Giorgio Ubbriaco – matricola:209899 – 2° anno Ingegneria Informatica formativo

Progetto – Sudoku

Il progetto è composto da quattro file:

* Backtracking
* AppSudoku
* Cella
* Sudoku

Il primo file è collocato nel package poo.backtracking sviluppato durante le lezioni dal professore Nigro.

I rimanenti tre file si trovano, invece, all’interno del package progetto.sudoku.

Il file Sudoku estende la classe Backtracking avente tipi Cella e Integer. Tale classe ha diverse variabili di istanza: le costanti intere statiche (dichiarate come protected) N e INIT per un migliore “clean code” dove la prima indica il numero di righe e colonne, poiché la matrice che verrà inizializzata sarà una matrice quadrata NxN, mentre la seconda indica il valore a cui sarà inizializzato ogni elemento della matrice quadrata m. Inoltre, è presente anche la variabile statica intera NUMERO\_SOLUZIONI utilizzata per “dichiarare” il numero di soluzioni ottenute da “rappresentare” all’interno della GUI. Inoltre, la variabile statica intera SOLUZIONI\_MASSIME, inizializzata a 11 come default, utilizzata come vincolo, cioè indica il numero massimo di soluzioni da poter rappresentare (in questa maniera se un determinato sudoku ammette 20 soluzioni, quelle rappresentate saranno sempre le prime 11). È presente anche la collezione listaPuntiScelta inizializzata come ArrayList<Cella>() che servirà per una migliore gestione dei punti di scelta rimanenti dopo i vari tentativi del Sudoku. Infine, troviamo la matrice m che conterrà le possibili scelte fatte ad inizio “gioco” e i valori impostati per ogni tentativo di risoluzione fatto. Il costruttore di default inizializzerà la m come matrice quadrata di N righe ed N colonne dove ogni elemento m[i][j] conterrà il valore iniziale INIT. Inoltre, nel doppio ciclo di for (di inizializzazione della matrice), aggiungiamo alla lista listaPuntiScelta una nuova cella avente indici i e j (cioè gli stessi dell’elemento impostato subito prima nella matrice) per indicarla come cella “disponibile” alla quale potrà essere impostato un valore dall’utente o che potrà “ospitare” il valore ottenuto dal tentativo corrente. Il metodo pubblico puntiDiScelta() ritorna una nuova collezione di celle, cioè di punti di scelta. Il metodo scelte(Cella ps) ritorna una collezione delle possibili scelte che possono essere attribuite ad ogni Cella e, pertanto, trattandosi del gioco Sudoku sarà composta dai valori compresi tra 1 e 9 inclusi. Il metodo scriviSoluzione(Cella ps), in questo caso aggiorna il numero di soluzioni, incrementandolo (essendo che si sta scrivendo una nuova soluzione (nella classe Backtracking, il metodo tentativo)), ed, inoltre, aggiungiamo la soluzione i-esima considerata e salvata nella matrice m tramite il metodo aggiungiSoluzione della inner class AppSudoku.FinestraSudoku. Se volessimo semplicemente stampare le soluzioni tramite un main nella stessa classe Sudoku, si dovrebbe eliminare la riga di codice dove aggiungiamo la matrice tramite il metodo aggiungiSoluzione(int[][] m) e aggiungere una stampa del numero della soluzione corrente ed un doppio ciclo di for (con ovviamente un “ritorno a capo” \n per ogni volta che termina il secondo for). Tale metodo è stato implementato nella classe concreta Sudoku poiché dichiarato astratto nella classe astratta Backtracking. Il metodo imposta colloca alla riga “row” e alla colonna “col” il valore “valore” passati come parametri al metodo. In tale metodo viene fatto ovviamente un controllo sul valore passato il quale se è minore o uguale a 0 si solleva un’eccezione unckecked. Successivamente viene creata una nuova cella la quale verrà assegnata alla corrispondente riga e colonna della matrice e ovviamente rimossa dalla listaPuntiScelta poiché non più disponibile, dopo aver verificato se tale punto di scelta è assegnabile o meno tramite il metodo assegnabile (Cella ps, int valore). Il metodo “assegnabile” verifica inizialmente che tale cella non sia contenuta nella collezione listaPuntiScelta così da poter effettuare un nuovo tentativo passando al metodo tentativo la collezione “listaPuntiScelta” e la cella di posizione successiva a quella considerata precedentemente. Successivamente, si verifica che tale punto di scelta considerato sia assegnabile o meno. Una cella è assegnabile se il valore di essa non si trova già sulla stessa riga, colonna e sotto-matrice 3x3 (ricordando che le sotto-matrici possibili sono quelle che hanno la cella più in alto a sinistra con indici pari a <0,0> , <0,3> , <0,6> , <3,0> , <3,3> , <3,6> , <6,0> , <6,3> , <6,6>). Ovviamente queste verifiche vengono effettuate tramite appositi metodi boolean denominati come stessaRiga(Cella ps, Integer scelta), stessaColonna(Cella ps, Integer scelta) e stessaSottoGriglia(Cella ps, Integer scelta). I metodi esisteSoluzione ed ultimaSoluzione vengono re-implementati nella classe concreta così da poter avere un corretto funzionamento. I metodi “assegna” e “deassegna” collocano nella cella passata come parametro rispettivamente la scelta passata come parametro e il valore zero. Infine, il metodo risolvi() richiama il metodo “tentativo” della classe Backtracking, il quale in maniera ricorsiva calcolerà le n soluzioni, passando come parametro la collezione listaPuntiScelta e la cella in posizione (0,0).

La classe Cella è composta da due variabili di istanza (la riga i-esima e la colonna j-esima), da un costruttore al quale passiamo due parametri (la riga e la colonna) i quali, prima di essere assegnati rispettivamente alla riga e alla colonna della cella, vengono “controllati” affinché essi non siano valori “non ammessi”. Sono presenti anche due getter (getI e getJ) ed i soliti metodi (toString, equals e hashCode) che fanno di essa (la classe) una classe “robusta”.

La classe AppSudoku contiene un main nel quale viene creato un nuovo JFrame tramite la inner class FinestraSudoku che estende infatti la classe JFrame. Tale finestra inizialmente presenta all’utente la possibilità di aggiungere delle possibili scelte da poter inserire all’interno delle matrici che poi successivamente conterranno le soluzioni del Sudoku. Infatti, l’utente troverà nel BorderLayout.NORTH un pannello contenente tre caselle (tre JTextField) identificati ognuno da un ToolTip così che l’utente possa capire di cosa si tratti semplicemente portando il puntatore sulla casella di cui vuole “informazioni”. Affianco a queste caselle troviamo un JButton che permetterà di aggiungere ogni scelta, se tale scelta ovviamente rispetterà i vincoli delle righe, delle colonne e del valore inseriti. Tanto è vero che se le caselle risultano essere vuote o contenente valori non ammessi (ad esempio una riga pari a 11 o a -1 o per assurdo ad una lettera (controllo effettuato tramite regex)), l’utente visualizzerà una finestra-errore (la classe ControllaValori che estende la classe JFrame) la quale segnalerà all’utente di controllare meglio i valori inseriti. Per evitare ulteriori errori, ogni volta che l’utente sbaglierà ad inserire i valori, oltre a visualizzare la finestra di avvertimento, verranno “azzerate” le caselle così da permettere all’utente il re-inserimento dei valori all’interno di esse. Se, invece, la riga, la colonna e il valore inserito rispettano la regex e risultano essere “idonei” da un punto di vista numerico, verrà creato un nuovo oggetto Oggetto (con i rispettivi campi riga i-esima, colonna j-esima, valore val) e una nuova JCheckBox che verrà inserita all’interno del Box presente nel pannello centrale. Se tale JCheckBox è gia presente (controllo effettuato tramite il metodo giaPresente(JCheckBox) che itera sulla collezione contenente tutte le JCheckBox presente fino a quel momento) allora l’utente visualizzerà una finestra-errore (la classe GiaPresente che estende la classe JFrame) la quale segnalerà all’utente di controllare meglio la scelta che si sta inserendo (ovviamente come spiegato precedentemente, le caselle verranno “azzerate”). I metodi aggiungiScelta(String s) e creaOggetto(String s) sono utili per migliore gestione delle scelte fatte dall’utente inizialmente. Rispettivamente, il primo metodo permette di aggiungere la scelta al Box di JCheckBox, mentre il secondo permette di creare, con l’ausilio di uno StringTokenizer, un nuovo oggetto Oggetto partendo da una stringa, passata come parametro al metodo. Nel BorderLayout.SOUTH troveremo alcuni JButton per una perfetta interazione dell’utente con la finestra. Tra questi troviamo il JButton “carica” che permette all’utente di caricare un vecchio salvataggio di Sudoku presente sul computer. Tutto ciò tramite l’ausilio di un JFileChooser che permetterà all’utente di scegliere il file da caricare e poi successivamente grazie al metodo “ripristina”, il quale inizializzerà un BufferedReader, che permetterà di leggere dal file ogni riga e caricarla nella finestra. Ovviamente in questo caso verranno caricate le eventuali scelte di celle che erano state fatte dall’utente durante la vecchia sessione di gioco. Affianco a quest’ultimo appena citato, troviamo il JButton “salva” che permetterà di salvare una sessione di gioco già salvata precedentemente sul computer e quindi di sovrascrivere tale file. Subito dopo troviamo anche il JButton “salva con nome” il quale tramite l’ausilio di un JFileChooser, di un metodo denominato “salva” nel quale viene inizializzato un PrintWriter proprio per scrivere sul File vuoto, permetterà all’utente di salvare le eventuali scelte di celle fatte all’interno di un nuovo File sul proprio computer. Salvare sul proprio computer una sessione di gioco Sudoku significa salvare le scelte delle celle fatte inizialmente. Inoltre, l’utente potrà visualizzare il JButton “rimuovi scelta” che gli permetterà di eliminare un’eventuale scelta fatta precedentemente (essendo che ogni scelta inserita è una JCheckBox) tramite il metodo privato elimina(). Ovviamente, premere tale JButton senza aver selezionato nessuna JCheckBox farà visualizzare all’utente una finestra-errore (la classe NessunSelezionato che estende la classe JFrame) segnalandogli di dover selezionare almeno una scelta da lui inserita (controllo effettuato tramite il metodo privato nessunSelezionato()). Infine, il JButton “risolvi sudoku” permette di risolvere il sudoku avente le eventuali scelte fatte dall’utente precedentemente. Appena viene premuto tale JButton, la finestra FinestraSudoku viene modificata automaticamente mostrando nella parte centrale la soluzione i-esima del sudoku. Nel BorderLayout.NORTH, invece, troveremo un JLabel contenente l’indice della soluzione che si sta visualizzando che cambia dinamicamente in base se l’utente decida di visualizzare una soluzione precedente o successiva premendo rispettivamente i JButton “precedenteSol” e “successivaSol”. Premendo, ovviamente, uno dei due JButton verranno visualizzate nel pannello centrale la nuova matrice contenente la soluzione i-esima. La matrice i-esima, visualizzata dall’utente, viene salvata in una matrice denominata “daAgg”, ottenuta dall’ArrayList di matrici-JLabel denominato “m\_daMostrare”. Tale ArrayList di matrici-JLabel è stato ottenuto tramite l’aggiunta, ogni qual volta che veniva calcolata una nuova soluzione, della matrice-soluzione i-esima ad un ArrayList di matrici intere denominato soluzioni (tramite il metodo aggiungiSoluzione(int[][] m) il quale, non solo aggiunge la soluzione i-esima alla collezione “soluzioni” ma, trasforma tale matrice i-esima in una matrice di JLabel aggiungendola, inoltre, nella collezione di matrici-JLabel). Ovviamente, ogni matrice-soluzione verrà aggiunta graficamente alla finestra FinestraSudoku tramite la creazione di un GridLayout, impostato sul pannello centrale, che con l’ausilio di un doppio ciclo di for, ci permetterà di aggiungere ogni cella e creando di conseguenza la nostra matrice NxN. Nel BorderLayout.SOUTH l’utente visualizzerà soltanto tre JButton: “salva”, “salvaConNome” e “nuovoSudoku”. L’ultimo JButton appena citato permetterà all’utente di “creare” una nuova sessione di gioco. Pertanto, verrà terminata la finestra corrente e inizializzata una nuova FinestraSudoku. Ovviamente se tale JButton (“nuovoSudoku”) dovesse essere premuto durante una sessione di gioco la quale era stata caricata precedentemente, non comporterebbe l’azzeramento dei progressi salvati nel file tranne se l’utente decida di salvare il nuovo contenuto nel file corrente.