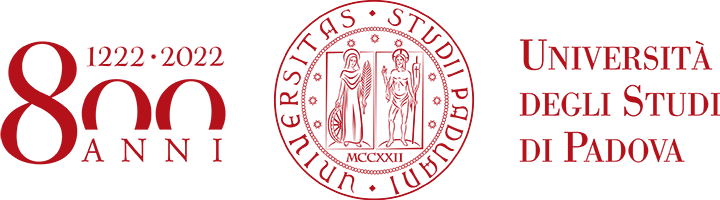
­­­­

PROGETTO

DI PROGRAMMAZIONE

AD OGGETTI

Anno Accademico 2021/2021

Realizzato da: Brugnera Luca 2014722

**Introduzione**

In qualsiasi ambito l’utilizzo di grafici aiuta ad avere una visione più chiara, semplice e completa, ChartManager permette proprio questo: è un’applicazione per la creazione, modifica e salvataggio di grafi a torta, linee e barre dotata di una GUI per semplificare l’interazione con l’utente.

Immagine che contiene testo

Descrizione generata automaticamente**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 di:

1. Aprire un grafo già creato
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 grafici sono divise in due sezioni. La **sezione dei dati** composta dall’insieme dei dati che identificano il grafo, a sua volta partizionata in: informazioni generali (titolo, funzionalità di selezione, aggiunta e rimossione di dati ed eventualmente nome degli assi) ed informazioni specifiche del dato selezionato; la **sezione di visualizzazione del grafo**. Per selezionare un elemento del grafo (come fetta, linea, barra…) si utilizzano dei menù a tendina i quali ne indicano colore e nome, in seguito alla selezione sono disponibili i suoi dati. 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 fare (salvare o cancellare) del grafo appena creato\modificato.

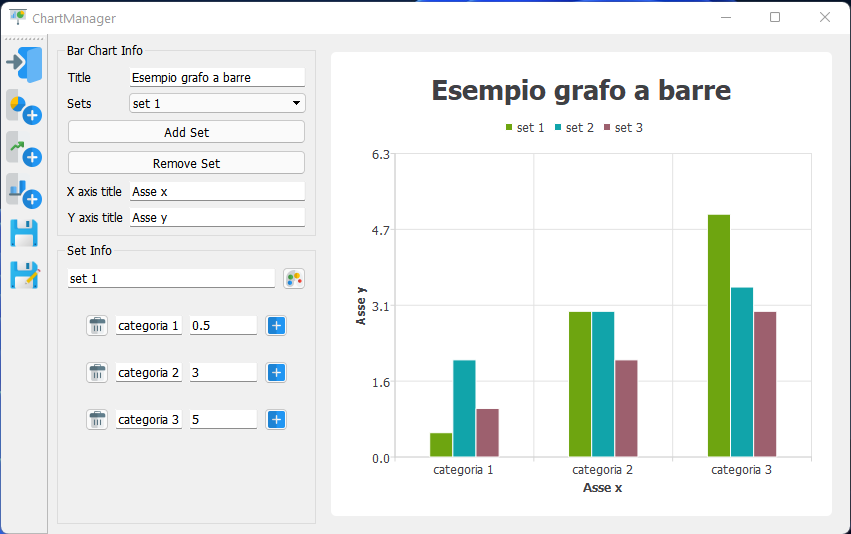
Per modificare una qualunque stringa o valore è necessario premere <Enter> (ogni oggetto QLineEdit connette come segnale di input solamente returnPressed() con uno slot)

**Grafo a torta**

Le informazioni generali si trovano nella sezione Pie Chart e comprendo titolo, fette presenti, bottone per aggiunta e rimozione di una fetta. Selezionando una fetta è possibile modificarne i dati attraverso la sezione di 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.

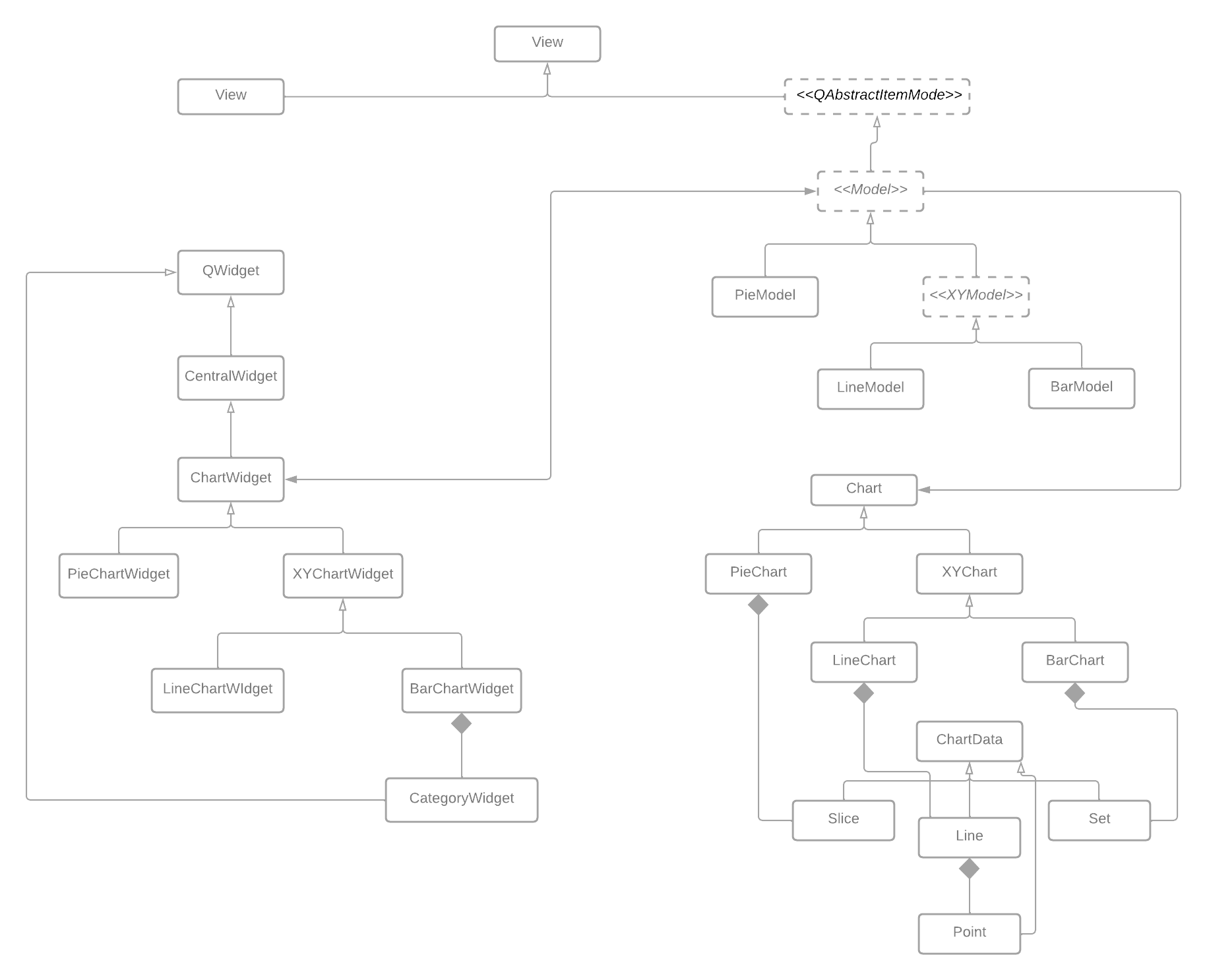
**Grafo a linee**

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), colore, insieme di punti (attraverso l’utilizzo del menù a tendina, pulsante di aggiunta, rimozione e rimozione di tutti). 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), 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 bottone per la cancellazione, nome della (il quale deve essere univoco), valore 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**

La classe View contiene un QMainWindow\* (ovvero la schermata dell’applicazione) e mette a disposizione dei metodi per la modifica del QToolBar e QMainWindow->centralWidget. La classe CentralWidget è un QWidget con un attributo View\* e un costruttore apposito creata per poter avere oggetti che si riferisco alla vista e quindi semplificare l’utilizzo di metodi e slot appartenenti a quest’ultima. ChatWidget mette a disposizione un'infrastruttura per la visualizzazione di un grafo e contiene gli elementi ricorrenti per tutti e tre tipologie (come per esempio il titolo), viene poi estesa dalle classi derivate PieChartWidget, XYChartWidget (le quali implementano elementi e funzionalità specifiche). XYChartWidget aggiunge alla sua classe base diretta gli assi ascisse ed ordinata (con annessa funzionalità per la modifica). Da XYChartWidget derivano LineChartWidget e BarChartWidget (classi specializzate nella visualizzazione di grafi a linee e barre rispettivamente). La gerarchia ChartWidget, tra le altre cose, connette segnali emessi dal modello al quale punta con suoi slot rendendo la GUI dinamica e sensibile alle modifiche sulla base di dati

CategoryWidget è un QWidget specializzato per una l’elemento categoria in un grafo a Barre. Tutti gli elementi che compongono la GUI sono figli della classe di appartenenza (che è un QWidget) quindi quando viene elimato l’oggetto vengono eliminati anche tutti i figli. 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 del metodo QAbstractItemModel -> data e di indici di tipo QModelIndex.

**Parte logica**

**Gerarchia Model**: contiene quelle classi il cui compito è comunicare sia con la base di dati (gerarchia Chart) sia con la vista. L’interafccia Model deriva da QAbstractItemModel, alla quale include metodi per prendere e modificare il titolo del grafo e salvarlo in un file json. PieModel implementa i metodi virtuali puri di Model creando così il modello specifico per i grafi a torta. XYModel contiene dei metodi per la creazione e modifica degli assi x e y. 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 modifca alla vista attraverso l’emit di segnali (i quali sono sia generici che specifici).

**Gerarchia Chart**: ricalca la gerarchia del Model e rappresenta la base di dati (conosciuta dal modello ma invisibile alla vista)

**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 QModeIndex& parent(const QModelIndex& child), QModelIndex& index(int row, int column, const QModelIndex& parent)…

**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 prevede, nel momento di costruzione da file json un minimo controllo sull’integrità (presenza di dati consistenti nei file) del file (nota: non sono state prese in considrezione tutte le possibili casiste 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 dichiarazione 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 alla creazione di un ChartWidget specifico (in seguito alla volontà dell’utente di aprire un grafo esistente da file) che configura la GUI

**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 che si segue è: 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 dove, per ogni scalino, si introducono informazioni riguardati al grafo. Esempio

QJSonObject \*BarChart::parsing() const{

QJsonObject\* obj = XYChart::parsing();

//insert in obj delle informazioni specifiche del bar chart

Return obj;

}

**Istruzione di compilazione**

Consegnato un file Chart-Application.pro (necessario utilizzare tale file e non generara un altro .pro). Inoltre sono state consegnate 2 cartelle supplementari (“icon” che contiene icone usate dall’applicazione e “chart samples” contiene un esempio di grafo per ogni tipologia) e il file Chart-ApplicationResource.qrc

Pacchetti necessari:

* Qt5-default
* Libqt5charts5-dex

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 berre (circa 3 ore) e in parte ad una sottostima delle funzionalità da implementare (le restanti 6). 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.