AirQuality Charts - Progetto OOP

Relazione
Ennio Italiano - mat. 1224819
Michele Cazzaro - mat. 1226303

Febbraio 2022

AirQuality Charts è un programma che permette di creare, modificare, scaricare e visualizzare dati orari relativi a componenti e qualità dell'aria di varie città in diversi periodi di tempo.

I dati possono essere inseriti nel programma attraverso un file JSON locale scelto dall'utente (o creato appositamente) o prelevati da internet attraverso richieste HTTP GET alle API del servizio OpenWeather.

Per quanto riguarda la visualizzazione, i grafici a disposizione dell'utente sono i seguenti:

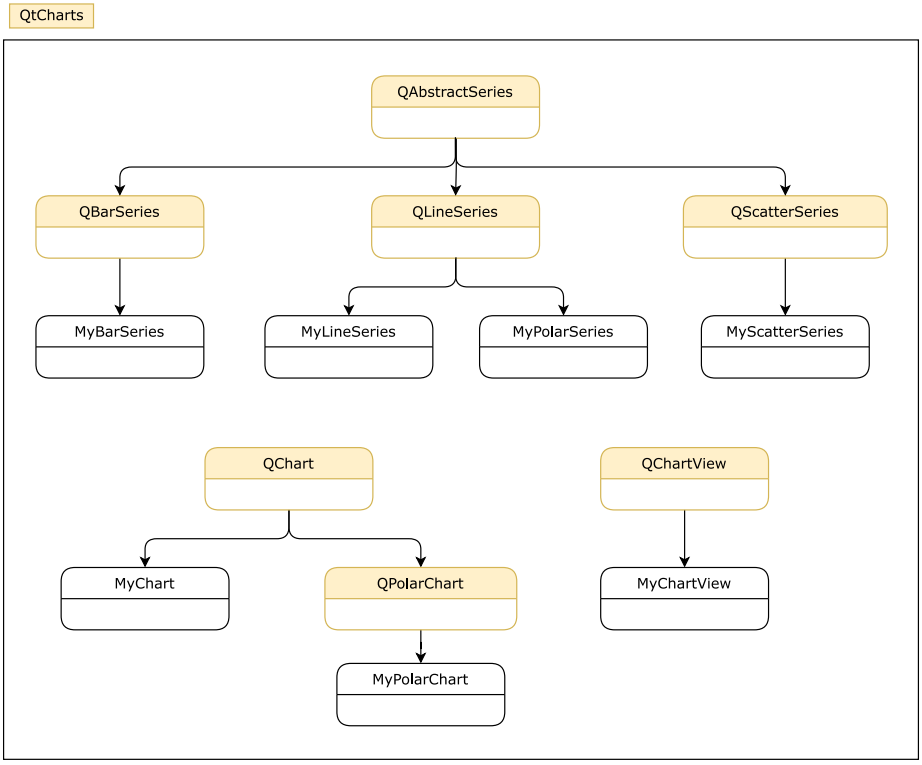
- grafico a linee;
- grafico ad area (in pila);
- istogramma;
- grafico a dispersione (plot);
- grafico radar.

Una breve descrizione di ciascuno di essi è disponibile all'interno del programma, nella finestra `DataViewer`. I dati sono anche visualizzabili in forma tabellare, modalità grazie alla quale è anche possibile apportare modifiche ai dati stessi. Una volta scaricati e/o modificati, è possibile salvare i dati all'interno di un file JSON.

1 Gerarchie di tipi

Di seguito sono riportate le diverse gerarchie di tipi implementate nel programma. Si riportano per completezza (in giallo) le classi già presenti in Qt utilizzate come basi per le nostre gerarchie.

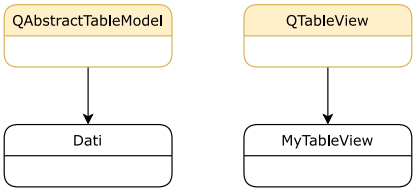
1.1 Grafici



Tutti i tipi di `QAbstractSeries` da noi utilizzati sono stati ridefiniti per adattarsi al meglio alle nostre esigenze ed essere riutilizzati in contesti diversi all'interno del programma. Esempio lampante di ciò è `MyLineSeries`; oggetti di questo tipo sono infatti utilizzati sia come singole linee nel grafico corrispondente, sia come "delimitatori" per le aree nell'omonimo grafico.

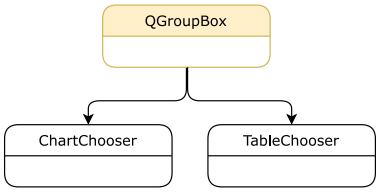
Le serie definite in questo modo sono quindi usate per costruire correttamente grafici di tipo `MyChart` (o `MyPolarChart` nel caso del grafico radar) che verranno poi visualizzati grazie alla classe `MyChartView` (ereditata da `QChartView`).

1.2 Model e Table



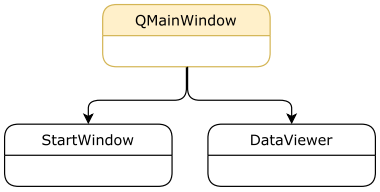
Per gestire al meglio il rapporto tra model e visualizzazione dei dati in tabella si è scelto di seguire l'implementazione del framework Model/View consigliata dalla documentazione di Qt. Per farlo, la classe `Dati` (rappresentante il model) è stata ereditata da `QAbstractTableModel` e la classe `MyTableView` da `QTableView`. Ciò ha permesso una profonda integrazione tra vista tabellare e modello. La documentazione stessa suggerisce inoltre i metodi virtuali delle classi base utilizzate dei quali è necessario fare *overriding* per una corretta integrazione; questo aspetto è trattato nel dettaglio nella sezione Polimorfismo.

1.3 Controlli



Si è scelto di creare due `QGroupBox` personalizzati, `ChartChooser` e `TableChooser`. In tal modo si sono creati i due form di controllo per visualizzare e gestire grafici e tabella. I due form sono poi visualizzati all'interno della finestra `DataViewer`, dove verranno anche mostrati i dati nella forma scelta.

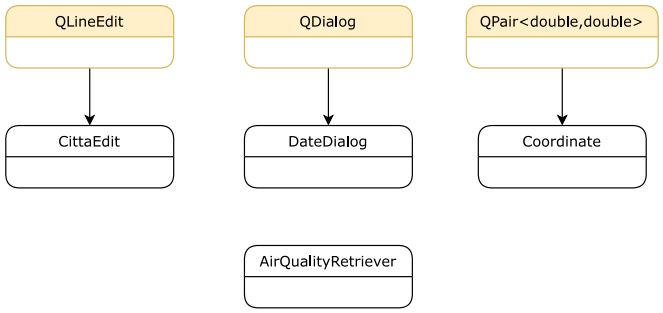
1.4 Finestre



Le uniche due finestre vere e proprie che vengono mostrate sono `StartWindow` e `DataViewer`, entrambe ereditate da `QMainWindow`. La prima è quella iniziale, che consente di scegliere tra importazione, creazione o download dei dati e mostra

una breve descrizione del programma; la seconda si occupa invece di mostrare i controlli riguardanti la visualizzazione dei dati, una breve descrizione di ciò che si andrà a visualizzare e di mostrare i dati nella forma selezionata.

1.5 Utilities



Le restanti classi da noi create e utilizzate sono:

- `CittaEdit`: `QLineEdit` personalizzato, consiste in un campo di testo che nel nostro caso è correlato di un `QCompleter` che rende più semplice la selezione di una città. `CittaEdit` viene utilizzata in `DateDialog` (per consentire la scelta della città per la quale creare un nuovo file) e in `StartWindow` (per consentire la scelta della città di cui scaricare i dati da internet).
- `DateDialog`: `QDialog` mostrato all'utente nel momento in cui deve scegliere data di partenza e città per cui creare un nuovo file.
- `Coordinate`: semplice `QPair` di latitudine e longitudine creato per rendere il programma indipendente dal pacchetto `positioning` di Qt ed evitare inconvenienti di compilazione su macchina virtuale.
- `AirQualityRetriever`: classe molto importante (ereditata direttamente da `QObject` per consentire l'uso di `slot` e `signals`) che si occupa di effettuare le richieste `HTTP GET` ai server di OpenWeather e gestire le risposte a tali richieste.

2 Polimorfismo

Due chiamate polimorfe sono presenti nel costruttore `MyTableView (Dati*)`. All'interno delle due `connect` presenti vengono chiamati i metodi (`SLOT`) `appendRows()` e `removeRows()` sull'oggetto restituito da `this->model()`; quest'ultimo è un puntatore a `QAbstractItemModel`, superclasse di `QAbstractTableModel` da cui è a sua volta derivata la classe `Dati`. Le versioni dei metodi virtuali `appendRows()` e `removeRows()` che vengono correttamente chiamate sono quindi quelle di cui si è fatto *overriding* nella classe `Dati`.

La documentazione di Qt riguardante l'implementazione del framework Model/View applicato all'uso di tabelle dichiara inoltre necessario o consigliato effettuare l'*overriding* di alcuni metodi virtuali puri presenti nella classe `QAbstractModelItem`, come per esempio `rowCount`, `columnCount`, `data`, `setData`, `headerData` e `flags`; dato che il collegamento tra la classe derivata da `QAbstractTableModel` e quella derivata da `QTableView` avviene attraverso una semplice chiamata `setModel(model)` nel costruttore di quest'ultima classe, è probabile che il framework stesso utilizzi delle chiamate polimorfe ai metodi appena citati per disegnare e gestire correttamente la tabella.

All'interno di `MyChart` sono inoltre presenti più metodi che hanno come parametri classi base (`QAbstractSeries` o derivate) ai quali in molti contesti vengono passate classi derivate su cui si effettuano correttamente operazioni di vario genere.

3 Formati di file per l'I/O

Per l'input/output di dati su file si è scelto il formato JSON. In particolare, i file che il programma può aprire correttamente e che genera in caso di salvataggio sono formattati nel seguente modo:

```
{
    "coord": {
        "lat": 0,
        "lon": 0
    },
    "list": [
        {
            "components": {
                "co": 0,
                "nh3": 0,
                "no": 0,
                "no2": 0,
                "o3": 0,
                "pm10": 0,
                "pm2_5": 0,
                "so2": 0
            },
            "dt": 0,
            "main": {
                "aqi":0
            }
        }
    ]
```

dove ogni 0 può essere un `double` (compreso il campo `dt`; le date vengono poi gestite attraverso metodi appositi) o un intero tra 1 e 5 (nel caso di `aqi`). `list`

deve contenere almeno un elemento con le caratteristiche di cui sopra.

Come si può notare, il programma è in grado di fornire informazioni riguardo alle seguenti sostanze presenti nell'aria (i cui valori corrispondono a $\mu g/m^3$):

- co (monossido di carbonio);
- nh3 (ammoniaca);
- no (ossido di azoto);
- no2 (diossido di azoto);
- o3 (ozono);
- pm10 (polveri sottili di diametro $\leq 10\mu m$);
- pm2.5 (polveri sottili di diametro $\leq 2.5\mu m$);
- so2 (anidride solforosa).

Viene anche considerato l'Air Quality Index, un indice che assume valori interi da 1 a 5 inversamente proporzionali alla pulizia dell'aria.

4 Ore impiegate

Io (Ennio Italiano) ho impiegato circa 66 ore, suddivise come di seguito:

Analisi preliminare del problema	$\sim$ 2 ore
Progettazione modello e GUI	$\sim$ 5 ore
Apprendimento libreria Qt	$\sim$ 10 ore
Codifica modello e GUI	$\sim$ 37 ore
Debugging e testing	$\sim$ 12 ore

Le ore in eccesso sono state impiegate principalmente per una migliore gestione delle eccezioni, per rendere la GUI più robusta e resistente a ridimensionamenti e per la creazione di classi "Utilities" che aumentassero la modularità del progetto.

5 Suddivisione del lavoro progettuale

Il lavoro non è stato suddiviso in modo esplicito, ma una distinzione generale può essere fatta in modo approssimativo nel seguente modo:

- Ennio Italiano
  - `ChartsChooser`
  - `Dati`
  - `MyTableView`
  - `TableChooser`

  - `DataViewer`
  - `StartWindow`

- Michele Cazzaro
  - `AirQualityRetriever`
  - `CittaEdit`
  - `Coordinate`
  - `DateDialog`

Per quanto riguarda le classi contenute nelle cartelle `MyCharts` e `MySeries` e la classe `MyChartView`, io (Ennio Italiano) ne ho realizzato una prima versione/bozza, mentre il mio collega le ha sviluppate fino alla versione finale in cui si trovano attualmente.

Va comunque considerato che tutti i file hanno subito molteplici modifiche e revisioni da parte di entrambi.

Per gestire il lavoro collaborativo è stato usato il sistema di version control `git` e il codice è stato scritto/formattato seguendo gli GNU Coding Standards per una migliore leggibilità.

6 Ambiente di sviluppo

Il programma è stato scritto e testato in un ambiente di sviluppo con le seguenti caratteristiche:

- sistema operativo: `Ubuntu 21.10`
- versione Qt: `5.15.2`
- compilatore: `g++ 11.2.0`

utilizzando l'IDE Qt Creator. È stato inoltre correttamente compilato ed eseguito sulla macchina virtuale fornita, utilizzando le istruzioni di compilazione riportate sotto.

7 Istruzioni di compilazione

Per compilare correttamente il progetto è richiesta l'installazione di `qt5-default` e `libqt5charts5-dev`. È inoltre richiesto l'utilizzo del file `AirQuality.pro` (e NON del file `.pro` generato automaticamente dal comando `qmake -project`) che garantisce una corretta compilazione. Dalla root directory del programma, saranno quindi sufficienti i comandi

```
qmake
make
./AirQuality
```

per eseguire il programma. Si forniscono inoltre i seguenti file:

- `fileVuoto.json`, necessario al programma per la creazione di file vuoti;
- `worldcities.json`, contenente l'elenco di città disponibili con le relative coordinate e utile per testare un file non supportato dal programma;
- `provaDati.json`, utile per test come file correttamente leggibile dal programma.