Progetto di Programmazione ad Oggetti - Relazione

Andrea Coletti - matricola 1096089

2017/2018

Contenuti

1 Note generali

L' applicazione è una calcolatrice tra alberi. L' applicazione è stata sviluppata tenendo a mente il design pattern MVC. Lo sviluppo ha fatto uso del IDE Qt Creator. Le ore impiegate per la creazione dell'applicazione sono rendicontate e non cronometrate. E' incluso un file progetto.pro necessario per la corretta compilazione ed esecuzione del programma. Note Java: La classe Use.java stampa i risultati su un file "Use.txt". Per stampare gli alberi , indica i nodi secondo la seguente convenzione :
nodo sinistro = l (left)
nodo destro = r (right)
nodo centrale = m (middle)

Specifiche macchina di sviluppo :

versione QMake : 3.0
versione Qt : version 5.5.1
versione gcc : version 5.4.0
sistema operativo : GNU/Linux

2 Metodo di lavoro

Lo sviluppo dell' applicazione ha seguito le seguenti fasi : creazione del modello , creazione della view , creazione del controller. In ogni fase sono state svolte le seguenti attività , ordinate , in modo ciclico: progettazione , sviluppo/codifica , test. L' attività di progettazione concerne : l' apprendimento delle librerie Qt impiegate , le scelte fatte e il loro perchè(esempio: uso di un metodo , membri delle classi , interazione tra classi...). L' attività di sviluppo/codifica riguarda la creazione e implementazione degli algoritmi usati , la scrittura effettiva delle classi. L'attività di test prevede : instanziazione di oggetti e invocazione metodi , l' uso degli strumenti di Qt Creator per cercare eventuali problemi(debugger e analyzer). Tempo impiegato:
Prima fase : progettazione = 3 ; sviluppo = 3 ; test = 2; totale = 8 ;
Seconda fase : progettazione = 7 ; sviluppo = 6 ; test = 3; totale = 16 ;
Terza fase : progettazione = 10 ; sviluppo = 8 ; test = 5; totale = 23 ;
Implementazione Java : 3 ;
Totale = 50 ore ;

3 Descrizione classi

Modello

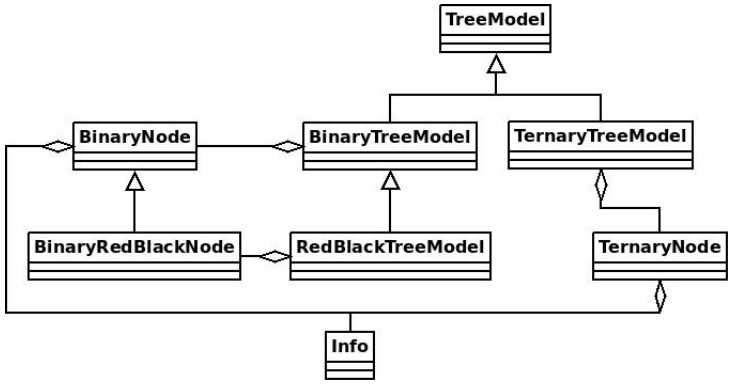

Figura 1: gerarchia

Le classi nodo mettono a disposizione dei metodi per ottenere i puntatori ai figli e al padre. Questi metodi sono virtuali nelle classi da cui si vuole far ereditare(BinaryNode , TernaryNode). Quindi le sottoclassi di BinaryNode e TernaryNode possono fare l' overriding delle funzioni e cambiare il tipo di ritorno al loro tipo specifico. Stesso discorso è valido per i metodi Sum(const TreeModel) e Intersection(const TreeModel) e insertValues(QVector). Il metodo insertValues(QVector) è un metodo d'utilità pensato per inserire i

valori in un albero. Tal metodo lavora sotto le seguenti assunzioni: il vettore in input è non vuoto , il primo elemento è il valore della radice , l'albero da riempire è completo. l'algoritmo di riempimento lavora con ordine prefisso. per ogni nodo 'i' list[i] è il nodo sinistro , list[i+1] è il nodo destro , list[i+2] è il nodo centrale(a seconda del tipo di albero). Le operazioni sugli alberi sono : "SumValues" = somma di tutti i valori di tutti i nodi contenuti nell'albero; assume che sia definto l'operatore di somma per la classe a cui il template è istanziato. "Sum" = somma tra due alberi alberi; inserisce l'albero passato come paramento nell'albero d'invocazione. "Intersection" = intersezione tra due alberi; trova i nodi comuni e con quelli crea un nuovo albero. "SumMiddle" = disponibile solo per alberi ternari; somma i valori di tutti i nodi "middle" dell'albero d'invocazione. "BlackHeight" = disponibile solo alberi rosso-neri. Ottiene l'altezza Nera dell'albero , se l'altezza rosso-nera è uguale per ogni cammino da ogni nodo verso le foglie torna il valore dell'altezza rosso-nera altrimenti torna "-1". L' altezza rosso-nera è il numero di nodi neri su un cammino da un nodo alla foglia. Per gli alberi binari è disponibile un metodo "Rotation()" che ritorna una copia dell'albero d'invocazione coi nodi scambiati. Questo metodo è virtuale così da permettere alle sottoclassi di ridefinirlo e tornare il loro tipo specifico. "Rotation()" si avvale dell' aiuto di più metodi tra cui "RotationHelp()". "RotationHelp()" crea una copia della radice e lascia la responsabilità di ruotare i sottoalberi sinistro e destro a una funzione ausiliaria. Infine ritorna un albero binario , le sottoclassi tornano il loro tipo specifico.

CreateTreeArea

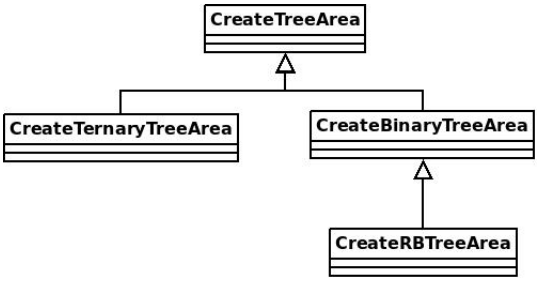

Figura 2: gerarchia

Tale gerarchia rappresenta la view dell'albero che accetta gli input dall' utente. Le sottoclassi decidono come creare effettivamente l'albero e creano i nodi tramite il metodo "createNode" che prende come parametri il testo del nodo da creare e il padre del nodo. Il metodo è virtuale in modo tale da permettere alle sottoclassi di ridefinirlo se necessario.

ResultTree

L'obbiettivo di questo insieme di classi è : ricevere un modello in input e mostrare una view corrispondente all' utente. Tale view deve avere la stessa struttura e gli stessi valori del modello ricevuto in input. Come viene

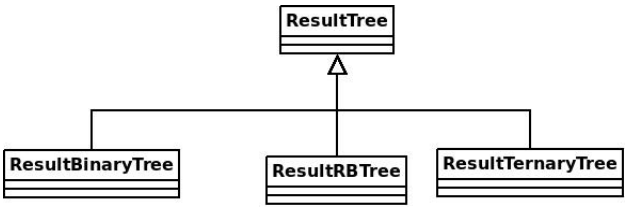

Figura 3: gerarchia

effettivamente costruita la view del risultato: nella classe base è presente un metodo virtuale astratto pubblico buildResultTree() che viene ridefinito nelle sottoclassi. Le sottoclassi rappresentano le view di uno specifico tipo di albero(binario,ternario...). Quindi le sottoclassi concrete sono composte con un puntatore costante ad un modello che , a sua volta , è un albero dello stesso tipo che si vuole rappresentare con la view. Quindi la view viene effettivamente istanziata solo quando viene invocata buildResultTree() che , grazie al polimorfismo , invoca la versione del tipo specifico. I metodi "createResultSingleNode" , "createResultNode" , "createRootResultNode" sono virtuali così da concedere alle sottoclassi la possibilità di ridefinirli se necessario.

Operation

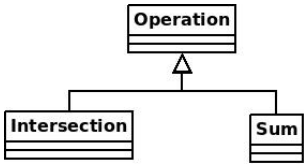

Figura 4: gerarchia

Questa gerarchia è stata realizzata per implementare in modo efficiente le operazioni binarie tra alberi. Operation è un operazione generica mentre le sottoclassi sono operazioni specifiche(Sum , Intersection). La classe ha un metodo virtuale astratto execute(TreeModel,TreeModel) ridefinito nelle sottoclassi. Ogni sottoclasse rappresenta un' unica operazione. La sottoclasse invoca l'operazione sui modelli , passati come argomenti in execute(TreeModel,TreeModel).

4 Manuale utente

Per spiegare la calcolatrice è possibile dividerla in aree colorate(vedasi immagine sottostante). L'area riquadrata in rosso serve per creare gli alberi. Si possono creare tre tipi : binari , ternari , binari-Rosso-neri. E' possibile creare solo alberi completi. I nodi contengono interi compresi tra 0 e 10000.

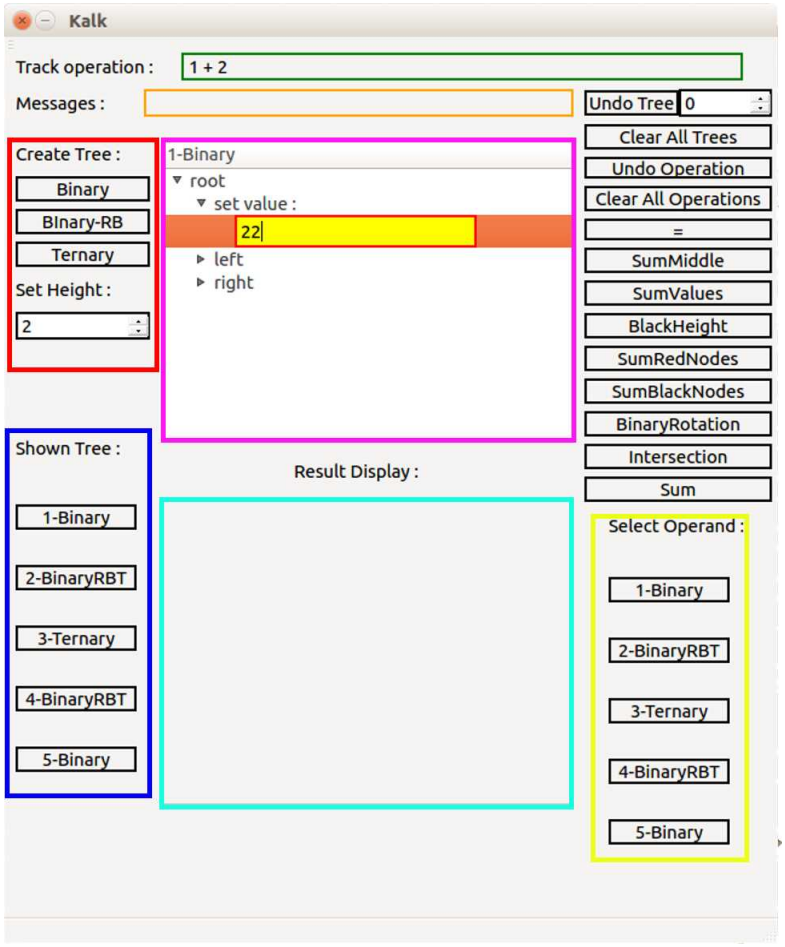

Figura 5: vista generica

Per creare un albero bisogna specificarne l'altezza(massimo 4). E' possibile creare e mantenere contemporaneamente fino a 5 alberi. Per gli alberi rosso-neri vale che: la radice è nera , ogni nodo è rosso o nero , le foglie sono nere. E' possibile decidere quale albero eliminare tramite il bottone "Undo Tree" indicando l'indice nella spinBox a lato(0 indica nessuno albero). Se si cerca di crearne altri viene stampato un messaggio d'errore. L'area riquadrata in viola mostra l'albero appena creato. In alto viene stampato l'indice dell'albero e il suo tipo nella forma : "numero-tipoAlbero". L' indice dell' albero viene usato come riferimento per eseguire le operazioni. Le operazioni sono di due tipi: unarie , n-arie. Le operazioni unarie vengono eseguite sull' albero correntemente mostrato nell'area riquadrata in viola. Le operazioni unarie sono: BinaryRotation = ruota i valori dell'albero(solo alberi binari con

sottoalbero destro).
SumValues = somma dei valori dell'albero.
SumMiddle = somma di tutti i nodi "middle" dell'albero(solo alberi alberi ternari).
SumRedNodes = somma valori dei nodi con colore rosso.(solo alberi rosso-neri).
SumBlackNodes = somma valori dei nodi con colore nero.(solo alberi rosso-neri).
BlackHeight = ottiene l'altezza rosso-nera dell'albero(solo alberi rosso-neri). L'altezza rosso-nera è il numero di nodi neri per ogni cammino dalla radice fino alle foglie. Le operazioni n-arie sono: Sum = dati due alberi T1 e T2 , inserisce T2 dentro T1. Intersection = dati due alberi T1 e T2 , trova i nodi comuni e con quelli crea un nuovo albero T3. Se T3 è vuoto mostra un messaggio nell' area arancione. L'area riquadrata in giallo contiene dei pulsanti con gli stessi nome dei bottoni nell'area blu. Digitando un bottono viene mostrato il corrispondente indice numerico nell' riquadro verde in alto("Track operation"). Tali bottoni servono per decidere gli operandi con cui eseguire le operazioni n-arie. Quindi solo "Sum" e "Intersection" generano un simbolo , rispettivamente "+" e "I" , nel riquadro "Track operation". Una volta terminata l'espressione desiderata (esempio : 1 + 2 I 3) bisogna digitare il pulsante "=" per portare a termine il calcolo. Se il calcolo è andato a buon fine viene mostrato il risultato nell' area azzurra(tutti i risultati sono qui mostrati). Altrimenti se si è verificato un errore viene mostrato un messaggio nel riquadro "Messages". Per far si che un calcolo esegua , l'espressione nell' area "Track operation" deve obbedire ai seguenti vincoli: alternanza tra operando operatore , il primo e l'ultimo elemento dell' espressione deve essere un operando , non devono esserci alberi binari e ternari nella stessa espressione. L'area riquadrata di blu contiene dei pulsanti col nome del operando. Digitando tali pulsanti è possibile vedere l'albero con tale indice. In alto ci sono due aree: una di bordo verde e una di bordo arancione. L'area verde mostra le operazione n-arie attualmente eseguite dall'utente. Nell' area arancione vengono mostrati i messaggi all'utente. I messaggi possibili sono: "Malformed expression" = espressione malformata "Can't Sum incompatible trees" = impossibile sommare ternari e binari "Can't Intersect incompatible trees" = impossibile intersecare binari e ternari "SumMiddle works only on Ternary trees" = impossibile eseguire l'operazione su alberi che non sono ternari. "BlackHeight works only on RedBlack trees" = impossibile ottenere altezza rosso-nera su alberi che non sono rosso-neri "Empty Intersection" = intersezione tra alberi vuota. L'espressione precendete ad un risultato viene mostrata nell'area cerchiata in rosso("Result expression").

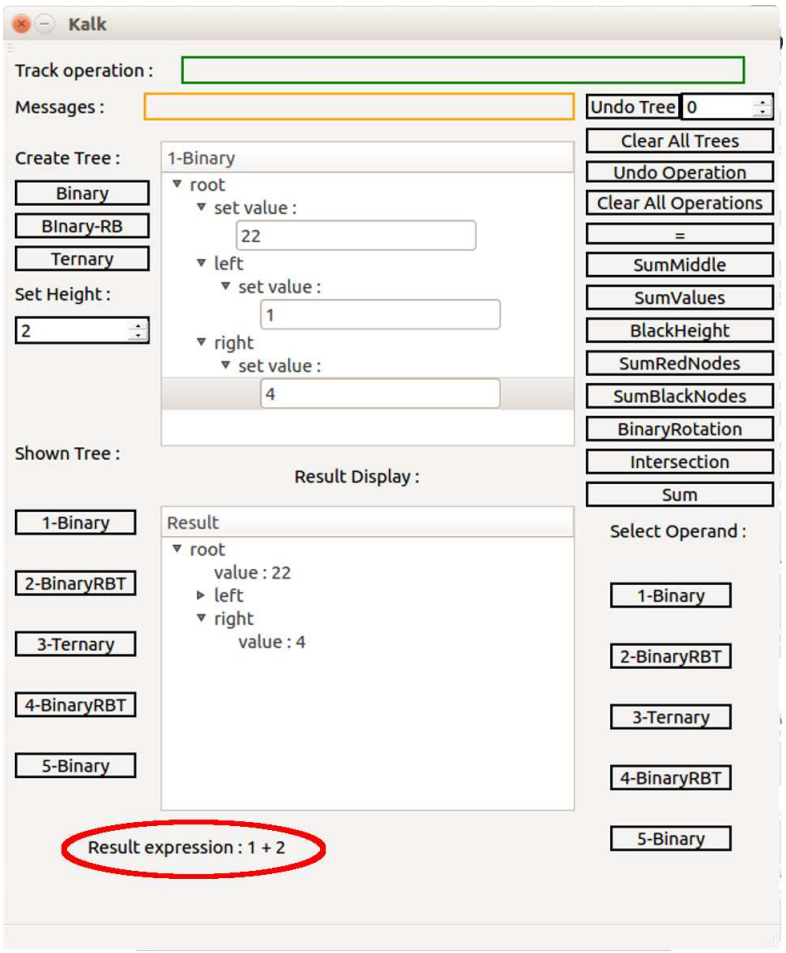

Figura 6: esempio risultato

5 Problemi e limitazioni note

1)Un grosso limite è sicuramente la mancanza di un' interfaccia comune Node , possibilmente classe astratta , che funga da interfaccia comune alle classi Nodo.
2)La view degli alberi , in cui l' utente inserisce i valori , viene creata chiedendo solo l'altezza in input. Quindi crea alberi completi e non fornisce all'utente la possibilità di creare alberi incompleti decidendo i nodi da effettivamente instanziare. Ciò è stato fatto per limiti di tempo e complessità. In fase di progettazione è stato riscontrato che non era possibile ottenere gli effetti sopra elencati senza sforare il limite delle 50 ore complessive per la portata a termine del progetto.
3)La classe che gestisce l' eccezioni è unica e non una gerarchia di classi. Non è

stata fatta una gerarchia perchè il progetto ha comumque dimensioni limitate e quindi non era necessario , tuttavia nel caso i messaggi d' errore aumentassero sarebbe meglio diversificarli a seconda del tipo(messaggio di log di sistema , messaggio relativo a un tipo specifico , errore utente...).
4)La classe MainWindow è sicuramente troppo grande , ha troppe resposabilità e sarebbe meglio spezzarla in classi più piccole. Questo non è stato fatto per mancanza di tempo e per evitare di dividere la classe in modo insensato o creare eccessive e insensate dipendenze tra classi.
5)Nella classe MainWindow è presente un membro Controller con parametro di template T=int. In questo modo si lega il controller allo specifico tipo int diminuendo la flessibilità del codice. L' idea originale era quella di creare un sistema che chiede all'utente quale tipo vuole istanziare tra delle alternative proposte (int , double , string ,regex ,...) , creare un QValidator per controllare che inserisca effettivamente ciò che vuole , e poi instanziare un controller con T = "tipo in input dall'utente". Ciò non è stato implementato per mancanza di tempo e per la complessità aggiuntiva nei calcoli. Difatti avendo più tipi diversi si dovrebbe prestare attenzione ad operazioni tra alberi contenti nodi con valori di tipo diverso. Una possibile soluzione al problema è la seguente: fornire un sistema all'utente per indicare il tipo che desidera avere nell'albero , ad esempio tramite delle checkbox che indicano il tipo da inserire. In MainWindow inserire un controller , per ogni tipo diverso , inizialmente vuoti per poi istanziare solamente i controller effettivamente usati. Un' altra possibile soluzione è quella di spezzare il controller in sottoclassi creando una gerarchia che gestisca meglio il problema e il calcolo stesso delle operazioni. In effetti la classe Controller è troppo grande con troppe responsabilità e sarebbe meglio suddividerla , non è stato fatto perchè avrebbe richiesto troppo tempo o comumque si sarebbe sforato il limite delle 50 ore.
6)Alcuni metodi e certi costruttori prendono troppi parametri in input(tre o quattro) , esempio i costruttori delle sottoclassi di CreateTreeArea. Ciò è sicuramente negativo per la mantenibilità del codice e per i programmatori che devono usare tali metodi e costruttori dato che devono obbedire ad una particolare segnatura troppo lunga e difficile da ricordare.
7)La classe ExceptionHandler non eredita da classi già definite per gestire le eccezioni(esempio : std::exception).