Riccardo Bernucci Matricola 1121331

KALK

Progetto per il corso di programmazione ad oggetti

ANNO ACCADEMICO 2017/2018

1. INTRODUZIONE

1.1 Scopo del progetto

Lo scopo del progetto è creare una calcolatrice per il calcolo dei polinomi e delle equazioni. Per la sezione dedicata ai polinomi non vi è un limite ai gradi gestiti, mentre per la parte delle equazioni si accettano solo quelle di primo e secondo grado.

1.2 Istruzioni di compilazione ed esecuzione

Il progetto è stato sviluppato in ambiente Windows 10 Pro 64bit, con gcc in versione 4.9.2 e libreria Qt in versione 5.6.2. Compilazione ed esecuzione sono state testate anche su Ubuntu 16.04 64bit tramite la macchina virtuale fornita e i programmi compilano correttamente.

1.2.1 Progetto C++

Il progetto C++ è contenuto nella cartella con il nome "C++". Viene fornito un file .pro (**progetto.pro**), necessario per compilare il progetto correttamente, di conseguenza i comandi per la compilazione dovranno essere: qmake → make.

1.2.2 Progetto Java

IL progetto Java è contenuto nell'omonima cartella. Per compilare ed eseguire correttamente il progetto devono essere adottati i seguenti comandi nell'ordine indicato: javac use.java → java use.

2. VINCOLI OBBLIGATORI

2.1 Separazione tra modello e grafica

Le specifiche recitano che il progetto deve essere separato perfettamente tra parte grafica e parte logica. Il pattern che ho deciso di adottare è di tipo ***model-view***. Le cartelle create sono due ***model*** e ***view***, le quali mostrano la separazione che è stata applicata nel progetto. Nella cartella *model* sono contenute le classi necessarie per effettuare i diversi calcoli previsti per la calcolatrice, fatta eccezione per *modelPolinomio* e *modelEquazioni*, due classi richiamate dall'interfaccia grafica per varie operazioni. La cartella *view* contiene l'aspetto grafico del progetto con qualche accorgimento per permettere il corretto funzionamento ed uso della calcolatrice.

2.2 Struttura modello logico

2.2.1 Classi necessarie

- **razionale:** è la classe di partenza utilizzata dall'intero modello logico. Essa contiene due campi interi che indicano numeratore e denominatore. Sono stati implementati i metodi set, get e un metodo di semplificazione reso privato ed utilizzato all'interno dei costruttori. Inoltre, tra i costruttori ve ne è uno che effettua il parsing della stringa in modo tale da semplificare la creazione dei dati dall'interfaccia grafica. Infine, le operazioni aritmetiche rese disponibili per la seguente classe sono somma, sottrazione, moltiplicazione, divisione e potenza razionale (m/n^x/y), mentre per la parte logica vi sono uguale, diverso e minore.

- **monomio:** è una classe che ha come campi dati due razionali, coefficiente ed esponente. La seguente classe oltre ai metodi set e get, presenta anche essa un costruttore che richiede una stringa in input, la quale deve essere nel formato r1, x^r2 oppure r1x^r2, dove r1 e r2 sono razionali (Attenzione: se inserita la x deve essere inserito anche l'esponente). Infine, le operazioni fornite sono somma, sottrazione, divisione, moltiplicazione.

- **polinomio:** presenta un unico campo dati che è una list di monomi. Il polinomio ha un costruttore vuoto nel quale viene istanziato unicamente il primo monomio a zero. Contiene metodi per la creazione e l'ampliamento del polinomio:
  1. somma, sottrazione, moltiplicazione, divisione tra monomio e polinomio;
  2. somma, sottrazione, moltiplicazione, divisione tra polinomio e monomio;
  3. somma e sottrazione tra polinomio e polinomio.

  Vi sono poi tre metodi necessari per le classi che useranno questo oggetto e sono:
  1. *void ordina()*: utilizzando l'operatore < sovraccaricato prima nella classe razionale e poi in quella di monomio, ho potuto sfruttare il metodo sort() del contenitore list nel modo desiderato ovvero, ordinare i monomi in base all'esponente in ordine crescente;
  2. *void eliminaZeri()*: permette discorrere la lista e cancellare quei monomi che sono andati a zero;
  3. *monomio getMonomio_Pos(int)*: restituisce il monomio alla posizione desiderata.

  Inoltre, presenta un metodo di calcolo della derivata prima e del valore del polinomio se la x assumesse il valore di un opportuno razionale da inserire.

Scelta implementativa

Ho scelto di adoperare una lista poiché tutti i metodi da me ideati devono scorrere la lista per intero e solo il metodo *getMonomio_pos(int)* risulta essere rallentato rispetto all'uso di un contenitore ad accesso casuale, come il vector. Inoltre, vi è un notevole risparmio di tempo nell'eliminazione degli zeri, cosa che avviene molto spesso nelle classi che derivano da equazione (vedi sezione 2.2.3)

2.2.2 Gestione degli errori e classi ausiliarie

Per la gestione degli errori ed eventuali classi ausiliarie ho optato per la creazione di un file .h. Il file in questione è *config.h*, esso presenta al suo interno le classi di gestione degli errori: **divisioneZero**, **erroreRadice**, **esponenteDiverso**, **erroreGrado**, **x_Assente**, **erroreSimbolo**, **errorePotenza**, **nonGestito**.

Una classe di errore è implementata secondo la seguente struttura:

class nomeClasse: public std::exception{};

Le classi ausiliarie adoperate, invece, sono: **simbolo** e **tipoEquazione**. Sono state ideate per permettere di tenere traccia dei simboli che possono essere inseriti nel programma e dei tipi di equazione che possono essere gestite.

Questo tipo di classi è stata implementata con la struttura:

enum class nomeClasse{ valore1 = a, valore2 = b, … };

2.2.3 Gerarchia obbligatoria

Il progetto richiedeva l'ideazione di una gerarchia con almeno tre tipi B, C1 e C2 tali che C1 e C2 fossero sottotipi di B.

- **equazione**: è la classe base astratta della gerarchia. Ha un distruttore virtuale e due campi privati polinomio. Inoltre, vi sono 4 metodi virtuali puri, utili per conoscere il grado, il tipo di equazione, il risultato e la derivata prima. Per ultimi ci sono tre metodi implementati che permettono di stampare, sostituire le x con un razionale per verificarne l'uguaglianza ed unire nella forma c + bx^1

+ ax^2 + … + kx^n tutte le equazioni.

- **primoGrado:** classe derivata da *Equazione*. Essa implementa i metodi virtuali puri e nel suo costruttore viene effettuato un controllo per verificare se è effettivamente di primo grado. Il metodo per calcolare la derivata poiché sarebbe di grado zero, ho optato per restituire un valore nullo.

- **secondoGrado:** classe derivata da *Equazione.* Essa implementa tutti i metodi virtuali puri della classe base e nel suo costruttore viene effettuato un controllo per verificare se è effettivamente di secondo grado. Inoltre, aggiunge una funzionalità il calcolo del delta e il metodo della derivata che questa volta è stato implementato in modo da restituire un'equazione di primo grado.

- **pura:** classe derivata da *secondoGrado.* Essa implementa tutti i metodi virtuali puri della classe base e nel suo costruttore viene effettuato un controllo per verificare se è effettivamente pura di secondo grado. Inoltre, viene sovrascritto il calcolo del delta.

- **spuria:** classe derivata da *secondoGrado.* Essa implementa tutti i metodi virtuali puri della classe base e nel suo costruttore viene effettuato un controllo per verificare se è effettivamente spuria di secondo grado. Inoltre, viene sovrascritto il calcolo del delta.

2.3 Java

Le classi create in c++ sono state ideate anche in java. Tuttavia, a seguito delle differenze tra java e c++, sia come linguaggio sia come specifiche del progetto ci sono state delle modifiche:

- Non sono stati creati tutti gli errori che sono stati ideati per c++ poiché non era necessario creare una interfaccia grafica;
- La classe simbolo non è stata ideata si è usato unicamente un simbolo *char* e dei controlli per verificare che rientrassero tra i seguenti +, -, : e *.
- Nella classe polinomio si è utilizzato un *ArrayList* di oggetti monomio.

Per tutte le classi si è cercato di mantenere lo stesso metodo di scrittura che si è adottato in c++. La classe ***use*** ovvero, la classe che contiene il main del programma, è stata scritta in modo da mostrare il corretto funzionamento di ogni metodo creato e, per mostrare che anche gli errori funzionano, ho inserito dati sbagliati in blocchi *try catch*.

3. MODEL e VIEW

3.1 Model

Sono stati ideati due oggetti **modelPolinomi** e **modelEquazioni** che presentano tutti i dati necessari per la corretta esecuzione dell'interfaccia. Tutti i dati eccetto i booleani sono puntatori che verranno allocati nello heap. Queste due classi contengono tutti i metodi che permettono di collegare l'insieme di classi create e la parte grafica. Infatti, la parte grafica quando sfrutta questi oggetti, richiama i metodi per ottenere dei risultati o per dare in input delle informazioni che dovranno essere controllate segnalando eventuali errori.

3.2 Interfaccia grafica

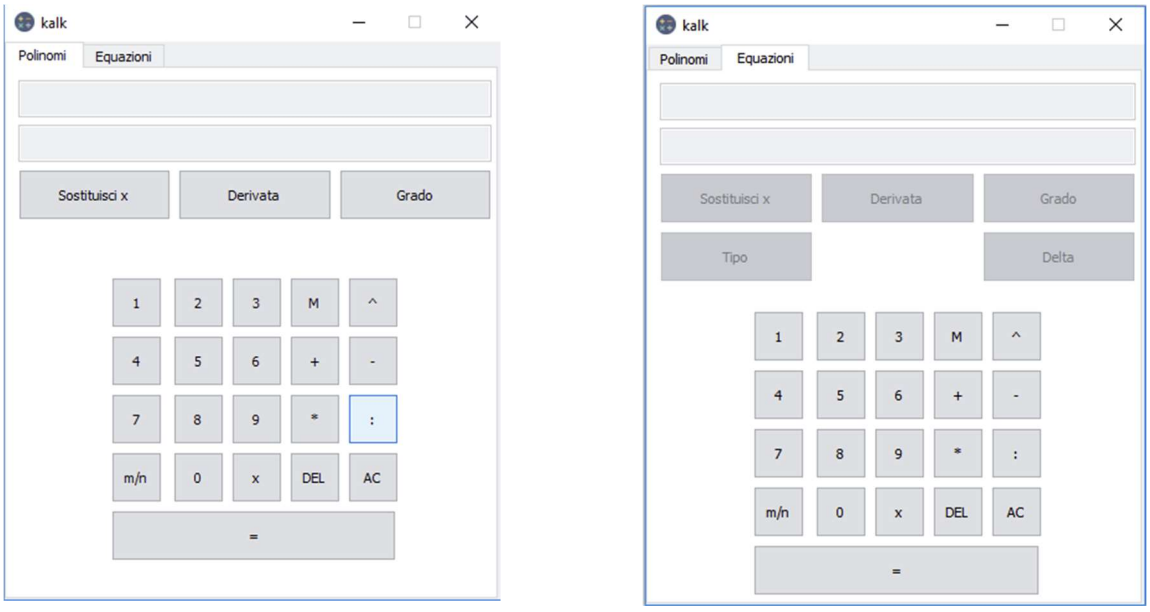
Per quanto riguarda l'aspetto grafico si è deciso di fare affidamento sul framework Qt così come indicato dalle specifiche del progetto. Le classi ideate per questo programma derivano tutte da *QWidget*, tranne che per l'ultima dell'elenco. Sono:

- **tastiera**: è la classe che contiene i 21 pulsanti dei numeri e delle principali operazioni. È stata ideato questo widget per evitare di generare codice identico in due classi distinte.

- **KalkPolinomi**: questa classe ha il preciso compito di generare l'aspetto grafico della sezione dedicata ai polinomi. Utilizza la classe tastiera appunto per generare la parte numerica e in essa sono contenuti tre pulsanti dedicati a: sostituire le x con un razionale, calcolare la derivata e mostrare il grado dell'operazione. Nel file .cpp vi è un continuo contatto con il controller, puntatore a modelPolinomio, per la gestione degli input e degli output. Inoltre, vi sono blocchi di tipo try catch per generare i messaggi di errore.

- **KalkEquazioni**: questa classe genera l'aspetto grafico della sezione dedicata alle equazioni. Utilizza la classe tastiera e sono contenuti 5 pulsanti dedicati a: sostituire le x con un razionale, calcolare la derivata, mostrare il grado dell'operazione, mostrare il tipo di equazione generata e il delta quando questo può essere calcolato. Nel file .cpp vi è un continuo contatto con il controller, puntatore a modelEquazioni, per la gestione degli input e degli output, esattamente come nell'oggetto precedente. Infine, vi sono blocchi di tipo try catch per generare i messaggi di errore.

- **Kalk**: questa classe estende QMainWindow e richiama tramite puntatori kalkPolinomi e KalkEquazioni, inoltre organizza la pagina principale in sezioni adoperando un QTabWidget.



(Immagini delle due sezioni dell'applicazione)

4. ORIENTAMENTO AGLI OGGETTI

4.1 Incapsulamento

In tutte le classi, i campi dati sono stati inclusi nella parte privata. Nella classe razionale e modelEquazioni, oltre ai campi dati, troviamo metodi utili unicamente alla classe stessa.

Per quanto riguarda razionale i metodi sono:

- *void semplifica()*: metodo che permette di semplificare i numeri razionali;
- *int stringToNumber(const string&)*: metodo che permette di convertire una stringa in un numero intero;
- *string numberToString(int)*: metodo che permette di convertire un numero in stringa.

Nella classe *modelEquazioni*, invece, i metodi che si sono ideati sono necessari a far puntare il puntatore equazione alla corretta classe derivata e in base all'esito ritorna un valore true o false.

3.2 Modularità

Per tutte le classi si è cercato di mantenere la massima separazione tra dichiarazione e definizione dei metodi. Le classi sono state tutte definite in un file header (.h) con lo stesso nome della classe, mentre la definizione dei metodi anch'esso con lo stesso nome della classe ma con estensione .cpp. Le classi sono posizionate all'intero di due cartelle model e view, come indicato in precedenza.

3.3 Estensibilità, evolvibilità e riutilizzo del codice

L'applicazione è strutturata in modo da essere predisposta ad eventuali estensioni/espansioni della stessa. Se si volessero, ad esempio, creare nuovi tipi di equazioni di grado superiore al secondo sarebbe sufficiente inserire nella classe *tipoEquazione* il nuovo tipo di equazione e nel *modelEquazioni* modificare calcola inserendovi un metodo di test, mentre nella view cambiare solo le stringe di messaggio statiche.

Per agevolare la riusabilità del codice, inoltre, si è deciso di usare il meno possibile il framework Qt. Ciò significa che soltanto le classi inerenti alla GUI utilizzano la libreria Qt. Le altre classi utilizzano il quanto più possibile la libreria STL. Questa scelta, se da un lato obbliga ad effettuare più conversioni, dall'altro permette una miglior riusabilità del codice, in quanto si è svincolati dalla libreria Qt.

3.3.1 Polimorfismo e cast a runtime

Il polimorfismo è stato implementato nel seguente modo: un puntatore di tipo equazione, viene istanziato all'opportuna classe derivata della gerarchia nei metodi test (es. primoGrado_test()).

Una volta istanziati per evitare di effettuare una chiamata polimorfa ogni volta che si preme un pulsante ho preferito istanziare già i dati grado e tipo.

```
251 void modelEquazioni::calcola(){
252     if(primoGrado_test()){
253         grado= new razionale(e->getGrado());
254         tipo= new tipoEquazione(e->getTipo());
255         return;
256     }
257     if(spuria_test()){
258         grado= new razionale(e->getGrado());
259         tipo= new tipoEquazione(e->getTipo());
260         return;
261     }
262     if(pura_test()){
263         grado= new razionale(e->getGrado());
264         tipo= new tipoEquazione(e->getTipo());
265         return;
266     }
267     if(secondoGrado_test()){
268         grado= new razionale(e->getGrado());
269         tipo= new tipoEquazione(e->getTipo());
270         return;
271     }
272
273     if(!e){
274         grado=0;
275         tipo=new tipoEquazione(tipoEquazione(4));
276     }
277 }
```

Il metodo *valoreX()* effettua una chiamata polimorfa al metodo *risolvi()* che permette di stampare il risultato di ciascun tipo di equazione gestita dall'applicazione.

```
280 string modelEquazioni::valoreX(){
281     if(e)
282         return e->risolvi();
283     return "Non risolvibile";
284 }
```

Viene utilizzato il *dynamic_cast*, perché in fase di compilazione non si conosce il tipo dinamico dell'oggetto puntato da e. Il metodo *eseguiDelta()*, presente solo nelle equazioni di secondo grado, calcola il grado delle equazioni pure, spurie e normali di secondo grado.

```
292 string modelEquazioni::eseguiDelta(){
293     return dynamic_cast<secondoGrado*>(e)->getDelta().toString();
294 }
```

Il metodo *eseguiDerivata()* effettua una chiamata polimorfa al metodo derivata() che permette di passare da una equazione di secondo grado ad una di primo grado e quindi stamparne il risultato, cambiando gli opportuni dati.

```
312 string modelEquazioni::eseguiDerivata(){
313     if(e)
314         e=e->derivata();
315     else
316         throw nonGestito();
317     if(e){
318         delete grado;
319         delete tipo;
320         grado=new razionale(e->getGrado());
321         tipo=new tipoEquazione(e->getTipo());
322     }
323     else{
324         throw x_Assente();
325     }
326     return e->getEquazione();
327 }
```

3.4 Efficienza e robustezza

Per quanto riguarda l'efficienza, mi soffermo a come è stata gestita la memoria. La parte logica, gerarchia e classi, si è deciso di istanziarle nello stack. Quando però ci si sposta nei due model e nella parte grafica, a parte per i valori booleani, ho optato per l'uso dello heap come spazio di memorizzazione dei dati. La memoria non è condivisa, ma viene lasciata la gestione standard di quest'ultima poiché il progetto non necessita di un tipo di memoria condivisa, dove più puntatori indirizzano ad una stessa zona. Si sarebbe potuto migliorare la gestione della memoria, sfruttando unicamente puntatori per tutti gli oggetti, ma per questioni di tempo non ho affrontato tale soluzione.

Per cercare di generare un'applicazione resistente agli errori, è stata creata in modo tale da catturare ogni eccezione e segnalare attraverso un QMessageBox l'errore che si è commesso e, in alcuni casi, un suggerimento per il corretto inserimento. Nella sezione equazioni si è optato di rendere più forte questi controlli, bloccando a prescindere i pulsanti fino all'inserimento di una equazione corretta. Tuttavia, anche a

quel punto l'equazione inserita potrebbe non essere gestita nel qual caso appariranno appositi messaggi di errore.

4. FUNZIONALITÀ

Le funzionalità fornite dal programma sono quelle tipiche di una calcolatrice, ovvero, somma, sottrazione, divisione, moltiplicazione. A queste si aggiunge la memoria, ovvero la possibilità di memorizzare un polinomio e riutilizzarlo in un secondo momento. Essendo una calcolatrice di polinomi ed equazioni altre operazioni fornite sono l'individuazione del grado, della derivata quando possibile e della possibilità di sostituire la x con un razionale. Nel lato equazioni invece, oltre a tutte queste funzionalità, vi è la possibilità di vedere il loro tipo, il delta delle equazioni di secondo grado e ottenere la soluzione quando questa sia possibile.

5. MANUALE

Per iniziare ad inserire un numero bisogna necessariamente inserire "+" o "–", altrimenti, il risultato proposto sarà 0 o un messaggio di errore se il valore non fosse corretto.

Un razionale può essere inserito sia inserendo solo il numeratore sia inserendo numeratore e denominatore separati dalla /, che è inseribile tramite il pulsante "m/n".

Un monomio può essere scritto nelle seguenti forme:

- Ax^B, dove A e B devono essere razionali;
- x^B, dove B deve essere un razionale.

Sezione polinomi

Il pulsante "=" serve a calcolare il risultato.

Il pulsante M può essere usato dopo aver premuto i pulsanti "+" e "-" se vi è un polinomio già inserito, "+", "-", "*" e ":" se vi è un monomio davanti, o quando la schermata è vuota e si desidera inserire il polinomio che si è salvato come punto di partenza. Altrimenti vi è una segnalazione di errore.

Sezione equazioni

Il pulsante "=" serve a inserire il simbolo di uguaglianza dopo aver inserito tutti i dati del polinomio sinistro. Dopo il primo click, esso cambia in "Calcola" e a questo punto alloca opportunamente l'equazione e si rendono disponibili tutti i pulsanti prima bloccati.

Il pulsante "AC" cancella tutto e ripristina alla situazione iniziale la calcolatrice fatta eccezione per quanto salvato in memoria "M".

Il pulsante M può essere usato solo dopo aver inserito un polinomio a sinistra e prima di cliccare "=". Poi è possibile cliccare la "M" nuovamente sia a destra che a sinistra, secondo le modalità previste anche alla sezione polinomi.

6. ORE DI LAVORO IMPIEGATE

- **Progettazione modello e GUI:** circa 5 ore;
- **Codifica modello e GUI:** circa 31 ore.
- **Debugging e testing**: dopo ogni implementazione o modifica delle classi venivano subito collaudate. Arrivati alla parte grafica e ai file modelPolinomio e modelEquazioni con cui interagisce si è perso un po' di tempo per sistemare qualche errore e migilorare alcune parti, ma stimo più o meno 8 ore.
- **Java**: circa 6 ore.

Il totale del monte ore è quindi di 50, in linea con quanto previsto dalla consegna.