# Programmazione ad Oggetti: Kalk

Schiabel Alberto - Matricola 1144672

8 luglio 2018

## Indice



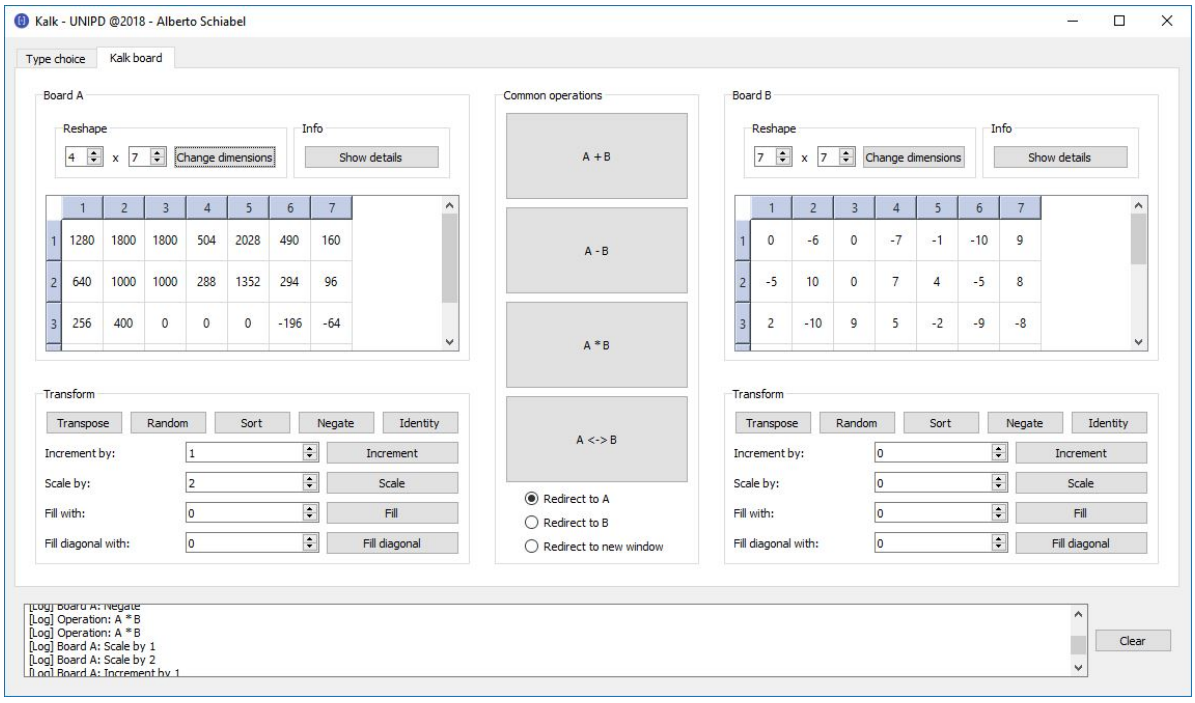



**Figura 1:** Kalk

# 1 Premessa

## 1.1 Compilazione ed esecuzione

### 1.1.1 Progetto Java

Il progetto Java è stato dotato di uno script bash per la compilazione. È sufficiente lanciare da terminale `./kalk-java/build.sh` dopo aver creato la cartella `./kalk-java/out`.

### 1.1.2 Progetto C++ con UI

L'applicazione Qt è dotato di un apposito file di progetto `kalk-gui/kalk.pro` che, ai fini della corretta compilazione, non deve essere assolutamente sovrascritto. Per compilare è sufficiente lanciare il comando `qmake` seguito da `make` all'interno della directory `kalk-gui`.

# 2 Abstract

**Kalk** è un progetto realizzato al fine di offrire un pratico calcolatore di matrici. Le strutture su cui è possibile operare direttamente sono divise in matrici generiche, quadrate e diagonali. È stata definita anche una classe per i vettori, che condivide una classe padre con le strutture sopra citate, ma essa è stata usata solo come strumento d'appoggio per alcune operazioni spiegate nelle prossime pagine. È possibile sia operare o mutare una singola struttura, ad esempio attraversandola con un apposito iteratore o ridimensionandola, sia combinare tra loro matrici diverse con operazioni di tipo algebrico. Determinate operazioni impongono dei vincoli: ad esempio, non è possibile sommare tra loro due matrici aventi dimensioni diverse, o moltiplicare due matrici a meno che non siano nella forma $A_{m,n}$ e $B_{n,p}$. Ogni classe di strutture dispone di alcuni metodi e, in alcuni casi, di iteratori propri. Si è cercato di seguire al meglio la filosofia SOLID, come descritto nel seguente paragrafo.

# 3 Operazioni principali

Nel seguito si va a definire l'elenco dei metodi più importanti di ogni classe della libreria realizzata. Non sono stati citati alcuni getter e setter e altri metodi triviali, che sono comunque riportati commentati nel codice allegato a questa relazione. Tutte le operazioni marcate dal simbolo (Δ) sono ridefinite nella classe `abstract_diagonal_matrix` e applicate solo agli elementi della diagonale principale.

## 3.1 Operazioni definite per ogni Contenitore

Nella *Tabella 1* sono elencate le operazioni principali definite nell'interfaccia `i_container` e implementate nella classe astratta `abstract_container`.

**Tabella 1:** Operazioni più importanti definite per ogni Contenitore

| Operazione | Descrizione |
|---|---|
| **clone** | Ritorna una nuova istanza del tipo dinamico dell'oggetto corrente. |
| **vswap** | Se il tipo dinamico di `A` e `B` è lo stesso, `A.vswap(B)` effettua l'equivalente di `std::swap` tra i due contenitori; ovvero, vengono scambiati i valori delle dimensioni, i puntatori ai rispettivi storage interni, e altri eventuali membri, il tutto solo copie shallow. Se i tipi dinamici non sono identici, viene lanciata l'eccezione `std::bad_cast`. |
| **scale** (Δ) | Moltiplica ogni elemento del contenitore per un valore scalare. |
| **increment** (Δ) | Somma un valore scalare ad ogni elemento del contenitore. |
| **fill** (Δ) | Ogni elemento nel contenitore assume uno stesso valore scalare. |
| **negate** (Δ) | Ogni elemento cambia di segno. |
| **sort** | Tutti gli elementi del contenitore vengono disposti in ordine ascendente. |
| **random** (Δ) | Ad ogni elemento del contenitore viene assegnato un numero casuale in un certo range. |
| **norm_infinity** | Ritorna il valore di $\|A\|_\infty$, dove $A$ è il contenitore. |
| **max** | Ritorna il massimo valore nello storage. |
| **min** | Ritorna il minimo valore nello storage. |
| **sum** (Δ) | Ritorna la somma di tutti i valori dello storage. |
| **max_abs** | Ritorna il massimo valore assoluto nello storage. |
| **min_abs** | Ritorna il minimo valore assoluto nello storage. |
| **sum_abs** (Δ) | Ritorna la somma di tutti i valori assoluti dello storage. |

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \\ a_{3,1} & a_{3,2} \\ a_{4,1} & a_{4,2} \end{bmatrix}$$

Matrice prima del ridimensionamento: $A_{4,2}$

$$A' = \begin{bmatrix} a_{1,1} & a_{1,2} & 0 & 0 \\ a_{2,1} & a_{2,2} & 0 & 0 \\ a_{3,1} & a_{3,2} & 0 & 0 \end{bmatrix}$$

Matrice dopo il ridimensionamento: $A'_{3,4}$

**Figura 2:** Esempio di operazione reshape

Nel caso dell'operazione *random*, è necessario passare un funtore che si faccia carico dell'operazione che deve necessariamente implementare *abstract_ random*, definita nel file *i_ container.h*. Questo step è necessario poiché non è altrimenti possibile desumere come generare una distribuzione uniforme di un tipo T qualsiasi.

Si noti inoltre che $\|A\|_\infty$ è definita diversamente a seconda che $A$ sia un vettore o una matrice: Infatti `norm_infinity` di una matrice ritorna la massima somma assoluta degli elementi di ogni riga, mentre nel caso di un vettore, è equivalente al massimo valore assoluto di tale vettore.

Nella classe `abstract_container` si è fatto largo uso di helper, definiti statici ove possibile, che hanno permesso di scrivere in maniera chiara e concisa le implementazioni dei metodi riportati nella tabella. Questi helper hanno sfruttato funzioni di libreria come `std::transform` e `std::sort`, passando gli appositi iteratori e *lambda functions* come parametri. Si noti che al fine di ottenere il massimo riutilizzo del codice, le *lambda functions* sono state esposte come metodi statici protetti in cui è stata applicata la type deduction automatica (che in molti casi può permettere al compilatore di evitare l'overhead di alternative tipizzate come `std::function`).

## 3.2 Operazioni definite per ogni Matrice

Nella *Tabella 2* sono elencate le operazioni principali definite nell'interfaccia `i_matrix` e implementate nella classe astratta `abstract_matrix`. Ogni matrice generica eredita le operazioni valide per i contenitori generici.

**Tabella 2:** Operazioni più importanti definite per ogni Matrice

| Operazione | Descrizione |
|---|---|
| **get_row_at** | Ritorna una nuova istanza di un vettore che rappresenta la i-esima riga della matrice. |
| **transpose** ($\Delta$) | Traspone la matrice corrente: $A = A^T$. |
| **reshape** | Ritorna una nuova matrice con dimensioni diverse, preservando almeno parte dello storage di partenza. Viene usato lo 0 come valore di padding. Si veda l'esempio. |
| **add** ($\Delta$) | `A.add(B)` ritorna il risultato dell'operazione $A + B$. Se le dimensioni non sono compatibili, viene lanciata l'eccezione `algebric_sum_dimensions`. |
| **subtract** ($\Delta$) | `A.subtract(B)` ritorna il risultato dell'operazione $A - B$. Se le dimensioni non sono compatibili, viene lanciata l'eccezione `algebric_sum_dimensions`. |
| **multiply** ($\Delta$) | `A.multiply(B)` ritorna il risultato dell'operazione $A * B$. Se le dimensioni non sono compatibili, viene lanciata l'eccezione `multiplication_dimensions`. |

## 3.3 Operazioni definite per ogni Matrice Quadrata

Nella *Tabella 3* sono elencate le operazioni principali definite nell'interfaccia `i_square_matrix` e implementate nella classe astratta `abstract_square_matrix`. Ogni matrice quadrata eredita le operazioni valide per le matrici generiche.

**Tabella 3:** Operazioni più importanti definite per ogni Matrice Quadrata

| Operazione | Descrizione |
|---|---|
| **trace** | Ritorna la traccia della matrice quadrata, definita come la somma degli elementi sulla diagonale principale. |
| **determinant** ($\Delta$) | Ritorna il valore del determinante. A seconda dell'ordine di dimensioni della matrice quadrata di partenza, viene applicata una formula diversa. Ad esempio, per le matrici $3X3$ viene applicata la regola di Sarrus, mentre per le matrici di dimensioni maggiori viene applicata l'espansione di Laplace. |
| **fill_diagonal** | Ogni elemento nella diagonale principale assume uno stesso valore scalare. |

### 3.4 Operazioni definite per ogni Matrice Diagonale

Nella *Tabella 4* sono elencate le operazioni principali definite nell'interfaccia `i_diagonal_matrix` e implementate nella classe astratta `abstract_diagonal_matrix`. Ogni matrice diagonale eredita le operazioni valide per le matrici quadrate. Si noti che nel caso delle matrici diagonali il metodo `determinant` è stato ridefinito semplificato, in quanto il determinante di una matrice diagonale è dato semplicemente dal prodotto degli elementi della diagonale.

**Tabella 4:** Operazioni più importanti definite per ogni Matrice Diagonale

| Operazione | Descrizione |
|---|---|
| **to_identity** | Trasforma la matrice diagonale corrente in un'identità, ovvero tutti gli elementi sulla diagonale principale assumono il valore 1. |
| **is_identity** | Ritorna vero *se e solo se* la matrice è un'identità. |
| **diagonal** | Ritorna una nuova istanza di un vettore che rappresenta la diagonale principale della matrice. |

## 4 Struttura della libreria

Sia nella versione C++ che Java, il progetto fa largo uso di strumenti come lambda functions ed iteratori. In C++ questi due elementi del linguaggio sono stati usati in maniera estensiva grazie alle funzioni di libreria standard in grado di sfruttarli, come ad esempio `std::transform` (presente nell'header `<algorithm>`), che permette di applicare una funzione ad un certo range (determinato da un iteratore iniziale e un altro finale) salvandone i risultati nel container definito da un altro (o dallo stesso) iteratore iniziale. Si noti che tutte le definizioni dei metodi che richiedevano di attraversare e modificare lo storage interno (come ad esempio `scale`, `fill` o `negate`) sono state definite in una classe che non ha concezione di tale storage, e che pone come unico vincolo l'esistenza di iteratori di tipo begin/end, sia costanti che non. Tale vincolo è stato assegnato definendoli come metodi virtuali puri.

Nella versione Java sono stati usati gli Stream (disponibili da Java 8) per offrire un'interfaccia quanto più universale e comoda possibile all'end user per l'inserimento di valori nelle matrici; questa scelta ha inoltre permesso di adottare uno stile funzionale nella validazione e importazione dell'input nello storage interno, rendendo il codice che gestisce l'import dei dati conciso e di facile lettura.

### 4.1 Tipi generici

La libreria core del progetto, `linear_algebra`, è strutturata in maniera completamente templatizzata, per garantire il cosiddetto polimorfismo parametrico e per non vincolare l'end user ad alcun tipo predefinito. A causa delle differenze tra C++ e Java, sono stati adotatti due approcci leggermente differenti:

#### 4.1.1 Gestione Template in C++

Grazie al supporto di operatori aritmetici che il linguaggio offre, i requisiti del tipo `T` (individuabile anche appendendo `::value_type` ad una qualsiasi instanza di classe all'interno del `namespace linear_algebra`) sono implicitamente assumibili a tempo di compilazione e non è necessario specificare alcuna classe wrapper dedicata. Questi requisiti implicano la possibilità di effettuare addizioni (`operator+/=`), sottrazioni (`operator-/=`), moltiplicazioni (`operator*/=`), confronti (`operator<, >, <=, >=, ==, !=`) e cast a tre numeri interi: 0, valore neutro della somma, 1, valore neutro della moltiplicazione, e -1. Non è richiesto il supporto cast per altri interi. Nel caso dell'operazione `random`, è necessario passare un funtore che si faccia carico dell'operazione che deve necessariamente implementare `abstract_random_function`, definita nel file `i\_container.h`. Questo step è necessario poiché non è altrimenti possibile desumere come generare una distribuzione uniforme di un tipo T qualsiasi.

#### 4.1.2 Gestione Generics in Java

Per Java si è resa necessaria la creazione dell'interfaccia `NumberWrapper`, che accetta un tipo generico `T`, il quale deve implementare l'interfaccia `Comparable` (deve quindi disporre dell'equivalente degli operatori di confronto). Sono state quindi definite le segnature dei metodi aritmetici (anche in forma variadica) e di metodi come `zero()` e `one()` per ottenere i valori neutri di somma e moltiplicazione. Sono stati inoltre definiti alcuni metodi di default per lanciare eccezioni in caso di overflow e per calcolare il valore assoluto. Ai fini della dimostrazione del progetto, è stata creata la classe `IntegerWrapper` che implementa `NumberWrapper`.

### 4.2 Iteratori

La libreria dispone di due classi di iteratori: una per percorrere tutti gli elementi di una qualsiasi matrice, e una per percorrere gli elementi presenti sulla diagonale; in quest'ultimo caso è richiesta una matrice almeno quadrata. Non si

è ristretto il vincolo d'uso di quest'ultimo iteratore alla sola matrice diagonale, poiché esso è stato sfruttato anche per definire il metodo `trace()` di `abstract_square_matrix`. Nella modellazione logica in entrambi i linguaggi, si è deciso di slegare la definizione degli iteratori alla definizione dello storage su cui iterare, al fine di ottenere il massimo incapsulamento possibile. È possibile infatti cambiare la classe di storage in `abstract_dense_matrix`, attualmente composta da un `std::vector<T>`, senza andare a toccare né la classe `abstract_matrix` né `abstract_container`. Volendo, è inoltre possibile creare una nuova serie di classi che implementi matrici di tipo *Sparse*, e anche in quel caso le classi da `abstract_matrix` in su non necessiterebbero di modifiche. Quanto appena descritto rispecchia la definizione di Open Closed Principle: *Le entità del software dovrebbero essere aperte all'estensione, ma chiuse alle modifiche.*

#### 4.2.1 Gestione Iteratori in C++

Nell'interfaccia `i_container.h` stati dichiarati i metodi virtuali puri `begin()` ed `end()`, con le relative controparti `const`, ispirate alla nomenclatura di STL, mentre in `i_square_matrix.h` sono stati dichiarati i metodi - sempre virtuali puri - `begin_diagonal()` ed `end_diagonal()`. I primi prevedono l'attraversamento di tutto lo storage, e permettono la shortcut `for (auto item : *this) {}`, mentre i secondi richiedono l'utilizzo del classico `for (auto it = this->begin_diagonal(); it != this->end_diagonal(); ++it) {}`. Nelle implementazioni concrete, a `begin()` e `end()` sono stati mappati gli stessi metodi lanciati sul container STL che fornisce lo storage per le matrici (in `abstract_dense_matrix.h`), mentre per l'implementazione di `begin_diagonal()` ed `end_diagonal()` (in `abstract_square_matrix.h`) si è utilizzato un iteratore custom. Quest'ultimo, chiamato `custom_stride_iterator`, estende la classe `std::iterator` specializzando la categoria `std::forward_iterator_tag`. Inizialmente non era stata estesa alcuna classe, ma questo passaggio si è reso necessario al fine di poter far accettare al compilatore l'utilizzo di `custom_stride_iterator` in funzioni di libreria come `std::transform`.

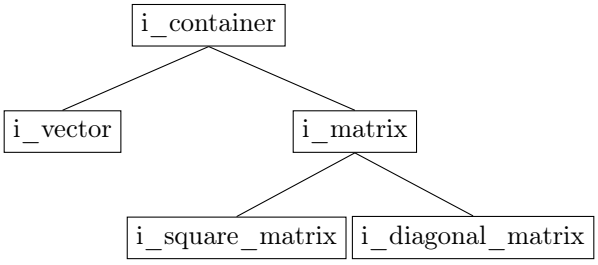
#### 4.2.2 Gestione Iteratori in Java

In Java la creazione degli iteratori è stata molto più immediata. Semplicemente, sono state definite estensioni di `Iterator` e `Iterable`. Il requisito minimo di un `Iterator` è la definizione di 3 metodi:

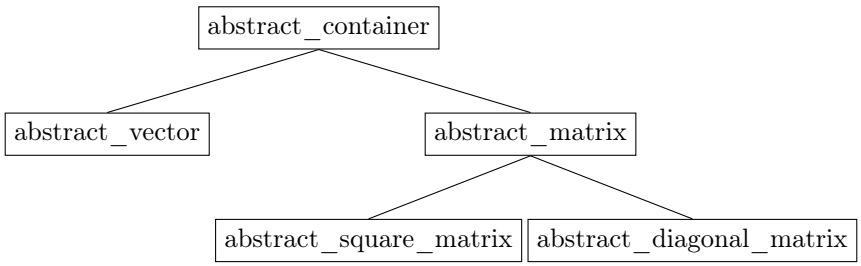
- **hasNext()**: Metodo booleano che deve ritornare `true` se e solo se sono terminati gli elementi da iterare.
- **next()**: Metodo che ha come scopo quello di ritornare il successivo elemento dell'iterazione. Per praticità d'uso sono stati definite delle interfacce custom che permettono la modifica e l'interrogazione (invocazione di `getters`) degli elementi iterati.
- **iterator()**: Metodo triviale, è sufficiente ritornare `this` all'interno della classe che implementa `Iterator`.

### 4.3 Gerarchia di classi

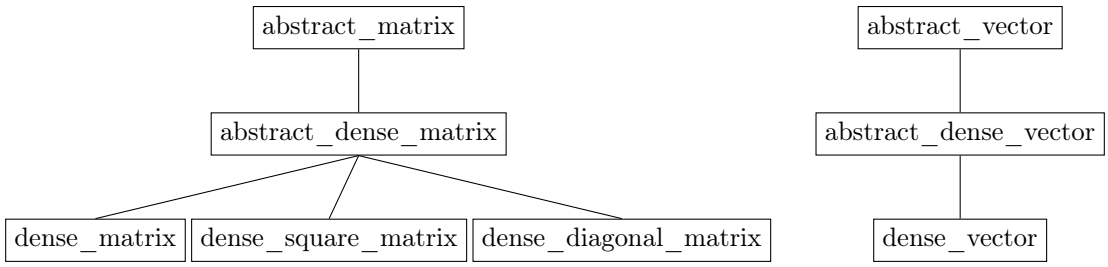
Al fine di seguire i principi di *Interface Segregation* e *Single Responsibility*, è stata una gerarchia piuttosto complessa dal punto di vista della definizione, ma in cui ogni classe definisce un `trait` o un set di operazioni ben definite. Al momento della progettazione, l'obiettivo è stato avere una delle interfacce il cui scopo primario fosse definire metodi virtuali puri da implementare in un'altra classe. Inoltre, per non dovere ripetere codice, si è cercato, per quanto possibile, di far risiedere l'implementazione di molti metodi in classi non consapevoli dello storage interno per contenere gli elementi delle matrici e dei vettori. Queste considerazioni hanno portato alla scelta quasi obbligata di ricorrere all'ereditarietà multipla; di pari passo, a causa dell'insorgere di collegamenti "a diamante", si è dovuta utilizzare anche l'ereditarietà di tipo virtuale. È più semplice visionare la gerarchia sezione per sezione. Sono stati creati quindi due rami di interfacce che partono da `i_container`.




Esiste poi la gerarchia delle implementazioni di tali interfacce, rappresentata da classi astratte:




Si è poi modellato il tipo di storage interno: tra le alternative *Dense* e *Sparse*, si è scelto *Dense*, poiché solitamente si utilizzano matrici compatte quando esse hanno pochi elementi, e dovendo realizzare un'applicazione grafica, matrici con pochi elementi sono semplici e comode da visualizzare. Ecco due porzioni di gerarchia, che definiscono strutture *Dense* a partire da `abstract_matrix` e `abstract_vector`.




**(a)** Matrici dense **(b)** Vettori densi

Se si compone l'intera gerarchia, considerando anche l'ereditarietà virtuale laddove è necessario risolvere il problema del "Diamante", si ottiene la seguente figura:

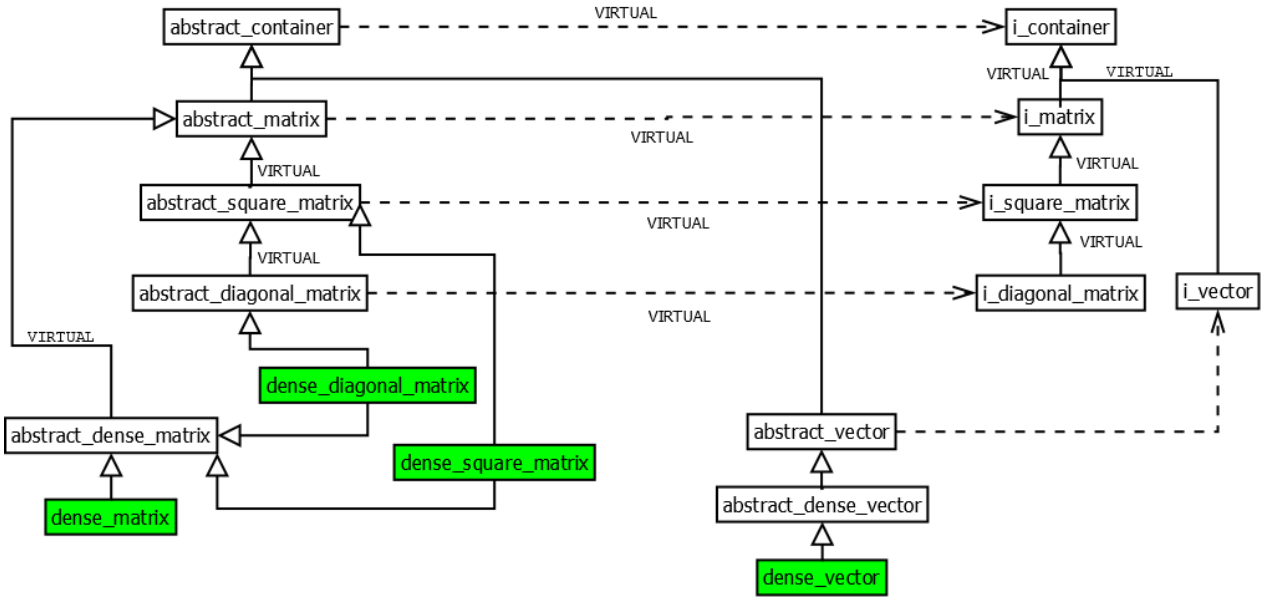



**Figura 3:** Gerarchia completa delle classi di linear_algebra

# 5 Polimorfismo

Il distruttore delle due classi basi principali, `abstract_container` e `i_container`, è dichiarato virtuale, in maniera da implementare correttamente il polimorfismo virtuale e per far sì che i distruttori delle classi figlie vengano chiamati correttamente, evitando *memory leaks*. Tutti i metodi pubblici dichiarati nelle interfacce `i_container`, `i_matrix`, `i_square_matrix`, `i_diagonal_matrix`, `i_vector` sono virtuali, e spesso sono virtuali puri (viene dichiarata solo la segnatura del metodo, e la definizione è delegata alla classi che implementano tali interfacce). È quindi possibile dichiarare un oggetto avente tipo statico `A*` (oppure `A&`), e tipo dinamico `B*` (oppure `B&`), con $T(A) < T(B)$, e con `A->metodo()` sarà possibile invocare l'implementazione di tale metodo definita in B o nella classe padre più vicina a B. Il cosiddetto polimorfismo parametrico è invece già stato discusso in precedenza. Di seguito sono definiti alcuni (non tutti) esempi di polimorfismo applicati in questo progetto.

### 5.1 Esempi di polimorfismo nella libreria

1. In `i_container.h` è stata definito il funtore virtual puro `abstract_random_function`. Il metodo `random` dichiarato in `i_container` si aspetta un puntatore ad un'istanza di `abstract_random_function` come terzo parametro. Vi è un esempio di tale istanza (specializzata per il tipo `int`) chiamata `random_fun` nel file `linear_algebra.h`.

2. In `abstract_container` sono stati dichiarati metodi virtuali puri che abilitano l'utilizzano di iteratori per tutte quelle classi che ereditano tale classe. L'implementazione effettiva avviene qualche livello più in basso se si considera l'albero delle gerarchie, precisamente in `abstract_dense_matrix` per le matrici e in `abstract_dense_vector` per i vettori. Il polimorfismo consente di inizializzare un oggetto `cont` di tipo `abstract_container*` con un'istanza di tipo concreto come `dense_matrix*` o `dense_vector*`, e di poterlo attraversare con un semplice `for (auto& item : *cont) {}`, che andrà ad invocare rispettivamente i metodi costanti `begin` ed `end` della classe `dense_matrix` o di `dense_vector`.

Si noti che altri esempi di questo tipo sono già stati nella sezione *Operazioni Principali*.

### 5.2 Esempi di polimorfismo nell'applicazione GUI

1. Il costruttore del widget `MatrixTableView` richiede un parametro `QAbstractTableModel* model`: è quindi possibile passare qualsiasi puntatore ad un oggetto avente tipo dinamico figlio della classe `QAbstractTableModel` (infatti, in occasioni diverse sono stati passati oggetti di tipo `KalkMatrixModel*` e oggetti di tipo `KalkResultMatrixMod` dove questi ultimi due tipi estendono la classe astratta QAbstractTableModel).

2. Nel widget `KalkOperandBoard`, il membro privato `matrixTableView` è dichiarato come `QTableView*`, ma è istanziato come `MatrixTableView*`, che è un puntatore ad un widget che estende la classe concreta `QTableView`.

3. Nel file `utils/layout/button.h` sono stati definiti due metodi `newButtonWithText`, in cui uno è l'overload dell'altro. Lo scopo del metodo è creare un'istanza (il cui tipo effettivo è definito dal template) di un figlio di `QAbstractButton` con un certo testo e, opzionalmente, un tooltip. Sono presenti dei controlli di tipo *SFINAE* allo scopo di contenere l'uso di `static_cast` all'interno dell'helper, senza doverlo ripetutamente riscrivere all'interno del codice delle views. Poiché sia `QPushButton` che `QRadioButton` sono figli di `QAbstractButton`, è stato possibile utilizzare la stessa funzione sia per settare testo e tooltip dei pulsanti dell'applicazione, sia per impostare testo e tooltip dei selettori radio. È quindi un caso di polimorfismo parametrico.

## 6 GUI - Interfaccia Grafica

È stato adottato il pattern Model-View-Controller, con una forte tendenza alla creazione di piccoli componenti specializzati per favorire la cosiddetta *Separation of concerns*, i quali sono responsabili di emettere `SIGNAL` in risposta ad eventi come click su pulsanti o cambi di valore su campo di input, che sono a loro volta propagati dalle classi container direttamente collegati al controller principale (`KalkBoardWidget`).

Sono stati utilizzate finestre di tipo Modal per tutte quelle azioni che richiedono il focus, come ad esempio l'emissione di una dialog d'errore (gestita dalla classe KalkViewManager, che binda il signal `onError` al metodo `showErrorDialog` della classe `KalkBoardWidget`) o la visualizzazione dei dettagli di una certa matrice (triggerata dal pulsante che reca la scritta *Show details*, definito in `KalkOperandBoardHeader`, il `SIGNAL` dispatchato provoca la visualizzazione di `OperandInfoDialog`).

#### 6.0.1 Log widget

Per agevolare il debugging e ridurre quindi i tempi di sviluppo, è stato realizzato un semplice componente dedicato al logging, che ad ogni evento mostra a video l'azione che ci si aspetta venga eseguita. È definito nella classe `LogWidget`. In un'ottica quasi di *Unit Testing*, non è stato iniziato lo sviluppo degli `SLOT` veri e propri collegati ai `SIGNAL` dell'applicazione fintantoché tutti i log mostrati non si sono rivelati privi di errori logici (quando si hanno $n$ `SLOT` in serie da connettere, è molto facile commettere sciocchezze come collegare il pulsante "Random" all'azione "Negate", e il fatto che cliccando sul pulsante legato all'operazione "Random" venisse loggata invece l'azione "Negate", ha permesso di mettere in evidenza dove fossero i bind errati).

## 6.1 Gestione configurazione applicazione

Al fine di poter ottenere un'interfaccia centralizzata di configurazione, evitando così di "inquinare" il codice con valori hardcoded disorganizzati, è stata creata la classe Settings, che legge le seguenti proprietà da un file di configurazione (presente tra le risorse interne dell'app) e le espone mediante metodi getter. Tali proprietà sono:

1. minMatrixDimension: Valore minimo che una riga o una colonna della matrice può assumere. È utilizzato per settare il valore minimo ai componenti QSpinBox che regolano la dimensione delle matrici renderizzate nel QTableView.
2. maxMatrixDimension: Come sopra, ma mappa il valore massimo di una riga o colonna.
3. defaultRowsDimension: Numero di righe con cui viene istanziata una matrice non quadrata la prima volta.
4. defaultColumnsDimension: Numero di colonne con cui viene istanziata una matrice non quadrata la prima volta.
5. defaultSquareDimension: Numero di righe e colonne con cui viene istanziata una matrice quadrata la prima volta.
6. random.min: Limite inferiore del metodo `random`
7. random.max: Limite superiore del metodo `random`
8. transform.min: Valore minimo assumibile dalle `QSpinBox` in `InlineOperationRow`
9. transform.max: Valore massimo assumibile dalle `QSpinBox` in `InlineOperationRow`

La classe Settings non è esposta direttamente, bensì si è usata la classe SettingsProvider come "proxy" al fine di poter applicare il Singleton Pattern con lazy initialization. Poiché SettingsProvider esponse Settings tramite un metodo statico, che istanzia Settings nella heap (con l'operatore new) oppure ritorna l'istanza già esistente, si è adoperato uno smart pointer statico, che al termine del suo lifetime libera la memoria dinamica utilizzata per instanziare Settings.

# 7 Componenti principali e manuale utente

### 7.0.1 Premessa

Si è cercato di disegnare l'applicazione in maniera tale che possa essere quanto più intuitiva e di immediato utilizzo pratico. Tutti i componenti che richiedono un'interazione (quindi pulsanti, tabelle, menu a finestra e via dicendo) sono stati dotati di un tooltip esplicativo al fine di agevolare l'utente finale più inesperto, migliorando quindi l'UX del programma. Inizialmente era stata valutata anche l'idea di aggiungere un'icona ai pulsanti principali, ma trattandosi di una calcolatrice di matrici, sarebbe stato troppo arduo trovare icone che potessero esprimere i calcoli in maniera intuitiva.

## 7.1 Scelta tipo iniziale

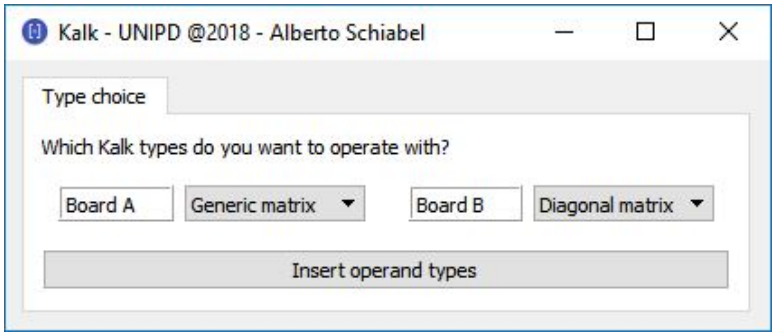



**Figura 4:** Schermata di scelta tipo iniziale

Al primo avvio è disponibile solamente la tab "Type choice", in cui è necessario scegliere le tipologie di matrici per la Board A e la Board B. Una volta confermata la propria scelta cliccando sul pulsante sottostante, si verrà rediretti alla nuova tab "Kalk board", che costituisce il piano di lavoro della calcolatrice. Ogni sezione è raggruppata in un widget QGroupBox e marcata da un titolo significativo.

## 7.2 Board

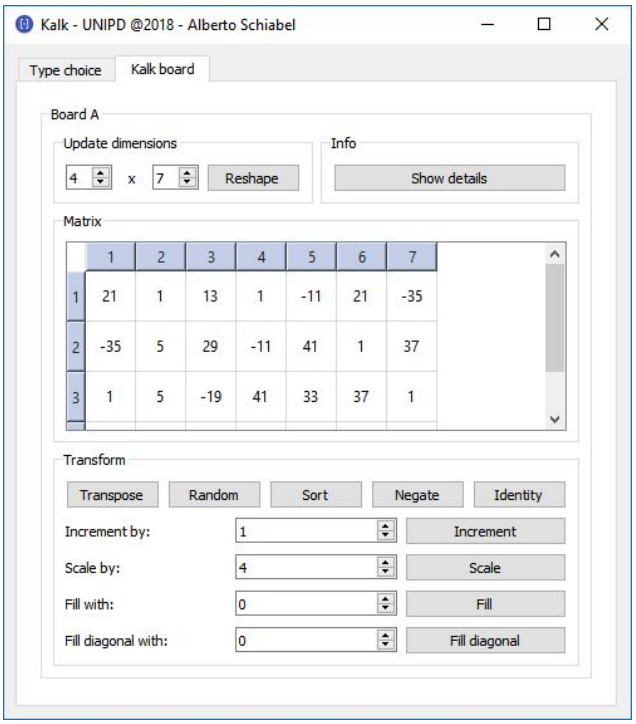



**Figura 5:** Dialog di informazioni di una matrice diagonale

Definita nella classe `KalkOperandBoard`, una board contiene la matrice su cui operare. È possibile fare doppio click su una cella della tabella che ospita la matrice (definita nella classe `MatrixTableView` e avente come modello un'istanza di `KalkMatrixModel`) per poter modificare il valore della matrice negli indici selezionati.

### 7.2.1 Update dimensions

Utilizzando le `QSpinBox` nella sezione **Update dimensions** e confermando la scelta cliccando sul pulsante "Change dimensions", è possibile ridimensionare la matrice della Board corrente. La dimensione minima è $1x1$, quella massima $10x10$.

### 7.2.2 Info

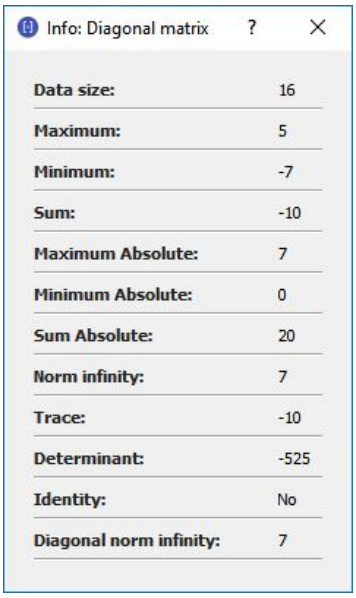



**Figura 6:** Dialog di informazioni di una matrice diagonale

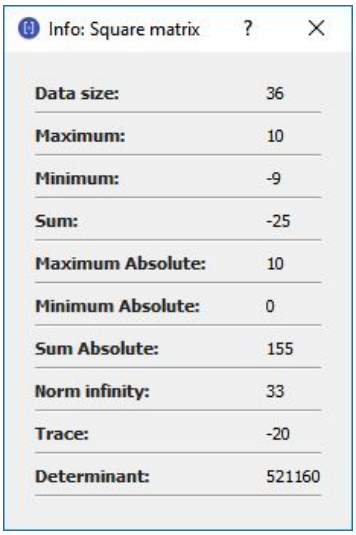



**Figura 7:** Dialog di informazioni di una matrice quadrata

Cliccando il pulsante nella sezione **Info** apparirà una dialog di tipo modal (definita nella classe `OperandInfoDialog`) che elencherà le principali proprietà della matrice. L'elenco di proprietà definite è più o meno ampio a seconda del tipo dinamico della matrice corrente; tale tipo sarà riportato nel titolo della dialog. Le informazioni della dialog sono popolate dinamicamente da un vettore di tuple di stringhe ottenuto dalla classe `MatrixInfoEnumerator`.

### 7.2.3 Transform

Questo widget (definito nella classe `TransformWidget`) consente di mutare la matrice della Board corrente. Cliccando sulla prima fila di pulsanti, è possibile eseguire le seguenti operazioni:

- **Transpose**: Transpone la matrice causando anche l'aggiornamento delle dimensioni nella sezione "Update dimensions".
- **Random**: Popola la matrice con elementi casuali.
- **Sort**: Ordina gli elementi della matrice in ordine ascendente.
- **Negate**: Scambia il segno agli elementi della matrice
- **Identity**: Trasforma la matrice corrente in un'identità. Funziona solo con matrici diagonali.

I successivi widget che consentono di eseguire operazioni leggendo un argomento $x$ come input sono definiti nella class `InlineOperationRow` e consentono le seguenti operazioni:

- **Increment** $x$: Incrementa di $x$ il valore degli elementi della matrice
- **Scale by** $x$: Moltiplica per $x$ il valore degli elementi della matrice
- **Fill with** $x$: Setta tutti gli elementi della matrice allo stesso valore $x$
- **Fill diagonal with** $x$: Setta tutti gli elementi della diagonalo principale allo stesso valore $x$. Funziona solo con matrici quadrate (quindi anche con le diagonali).

### 7.3 Operazioni tra matrici

Il widget definito nella classe `CommonOperationsWidget` permette di eseguire le seguenti operazioni di addizione, sottrazione, moltiplicazione e swap già descritte nei capitoli precedenti. Dove viene assegnato il risultato di tali operazioni (tranne lo swap) viene deciso in base al valore selezionato nel QButtonGroup che contiene 3 possibili QRadioButton di scelta. Sia $\oplus$ una generica operazione tra quelle appena citate:

- **Redirect to A**: $A = A \oplus B$
- **Redirect to B**: $B = A \oplus B$
- **Redirect to new window**: $C = A \oplus B$, $C$ viene renderizzato in una dialog modalless definita nella classe `ResultsWidget`.

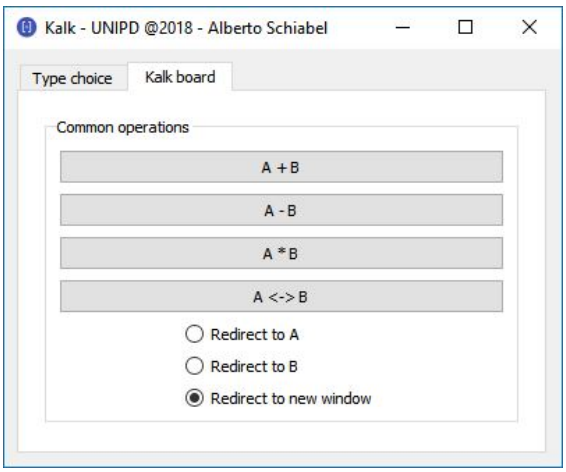



**Figura 8:** Schermata di selezione operazioni tra matrici e redirezione del risultato

## 8 Note Finali

### 8.1 Ambiente di sviluppo

| | |
|---|---|
| **Sistema operativo** | *Windows 10 Education* |
| **Compilatore C++** | *MSVC 2015, C++14* |
| **Compilatore Java** | *javac 1.8.0_102* |
| **Editor GUI C++** | *QtCreator 4.6.1, basato su Qt 5.10.1* |
| **Editor libreria C++** | *Microsoft Visual Studio Community 2015* |
| **Editor Java** | *Intellij IDEA Community Edition 2018.1.4* |

Si noti che tutti i file contenuti in *kalk-c++/lib/linearalgebra* sono stati sviluppati in Visual Studio col supporto del plugin Intellij ReSharper, che mi ha aiutato a rispettare le guideline di C++ e ad evitare pitfall comuni grazie ai suoi strumenti integrati di analisi statica del codice sorgente.

#### 8.1.1 Nota sulla gerarchia

A posteriori posso osservare che la gerarchia di classi si sarebbe potuta semplificare sacrificando alcune astrazioni e derivando anche da classi concrete (invece di applicare alla lettera il principio "Derive from abstractions, not concretions".). Tuttavia, data la natura accademica del progetto e il fatto che quest'anno ereditarietà multipla e virtuale non sono state discusse appieno, questa si è rivelata un'ottima occasione per approfondire questi argomenti e affrontare tipologie di problemi che potrebbero ricapitarmi nell'immediato futuro.

#### 8.1.2 Stima ore

La realizzazione dell'applicativo ha richiesto approsimativamente 55 ore (non cronometrate), di cui

- 10 ore di progettazione

- 2 ore di mockup UI
- 5 ore per realizzare la gerarchia di Java
- 25 ore per realizzare la gerarchia di C++
- 15 ore per realizzare e testare l'applicazione GUI

## 8.2 Differenze tra i compilatori C++

È stato perso più tempo del previsto per capire come mai alcuni errori o warning apparissero solo utilizzando il compilatore g++ e non in MSVC. Da alcune ricerche su StackOverflow è emerso che MSVC non segue fedelmente i dettami di ISO-C++, e dunque in alcuni casi accetta e compila anche codice che non dovrebbe essere valido.