Progetto di Programmazione ad Oggetti A.A. 2017/2018

Favaro Marco Matricola: 1123187

Clerici Michele Matricola: 1122656

Relazione di Favaro Marco

Progetto: Kalk

Indice:

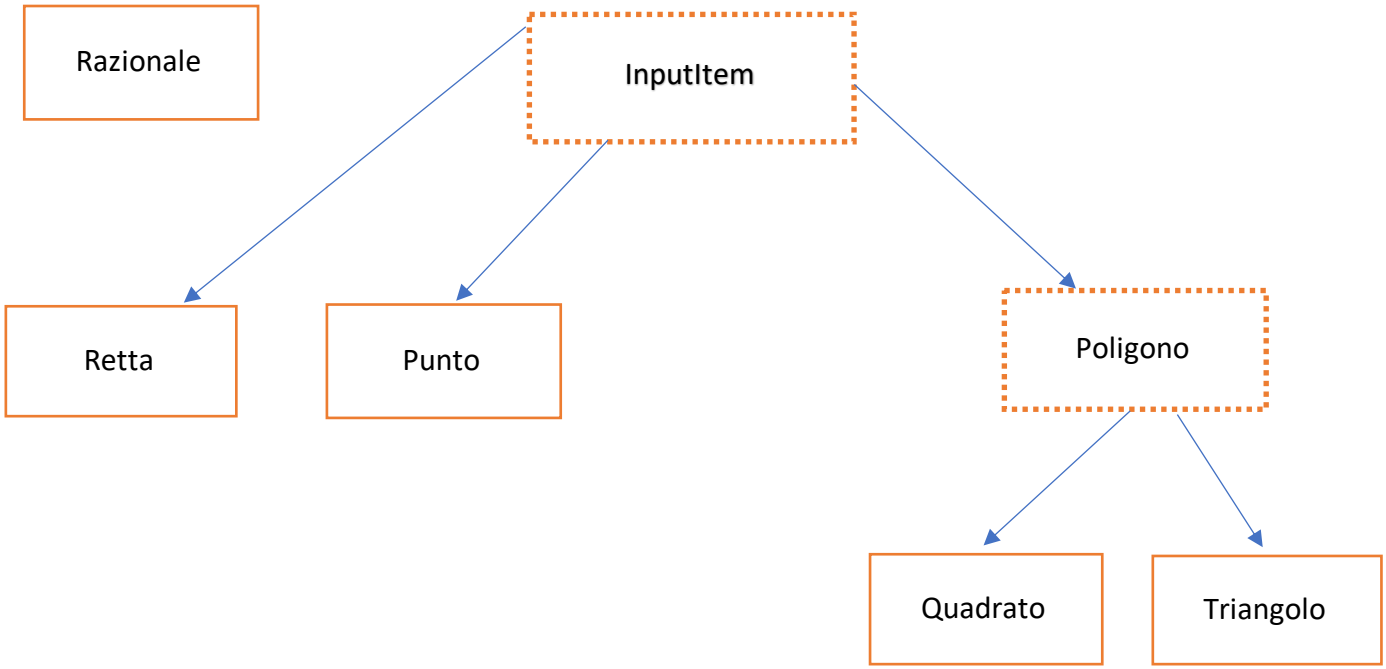
1 Abstract

La calcolatrice effettua operazioni su figure geometriche, precisamente su punti, rette e poligoni fino a quattro lati. Può calcolare la distanza e l'intersezione fra tutti i tipi, calcolare l'area e il perimetro dei poligoni, la retta passante per due punti, e, dati un punto P e una retta R può calcolare la retta passante per P e parallela o perpendicolare a R.
La gerarchia è stata pensata e costruita in modo tale che chi volesse, in seguito, può ampliarla aggiungendo il proprio tipo nel modello, a patto che implementi operazioni adeguate per il proprio tipo se non dovessero bastare quelle già presenti. Inoltre, con pochissime linee di codice aggiuntive è possibile aggiungere nella barra laterale di sinistra nuove azioni sul nuovo tipo di dato.

2 Descrizione/uso gerarchia e codice polimorfo

MODEL

La gerarchia utilizzata nel progetto rappresenta, come spiegato nell'Abstract, più figure geometriche "dominate" da una classe padre Inputitem.



INPUTITEM :

È la classe base astratta che ha come unico scopo dare un'identità iniziale all'input inserito dall'utente. La classe non ha campi dati ma solo la definizione e implementazione di due metodi statici *pars_start()* e *iniz_input()* e la definizione di due metodi virtuali puri, *intersect()* e *distance().*
Le prime due sono statiche per il semplice motivo che vengono invocate quando l'oggetto deve ancora essere creato.
La funzione pars_start() ha il compito di identificare il tipo di figura in input, svolge semplicemente un'operazione di parsing preliminare sulla stringa inserita. Il tipo di ritorno può essere un punto, una retta, o 0. Pars_start() viene invocato all'interno di iniz_input() dove, in base al suo tipo di ritorno decide se invocare il parser di punto , retta o poligono e quindi creando effettivamente l'oggetto che verrà ritornato al chiamante di iniz_input().
I due metodi virtuali puri sono implementati all'interno di punto, retta e poligono.
Il distruttore della classe è virtuale per permettere la distruzione di oggetti in presenza di puntatori polimorfi.

PUNTO :

La classe *punto* è derivata pubblicamente da *inputitem*. Contiene due campi di tipo razionale che rappresentano la coordinata x e la coordinata y del punto.
Il metodo *pars_point*() viene invocato da *iniz_input*() e passandogli una stringa contenente il punto inserito dall'utente, riconosce i valori della coordinata x e y. Al suo interno solleva eventuali eccezioni per questo nel metodo di invocazione è invocata in un blocco try per un corretto funzionamento.
La classe *punto* è concreta, poiché implementa i metodi virtuali puri della classe base *inputitem, intersect*() e *distance*(). Intersect() ha il compito di verificare se un punto (oggetto di invocazione), e qualsiasi altro oggetto passato con un puntatore polimorfo di tipo statico *inputitem,* si intersecano: se il puntatore passato ha come tipo dinamico *punto,* allora viene invocata la funzione *distanceTwoPoints()* tra i due punti e se ritorna 0 significa che si intersecano (ovvero sono uguali), quindi ritorna un vector contenente il punto passato. Se il tipo dinamico è diverso da punto allora viene invocata nuovamente *intersect()* ma con oggetto di invocazione l'oggetto puntato dal puntatore passato alla funzione cosi da invocare metodi opportuni implementati in quella classe. *Distance*() funziona con la stessa idea di *intersect*(), se il puntatore polimorfo ha tipo dinamico *punto* ritorna il risultato di *distanceTwoPoints()* al chiamante, altrimenti rilancia l'invocazione *distance()* con oggetto di invocazione l'oggetto a cui punta il puntatore e come parametro *this*.
Il distruttore della classe è virtuale per permettere la distruzione di oggetti in presenza di puntatori polimorfi.

RETTA:

La classe retta è derivata pubblicamente da inputitem. Contiene tre campi dati privati di tipo razionale: a, b, c, che rappresentano i coefficienti di una equazione lineare in forma implicita del tipo: ax + by + c = 0.
È presente un metodo *pars_rect*() che, passatogli una stringa contenente la retta, modifica la retta di invocazione riconoscendo i coefficienti e assegnandoli all'oggetto di invocazione.
Particolare attenzione è rivolta ai metodi *intersect*() e *distance*(), entrambi come nella classe *punto* vengono invocati nel momento in cui è necessario sapere l'intersezione o la distanza tra una retta e un qualsiasi altro oggetto nel piano cartesiano. Le due funzioni si comportano alla stessa maniera, se il puntatore passato ha come tipo dinamico *punto* o *retta* sfrutta le funzioni definite nella classe *retta* (*distanceRettaRetta(), Intersect_rette(), distancePuntoRetta()*), altrimenti se è un poligono rilancia la chiamata di funzione invertendo l'oggetto di invocazione e il puntatore passato per valore che avrà come tipo dinamico *retta*.
NB: le funzioni non richiamate all'esterno della classe sono state opportunamente dichiarate private.
Il distruttore della classe è virtuale per permettere la distruzione di oggetti in presenza di puntatori polimorfi.

POLIGONO:

La classe poligono è una classe astratta derivata pubblicamente da *inputitem*. È rispettivamente classe padre di *quadrato* e *triangolo*. Ha due attributi: uno privato di tipo int che contiene il numero dei lati del poligono, e un vector che contiene puntatori a punti che forma il poligono marcato protected. Il vector è protected perché deve essere accessibile alle sottoclassi. È stato scelto un vector poiché necessitavo di un accesso di tipo randomico ai punti.

La classe poligono è di tipo astratto perché contiene il metodo virtuale puro *getfisso*(), questo metodo è implementato nelle sottoclassi *quadrato* e *triangolo* e ritorna il numero fisso del poligono in considerazione (ogni poligono regolare ha un numero fisso definito per facilitare i calcoli su aree, lati, perimetro, ecc). *Poligono* è stata implementata per fare calcoli su ogni tipo di poligono regolare e sui triangoli di ogni tipo, per motivi progettuali e tempistici ci siamo soffermati solo su *quadrato* e *triangolo*.
Sono definite area(), perimetro(), lato() come metodi virtuali, questo perché esiste un metodo generale di calcolo dell'area, perimetro e del lato, (e questo metodo è stato implementato nella classe poligono) ma abbiamo preferito implementare dei metodi più "smart" e di complessità minore in *quadrato* e *triangolo.*
Sono implementate le funzioni distance() e intersect() della classe base (sono state implementate in *poligono* perché il loro comportamento è assolutamente lo stesso con tutti i poligoni). Le due funzioni hanno presso che lo stesso comportamento: in base al tipo dinamico del puntatore passato alla funzione, verrà invocata la funzione che calcolerà la distanza o l'intersezione con il relativo poligono di invocazione.
NB: l'intersezione viene verificata tramite la size() del vector ritornato, se size() == 0 allora le due figure non si intersecano, se size() > 0 allora le due figure si intersecano nei punti contenuti nel vector.
Il distruttore della classe è virtuale per permettere la distruzione di oggetti in presenza di puntatori polimorfi.
Il metodo *pars_pol*() data una stringa riconosce i punti del poligono e lo ritorna, al suo interno viene invocato iterativamente per ogni punto pars_point().

QUADRATO:
La classe quadrato è una classe derivata di poligono. Ha come unico campo dati statico *numerofisso* privato di tipo int. Gli unici metodi implementati sono la funzione che ritorna il numerofisso, e la funzione che calcola l'area (definita virtuale in poligono e overridata in quadrato).

TRIANGOLO:
La classe triangolo è una classe derivata di poligono. Ha come unico campo dati statico *numerofisso* privato di tipo int. Gli unici metodi implementati sono la funzione che ritorna il numerofisso, la funzione che calcola l'area, il perimetro e il lato (definite virtuali in poligono e overridate in triangolo).
NB: se il triangolo non è equilatero, la funzione lato() ritorna 0; poiché è una funzione utilizzata all'interno di altre non crea problemi.

ECCEZIONI:
È presente una gerarchia che gestisce le eccezioni per un corretto funzionamento della calcolatrice.

OSSERVAZIONE: Nella gerarchia tutti i metodi non richiamati all'esterno di essa sono stati marcati private o protected a seconda delle necessità.
All'interno della classe poligono sono presenti calcoli con complessità non banale, questo per il semplice motivo che non sono stati utilizzati algoritmi o librerie esistenti per il calcolo di intersezioni, distanze, ecc. Sono comunque presenti brevi commenti che descrivono gli algoritmi più complessi.

3 Manuale utente

Non è necessario il manuale. All'apertura di Kalk verrà visualizzato un wizard che aiuterà l'utente nel prendere dimestichezza con l'interfaccia e nell'inserire in maniera corretta l'input desiderato.

4 Analisi delle Tempistiche

Verranno specificate le tempistiche soggette alla mia parte. La soglia di ore disponibili è stata leggermente sforata. Su certi punti verrà giustificato l'ammontare di ore.

- analisi preliminare del problema(~05h);
- progettazione modello (~07h);
  La progettazione del modello svolta assieme al mio compagno
- sviluppo metodi senza codice (~10h);
  lo sviluppo dei metodi prima dell'implementazione.
- codifica modello (~23h)
  La codifica del modello ha occupato un tempo importante per gestire i calcoli in un piano cartesiano, come si può notare tutti i metodi sono stati implementati manualmente.
- debugging(~07h);
  Problemi con la gestione del garbage.
- testing(~05h).

5 Suddivisione del lavoro

L'applicazione è stata progettata e discussa in stretto contatto da entrambi i partecipanti, a partire dall'analisi del problema fino alla sua implementazione.

Il mio ruolo è stato lo sviluppo del modello c++ e la sua traduzione in Java, la gestione dei dati del modello e lo sviluppo di ogni funzione, quindi, sono di mia responsabilità.

La fase di test e di debugging è stata svolta assieme, soprattutto la gestione del garbage.

6 Ambiente di sviluppo e di test

A casa:

- Sistema operativo: Windows 10 Home 64-bit
- Compilatore: mingw492_32
- Libreria Qt: 5.5.1

In laboratorio:

- Sistema operativo: Ubuntu 16.04 64-bit
- Compilatore: gcc 5.4.0
- Libreria Qt: 5.5.1

7 Comandi per la compilazione ed esecuzione

Il progetto viene presentato con una cartella col progetto in C++ ed una col progetto in Java:

- **C++** Spostarsi col terminale nella cartella apposita (utilizzare il file .pro incluso) e dare i comandi:
    - qmake
    - make
    - ./Kalk

- **Java** Spostarsi col terminale nella cartella apposita e dare i comandi:

    1. javac use.java
    2. java use