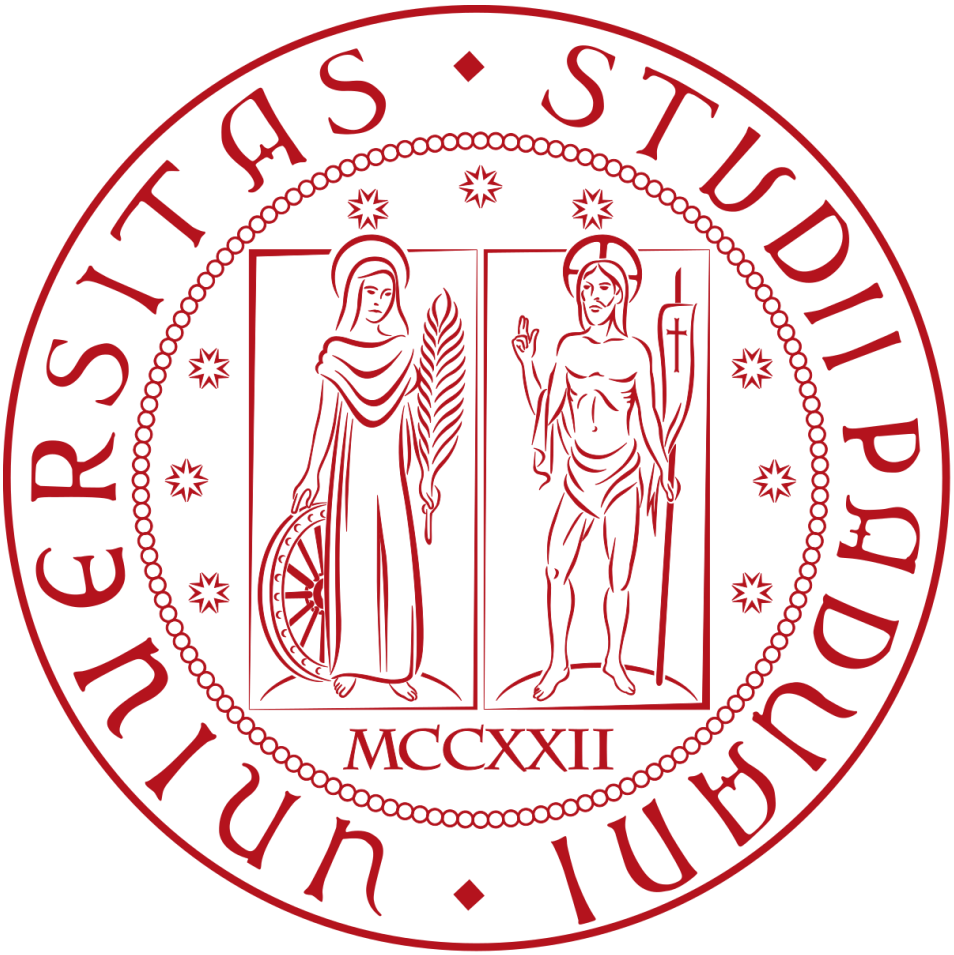
Progetto di Programmazione ad Oggetti

Anno accademico 2022/2023



Sviluppato da: **Enrik Rucaj 2016131** e Jude Vensil Braceros 2011068

Nome Progetto: Chart Project

Introduzione:

Si vuole realizzare un software dedicato alla creazione di grafici statistici tramite inserimento dei dati in una tabella, simile a quello che accade in Excel Microsoft ma in versione semplificata.

Nella prima versione rilasciata si è deciso di implementare cinque grafici nello specifico, i quali sono abbastanza diversi tra di loro proprio per cercare di soddisfare il più possibile ogni esigenza riguardo al tipo di analisi statistica che si vuole fare.

I grafici di cui si può far utilizzo sono:

- Un grafico ad Aree
- Un grafico a Barre
- Un grafico a Candela (alias grafico Azionario)
- Un grafico a Linee
- Un grafico a Torta

Si è scelto, ovviamente, di dare inoltre la possibilità al consumatore di poter salvare i vari grafici (in file di formato JSON) ed aprirli in un secondo momento.

Manuale d'uso:

Il software è abbastanza auto esplicativo riguardo alle funzioni principali di cui si vuole usufruire ma ci teniamo a descrivere altre funzioni secondarie di cui si potrebbe ignorare l'esistenza.

- In un grafico ad Aree, facendo doppio click su una qualunque area, è possibile cambiare il nome assegnata a quell'area.
- In un grafico a Barre, facendo doppio click su una qualunque barra, è possibile cambiare sia il nome della barra che il nome della categoria in cui si trova.
- In un grafico a Linee, facendo doppio click su una qualunque linea, è possibile cambiare il nome assegnata a quella linea.
- In un grafico a Torte, facendo doppio click su un una qualunque fetta, è possibile cambiare il colore della fetta (si è pensata a questa scelta diversa rispetto agli altri casi perché si è notato che nel caso vi fossero tante fette si potrebbe far fatica a distinguere i vari colori, dato che si tratta sempre di varie sfumature del blu).

Vi sono inoltre dei vincoli di cui tenere conto nel caso non si sapessero usare una certa tipologia di grafici.

- In un grafico ad Aree, i valori inseriti nella tabella che formano dei punti per una certa area devono sempre essere maggiori o uguali a quelli precedenti. Questo perché vi sono varie tipologie di grafici ad Aree in base all'uso che si vuole fare, principalmente vi sono quelli in cui le aree si sovrappongono e quelli in cui si accumulano a modo di pila; il software usa la seconda tipologia.
- In un grafico a Candela, i valori di open e close rappresentano il movimento delle azioni e nel caso il close sia minore di open il risultato sarà visibile in rosso mentre nel caso contrario sarà visibile in verde. I valori di high e low rappresentano invece il risultato massimo e minimo raggiunto nella giornata dal movimento delle azioni, si fa notare quindi che questi debbano essere obbligatoriamente i valori massimi (per high) e minimi (per low) in una certa riga. Il valore timestamp invece rappresenta da standard il numero di millisecondi passati dal 01/01/1970 ore 00:00, in seguito alleghiamo dei valori del timestamp con cui poter provare.
    - 1435708800000,
      1435795200000,
      1436140800000,
      1436227200000,
      1436313600000,
      1436400000000,
      1436486400000 ,
      1436745600000 ,

1436832000000 ,
1436918400000 ,
1437004800000 ,
1437091200000 ,
1437350400000 ,
1437436800000 ,
1437523200000 ,
1437609600000 ,

Architettura:

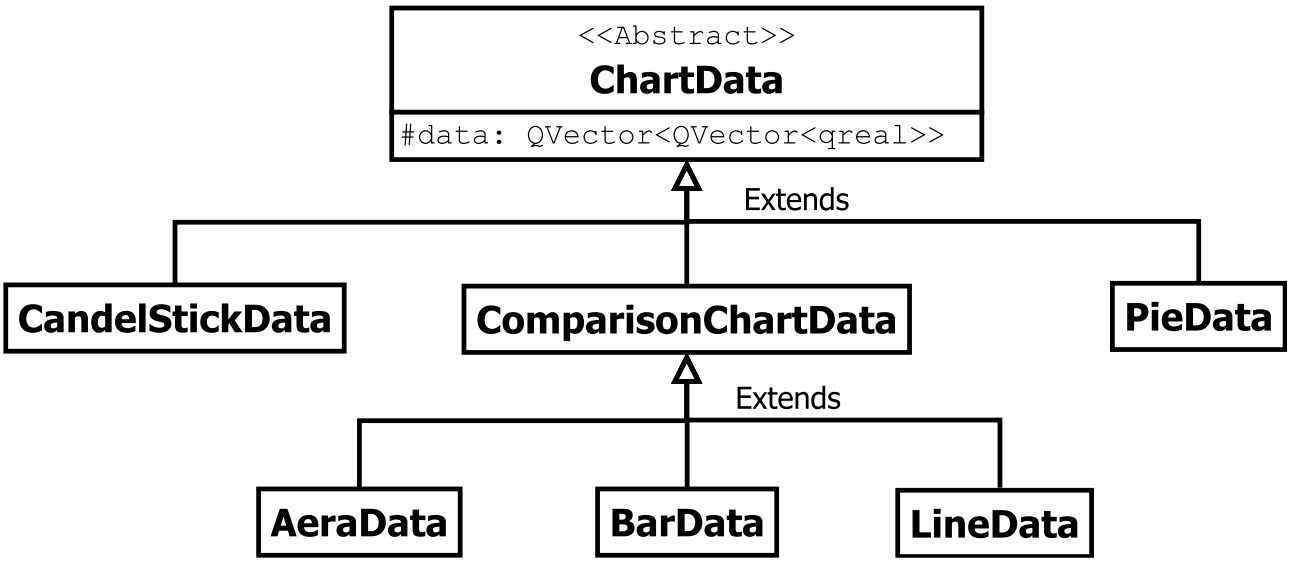
Per lo sviluppo si è preferito scegliere il pattern **Model – View** rispetto al **Model – View – Controller (MVC)** in quanto il secondo non è direttamente supportato da Qt.
La creazione delle classi per la sezione Model è risultata abbastanza grande e perciò si è scelto di concentrare in quel punto l'utilizzo di una gerarchia e applicazione del polimorfismo abbastanza efficiente.

Gerarchia e polimorfismo:

Si è scelto di dividere la sezione Model in tre sotto sezioni: **Data** – **TableModel** – **Model**.

- In cui le classi Data servono per creare una struttura dati in qui memorizzare le informazioni.
- In cui le classi TableModel servono per descrivere come si comporta la struttura dati al cambiamento di essi nella tabella.
- In cui le classi Model servono per descrivere come vengono interpretati i dati nella tabella per visualizzare la view.

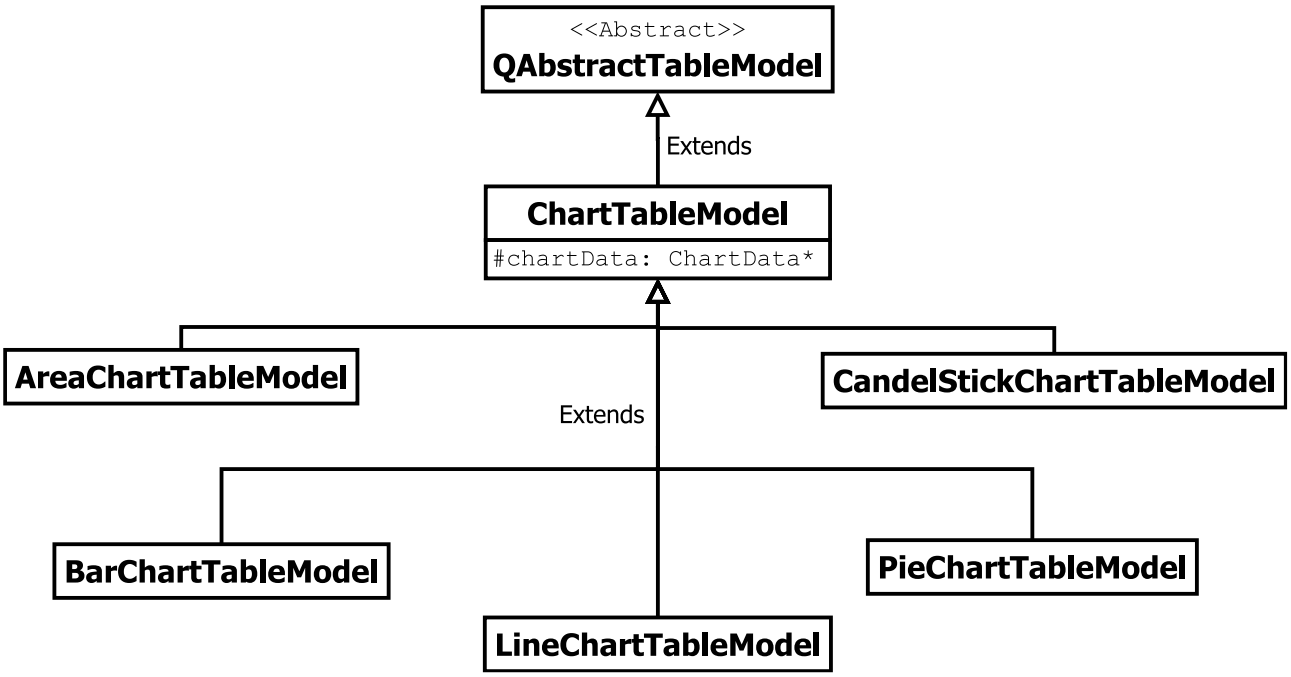
Sezione Data:



- ChartData è una classe astratta che estende tutte le altre classi che memorizzano le informazioni del loro tipo di grafico. È composta da due campi che sono comuni a tutti i grafici: Dati e Titolo.
  È astratta in quanto in essa viene usato un metodo virtuale puro dataInit() utile a interpretare le diverse strutture dati in un unico tipo universale composto da un vettore a due dimensioni del tipo QVector<QVector<qreal>> (si è scelto di non usare QVector2D in quanto la versione di qt usata nella macchina virtuale è la 5.9.5 e QVector2D viene usata nella versione 6 di qt).

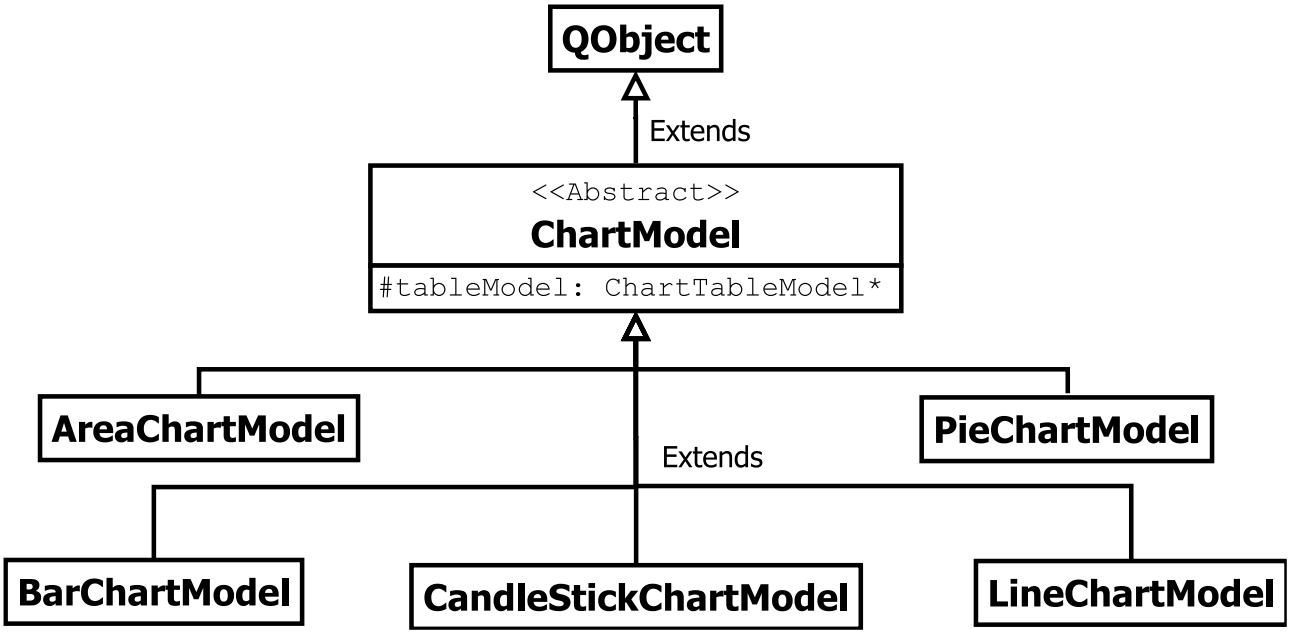
- ComparisonChartData è una classe base in cui non viene fatto un uso eccessivo del polimorfismo ma viene usata solo come classe ereditaria dai suoi figli. Essa rappresenta quei tipi di grafici in cui vi possano essere più "entità" che si confrontano tra di loro. Ad esempio in un grafico a linee, le entità corrispondono alle varie linee e/o in un grafico a barre, le entità corrispondono alle varie barre dello stesso tipo.

Sezione TableModel:



- QAbstractTableModel è una classe di qt di cui vi è già presente una documentazione dettagliata nel sito, viene citata qui soltanto per ribadire che tutte le classi di questa sezione si occupano soltanto di fare un override dei suoi metodi (puri e non).
- ChartTableModel è un semplice wrapper di QAbstractTableModel, esso aggiunge in più un campo puntatore alla classe base ChartData della sezione Data, utile per i fare le varie operazioni di inserimento/rimozione/modifica dati nelle tabelle. È importante specificare inoltre che questa classe non è astratta in quanto (involontariamente) fa un override dei metodi virtuali puri di QAbstractTableModel.

Sezione Model:



- ChartModel è l'unica classe degna di nota di questa sezione. È la classe base che accomuna tutte le altre tramite i loro due campi principali che verranno usate per la visualizzazione dei grafici: ChartTableModel (già discusso sopra) e QChart la quale conterrà una serie di dati che verranno visualizzati nella view. Essa è astratta perché contiene un metodo virtuale puro utile al salvataggio dei dati in formato JSON.
- Tutte le altre classi invece si occupano inoltre del mapping del modello con QChart tramite i ModelMapper forniti dalla documentazione.

Lato polimorfismo sono stati usati tantissimi metodi virtuali (oltre a quelli già citati) soprattutto nelle due classi astratte ChartData e ChartModel e nella classe ChartTableModel (anche se questa non conta molto dato che è una semplice estensione di QAbstractTableModel). Gli esempi più importanti da far notare sono:

- virtual void ***setData***(int x,int y, qreal val); **(In ChartData)** che permette di modificare qualsiasi cella di tabella in tutte le tipologie di grafici. Nel caso si volesse aggiungere un altro grafico, basterebbe fare un override di questo metodo per adattarlo ai propri bisogni nel caso venisse modificata una cella.
- virtual void ***salvaJson***() =0; **(In ChartModel)**. Una volta chiamato dalle view questo metodo slot permette di selezionare i corretti passaggi di salvataggio dei file dei vari tipi di grafici senza doverne creare un nuovo metodo per ciascuno (grafico).
- virtual void ***updateInsertRow***() {} **(In ChartModel)** che permette di aggiornare i dati memorizzati ogni qualvolta si debba inserire una nuova riga in qualsiasi tipo di grafico.

(Idem per altri metodi molto simili a questo i quali verranno soltanto elencati senza una spiegazione qui sotto.)

- virtual void ***updateInsertColumn***() **(In ChartModel)**
- virtual void ***updateRemoveRow***(int pos) **(In ChartModel)**
- virtual void ***updateRemoveColumn***(int pos) **(In ChartModel)**

Formato I/O per i file:

Per la scrittura (salvataggio) e lettura dei dati in file con una struttura dati organizzata, si è scelta la struttura JSON perché è un semplice formato per lo scambio di dati. Per le persone è facile da leggere e scrivere, mentre per le macchine risulta facile da generare e analizzarne la sintassi.

I dati interpretati come modelli vengono salvati nel formato JSON con una struttura ben specifica per ciascun tipo di grafico. Per il test principale che viene usato per capire durante l'apertura del file di quale tipo di grafico si tratta, si ispeziona il campo **{"Type": "…"}** dove al posto dei tre puntini viene inserito il tipo di grafico salvato (sotto forma di stringa).

Si fa notare inoltre che nel caso la struttura dati non risultasse come quella aspettata (file corrotto quindi), il file non viene aperto e in seguito vi è un warning dell'avvenuto fallimento.

Nella cartella graficisalvati sono stati messi a disposizione vari file in formato JSON sulle quali poter fare delle prove. Tra queste è stato aggiunto pure un file corrotto.

Note aggiuntive:

Si è preferito fare uso dei tipi definiti da Qt, ad esempio QString, QVector, QMap, qreal, ecc., in modo da ridurre le conversioni implicite e aumentare l'efficienza (al momento questa scelta non influisce nel tempo di esecuzione del programma ma si suppone che in futuro potrebbe essere utile).

Istruzioni di compilazione ed esecuzione:

Si suppone che si voglia compilare/eseguire il programma tramite terminale (linea di comando).

Prima di compilare si avvisa che servono i pacchetti **qt5-default** e **libqt5charts5-dev**.

Per la compilazione bisogna trovarsi nella cartella in cui è presente il file .pro ed eseguire su terminale i seguenti due comandi in ordine:

- **qmake**
- **make**

Per essere eseguito invece basta inserire il comando: **./Charts_project**

Divisione dei compiti nel gruppo:

Enrik Rucaj (Sottoscritto) si è occupato principalmente della definizione di gerarchie , polimorfismo e di alcune classi nella view (si intende l'uso e creazione di alcuni widget per poter visualizzare le view).

Jude Vensil Braceros si è occupato principalmente della separazione tra il Model e la View e salvataggio dei file in formato JSON.

Oltre a quello sopra specificato generalmente ci siamo occupati entrambi di un po' di tutto, soprattutto per la soluzione ad alcuni bug iniziali molto fastidiosi.

Ore di lavoro:

Analisi dei Requisiti	2 Ore
Documentazione Qt	6/7 Ore
Progettazione GUI	3 Ore
Progettazione e Sviluppo Gerarchie	30 ore
Aggiunta del Polimorfismo	7 Ore
Fase di Testing e Debuging	8 Ore

In totale sono servite sulle 56/57 Ore. Il problema principale per la quale sono state sfiorate le 50 ore consiste nell'aver commesso l'errore di pensare/sviluppare una gerarchia non all'inizio del progetto ma dopo un bel po'. Questo ha richiesto la ridefinizione di molte classi, operazione abbastanza longeva.

Ambiente di sviluppo:

Il progetto è stato sviluppato sul sistema operativo **Windows 10** con IDE **Qt Creator**, compilatore **Mingw 10.2.0(64 bit)** e versione di Qt **5.9.5.** Il programma è stato comunque testato sulla macchina virtuale con abbastanza facilità. Si è fatto uso inoltre di GIT per avere una storicizzazione dei vari cambiamenti che venivano fatti (tramite i commit) e per suddividere il lavoro con l'altro compagno tramite apposite branch.

Nota: Sono stato molto ostile all'uso di Qt Creator all'inizio, ma dopo averci preso la mano, direi che non poteva capitarmi di meglio grazie a molte funzionalità e semplificazioni che offriva.