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

**----- Matricola: -------**

**----- Matricola: -------**

**Relazione di -------**

**Progetto: Kalk**

**Indice:**

1. **Abstract**
2. **Descrizione/uso gerarchia e codice polimorfo**
3. **Manuale utente**
4. **Analisi delle tempistiche**
5. **Suddivisione del lavoro**

**1 Abstract**

La calcolatrice Kalk effettua operazioni sulle figure, precisamente su punti, rette e poligoni fino a quattro lati.

Con questi tipi può effettuare tali operazioni:

intersezione

distanza tra due punti

distanza generica tra due elementi

il perimetro

l’area

data un punto e una retta vedere se una retta è perpendicolare.

Tutte le figure sono costituite da punti dove a loro volta sono composti da coordinate definite tramite razionali per assicurare maggiore precisione.

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.

Triangolo

Razionale

Quadrato

Retta

Punto

Poligono

InputItem

L’utente inserisce una stringa nel box per l‘input, 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 disegnato. Infine si potrà selezionare un’azione con il tipo ricevuto sulla barra laterale sinistra.

Per questo motivo ogni elemento della gerarchia è figlio di InputItem ed è sempre per questo motivo che inputItem è una classa virtuale astratta.

Barra superiore: QToolBar.h

**VIEW**

Lo schema per la gui è la seguente.

Viene schematizzata solo “finestra.h” e “mainGui.h” (ogni entità ha [mini descrizione: nome classe]), le altre schermate sono immediate.

Dopo questa panoramica verranno descritte solo le classi interessanti dal punto di vista ingegneristico.

Gli elementi di “finestra” sono caratterizzati dal colore grigio, tutti gli altri elementi appartengono a “mainGui.h”, quest’ultima è contenuta in “finestra.h”.

Barra laterale: QToolBar.h

Errori dei vari slot: QLabel.h

Slot input: myQline.h

Slot input: myQline.h

Slot input: myQline.h

Oggetti Salvati:

QListView.h

Grafico: QCustomPlot.h

Display: QLineEdit.h

Bottoni di uso basilare

**finestra.h**

finestra.h agisce da pseudo-controller. Permette di collegare il model con la view ed inoltre collega le preferenze (impostazioni.h) ed il wizard di benvenuto (wizard.h). Un esempio utile di aver ingegnerizzato in questo modo la struttura è il check all’uscita delle finestre aperte, se si chiude la finestra principale verranno chiuse a cascata anche le altre (motivo per cui impostazioni.h e wizard.h sono state incapsulate in uno smart pointer).

**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 così fuori tema. I Warning presenti durante la compilazione sono dovuti solo a questa classe ed per l’appunto non essendo di nostra proprietà ci sentiamo esclusi dal mettere mano a questa parte di codice.

**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 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 (simile 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.

**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. 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. Ciò avviene nello slot “drawAndReturn()”. Il Parser, dopo aver analizzato l’input, se questo è corretto, ritorna un puntatore polimorfo ad “inputitem”. Successivamente vengono fatti dei controlli a Runtime per poter disegnare l’elemento. Questo puntatore viene analizzato a Runtime anche dai metodi della barra laterale di sinistra. È presente del polimorfismo anche su “remove\_qle()” per controllare l’elemento grafico da rimuovere.

**wizard.h**

Classe che permette di rappresentare una finestra di aiuto per l’utente all’apertura della calcolatrice. Viene ereditata da QWidget 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 ~~modello e~~ GUI(~20h);

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.

● apprendimento libreria Qt(~15h);

Ho cercato a fondo soluzioni per implementare il drag and drop e per implementare il sistema di salvataggio

● codifica ~~modello e~~ 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.

**4 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

**6 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: qmake make ./Kalk
* **Java** Spostarsi col terminale nella cartella apposita e dare i comandi javac…