Progetto Programmazione ad Oggetti 2021/2022

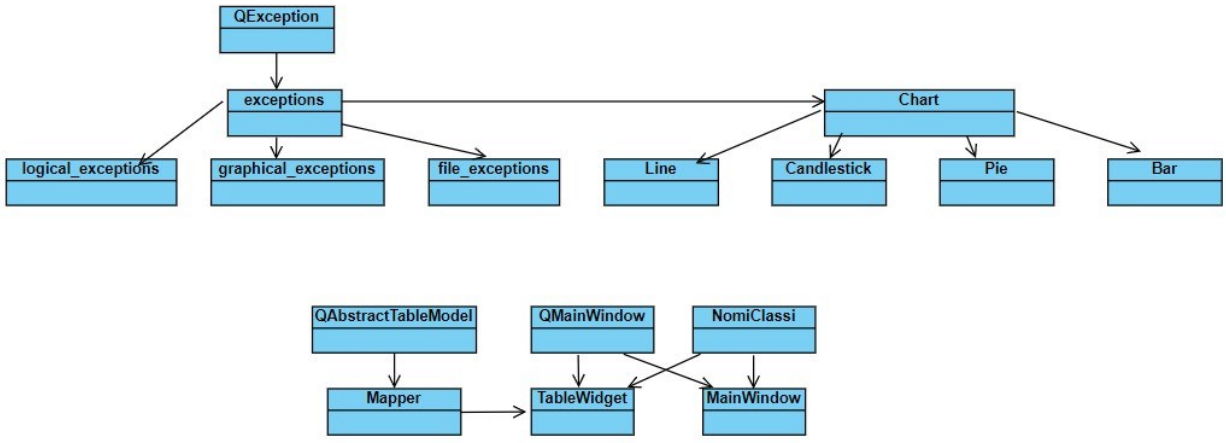
Qt Charts CR

qt/qtcharts

QtCharts module

Creatori progetto
Andrea Crocco – Matr 1226135
Gabriel Rovesti – Matr. 2009088

Introduzione/Abstract



L'applicazione sfrutta una gerarchia grafica composta da una superclasse "*Chart*" astratta che viene reimplementata dalle 4 sottoclassi (*Line, Pie, Bar, CandleStick*).

La classe *Chart* compone la parte logica dell'applicazione, in particolare prendendo in input dati definiti dall'utente e/o dati secondo una struttura definita da file. È una classe astratta in quanto implementa una serie di metodi virtuali che permettono di inserire dati sulla base dei tipi di grafici presenti, eventuale impostazione degli assi e impostazione finale del grafico una volta impostati i dati.

Per poter gestire in maniera estensibile la quantità di grafici ed opzioni presenti, l'applicazione si compone di appositi metodi e strutture dati; separatamente viene inclusa la gestione delle eccezioni (classe apposita derivata da *QException*), quindi situazioni/problematiche grafiche/runtime/file gestendo nella maniera più indicata la situazione eccezionale.

Per gestire la parte grafica, il progetto si compone di una gerarchia di gestione separata della finestra principale e dell'oggetto grafico tabella mostrato all'utente per permettere la modifica dei dati a schermo con superclasse comune. Nello specifico, la clase *TableWidget* permette, una volta inseriti logicamente i dati, di visualizzarli sotto forma di widget interattivi sfruttando il mapping fornito preventivamente da classe apposita. Allo stesso livello di questa classe ce n'è una apposita (*NomiClassi*) che consente di estendere a seconda delle necessità dell'utente i tipi di grafici disponibili. Parallelamente a questa esiste la classe Mapper che consente la gestione della modifica dell'utente in formato tabellare, mappando a seconda del tipo di grafico valori e colori, fornendoli alla classe widget apposita; questa classe deriva da *QAbstractTableModel* che fornisce un interfaccia personalizzabile sempre sulla base dei tipi presenti.

Si segnala inoltre nel metodo main la creazione dell'applicazione con gestione personalizzata dei segmentation fault con finestra di selezione e salvtaggio in un file log dei crash dell'applicazione, oltre che l'impostazione del titolo, dello splashscreen e della resa grafica della applicazione.
L'applicazione non aderisce a nessun pattern di modellazione specifico, in quanto sviluppata incrementalmente come descritto sotto; è pur da segnalare la visione della classe Mapper come Model tale da far visualizzare i dati alla View, essa descritta dalla classe TableWidget e MainWindow, che controlla ed imposta dinamicamente widget e slot sulla base dell'interazione specifica dell'utente.

Parte logica: specifica metodi/strutture

- *Chart*

Essa si presenta come la classe principale della applicazione, estendendo il plugin QT Charts, incluso nello stesso file header.
3 tipi di parametri utilizzati: gestione dati input, gestione estensibile di grafici/opzioni/serie, gestione dei file I/O. Struttura dei dati predefinita secondo il tipo QStringList, incapsulata dentro altre strutture per mantenere l'estensibilità. Per quanto riguarda gli assi, per l'asse X vengono gestite solo categorie con il tipo *QBarCategoryAxis* (coerentemente con le etichette) e per l'asse Y (di soli valori numerici), utilizzando il tipo *QValueAxis*. Gli assi hanno appositi valori di minimo e di massimo.

Descrizione metodi:

- ***insert***, metodo virtuale puro per inserire dati all'interno della struttura grafico;
- setting di etichetta/valore nelle serie/assi;
- setting personalizzato (aggiunta/rimozione valore) del singolo asse;
- setting personalizzato del nome del grafico;
- ***setaxis***, metodo virtuale pure di impostazione personalizzata degli assi;
- ***reloadGrafico***, ricrea il grafico sulla base dei valori/etichette presenti e utilizzato dal Model ;
- ***salvaGrafico***, permette di salvare lo stato delle variabili quando viene importato un file e quando si passa in altro tipo di grafico;
- ***svuotaGrafico,*** resetta le impostazioni qualora venga importato un file per evitare valori duplicati;
- ***setOption,*** in base alle opzioni presenti ed eventualmente aggiunte tramite la struttura QHash permette il salvataggio delle opzioni logiche per poterle riutilizzare graficamente;
- ***finalizza***, che imposta il grafico con serie e assi sulla base dei vincoli posti da QChart (nell'ordine, aggiunta valori, aggiunta serie, attaccare assi qualora non presenti). Ciò cambia in base al tipo di grafico;
- metodi di raccolta dei dati/file;
- ***getTipoGrafico***, dinamicamente gestisce il tipo grafico;
- ***importa,*** in base al file aperto dall'utente crea la struttura del grafico;
- ***esporta***, in base ai due tipi XML/JSON di file gestibili dal progetto legge la struttura dei dati e li esporta sulla base della struttura DOM di XML (struttura ad albero, da noi personalizzata) e JSON, adattando il tutto alle strutture QHash presenti.
- clonazione/distruzione polimorfa degli assi, valori, serie ed etichette;

- overloading dell'operator ==

Per i 4 sottotipi si descrivono i seguenti metodi in tutti presente:
- clonazione polimorfa del tipo di grafico attuale
- costruttore con valori di default predefiniti marcato esplicito
- metodi virtuali concreti per setting asse (*setAxis*), creazione del grafico e gestione modifiche qualora realizzate nella View (*reloadGrafico*), impostazione valori/serie del grafico a linee per mostrarli a schermo (*finalizza*) seguendo l'ordine descritto sopra.

Seguono i sottotipi:

1) Line: Usa un parametro privato di tipo QLineSeries per settare la serie coerentemente col suo tipo.
2) Pie: Usa un parametro privato di tipo QPieSeries per settare la serie e il parametro *isDonut*, fondamentale per discriminare se è un grafico a torta o a ciambella. Questo è utile nella fase di scelta nella GUI con apposita finestra.
3) Bar: Usa un parametro privato di tipo QBarSeries per settare la propria serie
4) *CandleStick:* Usa un parametro privato di tipo *QCandleStickSeries* per settare la propria serie e *nomeSerie* per gestire in modo personalizzato il nome della serie.

- exceptions

La gestione delle eccezioni, come anticipato, avviene con una classe apposita derivata da QException:
Usa un parametro privato di tipo *QString* per gestire in maniera apposita le eccezioni, a seconda che esse avvengano nella parte logica, grafica o avvengano nell'apertura/gestione dei singoli file.
A tale scopo è adibita una gerarchia di 3 sottoclassi, nello specifico *file_exceptions, logical_exceptions,, graphical_exceptions,* ciascuna dotata di apposito costruttore per poter distinguere il tipo di messaggio e poter correttamente gestire l'eccezione, terminando il programma nei casi eccezionali (ad esempio apertura errata/errore nel caricamento di file XML/JSON e rimozione di valori da assi/valori se vuoti).
Più nello specifico nel caso della classe *graphical_exceptions*, si cita la gestione gli errori possibili nel Mapper, ad esempio nel caso di una posizione di mapping fuori range (-1), comprendendo però la possibilità di valori nulli per rimuovere/aggiungere righe e colonne senza interrompere il programma oppure di errori negli widget nulli, nel caso non presenti per evitare possibili segmentation fault (errori di tipo Table).
Per tutte si gestiscono situazioni impreviste nel caso JSON/XML (*file_exceptions).*

Parte GUI: specifica metodi/strutture

- nomiClassi

Questa classe consente la creazione ed aggiunta dei tipi di classi sulla base dei possibili dati presenti. Si compone di 4 variabili private di tipo QString coerentemente con i tipi presenti e 4 metodi di get appositi. Il suo scopo è garantire l'estendibilità del codice sulla base delle strutture QHash presenti, garantendo la gestione ed acquisizione dei dati di input sia in seguito ad interazione dell'utente con la View/Model sia con l'importazione fornita da parte del Model.

- *Mapper*

La classe deriva direttamente da *QAbstractTableModel* e si occupa, con una serie di metodi ridefiniti, di settare la View utilizzata dall'utente per le modifiche, facendo calcoli specifici a seconda del tipo di grafico. Utilizza in particolare una struttura *QHash* per poter inserire l'area mappata ed editata secondo lo standard previsto dalla superclasse e una struttura di tipo *QVector*, indispensabile per calcolare il mapping della view a seconda del tipo di grafico selezionato. A tale scopo include quindi le librerie *QtCore* e *QtGui*, assieme ai file header dei tipi di grafico presenti.

Si dettaglia la presenza di un apposito costruttore con un *Chart** per poter costruire correttamente la view visualizzata, la gestione delle operazioni di inserimento/rimozione di riga/colonna personalizzata.
Seguono inoltre appositi metodi di impostazioni delle etichette nella TableView (*headerData)*, setting del colore della TableView (*data*), setting della parte editabile e impostazione finale del grafico (metodi *flags* e *setData),* numero di righe e colonne presenti (*rowCount/columnCount)*. Si segnala che nel caso del grafico a

linee sono settate delle apposite colonne doppie non editate rappresentanti le coordinate X e Y del tipo di grafico; ciò è stato settato in maniera apposita nei metodi descritti.
Dettaglio finale da descrivere è la presenza di un metodo *getOperationType* utilizzato all'interno della classe TableWidget e da tramite la stessa view e la finestra principale per la rimozione/inserimento di righe e colonne.

- *TableWidget*

Essa deriva dalle classi *QMainWindow* e *nomiClassi.* La classe corrisponde alla parte View dell'applicazione, fornendo in particolare la ChartView e la TableView creata dal Mapper all'utente per l'interazione finale (con appositi metodi di get, quindi *getTableWidget* e *getChartDisplayedWidget).* Essa è dotata di opportuno costruttore sfruttando la classe Chart, utile poi per creare il tipo di grafico visualizzato a schermo. A tale scopo include quindi *mapper.h* e *chart.h.*
Si dettagliano per questa classe le seguenti funzionalità principali:

- ***getDataFromFile***, che esegue una lettura del file di tipo JSON/XML selezionato dall'utente. Nel caso del file di tipo JSON, l'acquisizione avviene mediante una variabile di supporto che salva la struttura sotto forma di QByteArray, successivamente rielaborata a seconda di quanti tipi di grafico sono presenti nel file ed eseguendp per ognuno una scansione in base ai valori/etichette ed opzioni del grafico. Similmente nel caso XML, la lettura avviene secondo la struttura DOM con un insieme di classi che consentono di scansionare un file sequenzialmente, sempre salvando dati ed opzioni. Tutto ciò che viene letto viene salvato da parte della classe Chart, pulendo i dati letti in precedenza, per poter permettere la successsiva visualizzazione una volta che l'utente seleziona nuovamente un certo tipo di grafico. Da segnalare la presente di una QHash che permette di selezionare la classe presente nella superclasse *nomiClassi* e successivamente di scansionare il relativo tipo di grafico;

- ***createChartDisplayed***, sfruttando il model Mapper è in grado di creare una TableView interattiva e di mappare i dati a seconda del tipo di grafico. Nel caso del grafico a linee, essendo più serie presenti, il mapper agisce mappando più serie di volta in volta, permettendo di ciascuna la modifica. Ogni grafico sfrutta un mapper personalizzato di tipo verticale (*QVXYModelMapper, QVCandlestickModelMapper, QVBarModelMapper),* escludendo il grafico a torta che sfrutta un mapper di tipo orizzontale (*QHPieModelMapper*), in quanto consente la modifica in colonna del tipo di grafico a torta. Questo, similmente al grafico CandleStick, offrono una funzionalità di modifica ed interazione solo orizzontale in quanto non vengono aggiunte ulteriori serie nel grafico. Nel caso del grafico a linee e a barre l'aggiunta della serie avviene per riga, tranne che per il grafico a torta dove l'aggiunta dei valori avviene per colonna;

- ***setSeriesMapper***, sfrutta un mapping a colori apposito dei tipi di grafico presenti, colorando la TableView coerentemente al tipo di grafico visualizzato a schermo. Nel caso CandleStick, essendo il set della serie suddivisa in 4 punti ciascuno modificabile con un proprio colore, non è presente la visualizzazione a colori nella TableView. Questo metodo, in base al tipo di operazione scelta di modifica della TableView, esegue la modifica profonda del grafico selezionato, aggiungendo/togliendo valori alla serie/etichette e ricaricando a schermo il tipo di grafico presente. Si segnala nel caso del grafico a Torta il mapping che sovrappone la fetta già presente, basta dare un valore nella TableView presente per poter visualizzare correttamente la nuova fetta inserita, mentre per il grafico a linee l'aggiunta di un valore in riga, coerentemente al proprio tipo di Mapper, per poter visualizzare la serie inserita nelle opzioni di modifica della TableView.

- *MainWindow*

Essa deriva direttamente dalle classi *QMainWindow* e *nomiClassi*. Sfrutta un layout a griglia *QGridLayout* composto da una barra di menù, uno widget che compare in caso di eliminazione di tutti i dati, due widget di visualizzazione grafico e modifica dei dati con il model, due widget per la gestione di opzioni estetiche e modifica della TableView visualizzata. Essa comprende il Model e la View, includendo quindi *mapper.h* e *tablewidget.h*. Si descrive l'adattamento e l'uso della logica **segnali/slot** a seconda della specifica interazione dell'utente con l'applicazione, considerate tutte le casistiche ed implementazioni possibili, collegando i singoli widget ad un certo layout e successivamente aggiunti al layout a griglia con una certa funzionalità.

In particolare si compone di una serie di funzionalità così descritte:
- ***createEditMenu,*** menù verticale in basso a sx di aggiunta/rimozione righe/colonne;
- ***createChartMenu,*** menù orizzontale di selezione del tipo di grafico;
- ***createMenuBar,*** con le opzioni di importazione/esportazione grafico, qualora presente;
- ***createWidgets,*** creando il grafico visualizzato e il model di interazione;
- ***get/updateChartSettings,*** riempiendo i grafici mostrati sulla base delle strutture previste e discusse qualora venga importato un grafico (get) e aggiornandole dinamicamente in base all'interazione utente (update);
- ***handleEmptyTable,*** riempiendo in base al tipo grafico con valori/assi e ricreazione di widget, menù di modifica e salvataggio;
- ***handleEmptyAxis***, gestendo nel widget apposito assieme allo slot descritto prima di questo l'aggiunta di valori alla serie ed al grafico vuoto;
- ***closeEvent,*** con gestione personalizzata dell'evento chiusura e salvataggio dimensioni finestra;
- *transitioning,* pulisce il layout e pulisce il grafico nel passaggio tra un bottone e l'altro dei grafici;

In merito agli slot si dettagliano:
- sulla base del menù posto in apposito widget, la selezione di un apposito grafico con i 4 slot;
- presenti in base al tipo di grafico; nel caso della torta, compare la finestra di selezione se si sceglie grafico a ciambella oppure a torta;
- aggiunta/rimozione righe e colonne con slot appositi e reload del grafico visualizzabile;
- modifica del tema, aggiunta antialiasing/animazioni, visualizzazione della legenda del tipo di grafico presente;
- cambio nome grafico, colore serie (fornita nel caso del tipo di grafico CandleStick), appositamente indicate come opzioni secondarie di modifica, qualora il grafico sia esistente;
- slot personalizzati per importazione/esportazione XML/JSON mantenuti appositamente separati;

Si specifica inoltre che la presenza di variabili nel file *mainwindow.h* servono poiché utilizzate nella gestione degli slot e appositamente deallocate, peraltro come tutti gli widget già descritti, una volta cancellato tutto il layout con gestione standard della distruzione degli elementi.
Da dettagliare il fatto che la MainWindow svolga di per sé una logica da controller, gestendo tutto qualora la TableView visualizzata non abbia più valori e ricreando dinamicamente gli widget a seconda della specifica interazione dell'utente con quanto visualizzato a schermo.
Ogni elemento grafico dello widget, parlando in particolare di caselle testuali (*QTextEdit)*, combobox (*QComboBox)* e bottoni (*QPushButton*) è dotato di icona, slot e/o gestione apposita della propria funzionalità coerentemente al grafico mostrato e alla situazione presente a schermo (grafico vuoto, serie vuota, serie presente, ecc.). La creazione stessa dei grafici è mantenuta in slot separati per mantenere una separazione logico/grafica tra le singole classi della gerarchia descritta.

Manuale utente GUI

L'applicazione comprende una schermata principale con un menù verticale di selezione grafico, il menù vuoto di selezione riga/colonna e il menù riempito di modifica estetica del grafico e la barra dei menù per l'importazione/esportazione. L'utente seleziona un certo tipo di grafico; quando ciò accade vi è un menù apposito chiamato Prima impostazione che permette l'inserimento di un valore, settato per l'etichetta e l'eventuale impostazione personalizzata dell'asse presente nel grafico.
Fatto questo, è possibile inserire/togliere righe/colonne coerentemente alla presenza dei valori nel grafico presente, modificare esteticamente la View presentata e scegliere di importare/esportare il grafico ritornando alla schermata principale. È eventualmente possibile muoversi tra un grafico e l'altro, mantenendo il progresso attuale sul tipo di grafico presente.
Come detto, l'aggiunta della serie soltamente avviene per riga, ad eccezione del grafico a torta, il cui mapper comprende una gestione apposita della serie che la aggiunge per colonna ed il grafico finanziario, in cui è presente la singola serie coerentemente con i tipi di dati descritti e visualizzati.

Uso del polimorfismo

In merito all'estendibilità dell'applicazione e l'applicazione di metodi virtuali si descrivono nel dettaglio:

- la presenza dei metodi virtuali puri ***insert***, ***setAxis***, ***finalizza*** utilizzati e concretizzati come prima specificato dalle singole sottoclassi, ciascuna con le sue esigenze di implementazioni di inserimento valori/impostazione serie/assi. Gli stessi parametri di impostazione serie/assi sono disponibili tramite apposite strutture rese statiche e raccolte in strutture contenitive ed estensibili di tipo QHash per raccogliere impostazioni e tipi di grafico
- la presenza del metodo virtuale puro ***reloadGrafico***, che considera i cambiamenti avvenuti nella view e rimodifica ricreandosi autonomamente assi e serie a seconda del tipo di grafico selezionato rimuovendo i precedenti e facendo opportune aggiunte/modifiche/correzioni
- la presenza segnalata dei metodi di clonazione/uguaglianza/distruzione polimorfa
- in tutte le classi View e la stessa MainWindow della distruzione polimorfa virtuale degli oggetti presenti

Formati di file I/O utilizzati: XML/JSON

L'applicazione comprende, come introdotto, due tipi di file che possono essere importati e/o esportati da parte dell'app stessa, specificando l'esportazione qualora vi sia un grafico presente e l'importazione sempre possibile, ricaricando i dati corretti una volta selezionato un certo tipo di grafico.
Per entrambi i file è stata creata un'apposita struttura di esportazione ed importazione, in particolare avendo due tag di apertura *data* ed *options*, coerentemente importati dai grafici e poi visualizzati a schermo tramite la View. Distinguiamo quindi:

1) nel caso del formato **JSON**, nell'ordine il tipo identificativo sulla base dei tipi presenti, il tag *data* contenente i campi *label* e *values*, il titolo del grafico, *options* (opzioni) e l'asse (nel caso dell'asse X, con una serie di valori separati e impostato tramite il metodo apposito *setAxis*). In questo formato sono stati utilizzati i tipi *QJsonObject* e *QJsonArray* nella fase di esportazione e di importazione.

2) nel caso del formato **XML**, similmente, si prevede un tag contenitore di tutti i grafici, definito come tag *grafici*, un tag grafico comprensivo di titolo e tipo sulla base dei tipi presenti, un tag *data* contenente campo *labels* per le etichette e campo *values* con una serie di tag figli identificativi dei valori per una certa etichetta. Oltre a questo, presente il tag *options*, con le opzioni di modifica estetiche e l'asse X, implementata come tag apposito e similmente a prima, con una serie di tag figli comprendenti le etichette impostate dall'applicazione nell'apposito metodo di setting *setAxis*. In questo formato è stato utilizzato il tipo *QXmlStreamWriter* nella fase di esportazione, i tipi *QDomNodeMap, QDomNodeList, QDomElement, QDomAttr* nella fase di importazione per accedere a cascata ai singoli attributi e tag figli secondo la struttura appena descritta,

Si specifica che, a titolo di esempio, nei file XML e JSON consegnati come esempio nell'importazione sono presenti i singoli tipi di grafico per poter fare in modo di far visualizzare un esempio di importazione qualora l'utente clicchi un qualsiasi tipo di grafico.

Specifiche ambiente di lavoro/compilazione

L'applicazione è stata personalmente creata su un sistema operativo Windows 11, versione di Qt 5.9.9, compilatore MinGW 5.3.0 32 bit (usato anche Clang x86 64 bit), totalmente funzionante con queste specifiche.
È stata testata sulla macchina virtuale Linux tramite ISO fornita.
Nella macchina Linux, installata versione Qt 5.9.5 e si rendeva necessario, in una fase di compilazione, usare il comando "sudo apt install gcc-multilib g++-multilib, selezionare GCC x86 64 bit in usr/bin e successivamente adoperare il file .pro per far compilare il progetto correttamente.
Per la compilazione, si utilizzi appunto il file Qt_Charts_CR.pro presente nel progetto, in quanto contiene sia l'aggiunta di libreria XML che il plugin Qt Charts da noi usato per il progetto, più le singole parti di inclusione di widget e grafica. In esso sono presenti anche le indicazioni d'uso sulle icone e lo splash screen.
Si consegna tutta la cartella contenente tutti i file .h e .cpp assieme alla cartella "icons", utilizzata nella fase di estrapolazione grafica dal file .qrc di risorse utilizzato.

Si consegnano anche esempi di file importabili dalla applicazione (JSON/XML) ed esempi di file esportati dalla applicazione con titoli appositi e di riconoscimento.

Warning consapevoli

- *QxcbConnection: XCB error: 3 (BadWindow), sequence: 755, resource id: 37547253, major code: 40 (TranslateCoords), minor code: 0*

Ho provato a risolvere in vari modi settando kit e variabili d'ambiente senza purtroppo giungere ad alcuna soluzione; pare che abbia a che vedere con la risoluzione schermo ed è un bug presente solo sulla VM Ubuntu.

- *GtkDialog mapped without transient parent*

Questo è dato dalla finestra di tipo FileDialog in Linux, altro warning che ha richiesto una piccola modifica del codice e con la soluzione presente, cioè settare un *QFileDialog::DontUseNativeDialog* ai singoli bottoni risolve il warning ma entra in conflitto con la funzionalità di esportazione del file, in quanto vede il suffisso come nullo e non esporta come dovrebbe; non settando il flag, l'esportazione funziona correttamente nei due formati come richiesto. Come prima, il warning è presente solo nella VM Ubuntu.

Suddivisione del lavoro progettuale

Il lavoro progettuale è stato equamente suddiviso tra me e il mio collega Andrea, in particolare io mi sono principalmente occupato della parte grafica, la resa della MainWindow e gestione di slot/segnali, varie modifiche e implementazioni alle classi Model/View e della gestione nella fase di esportazione/importazione del formato XML. Ho curato anche la gestione della logica delle eccezioni e la creazione nella fase di creazione grafici principalmente della classe Pie con gestione dell'opzione ciambella/donut, delineando anche l'impostazione degli widget e la creazione della stessa MainWindow nel file "main.cpp" dell'applicazione.
Entrambi abbiamo poi corretto e modificato assenze/implementazioni mancanti all'altro, ciascuno curando singolarmente l'applicazione e scovando/segnalando possibili problematiche, lavorando a stretto contatto su ogni singolo caso.

Tempistiche di sviluppo e commenti

In merito allo sviluppo effettivo dell'applicazione, sono da entrambi state superate ampiamente le 50 ore monte previste per lo sviluppo del progetto, in particolare nella creazione delle View, del Model e della MainWindow stessa con le gestione di tutti gli slot, nonché nella creazione dei vari singoli grafici, per correzione di errori, aggiunte estetiche e ripetute prove e implementazioni carenti alla stessa documentazione di Qt, molto povera sotto tanti aspetti e ben poco descrittiva in merito all'uso specifico della creazione delle serie, dei grafici e dei mapper in particolare, specialmente il mapper del grafico a torta, la cui serie essendo gestita in modo diverso dalle altre, ha una creazione apposita e richiede una cancellazione specifica per poter far funzionare tutto il resto.
Nello sviluppo del progetto si è appreso incrementalmente entrambi da autodidatta realizzate l'uso di Qt e delle sue parti in crescendo di complessità, codificando in una prima fase la parte logica, poi implementando la View ed il Model, infine gestione completa della stessa MainWindow con tutta la gestione degli slot e dell'implementazione dei tipi di file I/O.
La stessa implementazione dei formati di file, XML in particolare, è particolarmente carente di esempi e ha costretto, assieme ai mapper stessi, una fase intensa di debugging e testing per capire motivazioni di errore e situazioni da gestire, essendo la documentazione assente di casi specifici o esempi effettivi di implementazione delle cose singole, al di là di fornire l'esistenza di metodi ma non dando esempi utili, anche sul semplice uso di widget e layout.
Sarebbe difficile quantificare un tempo effettivo per i motivi segnalati; entrambi abbiamo parallelamente lavorato dall'annuncio del progetto fino ad inizio febbraio fondamentalmente tutti i giorni almeno 2/3 ore, anche 4/5 ore nelle fasi di gennaio e febbraio tra fase di testing, debugging e gestione delle singole parti come motivato.