PROGETTO
DI PROGRAMMAZIONE
AD OGGETTI

Anno Accademico 2021/2021

Realizzato da: Brugnera Luca 2014722

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

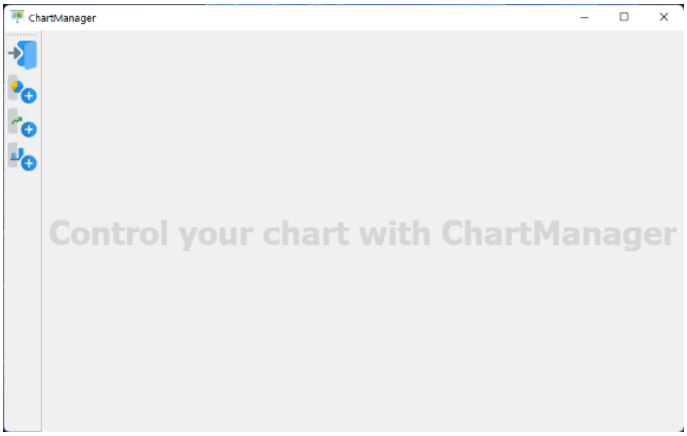
Introduzione

ChartManager è un'applicazione che permette all'utente di manipolare grafi a torta, a linee e a barre.

ChartManager dispone di un'interfaccia user-friendly per la creazione, modifica e cancellazione di grafi, ma non solo: sono disponibili anche le funzionalità di salvataggio e importazione da file.

Manuale d'uso gui

Schermata Iniziale



Una volta aperto il programma si visualizza la schermata iniziale, nella quale troviamo la barra multifunzione (a sinistra) che ci permette (in ordine) di:

1. Aprire un grafo da file
2. Creare un nuovo grafo a torta
3. Creare un nuovo grafo a linee
4. Creare un nuovo grafo a barre

Schermate di modifica dei grafi

Le schermate adibite alla modifica dei grafi sono divise in due sezioni pricipali: dati e visualizzazione.

La **sezione dei dati** è composta dall'insieme dei dati che identificano il grafo, i quali sono suddivisi in: informazioni generali (titolo, funzionalità di selezione, aggiunta e rimozione di dati ed eventualmente nome degli assi) ed informazioni specifiche per il dato selezionato

La **sezione di visualizzazione** la quale si occupa, come si deduce dal nome, della rappresentazione grafica del grafo.

Per selezionare un elemento del grafo (una fetta in un grafo a torta, una linea, una barra ecc) si utilizzano dei menù a tendina i quali ne indicano colore e nome.

L'applicato segue un meccanismo specifico per determinare la posizione del dato da aggiungere o rimuovere: quando si aggiunge un dato questo verrà aggiunto in seguito a quello selezionato, quando si elimina verrà eliminato quello selezionato.

Nella barra multifunzione vengono aggiunte in ordine le funzionalità di salva e salva con nome.

Quando si termina la creazione\modifica e se ne vuole iniziare un'altra (cliccando uno dei quattro pulsanti che lo permettono) appare una finestra popup la quale ci chiede cosa vogliamo farne (salvare o cancellare) del grafo appena creato\modificato.

Per modificare una qualunque stringa o valore è necessario premere <Enter> (ogni oggetto QLineEdit connette come segnale di input solamente returnPressed() con uno slot)

Grafo a torta

Le informazioni generali si trovano nella sezione “Pie Chart” e comprendono: titolo, fette presenti, bottone per aggiunta e rimozione di una fetta.

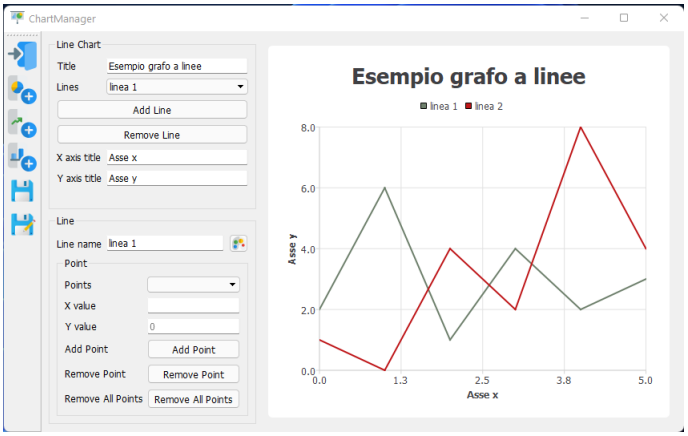
Selezionando una fetta è possibile modificarne i dati attraverso la sezione “Informazioni specifiche”, la quale è composta da: nome della fetta (che deve essere univoco), bottone per il cambio colore (possibile anche cliccando nel grafo la fetta di interesse) e valore.

Grafo a linee

Nella sezione “Line Chart” troviamo: titolo, insieme delle linee, pulsante per aggiunta e rimozione di quest’ultime e nome degli assi.

Selezionando una linea è possibile modificarne il nome (il quale deve essere univoco), il colore, l’insieme di punti di cui è composta (attraverso pulsanti e menù a tendina specifici).

Per i punti si è scelto di adottare una politica di accettazione che prevede di validare i punti che hanno valore di ascissa maggiore del punto precedente e minore di quello successivo.



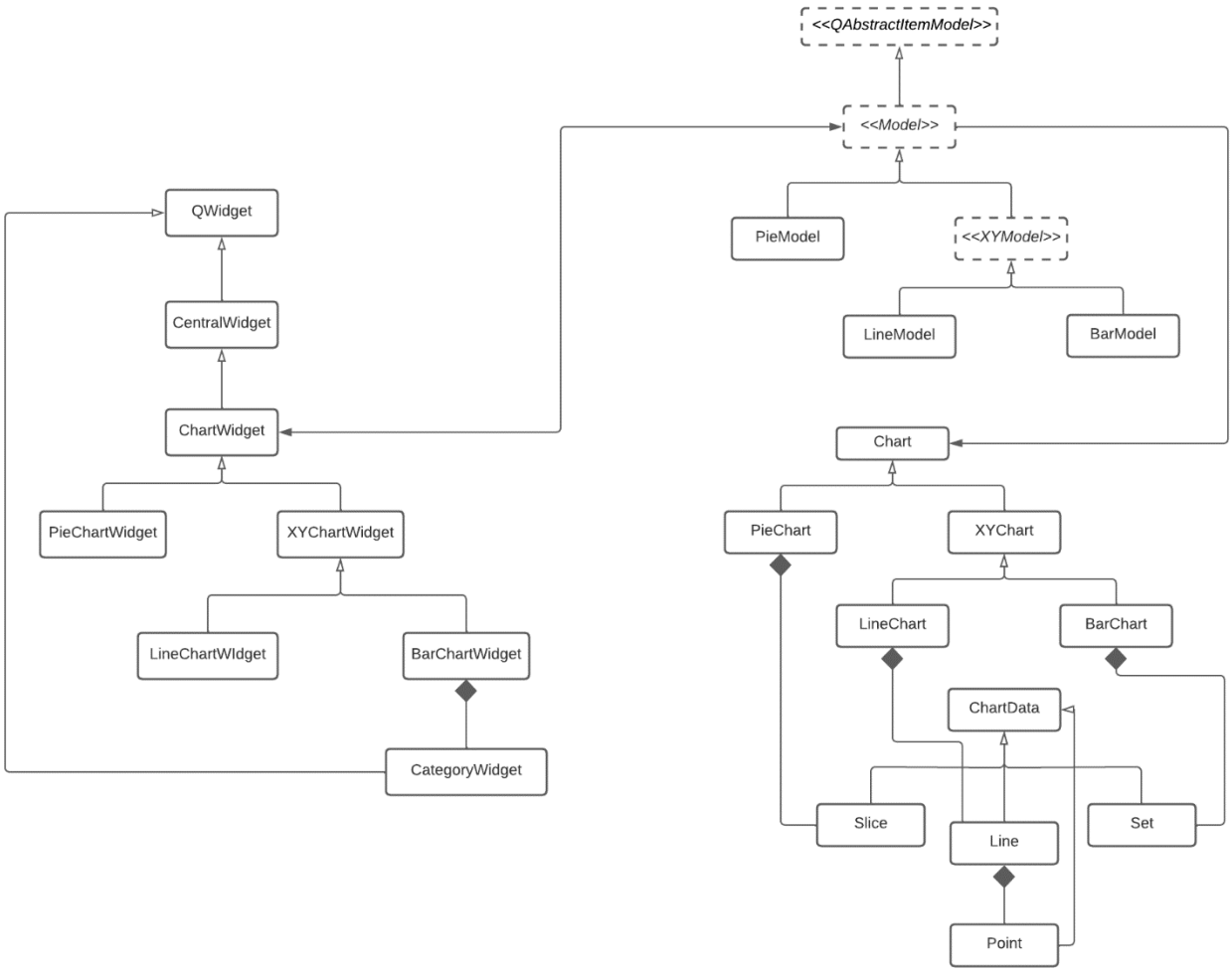
Grafo a barre

Come nel grafo a linee le informazioni generali sono composte da titolo, insieme di set, bottone per aggiunta e rimozione di set e nome degli assi.

Nelle informazioni specifiche troviamo il nome del set (il quale deve essere univoco), il bottone per il cambio colore (possibile anche cliccando una barra del set sul grafo) e l’insieme delle categorie.

Le categorie si affidano per la loro visualizzazione ad oggetti di tipo CategoryWidget i quali contengono: nome (univoco), valore del set selezionato, bottone per la cancellazione e bottone per la creazione di nuove categorie.

Gerarchia dei tipi



Si è deciso di utilizzare il pattern Model View sfruttando le classi QAbstractItemModel e QListView messe a disposizione dal framework Qt per dividere la parte logica da quella grafica.

La vista si interfaccia con l'utente e comunica al modello le modifiche da applicare alla base di dati, il modello applica tali modifiche ed emette segnali che permettono alla vista di aggiornarsi (coerenza tra base di dati e vista).

Parte grafica

View è la classe di maggior importanza nella vista in quanto contiene un puntatore all'oggetto QMainWindow (finestra principale dell'applicato) oltre a vari metodi per la modifica di suoi componenti (QToolBar, QWidget centrale).

La classe CentralWidget è un QWidget con un attributo View* e un costruttore apposito creato per poter avere oggetti che si riferiscono ad esso: sono quindi un'estensione di QWidget e verranno utilizzati come finestre centrali della finestra principale QMainWindow.

ChartWidget è un'infrastruttura per la visualizzazione della parte dati e di quella grafica di un grafo: viene poi estesa dalle classi derivate PieChartWidget per i grafi a torta e XYChartWidget per i grafi XY (a sua volta derivata in LineChartWidget e BarChartWidget) le quali implementano elementi e funzionalità specifiche.

La gerarchia ChartWidget, tra le altre cose, connette segnali emessi dal modello con suoi slot rendendo la GUI dinamica e sensibile alle modifiche sulla base di dati.

Tutti i singoli elementi (bottoni, spazi di testo, menù a tendina ecc) che compongono la GUI sono figli del QWidget (o classi derivate) di appartenenza, quindi, quando viene eliminato l'oggetto vengono eliminati anche loro.

I menù a tendina implementati attraverso QComboBox sono aperti al pattern model view visto che hanno una QListView integrata, la quale può essere collegata attraverso QComboBox -> setModel ad un QAbstractItemModel*. Il resto delle informazioni necessarie alla GUI sono richieste al modello attraverso l'utilizzo di metodi virtuali di QAbstractItemModel.

Parte logica

Per la parte logica si nota la presenza di 3 gerarchie: model, chart e chartdata.

Model: contiene quelle classi il cui compito è comunicare sia con la base di dati (gerarchia Chart) sia con la vista. L'interfaccia Model deriva da QAbstractItemModel, alla quale include metodi per salvare-importare grafi in-da file json, per la modifica del titolo del grafo ecc.

PieModel implementa i metodi virtuali puri di Model creando il modello specifico per i grafi a torta.

XYModel aggiunge dei metodi per la creazione e modifica degli assi ascisse e ordinata.

Da quest'ultima classe derivano LineModel e BarModel i quali implementano i metodi virtuali puri e danno origine ai modelli per i grafi a linee e a barre rispettivamente. I modelli comunicano una modifica alla vista attraverso l'emit di segnali.

Chart: questa gerarchia rappresenta la base di dati dell'applicazione

Gerarchia ChartData: rappresenta i dati che possiamo trovare all'interno delle varie tipologie di grafo (si dimostra particolarmente utile per semplificare metodi del modello quali QModelIndex& parent, QModelIndex& index ecc).

Formato dei file

Per salvare i grafi viene utilizzato il formato json, il quale facilità l'interscambio di dati basati su gerarchie di classi e oggetti in generale.

Ogni tipo di grafo importato da file json prevede, nel momento di costruzione, un minimo controllo sull'integrità del file (non sono state prese in considerazione tutte le possibili casistiche di manipolazione quindi un file fortemente compromesso potrebbe dar luogo ad errori tipo Undefined Behavior).

Per la manipolazione di file json sono state utilizzate le classe QJsonDocument, QJsonObject, QJsonArray e QJsonValue disponibili a partire da Qt 5.0.

Sono disponibili 3 file (uno per tipologia di chart) in "Chart-Application\chart sample"

Chiamate polimorfe

Distruttori

I distruttori di QWidget e QAbstractItemModel sono virtuali per definizione di Qt, il distruttore di Chart e ChartWidget è reso virtuale. Tali distruttori vengono invocati quando si cambia il grafo di interesse aprendone o creandone un altro (distruzione del ChartWidget -> distruzione Model -> distruzione Chart -> distruzione dati del Chart)

Metodo createChartFromModel

Metodo invocato dalla vista, più precisamente dallo slot privato openFromFile per la creazione di un ChartWidget specifico (in seguito alla volontà dell'utente di aprire un grafo esistente da file).

Metodi di QAbstractItemModel

QAbstractItemModel dispone di 5 metodi puri i quali rappresentano la base per lo scambio di informazioni con la vista (infatti sono utilizzati in maniera massiccia):

Model -> columnCount

Model -> data

Model -> index

Model -> parent

Model -> rowCount

Metodo parentItem

Metodo definito in ChartData il quale ritona il ChartData* padre, viene utilizzato nel modello per la definizione di QModelIndex& parent(const QModelIndex& child

Metodo parsing

Metodo che ritorna il QJSonObject* contenente il chart (da qui basta semplicemente inserirlo in un QJSonDocument). Di questo metodo viene fatto l'override in ogni classe appartente alla gerarchia Chart ed il pattern per la sua definizione è il seguente: ottenere il QJSonObject* dalla classe base diretta (la quale a sua volta lo ottiene dalla sua dando così il via ad un meccanismo a scalini) ed inserire i dati specifici della classe.

Esempio:

```
QJSonObject *BarChart::parsing() const{

QJsonObject* obj = XYChart::parsing();

//insert in obj delle informazioni specifiche del bar chart

Return obj;

}
```

Istruzione di compilazione

Oltre ai sorgenti ho consegnato:

- Un file Chart-Application.pro (necessario utilizzare tale file e non generarne un altro).
- Cartella “icon” contiene icone usate dall’applicazione
- File Chart-ApplicationResource.qrc
- Cartella “chart samples” contiene un esempio di grafo per ogni tipologia)

Pacchetti necessari:

- Qt5-default
- Libqt5charts5-dev

Istruzioni per la compilazione: qmake >> make

Ore di lavoro richieste

Analisi preliminare del problema	1 ora
Progettazione modello e GUI	5 ore
Codifica gerarchia Model	11 ore
Codifica gerarchia Chart + ChartData	11 pre
Codifica View e CentralWidget	2 ore
Codifica gerarchia ChartWidget	22 ore
Debugging	7 ore

Per un totale di 59 ore.

Le 9 ore in più sono causate in parte a problemi nella visualizzazione grafica di grafi a linea e barre (circa 4 ore) e in parte ad una sottostima delle funzionalità da implementare (le restanti 5).

La mia conoscenza di base della libreria Qt è stata approfondita durante la codifica dell’intero progetto

Software utilizzato

Sistema operativo	Windows 11
Versione compilatore MinGW	8.1.0
Versione Qt	5.9.9
IDE	Qt Creator
Version Control System	Git

Il progetto è stato testato anche nella macchina virtuale data in dotazione.