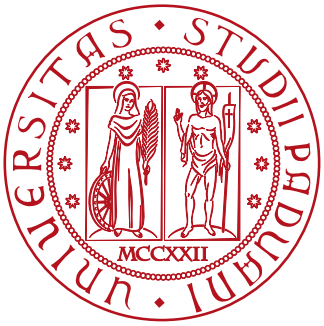

BasedCharts*

Progetto di Programmazione ad Oggetti

Augusto Zanellato

2000555

Anno Accademico 2021/2022

*Significato di *Based* (da UrbanDictionary): A word used when you want to recognize someone for being themselves, i.e. courageous and unique or not caring what others think.

Indice

1 Introduzione al progetto

1.1 Abstract

Si vuole realizzare un'applicazione che permette di lavorare con vari tipi di grafici usando il framework Qt in C++. Le operazioni implementate sono:

- Creazione
- Salvataggio
- Caricamento da file
- Modifica dei dati
- Modifica degli attributi del grafico, come ad esempio:
  - Colore delle serie di dati
  - Range degli assi con eventuale autoscaling
  - Titolo del grafico
  - Titolo degli assi
- Passaggio tra tema chiaro e scuro

I tipi di grafico supportati sono:

- Barre
- Linea (ad assi X collegati o indipendenti)
- Dispersione (detto anche Scatter) (ad assi X collegati o indipendenti)
- Spline (ad assi X collegati o indipendenti)

Ogni tipo di grafico supporta l'uso contemporaneo di multiple serie di dati.

1.2 Implementazione

Le scelte architetturali più significative del progetto sono:

- Supporto all'uso di schede per operare su più file contemporaneamente
- Utilizzo di JSON come formato di salvataggio dei dati
- Utilizzo del paradigma Model-View
- Lo sviluppo del contenitore custom `Vector2D` da utilizzare come storage per il modello
- Utilizzo di unit test per le parti più prone a bug, come ad esempio la struttura di dati `Vector2D`
- Utilizzo di linter (nello specifico `clang-tidy` e `clazy`) per evidenziare possibili bug o brutture
- Utilizzo dei *sanitizer*[1] di **clang** durante lo sviluppo per rilevare eventuali *undefined behavior* o accessi in memoria non validi e/o malformati.
- Supporto dell'applicazione a temi multipli
- Uso del pattern *Self-Registering Factory* (per ulteriori dettagli si veda 2.3.2)
- Ampio utilizzo di *namespace* per separare logicamente il codice.

Nella cartella `examples/` vengono forniti alcuni grafici di esempio, nella sottocartella `examples/invalid/` sono presenti alcuni file **non validi** usati per testare la correttezza della logica di deserializzazione.
NOTA: si è scelto di non fornire un manuale d'uso dell'applicazione in quanto la GUI è molto autoesplicativa dato che passando il cursore del mouse sulla maggior parte dei pulsanti appare nella *status bar* in basso una breve descrizione del comportamento del pulsante in questione.

[1] Di particolare interesse sono stati UBSan, MSan ed ASan

2 Architettura

Lo sviluppo adotta il pattern **Model-View** di Qt, non è stato usato il pattern **MVC** (**Model-View-Controller**) in quanto non direttamente supportato da Qt e avrebbe reso necessario aggirare alcune delle limitazioni da esso imposte. Oltre alla gerarchia di classi polimorfa usata per gestire i vari tipi di grafico supportati è stato sviluppato anche un modello per rappresentare i dati `ChartDataModel`, una struttura di dati per il salvataggio dei dati inseriti in forma tabellare. In questa sezione della documentazione verranno analizzate le classi principali e alcune delle scelte progettuali che hanno contribuito a definire la gerarchia utilizzata.

2.1 Progettazione

2.1.1 Gerarchia

La gerarchia principale del progetto (rappresentata nella figura 2.1.1) è quella che interessa le classi che gestiscono i vari tipi di grafico.

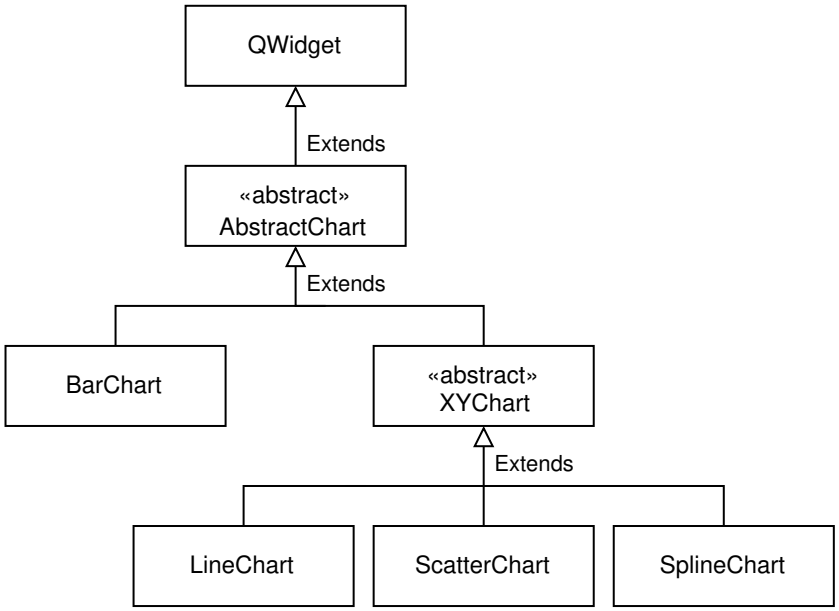

Figura 1: Parte della gerarchia delle classi d'interesse

È presente una classe base astratta (`AbstractChart`) che contiene la logica di base d'inizializzazione e gestione di un grafico comune ai vari tipi, `AbstractChart` eredita inoltre da `QWidget` in modo da renderla utilizzabile come widget di Qt.
`AbstractChart` contiene dei **campi pubblici costanti** che vengono inizializzati adeguatamente dalle sottoclassi i quali indicano se gli assi X ed Y devono essere personalizzabili oppure no (ad esempio un grafico a barre non consente la personalizzazione dell'asse X in quanto contiene delle categorie e non dei valori) e se le intestazioni di righe e colonne del modello devono poter essere modificabili.
Ogni sottoclasse di `AbstractChart` implementa alcuni **metodi virtuali puri** che forniscono alcune informazioni utili al resto del programma, come ad esempio se il grafico corrente richiede un numero pari di colonne nel modello (come ad esempio i grafici che estendono `XYChart` quando sono nella configurazione ad assi X indipendenti); altri metodi virtuali puri si occupano di serializzare e deserializzare gli eventuali attributi addizionali non presenti in `AbstractChart`; sono inoltre presenti alcuni *slot* astratti.

XYChart I grafici che estendono `XYChart` sono quei grafici i cui punti dati possono essere collocati mediante una coppia di coordinate cartesiane (X, Y). La scelta di usare un'astrazione comune è stata *presa in prestito* da Qt [2], è risultata una scelta vincente in quanto tutta la logica per gestire questo tipo di grafico

[2] Nel modulo Charts di Qt è presente una classe base astratta `QXYSeries` da cui derivano le classi `QLineSeries`, `QScatterSeries` e `QSplineSeries`

è in `XYChart`, le classi che la estendono devono solo implementare due metodi oltre al costruttore e alla registrazione con la factory: uno per fornire il tipo dinamico come stringa (si veda 2.3.1 per chiarimenti) e uno per creare una nuova serie di dati (questo metodo non è altro che un sottile wrapper attorno a i costruttori delle relative classi)

2.1.2 La struttura dati `Vector2D`

Per salvare i dati inseriti nella tabella è stato deciso di sviluppare una struttura dati *ad-hoc* che si comporta come un array a 2 dimensioni ma rispetto a quest'ultimo aggiunge alcuni metodi di utilità, come ad esempio la rimozione e l'inserimento di righe e colonne in determinate posizioni.
L'implementazione punta ad ottimizzare la località dei dati, viene infatti utilizzato un singolo array allocato dinamicamente nello heap, in questo modo tutti i dati contenuti saranno in locazioni di memoria contigue, a differenza di quanto si sarebbe ottenuto usando un `std::vector<std::vector<T>>` (o l'equivalente con `QVector`), in quanto ogni `std::vector` conterrebbe un puntatore alla locazione contenente i dati, ma nulla assicurerebbe la contiguità delle varie locazioni l'una rispetto all'altra.
Dettaglio degno di nota è il fatto che il costruttore di copia è eliminato, questa scelta è volta all'evitare onerose copie involontarie causate da sviste o dalla dimenticanza di un `&` in qualche firma di metodo.

2.1.3 Il modello

Il modello `ChartDataModel` estende la classe astratta `QAbstractTableModel` fornita da Qt, per memorizzare i dati della tabella usa un `Vector2D`. Oltre ai già citati dati memorizza anche le intestazioni sia delle righe che delle colonne (se previste ovviamente), salva anche eventuali colori di sfondo delle colonne i quali corrispondono ai colori delle corrispondenti serie di dati del grafico corrente.
Altri dati salvati dal modello degni di nota sono il numero di colonne minimo e il numero di colonne protette. Le colonne minime sono abbastanza autoesplicative, le colonne protette sono invece delle colonne che non possono essere eliminate: ad esempio i grafici che estendono `XYChart` in configurazione ad asse X singolo hanno la prima colonna del modello protetta (pur avendo un minimo di due colonne) in quanto eliminare la colonna che contiene i dati dell'asse X è un'operazione insensata.

2.1.4 `ThemeableAction`

Qt non supporta nativamente icone associate ad un tema che può cambiare a run-time, per aggirare questa limitazione è stata sviluppata la classe `ThemeableAction` che non è altro che un sottile wrapper attorno a `QAction`; in aggiunta ha uno slot che viene invocato quando l'applicazione cambia tema che provvede ad aggiornare l'icona della `QAction` in modo da usare le icone appropriate in base al tema in uso.

2.2 Polimorfismo

È stato fatto ampio uso del polimorfismo nella definizione della classe `AbstractChart`, soprattutto per quanto riguarda la serializzazione e deserializzazione dei dati (per ulteriori dettagli si veda 2.3). Altri usi del polimorfismo nell'ambito dei grafici sono ad esempio i metodi usati per determinare valore minimo e massimo degli assi quando viene effettuata la scala automatica degli assi. Sono inoltre definiti degli slot polimorfi (`updateLabels` e `modelResized`) i quali aggiornano il grafico in risposta a cambiamenti del modello in modo specifico al tipo di grafico in questione.

2.3 Gestione dei dati

Per il salvataggio e il caricamento di dati si è scelto l'uso del formato standard **JSON** in quanto è leggibile da umani e meno verboso rispetto a **XML**.
L'unico punto a sfavore dell'uso di **JSON** in Qt è che non è presente il supporto a *JSON Schema* [3], il quale avrebbe risparmiato la scrittura di molta della logica di validazione dei file.
La logica di serializzazione e deserializzazione è divisa in due parti:

- Riguardante il modello: si occupa dei dati contenuti nella tabella, le eventuali intestazioni di righe e colonne e i colori associati alle serie di dati.

[3] *Json Schema* è uno standard per la definizione della struttura di un documento *JSON* e la relativa validazione.

- Riguardante i grafici: si occupa delle proprietà specifiche dei grafici come il titolo, le proprietà degli assi ed eventuali altre proprietà specifiche

2.3.1 Serializzazione

La serializzazione dei grafici è gestita mediante il metodo pubblico

```
void serialize(QJsonObject& json) const
```

presente nella classe `AbstractChart`, il quale si occupa di serializzare gli attributi comuni a tutti i grafici (quelli quindi presenti in `AbstractChart`) tra cui il tipo dinamico (ricavato mediante un'invocazione al metodo virtuale puro `type()`), e di invocare il metodo virtuale

```
void internalSerialize(QJsonObject& json) const
```

il quale si occupa della serializzazione degli attributi specifici del tipo di grafico in uso; il metodo prevede un'implementazione di default vuota in quanto un grafico non deve per forza avere altri attributi da deserializzare.

2.3.2 *Self-Registering Factory* e deserializzazione

Per aumentare la facilità di estensione e per evitare di dover modificare la *factory* ad ogni aggiunta di un tipo di grafico si è pensato di adottare il pattern *Self-Registering Factory*. Differisce da una semplice *factory* in quanto le classi non vanno registrate centralmente ma ogni classe provvede a registrare se stessa. All'atto pratico questo consiste nell'avere un campo statico booleano all'interno di ogni classe che deve poter essere costruita dalla factory inizializzato così

```
bool registeredWithFactory = Factory::registerClass(MyClass::TYPE, &MyClass::deserialize);
```

È inoltre buona norma marcare questi campi con `__attribute__((unused))` in modo da comunicare al compilatore che è nostra intenzione che la variabile in questione non sia mai utilizzata, in modo da evitare warning inutili e per evitare che il compilatore elimini la variabile in caso il codice venga compilato con le ottimizzazioni abilitate. Il principale vantaggio di questa modalità rispetto ad una più classica *factory* è che per supportare un nuovo tipo di dato non servono modifiche alla *factory*.
Ciò che permette alla factory di funzionare a basso livello è una mappa (in questo caso viene usata una `QMap`, ma si avrebbe potuto usare anche una `std::unordered_map` senza particolari differenze) che ha come chiavi i tipi di dato che è possibile deserializzare (quindi il risultato delle varie implementazioni di `AbstractChart::type()`) e come valori i metodi statici delle rispettive classi che istanziano un nuovo grafico a partire da un modello (i quali altro non sono che un sottile wrapper attorno al costruttore), sul grafico così costruito viene chiamato il metodo virtuale

```
AbstractChart::deserialize(const QJsonObject& obj)
```

il quale si occupa della deserializzazione degli attributi del grafico.
Uno dei problemi incontrati durante lo sviluppo della *factory* è stato il fatto che in C++ l'ordine di inizializzazione dei campi statici è definito dall'implementazione e quindi non si possono fare assunzioni. Per evitare di ricadere nell'undefined behavior si è scelto di rendere la *factory* un *singleton* internamente, con i vari metodi pubblici statici che invocano dei metodi privati di esemplare che fanno le dovute operazioni.

2.4 La GUI (Graphical User Interface)

La finestra principale (`MainWindow`) della **GUI** è basata sulla `QMainWindow` fornita da Qt, è inoltre presente un *wizard* (`ChartWizard`) per aiutare l'utente a creare nuovi grafici e caricarne di esistenti.
Tutta la logica di presentazione del grafico è racchiusa nella classe `MainView`, inizialmente è stato fatto per aumentare il *decoupling* tra la parte grafica della finestra e la logica di raccordo tra GUI e l'accoppiata modello e grafico. In seguito questa divisione è tornata utile per l'implementazione delle schede che consentono di lavorare a più grafici contemporaneamente usando una sola istanza dell'applicazione.
Sono anche presenti alcuni dialoghi modali (`EditChartDialog` e `EditColorsDialog`) pensati per modificare le proprietà del grafico in modo da non inquinare troppo l'interfaccia utente.

2.5 Note di implementazione

- In `MainWindow` viene fatto uso solo di `static_cast` per i widget delle tab senza nessun genere di controllo in quanto non necessario dato che le tab possono essere solo di tipo `MainView*`
- Nella struttura dati `Vector2D` sono presenti degli usi di `assert()` che controllano la validità di determinate precondizioni per la correttezza delle funzioni dove sono poste. Le invocazioni sono state inserite nelle fasi di sviluppo iniziale del modello e mai rimosse in quanto non hanno nessun costo nelle *release builds* dell'applicazione.
- È stata intenzionalmente violata la regola di separazione della *business logic* dalla parte di GUI nel metodo `ChartDataModel::removeColumns` in quanto è il modello a mostrare un dialogo d'errore in caso di rimozione non valida, è stata una scelta pensata per ridurre al minimo la duplicazione del codice nei punti in cui quel metodo viene invocato (ovvero in `MainWindow` ed in `EditChartDialog`). Un altro motivo per cui si è preferita questa strada è che Qt in caso di rimozione fallita dal modello non prevede l'uso di eccezioni o di codici di errore ma di un banale `return false;` il quale non è abbastanza descrittivo e complica la futura estensibilità del codice in caso di aggiunta di motivi diversi per il fallimento della rimozione.
- Nei grafici che estendono `XYChart` non è imposto l'ordinamento dei valori della tabella sull'asse X, inizialmente questa scelta può provocare confusione ma è stata fatta per lasciare maggior libertà possibile all'utente.
- Dove possibile è stato preferito usare tipi di Qt (ad esempio `QString`, `QVector` e `QMap`) al posto degli equivalenti della **STL** per ridurre la necessità di onerose conversioni (implicite ed esplicite); questa decisione non ha ovviamente inficiato sulla separazione tra dati e logica di presentazione.
- Per il *mapping* del modello in serie di dati compatibili con la classe `QChart` sono stati utilizzati i `ModelMapper` forniti da Qt, i quali hanno ampiamente semplificato la gestione degli eventi del modello.
- Per semplicità implementativa si è scelto di non ammortizzare le allocazioni nella struttura dati `Vector2D`; ogni operazione che modifica le dimensioni del `Vector2D` provoca quindi una riallocazione (con conseguente copia degli elementi contenuti). È stata giudicata una scelta accettabile in quanto saranno sicuramente più frequenti gli accessi agli elementi rispetto ai ridimensionamenti.

3 Conclusioni

3.1 Timeline e conteggio ore di lavoro

Il progetto è stato iniziato a fine Novembre poco dopo la presentazione, in quel momento sono stati analizzati i requisiti e abbozzata la **GUI**.
Per quanto riguarda il computo delle ore di lavoro sono state necessarie **22 ore aggiuntive** rispetto alle **50 ore** previste dalla specifica di progetto, dovute principalmente al perfezionamento delle funzionalità e dell'interfaccia nonché al miglioramento della struttura del codice.

Fase progettuale	**Ore impiegate**
Analisi delle specifiche	3
Setup toolchain	3
Progettazione e bozza GUI	6
Studio del framework Qt	8
Progettazione, implementazione e unit testing della struttura dati	10
Progettazione e implementazione del modello	8
Progettazione e implementazione della gerarchia relativa ai grafici	10
Implementazione del salvataggio e caricamento dei dati in JSON	8
Sviluppo del wizard di creazione del grafico	2

Test e debugging[4]	6
Stesura documentazione	8
Ore Totali	**72 ore**

3.2 Ambiente di sviluppo

Nello sviluppo del progetto si è scelto di utilizzare come IDE **CLion**, un ambiente professionale sviluppato dalla compagnia JetBrains, essendo a pagamento è stata utilizzata una licenza gentilmente messa a disposizione dagli sviluppatori mediante il loro programma per studenti. Questa scelta ha portato inoltre all'uso di **CMake** come *build automation system* e di **Ninja** come *build system* invece dell'accoppiata **QMake**[5]-**Make**; questa scelta è stata presa anche per prendere confidenza con il tooling *standard de-facto* utilizzato nello sviluppo in C++.
L'utilizzo di CMake ha permesso inoltre l'uso agevole ed integrato con l'IDE di *linter* e *static analyzers*.
Lo sviluppo è avvenuto usando l'ultima versione di **Qt5** (**5.15.2**) [6] e l'ultima versione di **Clang** (**13.0.1**) in esecuzione su **Arch Linux**; sono stati effettuati anche periodici test sulla macchina virtuale del corso[7].

3.3 Compilazione ed esecuzione

È stata resa possibile la compilazione del progetto anche con **QMake** mediante file `.pro`, di seguito le istruzioni:

```
$ qmake
$ make
```

L'esecuzione di questi comandi porta alla compilazione del progetto con destinazione `./BasedCharts` dove `.` è la cartella dove è presente il file `.pro`, tutti i file intermedi generati dalla toolchain di Qt e dal compilatore sono posti nella cartella `./build/qmake`.

3.4 Note finali

Lo sviluppo del progetto è stato un'attività molto interessante che mi ha coinvolto molto piacevolmente, credo andrebbero inseriti più progetti del genere nel curriculum universitario.
Le maggiori difficoltà incontrate nello sviluppo sono state dovute alla documentazione lacunosa di Qt, soprattutto per quanto riguarda l'uso dei *ModelMapper*, si sono presentate diverse occasioni in cui è stato più produttivo visionare i sorgenti di Qt rispetto a leggere la documentazione.

[4] Stima molto approssimativa in quanto sono stati distribuiti durante tutto lo sviluppo

[5] Inoltre QMake sembra essere in via di abbandono dagli sviluppatori di Qt, con l'ultima versione *major* di Qt (6.0) sono infatti passati ad usare CMake per gestire le build di Qt stesso

[6] Fortunatamente questo ha portato solo ad esattamente due incompatibilità con la versione *target* di Qt, entrambe risolte rapidamente.

[7] Degno di nota è il fatto che l'installazione di Qt sulla VM non è completa, mancano molti pacchetti presenti nella distribuzione di default di Qt e soprattutto il pacchetto `qt5-default`, indispensabile per far compilare il progetto a **QMake**