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

KALK

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

Relazione di Luca Stocco

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 il relativo paragrafo 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;

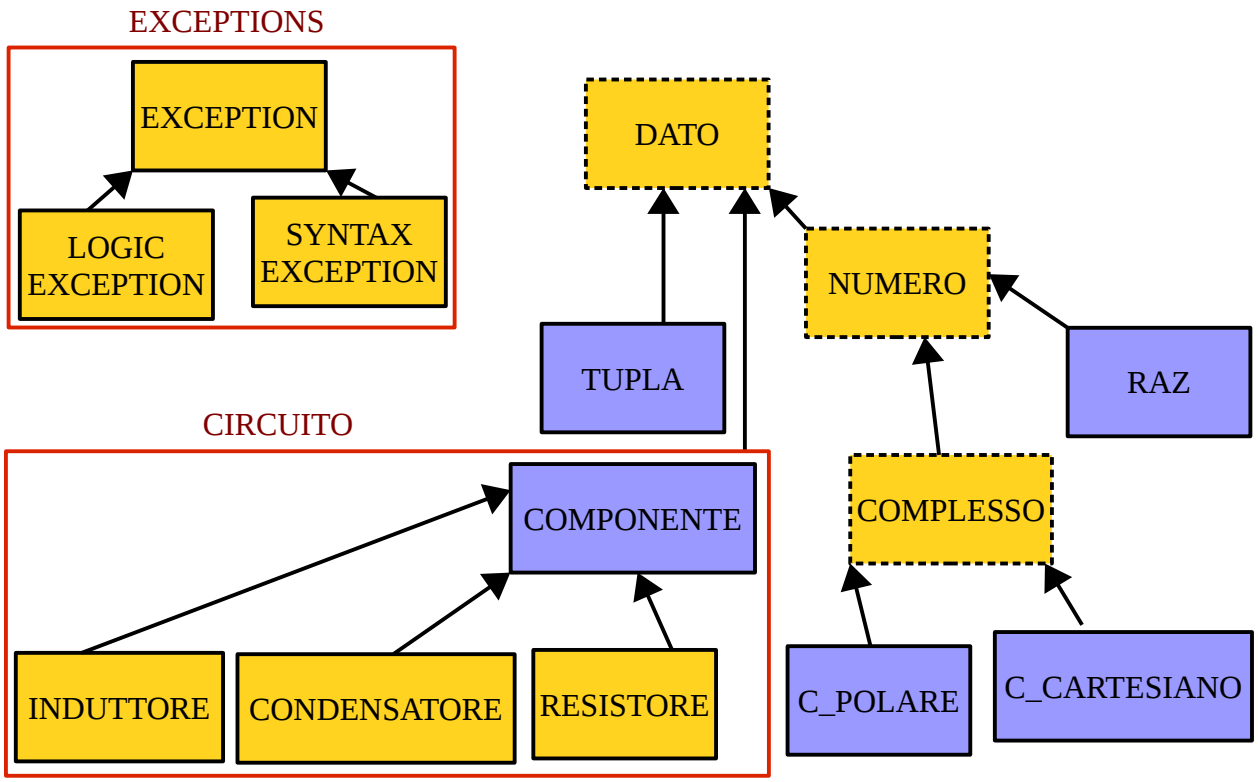
REALIZZAZIONE DELLA PARTE JAVA → il compito e' stato svolto dallo studente Niccolo' Vettorello;

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



Segue una descrizione delle classi che compongono il Modello: ogni classe e' corredata da una descrizione che ne presenta scopo e metodi principali. Si è preferito usare la gerarchia come "luogo" di definizione dei metodi virtuali, e inserire poi la maggior parte delle chiamate polimorfe nel parser, il vero "attore" che prende una decisione su chi chiamare e sul come farlo.
NB: i metodi *solve_operation* servono per il parser e saranno approfonditi nella sezione apposita.

Dato

Dato è una classe astratta che rappresenta un generico tipo di calcolo. Essa definisce un distruttore virtuale per la gerarchia, e possiede i seguenti metodi virtuali puri:

- toString() → il cui scopo è ritornare una stringa rappresentante l'oggetto;
- operator==(const Dato &) → overloading dell'operatore di uguaglianza;

L'esistenza della classe Dato è motivata anche dal funzionamento della classe parser, che sfrutta il fatto che tutti i tipi gestiti derivino da una sola classe

Tupla

Tupla è una classe concreta che deriva direttamente da Dato, rappresenta un insieme di coppie nella forma Metadato1,dato1,Metadato2, dato2,….,MetadatoN,datoN. Essa, oltre al costruttore da stringa, mette a disposizione vari metodi come exists_metadati(string), che verifica se un metadato è presente, o insert(string,string), che inserisce una entry Metadato,dato.
Le operazioni da notare sono le seguenti:

- l'overloading degli operatori '-', '/', '%' rispettivamente differenza, intersezione e join tra tuple (maggiori informazioni nel menù di aiuto);
- l'overriding dei metodi virtuali puri di Dato;

Circuito

Questa gerarchia rappresenta un generico Componente. La classe Componente è concreta e derivante da Dato, ma l'utente si troverà a gestire solo Resistori, Condensatori e Induttori. La motivazione di questa scelta progettuale è stata dettata in buona parte dalla logica del parser.
Oltre a implementare i metodi virtuali puri di Dato, la classe Componente definisce gli operatori di serie (+) e parallelo (-), e possiede due campi statici che rappresentano il voltaggio e la frequenza, i quali possono essere modificati o letti mediante gli opportuni setter e getter. Definisce inoltre un metodo virtuale impedenza();
Le classi Resistore, Induttore e Condensatore derivano da Componente e ne espandono le funzionalità ridefinendo l'operatore di uguaglianza, il metodo toString() e il metodo impedenza();

Gerarchia Numero

Numero è una classe astratta derivata da Dato. Rappresenta un generico numero e definisce la costante di classe pi. Possiede i seguenti metodi virtuali puri:

- operator+, per la somma tra numeri;
- operator-, per la differenza tra numeri;
- operator/, per la divisione tra numeri;
- operator*, per la moltiplicazione tra numeri;

Implementa inoltre due metodi statici per la conversione da gradi a radianti e viceversa.
Da Numero derivano due classi, Raz e Complesso.

Raz è concreta e rappresenta un numero razionale. Essa implementa le operazioni di somma, divisione, differenza, e moltiplicazione, elevamento a potenza, oltre all'operazione di uguaglianza e al metodo toString(). Fornisce poi le operazioni reciproco(), radice_quadrata(), e getter per il numeratore e denominatore.

La classe Complesso è astratta e aggiunge i metodi virtuali puri converti() e coniugato().

Le classi C_polare e C_cartesiano sono concrete e derivano da Complesso, e rappresentano numeri complessi scritti in forma polare e cartesiana, rispettivamente. Esse implementano tutti gli operatori derivati da Numero, le operazioni converti() e coniugato() derivate da Complesso, e l'operatore d'uguaglianza e il metodo toString() derivati da Dato.

5) ECCEZIONI

Per quanto riguarda la gestione delle eccezioni si e' deciso di operare nel modo seguente:

- E' stata definita una classe exception con un campo stringa, un getter per la stringa e un costruttore da stringa;
- Sono state fatte derivare da exception le classe syntax_exception e logic_exception;
- La classe syntax_exception rappresenta un errore di sintassi: sono definiti errori di sintassi gli errori causati da una interazione con l'utente non corretta. Per esempio, l'inserimento di una stringa con un operatore inesistente e' un errore di sintassi;
- La classe logic_exception rappresenta un errore logico, ossia un errore interno al programma stesso ed indipendente dall'utente;

La politica generale per un errore di sintassi e' quella di mostrare un messaggio tramite la Vista, che informi l'utente dell'eventuale errore e permetta l'inserimento di una nuova stringa.
La politica generale per la gestione di errori logici e' quella di visualizzare un messaggio tramite la vista che informi l'utente dell'avvenuto errore e successivamente chiuda il programma.
Le eccezioni vengono rilanciate fino ad arrivare al Controller, che decide quindi come gestire il problema.
NB: il parser è templatizzato per garantire che non si possano verificare eccezioni di tipo logico (cast falliti, ecc…). Tuttavia la gerarchia dei tipi lancia lo stesso questo tipo di eccezioni qualora ne dovesse rilevare. Ciò è dovuto al fatto che si è voluta mantenere una indipendenza tra il parser e il Modello stesso per quanto riguarda la rilevazione e gestione degli errori

5) GUI: DESCRIZIONE E MANUALE UTENTE

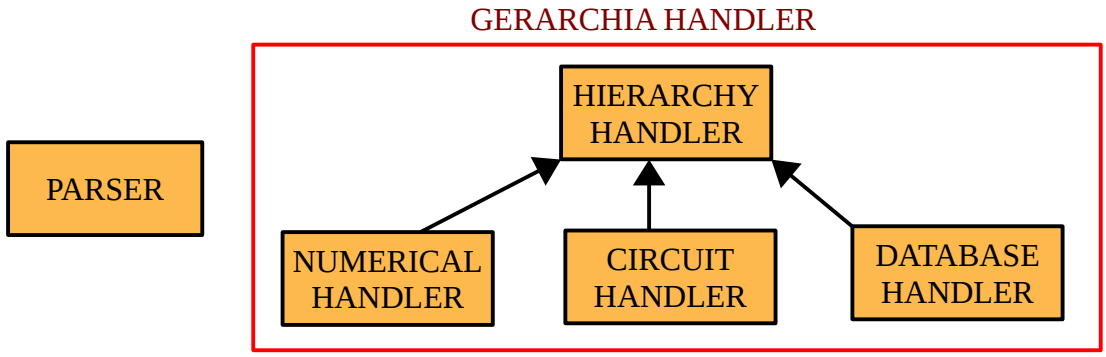
NB: la parte relativa alla GUI verrà trattata superficialmente in questa sede, in quanto compito dello studente Niccolò Vettorello.

La GUI è stata progettata per un utilizzo intuitivo da parte dell'utente. Si è cercato di rendere l'esperienza il più simile possibile a quella del classico applicativo "Calcolatrice" presente in ogni sistema operativo.
Avviando una istanza di KALK appare una schermata con quattro pulsanti, che permettono all'utente di scegliere quale vista specializzata usare. La pressione di uno qualsiasi di questi pulsanti porta alla generazione della vista appropriata.
In ogni vista specializzata sono presenti:

a) una etichetta che indica il nome della vista;
b) un pulsante "SERVE AIUTO?" che crea una schermata con indicazioni contestuali su come utilizzare la vista;
c) un pulsante "BACK" che serve a tornare alla selezione del tipo;
d) un display, sul quale l'utente può scrivere con la tastiera, oppure utilizzando i pulsanti che KALK fornisce;
e) i tasti AC (cancella tutto ciò che si trova nel display), DEL (cancella l'ultimo carattere nel display), ENTER;
f) tutte le viste tranne quella di Tupla hanno un tastierino numerico che permette di scrivere sul display;
g) ogni vista specializzata ha i propri operatori, che permettono di scrivere sul display;
h) La vista Complesso e la vista Razionale hanno tasti per operatori speciali (nel menu di aiuto della relativa vista vi sono anche informazioni su come usarli correttamente);

Il principio di funzionamento è molto semplice (varia leggermente per gli operatori speciali): è sufficiente scrivere sul display la stringa che rappresenta l'operazione desiderata e successivamente premere il tasto ENTER per ottenere il risultato.

6) PARSER



Dopo un'attenta analisi, si e' scelto di realizzare un parser templatizzato che fa uso di una gerarchia di classi chiamata Gerarchia Handler. Questa scelta e' stata dettata dal fatto che si e' voluto emulare il piu' possibile il comportamento di una classica calcolatrice: vi e' l' inserimento di una stringa da parte dell'utente, che viene parsata, elaborata, e il conseguente risultato viene mostrato dalla vista.
Segue una descrizione delle classi che compongono il parser.

Gerarchia Handler

La Gerarchia Handler e' formata da una classe base astratta Hierarchy_Handler, e da tre classi concrete derivate da quest'ultima.
La Gerarchia Handler permette al parser di creare oggetti di classi derivate dal tipo T. Inoltre ogni handler gestisce gli operatori del suo tipo specifico (tramite una lista di char inizializzata nel costruttore tramite load_operators()).
Da notare il metodo create(string) definito come virtuale puro in Hierarchy Handler, che chiama il corretto costruttore dell'oggetto da instanziare (se il parser e' instanziato a Complesso viene chiamato invece il metodo create_complex(string), dal momento che la classe Complesso e' astratta. Questo comportamento si sarebbe potuto evitare creando un Handler derivato da Numerical per la gestione del tipo Complesso).

Parser

La classe Parser crea un albero per gestire l'ordine e la priorita' delle operazioni. Cio' avviene nel seguente modo:

- si crea un oggetto di Parser instanziato al tipo desiderato T passando come parametro la stringa da analizzare e un oggetto fittizio di tipo statico Dato* e con tipo dinamico D<=T;

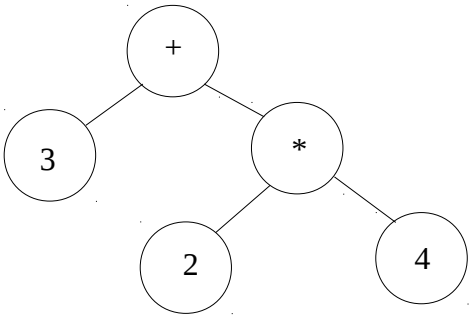
- se la stringa e' vuota viene lanciata una eccezione di sintassi, altrimenti viene creato l'handler appropriato tramite il puntatore all'oggetto fittizio;
- si procede alla costruzione dell'albero tramite build_tree(string);

Un albero di parsing appena creato e' formato da nodi aventi le seguenti caratteristiche:

1. hanno un puntatore di tipo T* all'oggetto creato;
2. hanno due puntatori di tipo Node* ai nodi figli destro e sinistro;
3. hanno campo char per l'operatore;
4. hanno un campo int per la precedenza dell'operatore;

Ogni nodo puo' rappresentare un operatore, il che implica che il suo campo char sia diverso dal carattere nullo e che esisteranno sicuramente un sottoalbero sinistro e destro, o un operando, che avra' il campo char nullo e sara' una foglia.

ES. 3 + 2 * 4



Il parser è stato progettato in maniera tale da essere estendibile con relativa facilità: per aggiungere un nuovo tipo T alla gerarchia ed implementare per esso una funzionalità di parsing da stringa è sufficiente:

- fare derivare T da Dato;
- definire la grammatica per un tipo T e creare un costruttore T(std::string) che si occupi di farla rispettare;
- definire un metodo statico T::solve_operations(Dato*, Dato*, char), presi due i Dato*, tenti il downcast a T*: se il cast riesce *solve_operation* deve invocare il corretto metodo, basandosi sul carattere ricevuto come parametro. *Il metodo solve_operation dovrebbe essere il metodo che effettua le (eventuali) chiamate polimorfe per la classe T*;
- estendere Hierarchy_Handler con una classe T_Handler, necessaria a definire i caratteri che rappresentano gli operatori per il tipo T;

La gestione della precedenza degli operatori viene effettuata dal metodo set_prec(char): essa e' stata implementata in modo statico, essenzialmente per motivi di tempo. E' possibile renderla dinamica spostando la sua logica dal parser alla gerarchia handler apposita, tramite una map o una struttura ad hoc, che associ ogni operatore alla sua precedenza.

Parser presenta inoltre il metodo resolve(Node*), dedicato alla risoluzione e gestione dell'albero di parsing: questo metodo e' ricorsivo e scorre l'albero in maniera infissa chiamando l'opportuno metodo statico solve_operation(Dato*, Dato*, char), risolvendo di volta in volta l'operazione richiesta.

I **metodi solve operation** sono caratteristici di ogni classe della gerarchia dei tipi e sono il vero nucleo del polimorfismo del progetto: si occupano infatti di associare l'operatore della stringa inserita dall'utente al giusto metodo del modello, *cercando di effettuare quante piu' chiamate polimorfe possibile*.

Ad esempio, il metodo solve_operation di Complesso, effettua un cast dinamico dei due parametri di tipo Dato* a Complesso*, e invoca il giusto operatore , con una chiamata polimorfa, senza bisogno di dettagli sul tipo dinamico degli stessi.

```
Complesso* Complesso::solve_operation(const Dato* a, const Dato* b, char o){
        auto l=dynamic_cast<const Complesso*>(a);
        auto r=dynamic_cast<const Complesso*>(b);
        if(l && r){
                switch(o) {
                case '+':
                        return static_cast<Complesso*>(l->operator+(r));
                case '-':
                        return static_cast<Complesso*>(l->operator-(r));
                case '*':
                        return static_cast<Complesso*>(l->operator*(r));
                case '/':
                        return static_cast<Complesso*>(l->operator/(r));
                default:
                        throw syntax_exception("Operatore non valido");
                }
        }
        else
                throw logic_exception("tipo di dati errato");
}
```

7) ANALISI DELLE TEMPISTICHE

Le seguenti sono le tempistiche relative alla parte svolta dallo studente Luca Stocco. 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 MODELLO→ *circa 10 ore;*
REALIZZAZIONE DEL CODICE DEL MODELLO → *circa 20 ore;*
REVISIONE → *circa 9 ore;*
TESTING E DEBUGGING → *circa 5 ore;*

8) AMBIENTE DI SVILUPPO

Oltre al computer del laboratorio, lo studente Luca Stocco ha usato un ambiente di sviluppo con le seguenti caratteristiche:

- SISTEMA OPERATIVO : OpenSuse Leap 15
- COMPILATORE : g++ 7.3.1
- LIBRERIA QT : non necessaria per la realizzazione del Modello

9) 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`