PROGETTO DI PROGRAMMAZIONE AD OGGETTI
A.A. 2017/2018

KALK

LUCA STOCCO **Matricola: 1125280**
NICCOLO' VETTORELLO **Matricola: 1122264**

Relazione di Niccolo' Vettorello

INDICE DEI CONTENUTI:

1) ABSTRACT

KALK e' una calcolatrice che permette l'esecuzione di operazioni sui seguenti tipi:

a) numeri razionali
b) numeri complessi in forma polare
c) numeri complessi in forma cartesiana
d) componenti (induttori, resistori, condensatori)
e) tuple (vedere l'altra relazione per una descrizione piu' dettagliata)

Le possibili operazioni sono contestuali al tipo corrente e spaziano dalla classica aritmetica tra tipi numerici fino ad arrivare ad azioni piu' specifiche per gli altri tipi gestiti (ad esempio serie e parallelo tra componenti, o operazioni tra tuple).

2) SUDDIVISIONE DEL LAVORO

I compiti che hanno portato alla realizzazione del progetto sono stati cosi' suddivisi:

ANALISI DEL PROBLEMA → e' stata svolta da entrambi i componenti del gruppo;

PROGETTAZIONE DEL PARSER → svolta da entrambi gli studenti;

REALIZZAZIONE DEL MODELLO
(COMPRENSIVO DI ECCEZIONI)→ il compito e' stato svolto interamente dallo studente Luca Stocco;

REALIZZAZIONE DELLA VISTA → il compito e' stato svolto dallo studente Niccolo' Vettorello;

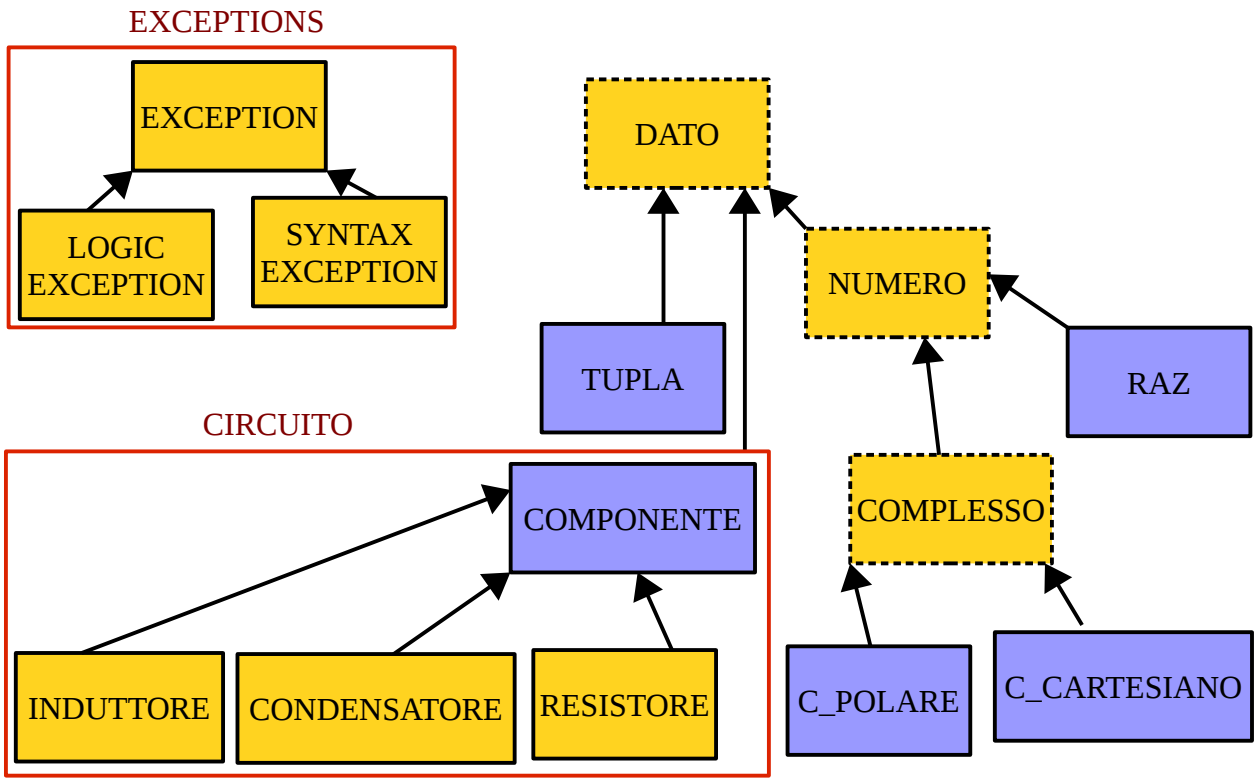
REALIZZAZIONE DELLA PARTE JAVA → il compito e' stato svolto dallo studente Niccolo' Vettorello;

REALIZZAZIONE DEL CONTROLLER → il compito e' stato svolto dallo studente Niccolo' Vettorello;

FASE DI COMPILAZIONE, TESTING E DEBUGGING → svolto da entrambi i componenti del gruppo;

3) GERARCHIA DEI TIPI: DESCRIZIONE E USO

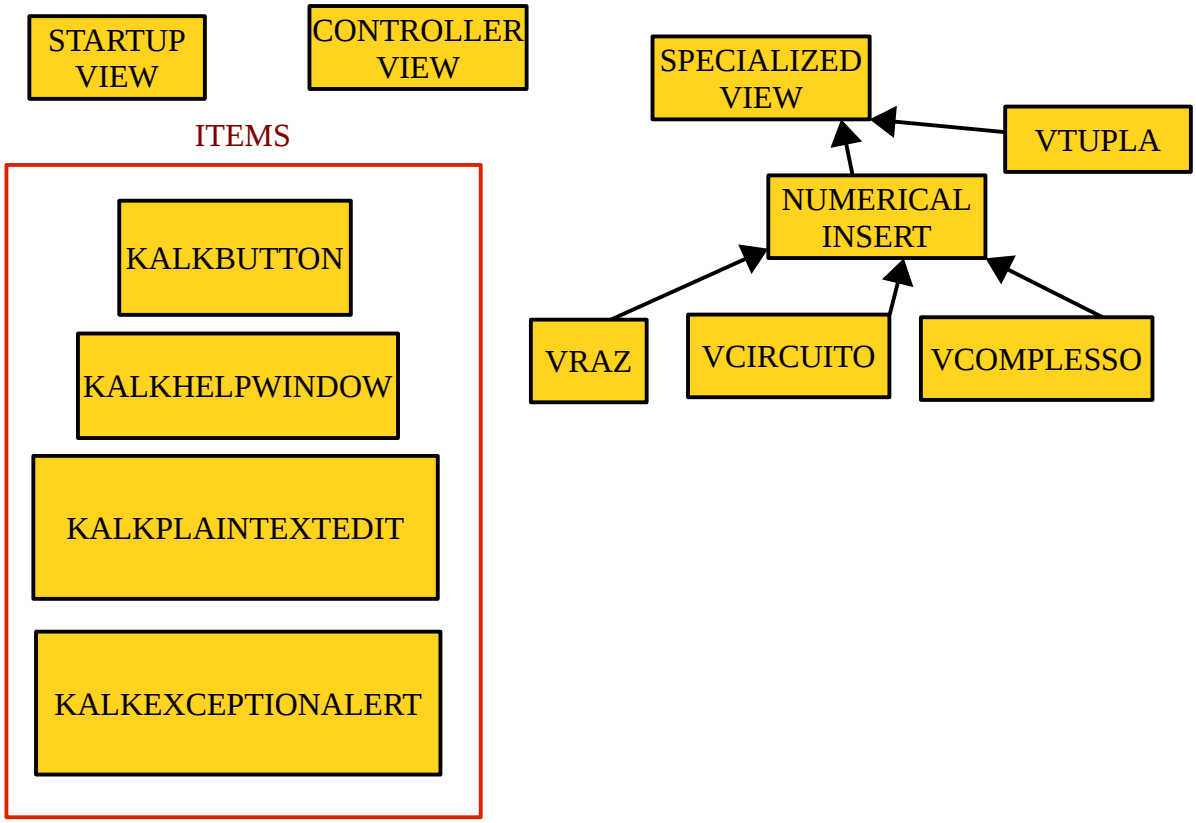
NB 1: nel grafico sottostante, le caselle in viola rappresentano le classi scelte come tipi gestiti da KALK, quelle tratteggiate sono classi astratte, quelle con linea continua sono concrete.
NB 2: con il termine Circuito ci si riferisce alla "mini" gerarchia con classe base Componente
NB 3: il compito e' stato realizzato dallo studente Luca Stocco, pertanto sara' trattato superficialmente in questa sede.



La totalita' dell'interazione dell'utente con il modello passa attraverso il parser (trattato piu' dettagliatamente nell'altra relazione). Infatti esso si occupa di interpretare la stringa ricevuta dall'utente (l'invio avviene tramite la pressione del

tasto ENTER), di segnalare eventuali casi eccezionali, principalmente errori di sintassi, e di effettuare le chiamate al modello piu' opportune se non vengono rilevati errori.
Il modello prevede una gestione degli errori interni (logici e sintattici) che possono avvenire durante l'esecuzione.
Il parser restituisce quindi un puntatore di tipo Dato* all'oggetto risultato (questo spiega perche' vi e' un unica classe base astratta), su cui viene effettuata una chiamata polimorfa per la conversione a stringa. Quest'ultima viene poi inviata alla Vista per essere visualizzata.
Si e' voluta quindi staccare completamente qualsiasi dipendenza diretta tra Vista e Modello: la Vista opera solo con QString, e non ha alcuna relazione con i tipi gestiti dal Modello.

4) GUI: DESCRIZIONE E MANUALE UTENTE



Viene presentata ora una descrizione delle classi che compongono la Vista: ogni classe e' corredata da una descrizione che ne presenta scopo e metodi principali, oltre a motivare scelte particolari di implementazione, qualora ve ne dovessero essere.
La GUI prevede inoltre un pulsante "SERVE AIUTO?" che mostra un menu' con indicazioni su come usare la vista corrente: non si ritiene quindi necessario inserire un manuale utente.
Per quanto riguarda la gestione della memoria si e' cercato di utilizzare il piu' possibile il meccanismo di "parenting" offerto da QT, intervenendo "manualmente" solo dove strettamente necessario.

KalkButton.h

KalkButton rappresenta l'implementazione di Kalk di QPushButton, dalla quale deriva. Un KalkButton e' un pulsante a cui e' associato un intero reperibile tramite il metodo getDataNumber(). I pulsanti KalkButton vengono usati estensivamente nella vista poiche' permettono un piu' facile scambio di informazioni.

KalkExceptionAlert.h

La classe KalkExceptionAlert deriva da QWidget e serve alla calcolatrice per visualizzare l'eccezione verificatisi.
Essa inoltre controlla il tipo di eccezione ricevuta: se è di tipo logico viene conseguentemente richiesta la chiusura dell'applicazione.

KalkPlainTextEdit.h

Rappresenta l'implementazione di Kalk del caratteristico display della calcolatrice. Si e' scelto di far derivare la classe da QplainTextEdit, in maniera tale da permettere un interazione con l'utente veloce e intuitiva. La classe fornisce uno slot pubblico write_on_kpte(), che implementa la logica necessaria per scrivere sul display (utilizzata appunto dai pulsanti delle viste che devono scrivere/operare con stringhe)

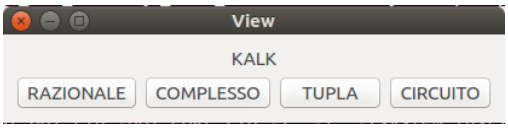
KalkHelpWindow.h

Deriva da QWidget. Rappresenta il menu di aiuto che compare una volta premuto il pulsante "SERVE AIUTO?". Presenta un QplainTextEdit in read only mode che fornisce indicazioni su come usare la vista corrente

controller_view.h

controller_view e' una classe creata per permettere la gestione delle viste startup_view e specialized_view: il suo scopo principale e' infatti quello di garantire che solo una delle due viste sia attiva in ogni momento, al fine di evitare sprechi di memoria.
controller_view deriva da QObject, in modo da poter implementare vari signal e slot (principalmente per la comunicazione di dati al Controller). I due slot principali sono createStartup(), che serve a creare startup_view per permettere la selezione del tipo, e createSpecialized(int), che controlla il tipo selezionato dall'utente e crea la vista piu' opportuna. In createStartup() abbiamo inoltre una **chiamata polimorfa al distruttore di specialized_view** (delete SpecializedV): specialized_view infatti eredita il distruttore virtuale da QWidget e ne fa override.

startup_view.h



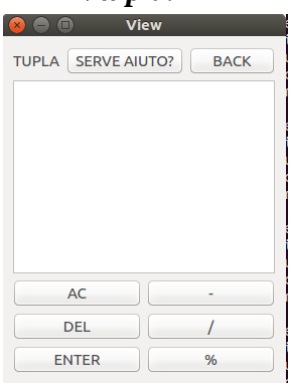
startup_view e' la classe che rappresenta la schermata iniziale della calcolatrice per la selezione del tipo. Essa deriva da QWidget ed implementa lo slot type_FButtonTStartup(), il cui compito e' quello di reperire l'intero associato al KalkButton premuto ed emetterlo tramite il segnale type_FstartupTController(int).

specialized_view.h

specialized_view e' la classe base della gerarchia delle viste. Il suo scopo e; quello di implementare tutti i QWidget comuni a tutte le viste e di fornire tutti gli slot e i signal per la comunicazione con controller_view. specialized_view deriva da QWidget.
I metodi in specialized_view possono essere divisi in tre macro gruppi: metodi locali alla vista, metodi per la comunicazione di stringhe verso il controller e viceversa, metodi per l'invocazione degli operatori speciali dei tipi.

- I metodi locali alla vista sono costituiti essenzialmente dagli slot che mostrano il menu di aiuto contestuale (helpraz(), helpComplesso(), helpCircuito(), helpTupla()) e dal segnale back(), che permette di ritornare alla startup_view per selezionare un altro tipo;
- I metodi per la comunicazione di stringhe sono gli slot raccogli_testo_corrente(), che estrae la stringa dal display e la emette mediante il segnale inviaQS(QString), e lo slot mostra_result(QString), che scrive sul display la stringa risultato;
- I metodi relativi alle operazioni speciali dei tipi sono gli slot specops_raz(), che estrae l'informazione sul pulsante premuto e la invia tramite il segnale inviaSORaz(int), specops_complesso(), il quale svolge una funzione analoga, e i segnali specialized_view_inviaVolt(double) e specialized_view_inviaFreq(double), che inviano l'informazione relativa ad un avvenuto cambiamento nei valori dei QdoubleSpinBox di Vcircuito;

Vtupla.h

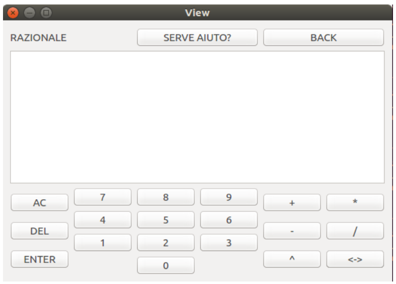


VTupla rappresenta la vista che gestisce il tipo Tupla. Essa deriva da specialized_view. Cosi come tutte le classi che implementano una vista per un tipo specifico, essa aggiunge i pulsanti relativi alle operazioni specifiche e crea per essi le connessioni opportune.

numerical_insert.h

numerical_insert deriva da specialized_view. Lo scopo di questa classe e' implementare il tastierino numerico per tutte le viste specifiche che ne fanno uso.

Vrazionale.h



Deriva da numerical_insert e rappresenta vista per la gestione del tipo Razionale. Aggiunge tutti i pulsanti degli operatori specifici ed effettua le connessioni opportune, tra cui quelle per gli operatori speciali di Razionale (vedi il paragrafo dedicato a specialized_view);

Vcomplesso.h

Deriva da numerical_insert e rappresenta vista per la gestione del tipo Complesso. Aggiunge tutti i pulsanti degli operatori specifici ed effettua le connessioni opportune, tra cui quelle per gli operatori speciali di Complesso (vedi il paragrafo dedicato a specialized_view);

Vcircuito.h

Deriva da numerical_insert e rappresenta vista per la gestione del tipo Circuito. Aggiunge tutti i pulsanti degli operatori specifici (incluse le spinbox) ed effettua le connessioni opportune, anche quelle per le QdoubleSpinBox (vedi il paragrafo dedicato a specialized_view);

NB 1: si e' deciso inoltre di non gestire le eccezioni relative a downcast falliti nella vista dal momento che la struttura della stessa preclude a prescindere il verificarsi di una tale situazione.

NB 2: lanciando l'applicativo con il comando:

```
valgrind --leak-check=full --show-leak-kinds=all ./kalk
```

viene rilevata una moltitudine di leak di memoria. La totalita' di questi errori tuttavia sembra riguardare librerie di Linux utilizzate da QT. Stante questo fatto, la presenza di questi errori e' stata completamente ignorata poiche' non riguardante l'applicativo presentato.

5) CONTROLLER

Il controller e' una parte fondamentale di Kalk, poiche' crea il collegamento tra Model e View. Deriva da Qobject per poter usare il meccanismo Signal&Slot e implementa una serie di slot che gestiscono la logica delle chiamate al Model.
Il controller "comunica" con il Model in due modi diversi:

- PARSER→ la creazione di un oggetto parser e l'utilizzo del suo metodo resolve() sono tutto cio' che serve al Controller per gestire le stringhe che riceve dalla GUI in maniera corretta (i dettagli sul parser sono contenuti nell'altra relazione);
- OPERAZIONI SPECIALI→ alcuni operatori, come quello di conversione tra razionale e double, o l'operatore coniugato su un complesso (oppure le spinbox in VCircuito) sono collegati a degli slot specifici che ne gestiscono il comportamento;

Il controller offre un segnale per la comunicazione della stringa alla vista (data_controller_to_GUI(QString));

La classe possiede inoltre i seguenti campi:

- cv → puntatore a controller_view che gestisce la vista;
- from_gui→ Qstring che rappresenta la stringa comunicata dalla GUI;
- tipo_corrente → e' il tipo corrente che si sta trattando;
- conv_raz → ogni volta che si produce un risultato di tipo Razionale ne viene salvata anche la versione double, per evitare che una serie di conversioni successive sullo stesso oggetto facciano perdere troppa precisione;
- conv_complesso→ puntatore a complesso che rappresenta la conversione di un risultato complesso ottenuto (per evitare di perdere troppa precisione in una serie di conversioni consecutive sullo stesso oggetto);

Il controller effettua alcune chiamate polimorfe:

1. [controller::data_GUI_to_controller(QString)]
   conv_complesso = (static_cast<Complesso> (result))→converti();*
   oggetto_corrente ha tipo statico Dato*, e tipo dinamico C_polare* o C_cartesiano*. Esso viene castato staticamente a Complesso* e viene effettuata una chiamata polimorfa all'operatore converti();
2. nell'implementazione di vari metodi viene invocato per più volte il metodo toString() sul puntatore Dato* result;
3. [controller::socomplesso_logic(int)]
   Complesso result_coniugato = result->coniugato();*
   aux ha tipo statico Complesso*, chiamata polimorfa a coniugato();

Per come e' stata progettata la classe, per avviare una instanza di KALK e' sufficiente creare un oggetto di controller nel main().

6) ANALISI DELLE TEMPISTICHE

Le seguenti sono le tempistiche relative alla parte svolta dallo studente Niccolò Vettorello. La soglia di 50 ore disponibili e' stata leggermente sforata: ciò è dovuto al fatto che la discussione e progettazione del parser ha richiesto un discreto investimento di tempo:

ANALISI DEL PROBLEMA → *circa 5 ore*;
PROGETTAZIONE DEL PARSER → *circa 15 ore;*
PROGETTAZIONE VISTA→ *circa 5 ore;*
REALIZZAZIONE DEL CODICE DELLA VISTA → *circa 5 ore;*

REALIZZAZIONE PARTE JAVA→ *circa 10 ore;*
REALIZZAZIONE DEL CONTROLLER→ *circa 4 ore;*
REVISIONE→ *circa 4 ore;*
TESTING E DEBUGGING → *circa 5 ore;*

7) AMBIENTE DI SVILUPPO

Oltre al computer del laboratorio, lo studente Niccolo' Vettorello ha usato un ambiente di sviluppo con le seguenti caratteristiche:

- SISTEMA OPERATIVO : Ubuntu 17.10
- COMPILATORE : g++ 7.2.0
- LIBRERIA QT : versione 5.9.1

8) ISTRUZIONI PER LA COMPILAZIONE

C++: eseguire i seguenti comandi per compilare:

1. spostarsi nella cartella KALK con: `cd KALK`
2. eseguire il comando: `qmake -project "QT += widgets" "CONFIG += c++11"`
3. eseguire il comando: `qmake`
4. eseguire il comando: `make`
5. lanciare l'applicativo con: `./KALK`

JAVA: eseguire i seguenti comandi per compilare:

1. spostarsi nella cartella Java con: `cd Java`
2. compilare con il comando: `javac Use.java`
3. avviare l'eseguibile con: `java Use`