Progetto di Programmazione ad Oggetti A.A. 2017/2018

Michele Clerici Matricola: 1122656

Favaro Marco Matricola: 1123187

Relazione di Michele Clerici

Progetto: Kalk

Indice:

1 Abstract

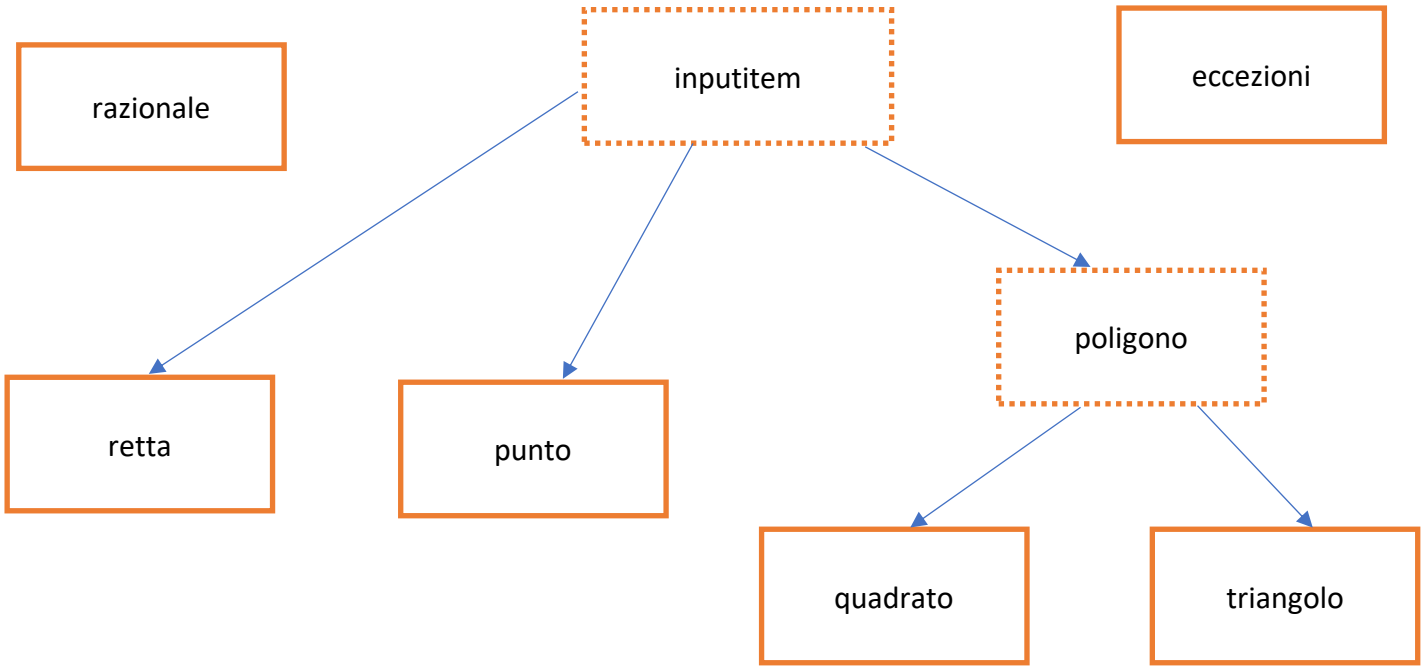
La calcolatrice effettua operazioni su figure geometriche, precisamente su punti, rette e poligoni fino a quattro lati ed esegue diverse operazioni su di esse ad esempio calcola la distanza e l'intersezione fra tutti i tipi o calcola l'area e il perimetro sui poligoni.
La gerarchia (sia per il modello sia per la GUI) è stata pensata e costruita in modo tale che chi volesse, in seguito, potesse ampliarla aggiungendo il proprio tipo a patto che implementi operazioni adeguate se non dovessero bastare quelle già presenti.

Note: Dopo un'attenta rianalisi sono state riviste e modificate diverse logiche appartenenti alla GUI. Si è cercato di seguire il più possibile il principio di singola responsabilità ovvero che ogni parte deve adempiere ad un solo compito e quello che offre deve essere strettamente inerente al proprio ambito. I cambiamenti sono segnalati in *corsivo*.

2 Descrizione/uso gerarchia e codice polimorfo

MODEL

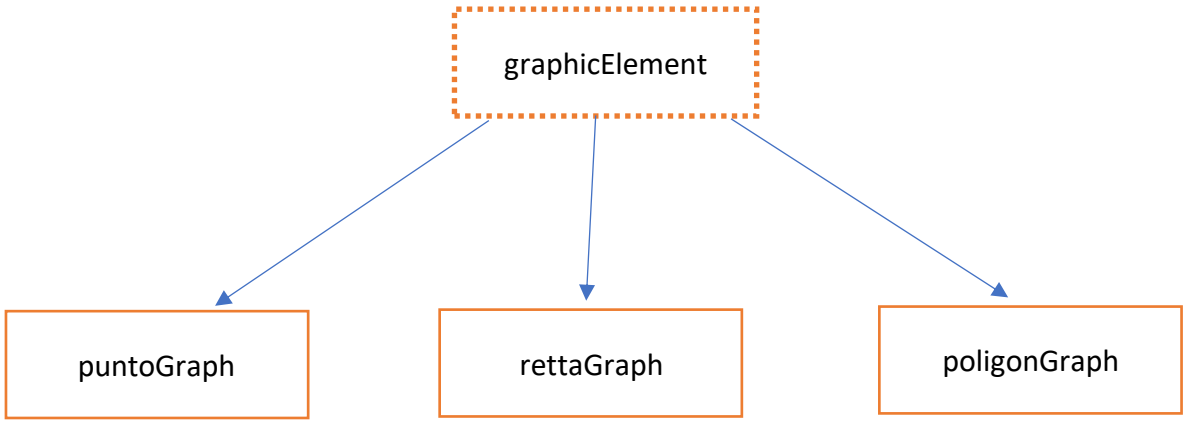
La gerarchia per il modello è la seguente (verrà trattata superficialmente perché è compito del mio compagno):



L'utente inserisce una stringa nel box per l'input e, se questo sarà corretto, non verrà sollevata nessuna eccezione ed il parser ritornerà un elemento di tipo inputItem. A questo punto verranno fatti i dovuti controlli a Runtime per capire di che elemento grafico stiamo parlando (punto, retta, triangolo, ecc.) e verrà ritornato. Per questo motivo ogni elemento della gerarchia è figlio di InputItem ed è sempre per questo motivo che inputItem è una classa virtuale astratta.
Ad un primo impatto potrebbe sembrare che ci siano vari usi del typeid in modi che limitano l'estensibilità (ad esempio retta::distance o punto::distance). Ma la nostra scelta è stata dettata dal fatto di dare a terzi la possibilità di implementare metodi come poligono::polipoli(…), poligono::polipunto(…), poligono::rettapol(…) poligono::distpuntopol(…) e così via e non i metodi inputitem::intersect(…) e inputitem::distance(…) poiché questi metodi richiamano le funzioni citate sopra.

VIEW

Dopo un'attenta rianalisi ho deciso di ricreare da zero la gestione degli elementi grafici. Innanzitutto, verrò presentata la nuova gerarchia implementata. Designata in questo modo per poter permettere il massimo dell'estensibilità a terzi che vorranno implementare la classe grafica.



graphicElement

Questa è la classe principale per la creazione di ogni tipo di elemento rappresentabile graficamente all'interno della calcolatrice. È una classe astratta, contiene due metodi virtuali puri che meritano una descrizione approfondita. Il primo è il parser grafico, ovvero dato un puntatore generico inputitem restituito, ad esempio, dal parser del modello ritorna un supertipo graphicElement*. Ottenuto quest'ultimo ad esempio è possibile invocare il metodo astratto draw(grafico*, int) dove passando solamente il grafico su cui disegnare e lo slot di appartenenza (ovvero gli Slot Input descritti graficamente più in basso nello schema generale della gui) disegna a Runtime l'elemento corretto. Com'è intuibile draw(…) è virtuale e protected (per ragioni di estensibilità) e per questo motivo è stata definita una funzione pubblica drawing(…) con i medesimi parametri di draw per permettere il disegno. Potrebbe essere presente un warning dovuto al parser di tipo [-Wreturn-type] "control may reach end of non-void function". Questo succede perché ci sono solo condizioni e mai un else finale ma non c'è problema perché il parser in questo punto troverà sicuramente un tipo dinamico esatto per creare la figura.*

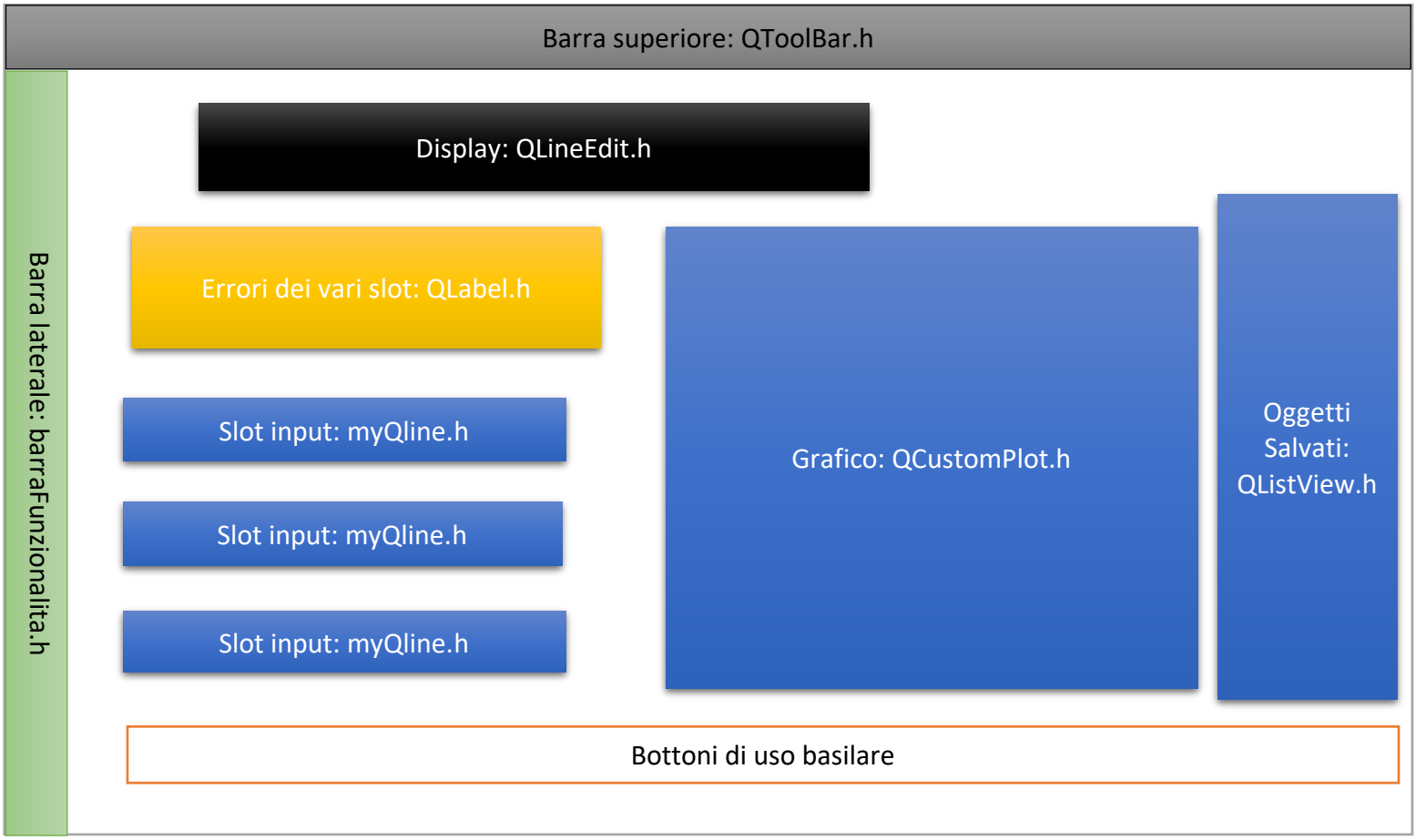
puntoGraph/rettaGraph

Di queste due classi è importante descrivere il costruttore. Rispettivamente per la prima c'è bisogno solamente di un punto mentre per la seconda è necessario passare un range (min, max) per costruire la retta poiché è impossibile rappresentare una retta di range infinito.

poligonGraph

poligonGraph è la classe che si occupa del quadrato e del triangolo. Potenzialmente il metodo draw overraidato in questa classe (ma tutta la classe in generale) è già prontò per gestire ogni tipo di poligono a patto che sia regolare. In ogni caso se questo non dovesse bastare bisogna estendere questa classe.
Per creare un oggetto di questo tipo è necessario passare solo il corrispettivo poligono che appartiene al modello (ritornato, ad esempio, dal parser…per questo motivo se si decide di estendere la gerarchia grafica con buona probabilità sarà necessario estendere anche il modello).

Viene ora presentato lo schema della GUI con lo scopo di dare un'idea generica degli elementi che sono inglobati nelle principali classi di visualizzazione ovvero finestra.h e mainGui.h.
Ogni entità ha [mini descrizione: nome classe], non verranno mostrati i pulsanti di banale comprensione, le altre schermate sono immediate.
Dopo questa panoramica verranno descritte solo le classi interessanti dal punto di vista ingegneristico.
Cominciamo con barraFunzionalità.h, barra (verde nel grafico sottostante, appartiene a finestra.h) che offre le funzionalità di calcolo disponibili, verrà descritta più dettagliatamente in seguito.
Gli elementi di finestra.h sono caratterizzati dal colore grigio (nota: mainGui è inglobata in finestra quindi tutti gli elementi dovrebbero avere anche il colore grigio ma per motivi di comprensione viene dato solamente il colore di appartenenza stretta), tutti gli altri elementi colorati tranne il verde ed il grigio appartengono a mainGui.h.



finestra.h

finestra.h agisce da "pseudo-controller" infatti permette di collegare il modello con la view. Collega le preferenze (impostazioni.h), il wizard di benvenuto (wizard.h) *ed è responsabile della creazione di barraFunzionalita.h.* Esempi utili per comprendere la scelta di questa struttura:

* Check all'uscita delle finestre aperte, se si chiude la finestra principale verranno chiuse a cascata anche le altre con un completo controllo del garbage (motivo per cui impostazioni.h e wizard.h sono state incapsulate in uno smart pointer).

** Permette la separazione ed il controllo dei componenti che formano la GUI (vedi appunto barraFunzionalita.h che in una revisione passata di Kalk apparteneva a mainGui.h).*

* Carica le impostazioni personali dell'utente all'avvio.

maingGui.h

Questa classe viene ereditata da QWidget principalmente per poter estendere gli slot. Anche qui si fa uso di QSettings.h per poter disegnare con le impostazioni settate precedentemente dall'utente (*impostazioni che sono state riviste e migliorate, ora sono più compatte, è stato rimosso tutto il codice ridondante, per esempio per evitare questo problema su loadColor(…) si è fatto uso di una QMap*). Rappresenta il fulcro dell'interfaccia dove avviene la maggior parte dell'interazione con l'utente ed è anche il punto della GUI dove viene fatto un massiccio uso del polimorfismo. Ad esempio nello slot drawAndReturn() il Parser del modello, dopo aver analizzato l'input, se questo è corretto, ritorna un puntatore polimorfo di tipo inputitem*. *Successivamente viene passato al parser grafico che procederà alla creazione dell'elemento grafico corrispondente ed infine, con il meccanismo precedentemente descritto verrà disegnato*. *Mentre il puntatore all'elemento grafico appena disegnato viene eliminato (non è più necessario) il puntatore ritornato dal parser del modello viene mantenuto alla fine di questo slot, anzi viene inserito in un vettore di puntatori ad inputitem* per poter essere analizzato nuovamente in futuro anche dagli slot di barraFunzionalita.h*, *verrà eliminato solamente dal distruttore all'uscita o quando necessario da remove_qle(). Riassumendo mainGui è stata rivista di molto, infatti è stata sollevata da diversi incarichi, (presi in carico da barraFunzionalita.h) sostanzialmente ora si occupa solamente di interpretare l'input dell'utente, di portare queste informazioni al resto dell'interfaccia ed infine di dialogare con il grafico per il plotting.*

barraFunzionalita.h

Questa è la barra che si trova a sinistra dell'interfaccia. È stato deciso di creare una classe a parte per separare ancor più le logiche per la visualizzazione. È presente una friendship di questa classe in mainGui per permettere l'accesso all'input dell'utente e per permettere di stampare il risultato nel display. Con questa scelta è stata aumentata anche l'estensibilità di tutta la GUI. Se un programmatore vuole aggiungere una funzionalità, dopo aver esteso il modello e opzionalmente anche il modello grafico, non deve fare altro che aggiungere la propria QAction ed un nuovo slot in questa classe.

grafico.h

La classe grafico.h è ereditata da QCustomPlot. **Classe che non fa parte del framework** di Qt per noi fondamentale, poiché, usando solo i metodi strettamente necessari ci permette di disegnare gli elementi da noi parsati.
Abbiamo optato per QCustomPlot per due ragioni: è un "semplice" file sorgente che garantisce assoluta compatibilità ma soprattutto perché avremmo sforato di troppo il tempo a disposizione per creare una struttura di plotting così complessa andando fuori tema. I Warning presenti durante la compilazione sono dovuti solo a questa classe che, per l'appunto, non essendo di nostra proprietà ci sentiamo esclusi nel mettere mano a questa parte di codice. L'aspetto interessante di questa classe è stata la scelta di creare una matrice "smart" (corredata di metodi per gestire il garbage, nota: gli elementi grafici devono essere eliminati con l'apposito metodo di QCustomPlot) che gestisce per ogni slot di input i segmenti (QCPItemLine*) che compongono la figura ottenuta dal parser.

myQLine.h

Degna di nota è la classe myQline.h derivata direttamente da QLineEdit.h della libreria grafica di Qt. Questa scelta è stata dettata dalla implementazione del drag and drop. L'idea era quella di poter trascinare i risultati salvati da una QListView.h direttamente su uno degli slot di input ed il percorso migliore era questo, ovvero ereditare la classe per poter ridefinire dragMoveEvent(…) e dropEvent(…) i due metodi virtuali protetti fondamentali per questo scopo.

impostazioni.h

Di questa classe l'unica cosa interessante da menzionare è stata l'implementazione del salvataggio. È stato creato un metodo di nome saveSettings() che fa uso di QSettings.h. Con tre cicli for scorro i bottoni di ogni slot di input e controllo se sono premuti, mi salvo il loro stato e alla prossima apertura della finestra impostazioni.h verrà caricato il salvataggio (simil discorso viene fatto per il range min. e max. del grafico). Ciò è possibile tramite finestra.h che, come precedentemente detto, agisce da controller. Ha uno slot showOption() che all'azione dell'utente crea la finestra e invoca loadSettings() per caricare il salvataggio (se questo esiste). Ho scelto di creare showOption() per aver un miglior controllo del garbage, questo metodo insieme ad altri controlli usati alla chiusura della finestra delle impostazioni mi assicurano che essa vive solamente quando è necessaria e non per tutta la durata della calcolatrice. È presente un bug nel framework di qt che causa in certi sistemi l'errore "setNativeLocks failed resource temporaly unavailable". Ciò è causato dalla chiamata del metodo sync() all'interno della classe QSettings solamente in contesti particolari ovvero quando il ".conf" file è localizzato in un dispositivo di rete (proprio il caso del laboratorio, mentre sulla macchina virtuale locale data dal professore e nell'ambiente di test e sviluppo non ci sono problemi). Riporto qui un altro utente che ha segnalato il bug all'interno di
QT bug tracker (https://bugreports.qt.io/browse/QTBUG-46762).

wizard.h

Classe che permette di rappresentare una finestra di aiuto per l'utente all'apertura della calcolatrice. Viene ereditata da QWizard al solo scopo di migliorare la lettura del codice.

3 Manuale utente

Non è necessario il manuale. All'apertura di Kalk verrà visualizzato un wizard che aiuterà l'utente nel prendere dimestichezza con l'interfaccia e nell'inserire in maniera corretta l'input desiderato.

4 Analisi delle Tempistiche

Verranno specificate le tempistiche soggette alla mia parte. La soglia di ore disponibili è stata leggermente sforata. Su certi punti verrà giustificato l'ammontare di ore.

● analisi preliminare del problema(~05h);

● progettazione GUI(~20h+*~5h*);
Diversi dubbi su come implementare lo schema model-view.
Problema su come implementare il grafico. Dopo diversi tentativi, come già specificato abbiamo optato per un codice sorgente esterno.
Re-design gerarchia view

● apprendimento libreria Qt(~15h);
Implementazione drag and drop e per implementazione sistema di salvataggio

● codifica GUI(~13h);

● debugging(~05h);
Problemi con la gestione del garbage.

● testing(~03h).

5 Suddivisione del lavoro

L'applicazione è stata progettata e discussa in stretto contatto da entrambi i partecipanti, a partire dall'analisi del problema fino alla sua implementazione. Inizialmente mi sono occupato dei primi abbozzi di gerarchia ma successivamente questa parte è stata completamente realizzata dal mio collega. Io, invece, mi sono dedicato principalmente alla progettazione e alla realizzazione della GUI con tutti i problemi e le decisioni ad essa annessi che vanno, ad esempio, dalla scelta di utilizzare QCustomPlot fino alla gestione del sistema di salvataggio delle preferenze personali. Il mio compagno, invece, si è occupato della realizzazione del modello sia in c++ che in java inclusa la completa gestione delle eccezioni. La fase di test e di debugging è stata svolta assieme, soprattutto la gestione del garbage.
Mi sono occupato inoltre di tutte le decisioni e i cambiamenti fatti dalla prima alla seconda consegna.

6 Ambiente di sviluppo e di test

A casa:

● Sistema operativo: Mac OS 10.13.5

● Compilatore: clang_64bit

● Libreria Qt: 5.5.1

In laboratorio:

● Sistema operativo: Ubuntu 16.04 64-bit

● Compilatore: gcc 5.4.0

● Libreria Qt: 5.5.1

7 Comandi per la compilazione ed esecuzione

Il progetto viene presentato con una cartella col progetto in C++ ed una col progetto in Java:

- **C++** Spostarsi col terminale nella cartella apposita (utilizzare il file .pro incluso) e dare i comandi:
  1. qmake
  2. make
  3. ./Kalk
- **Java** Spostarsi col terminale nella cartella apposita e dare i comandi:
  1. javac use.java
  2. java use