Programmazione ad Oggetti
Relazione progetto Kalk

Mihai Eni
Matricola 1101684

08-09-2018

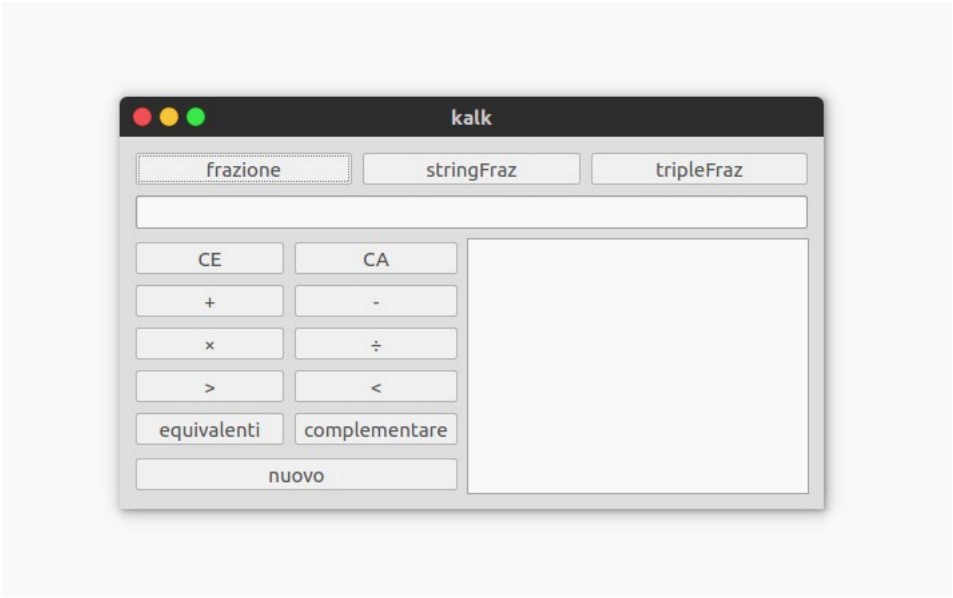

Figura 1: Schermata di default Kalk

Indice

1 Scopo del Progetto

La richiesta del progetto è la realizzazione di una calcolatrice che opera con diversi titpi di dato I dati disponibili nella calcolatrice sono i seguenti :

- **frazione**, sono frazioni dove numeratore e denominatore sono numeri reali, permettono di eseguire le principali operazioni algebriche e di confronto tra frazioni;
- **stringFraz**, sono frazioni con l'aggiunta di *mid* un parametro stringa, questo tipo di frazioni peremettono le stesse operazioni di *frazione*;
- **tripleFraz**, sono frazioni con l'aggiunta di *triple* un numero intero, anche quest'ultimoe peremettono le stesse operazioni di *frazione*;

La calcolatrice non cambia graficamente tra i tre tipi di dato,cambia l'implementazione.

2 Gerarchie utilizzate

2.1 Gerarchia dei tipi di dato

2.1.1 Classe Frazione

La classe Frazione rappresenta la divisione tra numeri reali mediante frazioni. Ha due campi dato di tipo intero che indetificano numeratore e denominatore.
Esegue l'overloading virtuale di diversi metodi:

- **virtual frazione* operator +(const frazione*) const;**: restituisce un puntatore ad una frazione uguale alla somma tra 2 frazioni;
- **virtual frazione* operator -(const frazione*) const;**: restituisce un puntatore ad una frazione uguale alla differenza tra 2 frazioni;
- **virtual fr azione* operator*(const frazione*) const;**: restituisce un puntatore ad una frazione uguale alla moltiplicazione tra 2 frazioni;
- **virtual frazione* operator/(const frazione*) const;**: restitutisce un puntatore ad una frazione uguale alla divisione tra 2 frazioni;
- **virtual bool operator==(const frazione*) const;**: restituisce un booleano che indica se le frazioni sono uguali;
- **virtual bool operator!=(const frazione*) const;** restituisce un booleano che indica se le frazioni sono diverse;
- **virtual bool operator<(const frazione*) const;**: restituisce un booleano che indica se una frazione è minore dell'altra;
- **virtual bool operator>(const frazione*) const;**: restituisce un booleano che indica se una frazione è maggiore dell'altra.

Definisce i seguenti metodi virtuali:

- **virtual frazione* clone() const;**: restituisce un puntatore alla copia della frazione;
- **virtual std::string toString()const;**: restituisce una *std::string* che rappresenta la frazione;
- **virtual int getMCD() const;**: restiruisce il massimo comun divisore tra numeratore e denominatore;
- **virtual void reduce();**: riduce ai minimi termini la frazione;
- **virtual frazione* minima() const;**: restituisce un puntatore alla copia della frazione ridotta ai minimi termine;
- **virtual double razionale() const;**: ritorna il razionale corrispondente alla frazione;
- **virtual frazione* complementare() const;** ritorna un puntatore alla frazione complementare.

2.1.2 Classe stringFraz

La classe *stringFraz*, derivata dalla classe *frazione*, rappresenta un tipo di frazione alla quale è stata aggiunta una stringa centrale.
Esegue l'override dei seguenti metodi virtuali della classe frazione:

- **stringFraz* clone() const;**
- **stringFraz* minima() const;**
- **stringFraz* complementare() const;**
- **stringFraz* operator +(const frazione*) const override;**
- **stringFraz* operator -(const frazione*) const override;**
- **stringFraz* operator *(const frazione*) const override;**
- **stringFraz* operator /(const frazione*) const override;**
- **bool operator==(const frazione*) const override;**
- **bool operator!=(const frazione*) const override;**
- **bool operator<(const frazione*) const override;**
- **bool operator>(const frazione*) const override;**

2.1.3 Classe tripleFraz

La classe *tripleFraz*, derivata dalla classe *frazione*, rappresenta una doppia frazione mediante l'aggiunta di un intero chaimato **triple**. Triple viene visto come un secondo numeratore di conseguenza si vengono a creare 2 frazioni con lo stesso denominatore e 2 numeratori che possono essere uguali.
Esegue l'override dei seguenti metodi virtuali della classe frazione:

- **tripleFraz* clone() const;**
- **int getMCD() const;**
- **void reduce();**
- **tripleFraz* minima() const;**
- **double razionale() const;**
- **tripleFraz* complementare() const;**
- **tripleFraz* operator +(const frazione*) const;**
- **tripleFraz* operator -(const frazione*) const;**
- **tripleFraz* operator*(const frazione*) const;**
- **tripleFraz* operator/(const frazione*) const;**
- **bool operator==(const frazione*) const;**
- **bool operator!=(const frazione*) const;**
- **bool operator<(const frazione*) const;**
- **bool operator>(const frazione*) const;**

2.2 Gerarchia del controllo input

La gerarchia del controllo input serve per controllare che l'input sia conforme per la successiva creazione dei tipi di dato. Il controllo input è formato da una classe base astratta **check**, da cui derivano tutte le classi concrete del controllo input come **checkFraz**, **checkString** e **checkTriple**

2.2.1 Classe check

La classe base astratta *check* serve per contenere il nome dei parametri da inserire nell'input
Definisce i seguenti metodi propri:

- **std::list<QString> getParametri() const;**: serve a ottenere il nome dei parametri dei tipi di dato;
- **void addParemetro(const QString);**: serve contenere il valore di ciascun parametro definito nell'input;

Definisce i seguenti metodi virtuali puri:

- **bool virtual parser(QString, QString&) =0;**: serve per controllare che l'input inserito sia corretto rispetto al tipo di dato;
- **virtual frazione* getFrazione(std::vector<QString>*)const=0;**: usato per creare il tipo di dato con i valori inseriti nell'input;

2.2.2 Classe checkFraz

La classe *checkFraz* derivata dalla classe astratta **check**: serve per definire il nome dei parametri e per il controllo del input del dato *frazione*.
Implenta i seguenti metodi:

- **bool virtual parser(QString, QString&);** serve per controllare che l'input del numeratore e del denominatore sia un numero intero;
- **virtual frazione* getFrazione(std::vector<QString>*)const=0;**: usato per creare una frazione una volta che il parser è stato eseguito su tutti i parametri;

2.2.3 Classe checkString

La classe *checkString* derivata dalla classe astratta **check** serve per definire il nome dei parametri e per il controllo del input del dato *stringFraz*.
Implenta i seguenti metodi:

- **bool virtual parser(QString, QString&);** serve a controllare che l'input per la stringa mid sia composto solo da lettere e numeri reali;
- **virtual frazione* getFrazione(std::vector<QString>*)const=0;**: usato per creare una stringFraz dopo che il parser è stato eseguito su tutti i parametri.

2.2.4 Classe checkTriple

La classe *checkString* derivata dalla classe astratta **check** serve a definire il nome dei parametri e per il controllo del input del dato *tripleFraz*.
Implenta i seguenti metodi:

- **bool virtual parser(QString, QString&);** usato per controllare che l'input per triple sia un numero reale;
- **virtual frazione* getFrazione(std::vector<QString>*)const=0;**: usato per creare una stringFraz dopo che il parser è stato eseguito su tutti i parametri.

3 Classe contenitore Kalk

La classe *Kalk* è la classe contenitore generale che viene usata per mettere in communicazione i tre tipi di oggetti che contiene al suo interno:

- **model**: gestore delle checkS;
- **input**: una vista per l'input;
- **view**: la vista della calcolatrice.

3.1 Classe model

La classe *model* permette lo scambio del tipo di dato tra *frazione*, *stringFraz* e *tripleFraz* a livello logico e gestisce le operazioni dei vari tipi di dato. Contiene al suo interno i *check* per fare l'input, tutte le frazioni che vengono create, 2 operandi per fare le operazioni tra frazioni, result che serve salvare il risultato delle oprazioni e i valori inseriti nell'input. Contiene i seguenti SLOT:

- **void setOperand(QListWidgetItem*);**: usato per assegnare agli operandi le frazioni immagazzinate;
- **void changeCheck(int);**: serve a cambiare il tipo di *check* e conseguenza tipo di dato usato;
- **void openInput();** usato per comunicare a **kalk** di aprire l'input;
- **void getNewValore(QString, QString);** uasato per immagazzinare l'utimo input inserito.

Contiene i seguenti SIGNALS:

- **void sendInputKo(QString);**: per communicare a **kalk** che l'input è errato;
- **void sendCloseInput();**: per communicare a **kalk** che l'input è corretto e quindi chiudere l'input;
- **void sendToVideo(QString);**: per mandare alla **view** una stringa da mettere a video;
- **void sendChangedCheck();**: comunica a **kalk** che il cambio della *check* è avvenuto correttamente.

3.2 Classe input

La classe *input* serve per poter inserire l'input e inviarlo al model. Contiene una *QLineEdit()* usata per inserire l'input e una *QLabel()* usata per comunicare quale parametro inserire;

3.3 Calsse view

La classe *view* serve per formare la GUI e contiene:

- **QLineEdit* constVideo;**: serve per fare vedere l'operatore selezionato, il risultato oppure l'errore avenuto all'iterno di un'operazione;
- **QListWidget* oggetti;**: serve per far vedere tutte le frazioni create e salvare i risultati;
- **QVBoxLayout* vista;**: layout generale usato per contenere *constVideo* ,*text* e tutti i pulsanti delle operazioni.

Contiene i seguenti SLOT:

- **void addToVideo(QString);** serve per cambiare il contenuto di *constVideo*;
- **void refreshFrazioni(std::vector<QString>);** serve per aggiornare il contenuto di *oggetti*.

Contiene i seguenti SIGNALS che vengono inviati quando si preme un pulsante nella tastiera e quindi indica l'operazione da fare:

- *void sendChangeCheck(int);*
- *void sendSelectOperand(QListWidgetItem*);*
- *void sendOpenInput();*
- *void sendSomma();*
- *void sendSottrazione();*
- *void sendMoltiplicazione();*

- *void sendDivisione();*
- *void sendMaggiore();*
- *void sendMinore();*
- *void sendUgualianza();*
- *void sendComplementare();*
- *void sendClearElement();*
- *void sendClearAll();*
- *void sendCloseInput();*

4 Descrizione del codice polimorfo

Nelle classi precedentemente descritte sono presenti metodi che eseguono chiamate polimorfe. Per fare esempi di chiamate polimorfe ogni chiamata fatta dalla classe *model* sugli operandi, il tipo statico del puntatore è di tipo *frazione** mentre l'oggetto puntato potrebbe essere di tipo *frazione*,*stringFraz* o *tripleFraz*. Altri esempi di chiamate polimorfe possono essere individuati nella classe *check* nelle operazioni di input, dopo vengono chiamati i metodi *parser* ed altri.

5 Manuale utente

5.1 Avvio

All'avvio dell'applicazione Kalk questa si troverà nella scheda *Frazione*, per gli altri tipi di dato basterà premere i tasti in alto a destra cambiando così tipo di dato su cui operare.

5.2 Struttura

La calcolatrice è divisa in tre aree principali:

- nell'area sulla destra è presente una lista che in base al tipo di dato su cui si sta lavorando mostra tutti gli oggetti creati fino a quel momento per il tipo di dato corrente;
- la seconda parte sotto i tipi di dato è la visualizzazione dei dati inseriti, dei risultati e degli erorri sulle operazioni;
- e una terza contenete tutti i pulsanti delle operazioni disponibili.

5.3 Dinamica operazioni

La dinamica di selezione di operandi e operazioni è leggermente diversa dalla normale calcolatrice:

- per fare una operazioni selzionare prima gli operandi sulla destra e premere poi il pulsante con l'operazione;
- per inserire un nuovo dato premere nuovo e poi si aprira l'input dove ci sono i nomi dei parametri e la barra dove inserire il valore del paramentro

6 Ore utilizzate

Sono state utilizzate le seguenti ore alla consegna:

- **Progettazione**: 2 ore;
- **Studio delle librerie di QT**: 20 ore;
- **Scrittura codice**: 23 ore;
- **Scrittura relazione**: 2 ore;
- **Test**: 1 ora;
- **Traduzione gerarchia tipi in Java**: 2 ore.

7 Ambiente di sviluppo

- Sistema operativo: 16.04 LTS
- gcc version 5.4.0 20160609
- Qt version 5.5.1
- QMake version 3.0