato nell'area destra del programma per rendere disponibile l'aggiornamento dei grafici tramite il *QPushButton* "Update Graph".

Si noti che il grafico di default, inserito al primo avvio è un line chart vuoto e se si desidera, una volta inseriti i dati, visualizzare l'andamento di quel grafico, come primo grafico, essendo la costruzione per uno stesso grafico visualizzato "bloccata" si deve passare per il QPushButton "Update Graph".

1 

2 

3 

4 

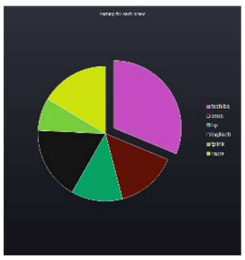
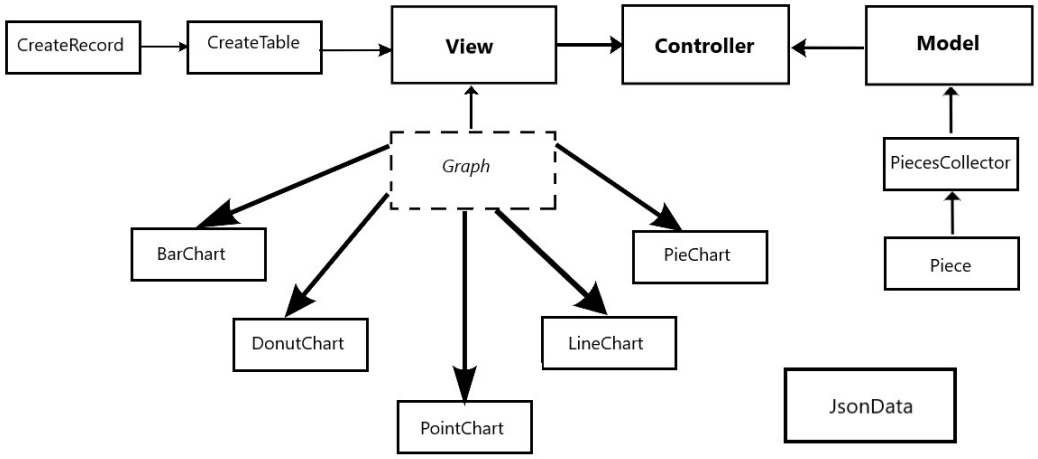
5 

Chart	Andamento visualizzato
Bar Chart	Quantità trimestrali vendute per brand
Donut Chart	Pezzi totali venduti per brand
Point Chart	Quantità vendute per ogni articolo
Line Chart	Guadagno totale per ogni articolo
Pie Chart	Guadagno totale per ogni brand (viene evidenziata la "fetta" relativa al brand che ha incassato di più)

3. Gerarchia di tipi e Salvataggio su file



Per dividere la parte logica dalla vista è stato scelto il pattern **MVC**.

Osservando la mappa contenente tutte le classi del progetto è possibile individuare tutte le classi che compongono il Modello (frecce verso Model) e tutte quelle che compongono la Vista (frecce verso View), direzionate entrambe dalla classe Controller che gestisce il controllo delle operazioni dell'utente.

La gerarchia di tipi è subito evidente osservando la mappa sopra-riportata; la gerarchia è stata impostata sui grafici da visualizzare relativi agli andamenti e alle statistiche nei dati della tabella.

La classe **Graph** è una ***classe base astratta*** che deriva da QWidget ereditando tutte le caratteristiche dei widget, utili nel creare e visualizzare i cinque grafici che ereditano da Graph; troviamo quindi **le classi sottotipo di Graph: Barchart, Donutchart, Pointchart, Linechart e Piechart**.

Graph è caratterizzata dal distruttore virtuale, ridefinito poi nei sottotipi, e dal metodo virtuale puro ***"virtual void update(const QVector<createrecord*>&) =0;"*** legato all'aggiornamento dei grafici tramite il QPushButton "Update Graph".

Una classe particolarmente rilevante è la classe **Jsondata**, una classe utilizzata per il supporto alla lettura e salvataggio su file e utilizzata solamente a questo scopo; la classe di fatto, per il suo utilizzo non fa parte né del Modello né della Vista ma diventa una classe di supporto che offre funzionalità lato file del programma.

Jsondata offre due metodi statici per la lettura e la scrittura da file in formato JSON:

- **PiecesCollector* readJ(const QString& path);**
  Permette, dato il cammino di un file, di leggere un file.JSON e ritorna un PiecesCollector* che rappresenta l'insieme dei record della tabella dal punto di vista logico, un valore che verrà poi utilizzato per tenere aggiornato il Modello.
  Sono stati inoltre implementati dei controlli per quanto riguarda l'apertura di *file.json* non appropriati all'utilizzo del programma.

  È possibile osservare l'utilità di tale metodo nello SLOT: ***void controller::openFile()***

- **void writeJ(PiecesCollector*,const QString& path);**
  Permette, dato il cammino di un file su cui salvare, di creare o scrivere su file.JSON un insieme di record logici dati dal parametro "PiecesCollector*" e che rappresentano i record attualmente visualizzati nella tabella al momento del salvataggio.

  È possibile osservare l'utilità di tale metodo negli SLOT:

  - ***void controller::saveFile()***

  - ***void controller::saveAsFile()***

4. Polimorfismo

- ## Distruttori virtuali

  Il distruttore di Graph, in cima alla gerarchia di tipi, è stato dichiarato *virtual*, in modo da invocare la distruzione polimorfa nei sottotipi.

  Tale implementazione viene utilizzata in diversi contesti per fare la *delete* su un puntatore di tipo statico **Graph*** avente tipo dinamico non definibile a priori ma sicuramente ∈ {**Barchart**, **Donutchart**, **Pointchart**, **Linechart**, **Piechart**}.
  Tale distruzione trova un utilizzo nell'aggiornamento del parametro *displayedGraph* della classe "View", contenente il puntatore al grafico attualmente visualizzato a schermo.
  È possibile osservare il suo utilizzo nel metodo: ***void view::setDisplayedGraph(graph* g)***, richiamato alla creazione di un nuovo grafico, tramite gli slot del controller "showTypeChart" (con Type ∈ {Barchart, Donutchart, Pointchart, Linechart, Piechart}), in modo da tenere aggiornato l'attributo *displayedGraph* con il puntatore al nuovo grafico appena creato.

- ## Metodo update

  graph::virtual void update(const QVector<createrecord*>&) =0;

  Il metodo è principalmente legato al *QPushButton* "Update Graph", il quale permette di aggiornare un grafico visualizzato nell'area destra del programma con i nuovi dati inseriti nella tabella.

  L'idea è stata quella di applicare il polimorfismo al *QPushButton "Update"* per invocare la corretta update relativa al tipo di grafico che è attualmente visualizzato nell'area di visualizzazione dei grafici.

  La corretta ***type::void update(const QVector<createrecord*>&)***, con ***type* ∈ {Barchart, Donutchart, Pointchart, Linechart, Piechart}**, verrà invocata **in base al tipo dinamico dell'attributo di tipo statico Graph* *displayedGraph* della classe "View"**, un attributo che tiene traccia del grafico attualmente visualizzato nell'area destra del programma e che viene regolarmente aggiornato in base al grafico attualmente mostrato.

  Tutto nasce dal fatto che si vuole visualizzare un solo grafico alla volta e al posto di avere diversi bottoni update, uno per grafico e connessi alla relativa funzione update, si è deciso di gestire questo comportamento tramite chiamate polimorfe.

5. Formato di input/output

Come accennato in precedenza una funzionalità offerta dal programma è il salvataggio e l'apertura da file in formato **JSON**. I file sono una rappresentazione dei dati contenuti nella tabella di registrazione delle vendite offerta dalla GUI.

Il contenuto dei file ha una struttura ben specifica che riflette quella della classe "PiecesCollector" che rappresenta un insieme di record *logici*; un record logico è rappresentato dalla classe "Piece". Tale struttura permette infatti di prelevare e salvare la tabella come se fosse un "PiecesCollector"; si osservino i metodi relativi alla classe "jsondata" riportati al punto 3 della relazione.

La struttura del file è la seguente: **{ "Pieces Collector": [ ] }**

Il programma consente inoltre, tramite la *QAction "Open"*, di poter leggere un file salvato in precedenza per andare ad aggiornarlo o visualizzarne nuovamente i relativi grafici.

Insieme al codice viene inoltre fornito, a scopo illustrativo, un file **esempio.json**, riguardante le vendite di un negozio di elettronica e informatica, utile ad osservare e testare le funzionalità offerte dall'applicativo.

6. Suddivisione del lavoro progettuale

Inizialmente ci si è riuniti insieme per discutere dell'impronta reale che si voleva dare al programma e per definire le classi che compongono il Modello, dato che queste sono state ritenute la base del progetto e si doveva concordare su questa base comune.

Dopo aver concordato e definito il Modello ogni componente del gruppo si è concentrato, almeno inizialmente, su parti diverse in modo da poter portare avanti il lavoro. Inizialmente mi sono concentrato sulla parte grafica e visiva, abbozzando quella che poi è diventata la base della GUI, mentre il mio collega ha posto più l'attenzione sull'input output da *file.json*. C'è stato un periodo di studio da entrambe le parti per realizzare in maniera corretta questa prima parte di lavoro e, una volta conclusa, sono state unite le due parti per vedere il risultato finale.

Concordato su questo risultato il lavoro rimanente, che consisteva nella realizzazione e nel collegamento delle componenti grafiche principali, tra cui la tabella e i cinque chart, è stato portato avanti insieme suddividendo equamente i compiti.

Sono stati infine effettuati dei test da entrambe le parti per verificare il corretto funzionamento dell'applicativo e correggere le eventuali problematiche riscontrate durante essi.

7. Compilazione e ambiente di sviluppo

Insieme al progetto è stato fornito il file **Graphs.pro** e un file **images.qrc**, contenente le immagini utilizzate nel programma.

Il comando per eseguire il programma da terminale è il seguente: **qmake Graphs.pro → make → ./Graphs**

L'ambiente principale di sviluppo, da me utilizzato, è stato ***Windows 11***, su cui è stato installato ***Qt versione 5.9.5*** e compilatore ***MinGW g++ (GCC) 9.2.0***.

Il progetto è stato poi correttamente testato sulla macchina virtuale "ssh.studenti.math.unipd.it", di cui è stata messa a disposizione l'immagine, ottenendo lo stesso risultato che si è osservato nell'ambiente di sviluppo principale tranne che per il modo in cui veniva visualizzata l'applicazione, ovvero non a schermo intero; è stata infatti aggiunta una macro sul file "*main.cpp*" per gestire la visualizzazione dell'applicazione in base al sistema operativo su cui si va ad eseguire l'applicativo, che sia linux o windows.

Si fa notare che nella macchina virtuale fornita mancavano i suddetti pacchetti utili al corretto funzionamento del programma e che sono stati quindi installati tramite i comandi:

- **sudo apt-get install qt5-default**
- **sudo apt install libqt5charts5-dev**

8. Ore di lavoro per la realizzazione

Analisi preliminare del problema	2 ore
Progettazione modello	3 ore
Progettazione GUI	4 ore
Apprendimento libreria Qt	16 ore
Codifica modello	7 ore
Codifica GUI	16 ore
Debugging	4 ore
Testing	2 ore
Totale	**54 ore**

Purtroppo sono state oltrepassate le 50 ore previste per i seguenti fatti:

- l'apprendimento della libreria Qt, allo scopo di capire quali componenti si adattavano meglio a ciò che si doveva realizzare e capirne il loro comportamento e caratteristiche, ha richiesto più tempo del previsto essendo appunto un nuovo ambiente.
- Durante i test sulla parte relativa ai chart, sono stati riscontrati dei crash dell'applicativo che hanno richiesto un'ulteriore supervisione del codice per poterne evidenziare la causa e poterli così risolvere.