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

Michele Clerici Matricola: 1122656

Favaro Marco Matricola: 1123187

Relazione di Michele Clerici

Progetto: Kalk

Indice:

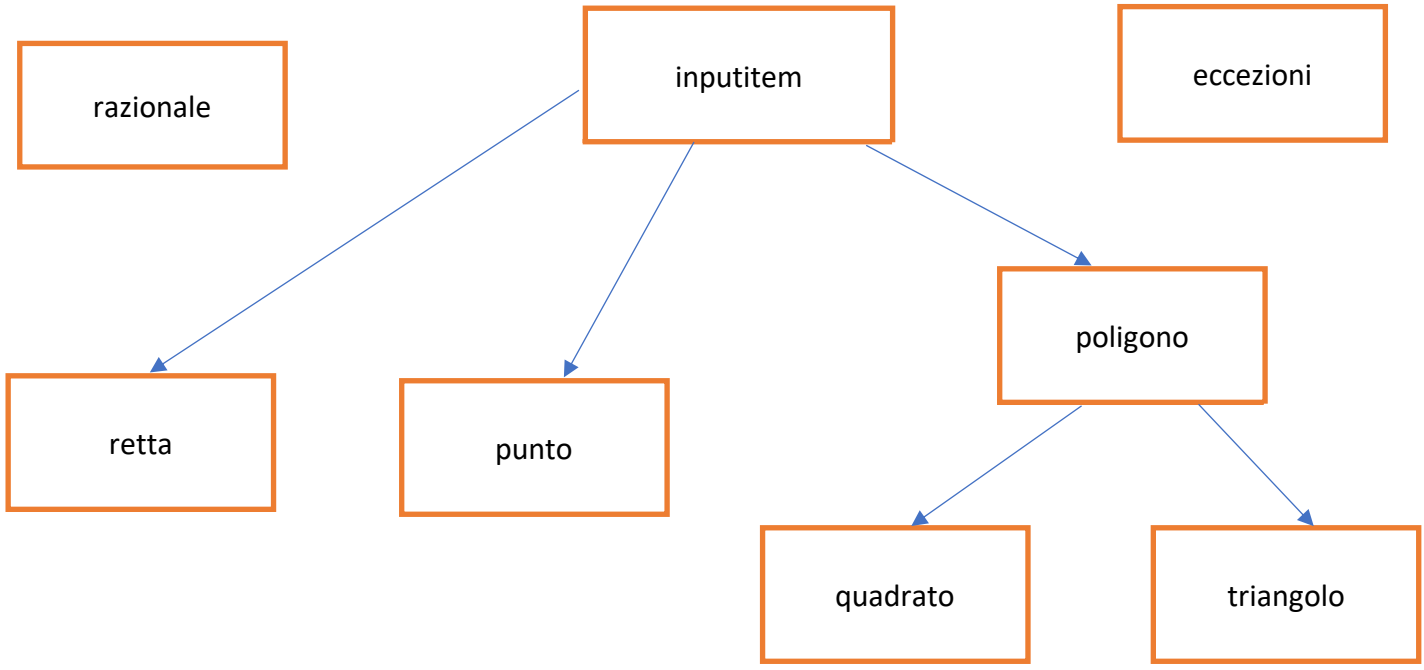
1 Abstract

La calcolatrice effettua operazioni su figure geometriche, precisamente su punti, rette e poligoni fino a quattro lati. Può calcolare la distanza e l'intersezione fra tutti i tipi, calcolare l'area e il perimetro dei poligoni, la retta passante per due punti, e, dati un punto P e una retta R può calcolare la retta passante per P e parallela o perpendicolare a R.
La gerarchia è stata pensata e costruita in modo tale che chi volesse, in seguito, può ampliarla aggiungendo il proprio tipo nel modello, a patto che implementi operazioni adeguate per il proprio tipo se non dovessero bastare quelle già presenti. Inoltre, con pochissime linee di codice aggiuntive è possibile aggiungere nella barra laterale di sinistra nuove azioni sul nuovo tipo di dato.

2 Descrizione/uso gerarchia e codice polimorfo

MODEL

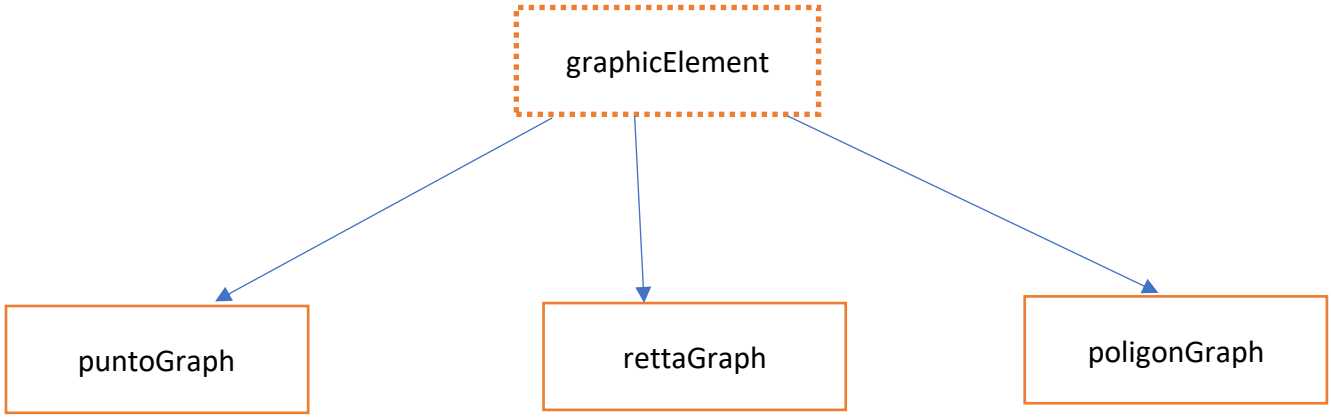
La gerarchia per il modello è la seguente (verrà trattata superficialmente perché è compito del mio compagno):



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.

VIEW

Dopo un'attenta rianalisi ho deciso di ricreare da zero la gestione degli elementi grafici. Innanzitutto, verrò presentata la nuova gerarchia implementata. Disegnata 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.

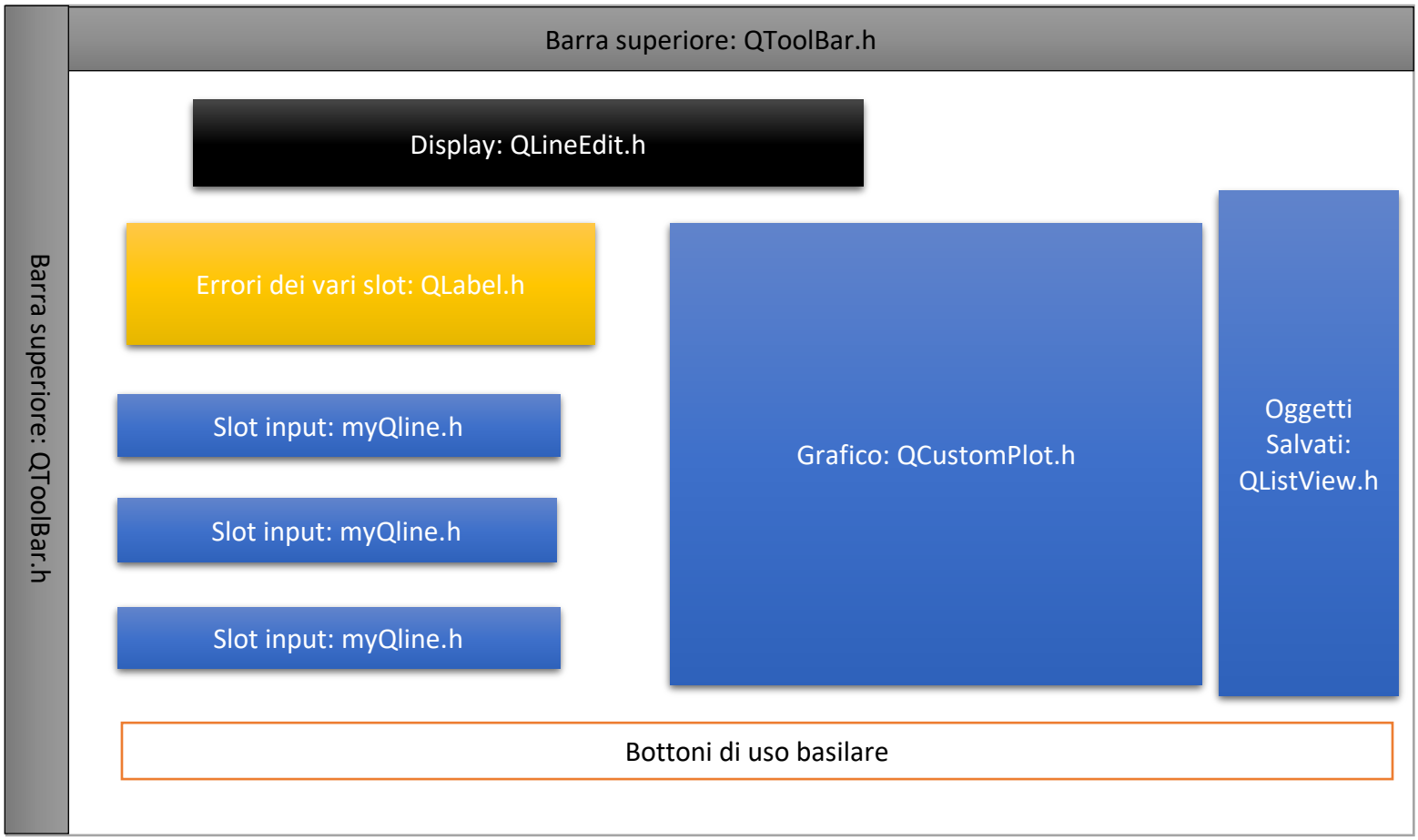
puntoGraph/rettaGraph

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 schematicamente lo schema della GUI.
Si prende in considerazione solo “finestra.h” e “mainGui.h” (ogni entità ha [mini descrizione: nome classe], inoltre non verranno mostrati i pulsanti di immediata comprensione), le altre schermate sono immediate.
Dopo questa panoramica verranno descritte solo le classi interessanti dal punto di vista del polimorfismo.
Gli elementi di “finestra.h” sono caratterizzati dal colore grigio, tutti gli altri elementi appartengono a “mainGui.h”, quest’ultima è contenuta in “finestra.h”.



finestra.h

finestra.h agisce da “pseudo-controller”. Permette di collegare il modello con la view ed inoltre collega le preferenze (impostazioni.h) ed il wizard di benvenuto (wizard.h). Un esempio utile per comprendere la scelta di questa struttura è il 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).

maingGui.h

Questa classe viene ereditata da QWidget principalmente per poter estendere gli slot. Anche qui si fa uso di QSettings.h per caricare eventuali impostazioni personali (*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 della gui 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 ad “inputitem”. *Successivamente viene passato al parser grafico che procederà alla creazione dell’elemento grafico corrispondente ed infine, con il meccanismo precedentemente descritto verrà disegnato*. *Il puntatore ritornato dal parser del modello non viene eliminato alla fine di questo slot, anzi viene inserito in un vettore di puntatori ad inputitem* per poter essere analizzato nuovamente in futuro* anche dai metodi della barra laterale di sinistra, *verrà eliminato solamente dal distruttore all’uscita.* Infine è presente del polimorfismo anche su remove_qle() per controllare l’elemento grafico da rimuovere.

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 gestisca 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 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);
Ho cercato a fondo soluzioni per implementare il drag and drop e per implementare il 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.

6 Ambiente di sviluppo e di test

A casa:

● Sistema operativo: Mac OS 10.13.4

● 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