# matriciKalculator

Progetto per l'esame di Programmazione ad oggetti

A.A. 2017-2018

Sviluppato da:

**Alessio Lazzaron**

1142247

&

**Samuele Gardin**

1143807

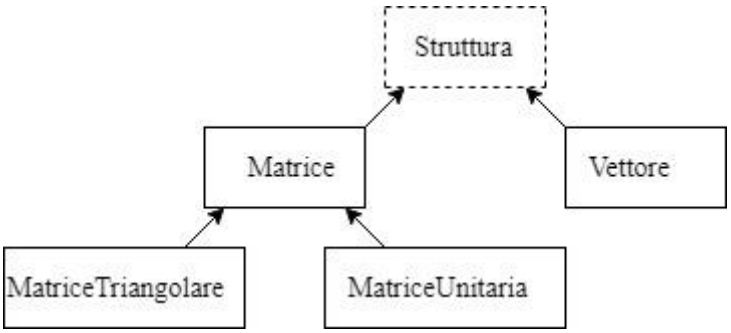
# Sommario

# Descrizione generale

Il progetto consiste in una calcolatrice che permetta di risolvere diverse operazioni tra matrici e vettori. Dati in input degli operatori e un tipo di operazione, la calcolatrice restituirà in output il risultato ottenuto. L'applicazione è in grado di eseguire le operazioni di somma, differenza, moltiplicazione, potenza e poter verificare se una matrice è ortogonale o meno. È stato ideato tenendo a mente i principi di estendibilità, permettendo così di poter inserire nuovi tipi di dato e innumerevoli operazioni.

# Analisi delle gerarchie

**Gerarchia utilizzata nel modello logico**




La classe base della gerarchia è **Struttura**. Essa rappresenta tutti i tipi di dato che possiamo dare in input alla calcolatrice. Essa contiene due campi dati:

- **QVector dim**, nel quale vengono salvate le dimensioni dei tipi inseriti. Abbiamo deciso di usare un tipo QVector per permettere una futura estendibilità a tipi con maggiori dimensioni.
- **Int id**, nel quale viene salvato un numero che identifica il tipo di oggetto a cui ci riferiamo.

**Struttura** è una classe base astratta il cui compito è quello di fornire una base di partenza per operare sui dati delle sue sottoclassi in modo polimorfo. Infatti, contiene quattro metodi virtuali puri costanti:

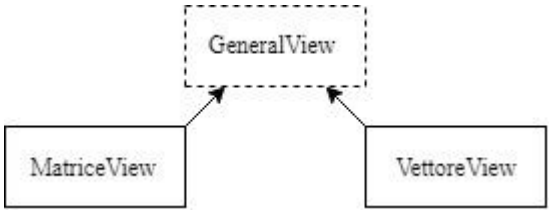
- **somma**
- **differenza**
- **moltiplicazione**
- **potenza**.

Questi verranno implementati nelle sue sottoclassi. Da **Struttura** derivano direttamente Vettore e Matrice i quali differenziano per il numero di dimensioni, Matrice ne ha due e Vettore una. In Matrice sono stati implementati due metodi in più rispetto a Vettore:

- **ortogonale**: restituisce un valore booleano, se la matrice lo fosse restituirebbe true, altrimenti false.
- **moltiplicazioneMV**: esegue la moltiplicazione tra una matrice e un vettore, e viceversa.

Da Matrice derivano direttamente MatriceUnitaria e MatriceTriangolare. Quest'ultima è caratterizzata da un campo dati **char tipo**, il quale se contiene il carattere **'s'** sta a indicare cha la matrice è di tipo triangolare superiore, mentre triangolare inferiore se il carattere fosse **'i'**.

**Gerarchia utilizzata nella GUI**




**GeneralView** è una classe base astratta da cui derivano direttamente **MatriceView** e **VettoreView**. L'interfaccia dove potremo scegliere quale tipo di operando usare è implementata da **GeneralView**, mentre le interfacce che si occupano di figurare il tipo di dato selezionato sono rispettivamente **MatriceView** e **VettoreView**. La GUI è stata ideata seguendo il principio dell'estendibilità, di fatto in modo semplice possiamo aggiungere un nuovo tipo di dato.

L'applicazione è formata principalmente da quattro parti, due che si occupano degli operandi, una per il tipo di operazione e quella più a destra ci farà vedere il risultato ottenuto. Il tutto è creato istanziando un oggetto della classe **Grafica.**

# Analisi del codice polimorfo

**Codice polimorfo nel modello**

La classe base **Struttura** dichiara quattro metodi virtuali puri costanti:

```
virtual Struttura* somma(Struttura* mat) const=0;
virtual Struttura* differenza(Struttura* mat) const=0;
virtual Struttura* moltiplicazione(Struttura* mat) const=0;
virtual Struttura* potenza(int exp) const=0;
```

Questi verranno implementati nelle varie sottoclassi (Matrice, MatriceTriangolare, MatriceUnitaria, Vettore). Il loro contratto è quello di restituire un puntatore all'oggetto della classe di riferimento contenente il risultato. Quest'ultimo è ottenuto dall'operazione svolta con gli input ricevuti. Poiché la classe base contiene almeno un metodo virtuale puro, in **Struttura** è definito il distruttore virtuale.

**Codice polimorfo nella GUI**

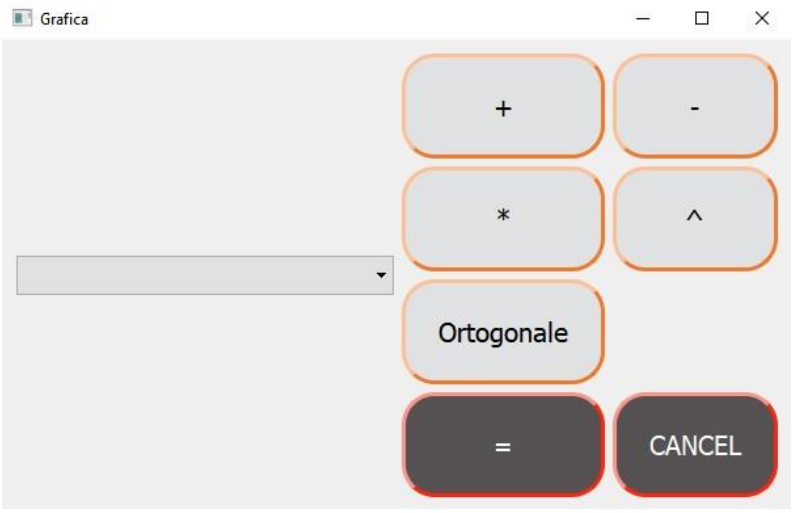
La classe base **GeneralView** dichiara quattro metodi virtuali puri:

```
virtual void addRow()=0;
virtual void addColumn()=0;
virtual void minusCells()=0;
virtual Struttura* getStruttura()=0;
```

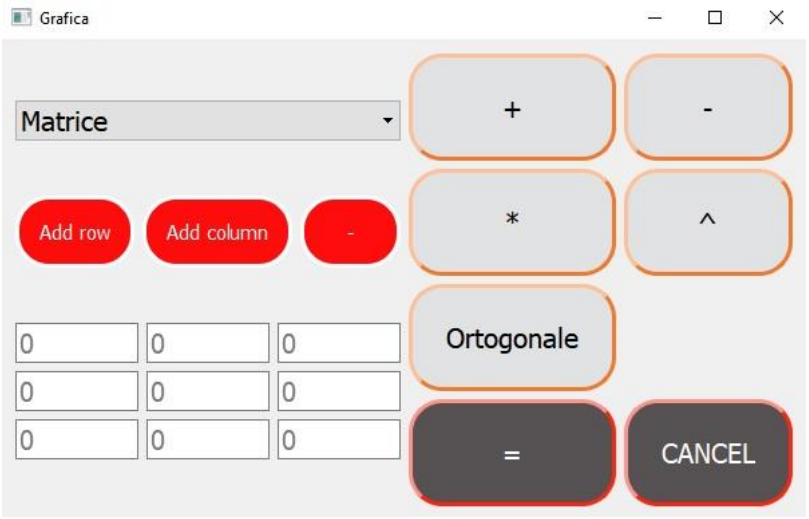
Questi verranno implementati nelle due sottoclassi, **MatriceView** e **VettoreView**. **AddRow** si occupa di inserire una riga se dovesse trattarsi di una matrice, altrimenti una cella in un vettore. **AddColumn** aggiunge una colonna alla matrice di riferimento. **MinusCells** elimina una riga e una colonna dalla matrice, oppure una cella in un vettore. . Poiché la classe base contiene almeno un metodo virtuale puro, in **GeneralView** è definito il distruttore virtuale.

# Manuale utente dell’applicazione

All’apertura dell’applicazione ci troviamo difronte ad un’applicazione semplice ed intuitiva. Alla sinistra ci troviamo un menu a tendina dove possiamo scegliere il primo operando. Cliccandoci sopra si aprirà la lista degli operatori che possiamo selezionare. A destra invece abbiamo dei bottoni tramite i quali possiamo scegliere quale operatore usare.



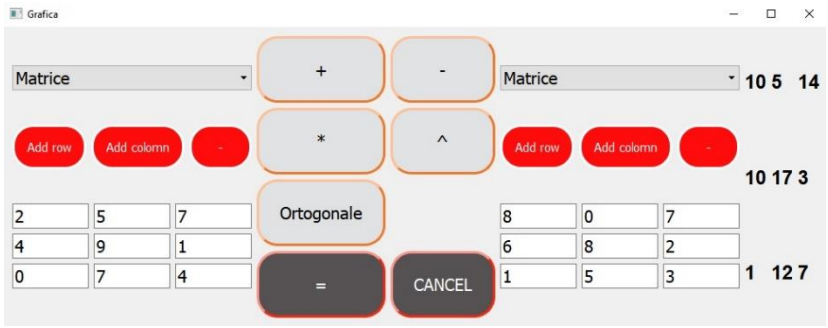

Di seguito possiamo vedere come viene presentata una matrice di default con dimensioni 3x3. Con il bottone “Add row” possiamo aggiungere una riga alla matrice, con “Add column” possiamo invece aggiungere una colonna e con “-” possiamo rimuovere contemporaneamente una riga e una colonna.



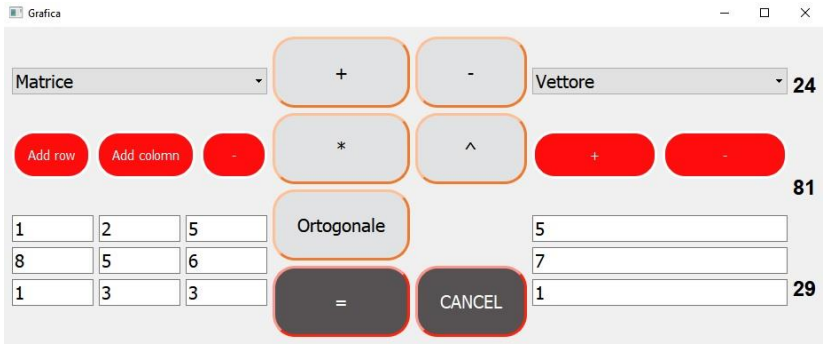
Una volta scelto il primo operatore e l'operazione da eseguire potremo trovarci in tre diverse situazioni:

- se abbiamo scelto somma, differenza o moltiplicazione si aprirà a destra un nuovo menu dove potremo scegliere il secondo operatore.
- se abbiamo selezionato il bottone di potenza si creerà una tabella di testo dove potremo inserire l'esponente.
- altrimenti scegliendo il bottone ortogonale ci apparirà una finestra che ci dirà se la matrice inserita lo è o meno.

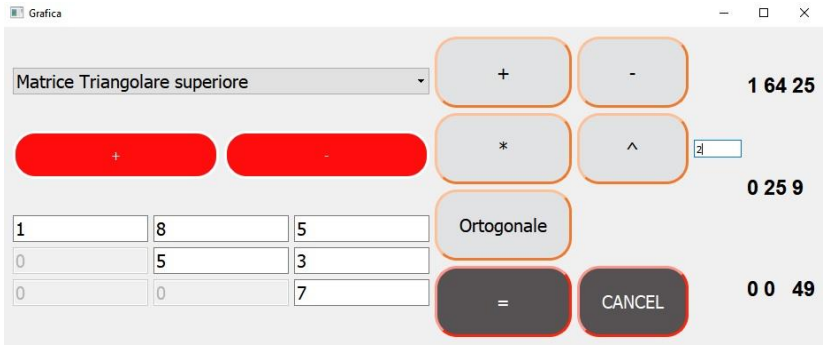
Arrivati a questo punto premendo il bottone '=' verrà restituito a destra il risultato ottenuto dall'operazione, altrimenti se abbiamo inserito delle dimensioni non corrette, verrà visualizzato su una finestra l'errore.




Esempio di somma tra matrici.

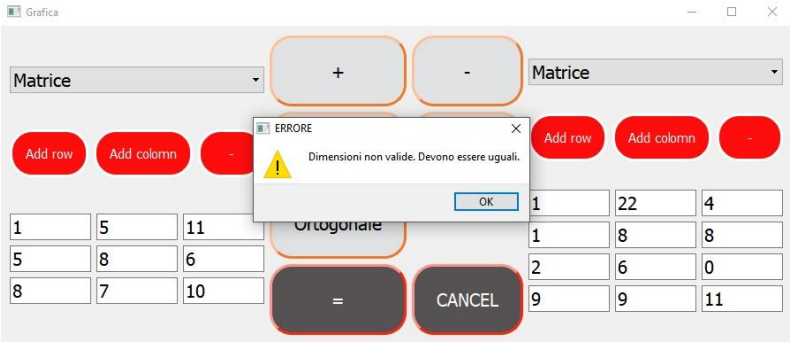



Esempio di moltiplicazione tra matrice e vettore.




Esempio di potenza con esponente 2 di matrice triangolare superiore.


Esempio di matrice non ortogonale.




Esempio di operazione con matrici di dimensioni diverse.

# Analisi delle ore di lavoro

| **Fasi progettuali** | **Ore richieste** |
|---|---|
| Analisi preliminare del problema | 5 ore |
| Progettazione modello | 3 ore |
| Progettazione GUI | 2 ore |
| Apprendimento libreria Qt | 8 ore |
| Codifica modello | 10 ore |
| Codifica GUI | 20 ore |
| Debugging | 4 ore |
| Testing | 3 ore |
| **TOTALE** | **55 ore** |

# Suddivisione del lavoro nella coppia

Dai seguenti compiti si ritengono implicitamente presenti per entrambi: progettazione del modello e della GUI, familiarizzazione con le API di Qt, debugging e testing.

| **Samuele Gardin** |
|---|
| Sviluppo del modello logico |
| Sviluppo del 50% del modello in java |
| Revisione della GUI |

| **Alessio Lazzaron** |
|---|
| Sviluppo dell'interfaccia grafica |
| Sviluppo del 50% del modello in java |
| Revisione del modello logico |

# Note

Il progetto è stato sviluppato all'interno di un calcolatore con le seguenti caratteristiche:

- OS: Windows 10 Pro (versione 1803 buid 17134.228)
- Informazioni su Qt: QtCreator 4.5.0 based on Qt 5.10.0 (MSVC 2015, 32 bit);
- Compilatore Java: Java 9.0.1

# Note per la compilazione

Per eseguire la compilazione del progetto si deve utilizzare il file progetto matriciKalk2.pro già esistente, secondo i seguenti comandi:

- qmake matriciKalk2.pro
- make