Progetto Ingegneria del Software # 2 - A.A. 2018/2019: FUTOSHIKI

Mattia Gatto

Matricola:182926

III anno Ingegneria Informatica

Il FUTOSHIKI

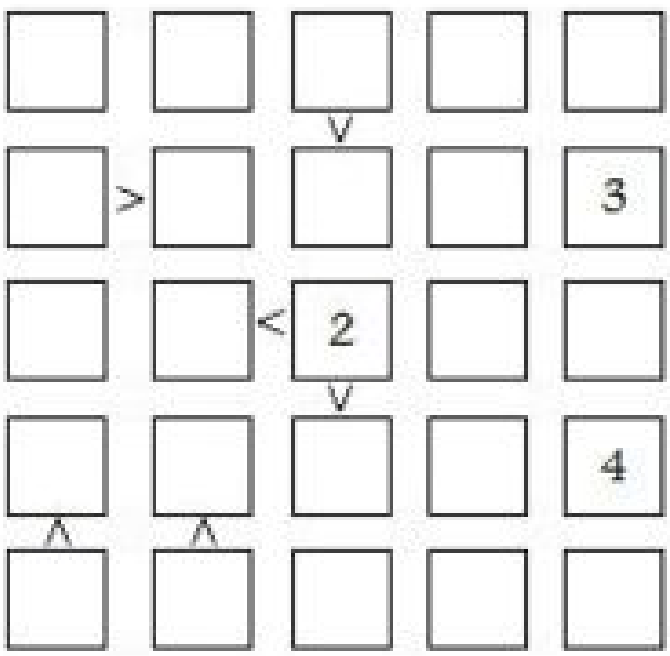
Si tratta di una delle evoluzioni più interessanti del Sudoku e si compone di una griglia contenente solitamente 5 caselle per lato, alcune contenenti un numero, altre separate da un segno > (maggiore di) o < (minore di).

Scopo del gioco è riuscire a completare lo schema, riempiendo tutte le caselle con numeri dall'1 al 5 e rispettando due semplici regole:

• le cifre presenti in due caselle adiacenti unite da un simbolo > o < devono rispettare l'ordine di grandezza che il simbolo rappresenta (se una casella è unita a un'altra da un simbolo >, il numero inserito dovrà essere necessariamente maggiore di quello della casella adiacente e viceversa nel caso del simbolo opposto);

• un numero non può apparire due volte nella stessa riga o colonna.

La difficoltà di uno schema dipende da due fattori: la dimensione dello schema e la quantità di numeri e simboli preinseriti.



Risolvere il gioco progettando e sviluppando un'applicazione Java basata su template method e la tecnica backtracking, realizzando una classe erede di Problema<P,S> specializzata al FUTOSHIKI. Prevedere una GUI con la quale si possa configurare il gioco (porre numeri nelle caselle, stabilire relazioni di precedenza tra caselle contigue etc.), definire il numero massimo desiderato di soluzioni, lanciare l'applicazione e, infine, navigare (con tasti tipo next/previous) sullo spazio delle soluzioni trovate, mostrando a video le varie soluzioni.

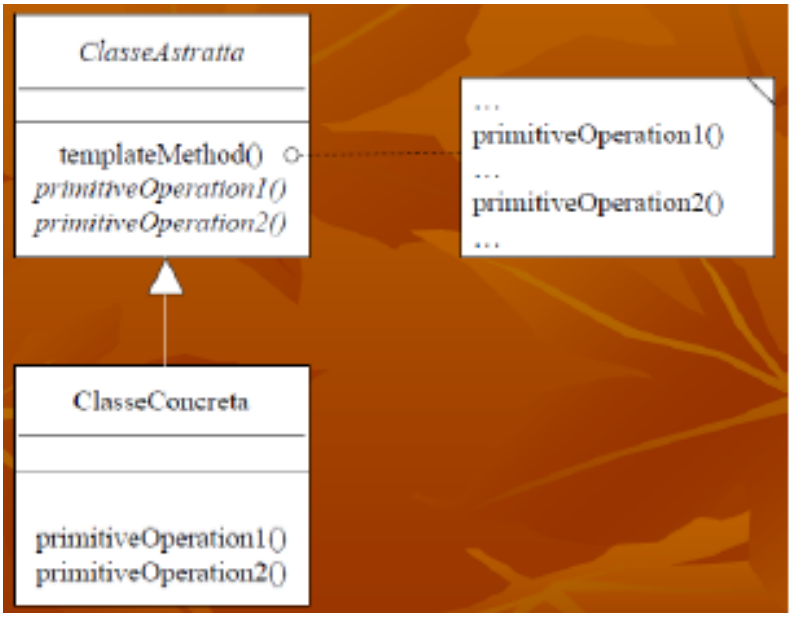
Nello sviluppo del progetto si devono utilizzare i Design Pattern ritenuti più adeguati motivandone opportunamente la scelta. le fasi del processo di sviluppo devono essere documentate ricorrendo, ove necessario, all'uso di diagrammi UML.

Si richiede inoltre di effettuare il testing di uno o più moduli significativi impiegando un opportuno criterio e sfruttando le funzionalità offerte dal framework JUnit.

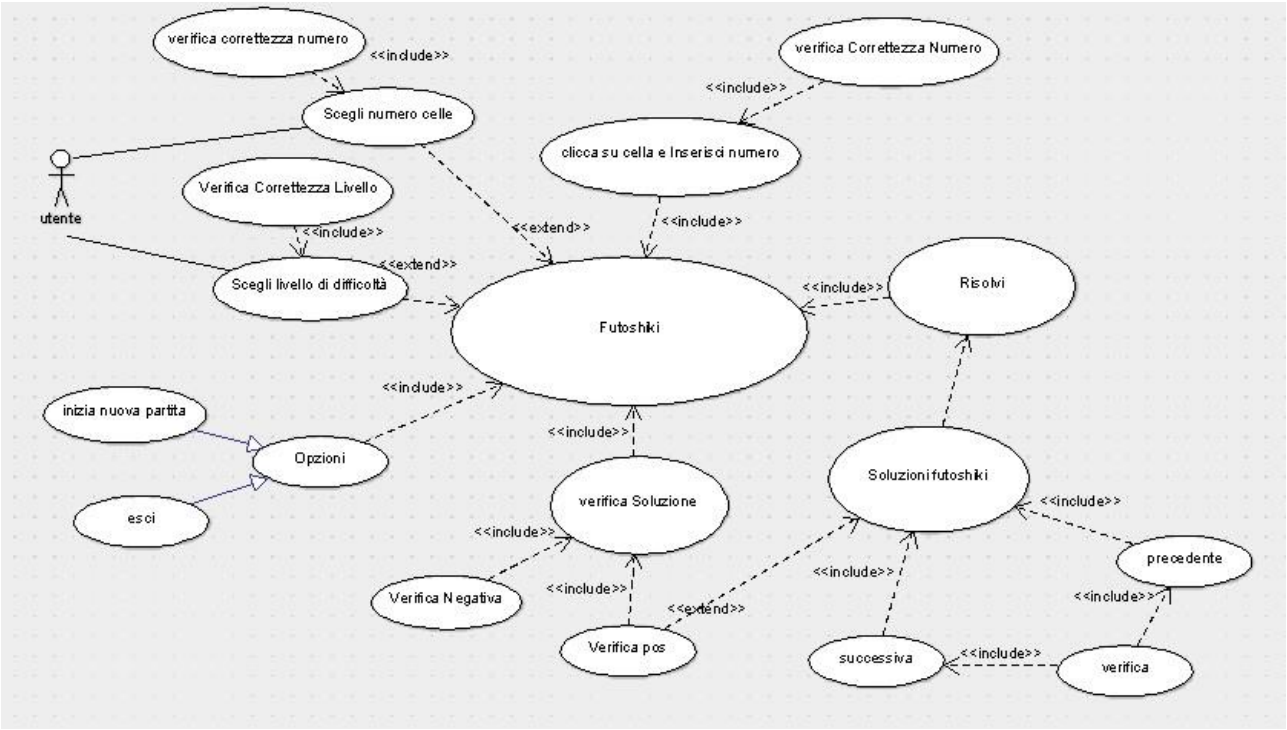
Relazione

Per la realizzazione del progetto ho fatto uso dei pattern:

- Template Method:
  Di norma esso è un Design pattern il cui scopo è di tipo comportamentale e il cui raggio d'azione è orientato alle classi, ossia usare l'ereditarietà per descrivere algoritmi e flussi di dati. Lo scopo è quello di definire la struttura di un algoritmo all'interno di un metodo di una classe, delegando i passi dell'algoritmo alle sottoclassi. Template Method lascia che le sottoclassi ridefiniscano alcuni passi dell'algoritmo senza dover implementare di nuovo la struttura dell'algoritmo stesso. Noi ad esempio lo applichiamo al backTracking: l'interfaccia definisce una serie di operazioni elementari e rimanda tutto al metodo concreto risolvi(). La struttura è molto semplice: abbiamo una classe astratta che definisce il templateMethod() che al proprio interno ha le operazioni che le sotto classi concrete dovranno implementare. I metodi template sono una tecnica fondamentale per il riuso del codice.
  La struttura quindi risulta essere così fatta:
  AbstractTemplate: definisce una classe astratta nella quale vi è un metodo templateMethod() che utilizza i metodi primitiveOperation1() e primitiveOperation2();
  ConcreteTemplate: estende la classe astratta AbstractTemplate e ridefinisce i metodi della classe astratta che conformeranno alla fine il metodo risolvi della classe astratta ;

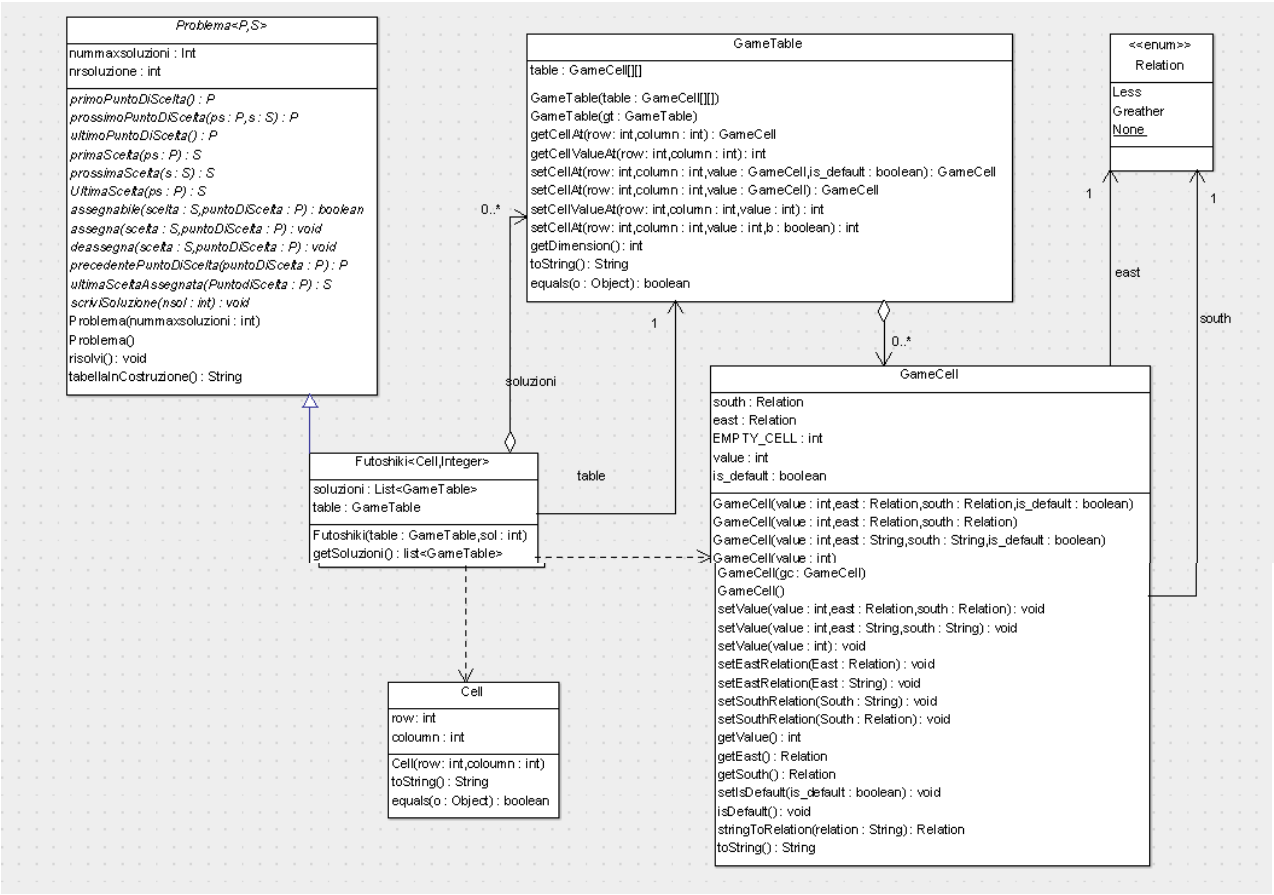


Per rappresentare la modalità di uso del programma faccio ricorso ad un caso d' uso ossia uno scenario generale che può avere l'utente con il programma attraverso l'interfaccia grafica sviluppata:



1. L'utente all'inizio si trova davanti un'interfaccia che presenta una casella di testo nella quale si può inserire il numero di celle che dovrà avere il nostro gioco e successivamente un'altra casella di testo dove si dovrà inserire un livello di difficoltà, il quale verrà verificato e solo in caso di risposta positiva verrà aperto il gioco. I livelli sono di tre tipi:
    a. Facile: mette a disposizione una tabella con una bassa probabilità che ci siano tanti segni.
    b. Medio: mette a disposizione una tabella con una media probabilità che ci siano tanti segni.
    c. Difficile: mette a disposizione una tabella con un'alta probabilità che ci siano tanti segni.
2. Viene aperto il Futoshiki che ha diverse modalità di sviluppo da parte dell'utente:
    a. Clicca su una cella: Mette a disposizione un'interfaccia nella quale è possibile inserire un valore da inserire in quella cella, e se rispetta i dovuti controlli viene accettata e inserita lì.
    b. Se si clicca su verifica Soluzione:
        i. Se soluzione positiva si va direttamente in 'Soluzioni Futoshiki'.
        ii. Se soluzione negativa si ritorna al Futoshiki.
    c. Se clicchi Risolvi:
        i. Si va in Soluzioni Futoshiki.
3. Soluzioni Futoshiki è una nuova interfaccia che ha un tasto successivo e un tasto prec:
    a. Se clicco tasto successivo: vedo la soluzione successiva;
    b. Se clicco tasto precedente: vedo la soluzione precedente;
    c. Se clicco su opzioni mi viene aperto un menu che ha due scelte:
        i. Esci: chiude l'applicazione.
        ii. Nuova partita: apre una nuova partita che riparte dalla scelta del livello di gioco.
4. Opzioni è un menù che ha due scelte:
    a. Esci: chiude l'applicazione.
    b. Nuova partita: apre una nuova partita che riparte dalla scelta del livello di gioco.

Ora parliamo di come sono state strutturate le classi all' interno del progetto attraverso l'utilizzo del class diagram:



Attraverso l'utilizzo della classe Futoshiki è stato possibile riuscire a collegare i singoli componenti tra di loro e ad concretizzare i metodi astratti della classe Problema di tipo <P,S>.

Ora mosto il metodo risolvi nella classe astratta Problema:

```
public void risolvi() {
                P ps = primoPuntoDiScelta();
                S s = primaScelta(ps);
                boolean backtrack = false, fine = false;
                do {
                        while (!backtrack && nrsoluzione < nummaxsoluzioni) {
                                if (assegnabile(s, ps)) {
                                        assegna(s, ps);
                                        if (ps.equals(ultimoPuntoDiScelta())) {
                                                ++nrsoluzione;
                                                scriviSoluzione(nrsoluzione);
                                                deassegna(s, ps);
                                                if (!s.equals(ultimaScelta(ps)))
                                                        s = prossimaScelta(s);
                                                else
                                                        backtrack = true;
                                        } else {
                                                ps = prossimoPuntoDiScelta(ps,s);
                                                s = primaScelta(ps);
                                        }
```

```
                        } else if (!s.equals(ultimaScelta(ps)))
                                s = prossimaScelta(s);
                        else
                                backtrack = true;
                }
                fine = ps.equals(primoPuntoDiScelta())
                                || nrsoluzione == nummaxsoluzioni;
                while (backtrack && !fine) {
                        ps = precedentePuntoDiScelta(ps);
                        s = ultimaSceltaAssegnata(ps);
                        deassegna(s, ps);

                        if (!s.equals(ultimaScelta(ps))) {
                                s = prossimaScelta(s);
                                backtrack = false;
                        } else if (ps.equals(primoPuntoDiScelta())){
                                fine = true;
                        }
                }

        } while (!fine);
    }
```

Questo è il tipico problema di backtraking che va a prendere piano piano le soluzioni e tramite i metodi si va ad accertare se si tratti di una soluzione ammissibile in tal caso incrementa di uno il numero di soluzioni e la aggiunge alla lista delle soluzioni.

```
public class Cell {
        int row=0,column=0;

        public Cell (int row,int column){
                this.row=row;
                this.column=column;
        }
}
```

La classe Cell non è altro che un oggetto che contiene una riga e una colonna e contiene un metodo equals e un metodo toString.

```
public class GameTable {
        private GameCell[][] table;
        public GameTable(int dim){
                this(new GameCell[dim][dim]);
        }
        public GameTable (GameCell[][] table){
                if(table.length==0){
                        this.table=table;
                        return;
                }
                this.table=new GameCell[table.length][table[0].length];
                for(int i=0; i<table.length;i++){
                        if(table.length!=table[i].length)throw new IllegalArgumentException();
                        for(int j=0;j<table[i].length;j++){
                                if(table[i][j]==null){
                                        this.table[i][j]=new GameCell();
                                }else {
                                        this.table[i][j]=new GameCell(table[i][j]);
                                }
                        }
```

```
		}//for
	}//GameTable

	public GameTable(GameTable gt){
		this(gt.table);
	}

	public GameCell getCellAt(int row, int column){
		return new GameCell(table[row][column]);
	}

	public int getCellValueAt(int row, int column){
		return table[row][column].getValue();
	}

	public GameCell setCellAt(int row,int column, GameCell value, boolean is_default){
		GameCell g = table[row][column];
		table[row][column]=new GameCell(value);
		table[row][column].setIsDefault(is_default);
		return g;
	}

	public GameCell setCellAt(int row,int column, GameCell value){
		return setCellAt(row,column,value,value.isDefault());
	}


	public int setCellValueAt(int row , int column, int value){
		GameCell c=table[row][column];
		return setCellValueAt(row,column,value,(c==null)?value!=GameCell.EMPTY_CELL:c.isDefault());
	}


	public int setCellValueAt(int row, int column, int value, boolean b) {
		int i = table[row][column].getValue();
		table[row][column].setValue((value<=0)?GameCell.EMPTY_CELL:value);
		table[row][column].setIsDefault(b);
		return i;
	}

	public int getDimension(){return table.length;}
}
```

La classe GameTable va a creare la cosiddetta tabella di gioco ossia va a prendere in modo particolare una matrice di GameCell. Sopra sono riportarti costruttori di default costruttori di clonazioni costruttori in caso di matrice vuota con verifiche sugli elementi e sulle colonne. In più sono esposti metodi per settare o prendere i valori dato riga e colonna della cella di gioco.

```
public class GameCell {
	private Relation south=Relation.NONE;
	private Relation east=Relation.NONE;
	/**indica una cella vuota*/
	public static final int EMPTY_CELL=0;
	/**valore della cella*/
	private int value=EMPTY_CELL;
	private boolean is_default=false;

	public GameCell(int value,Relation east,Relation south,boolean is_default){
		this.value=value;
		this.east=east;
		this.south=south;
```

```
        this.is_default=is_default;
}

public GameCell(int value,Relation east,Relation south){
        this(value,east,south,false);
        if(value!=EMPTY_CELL) is_default=true;

}

public GameCell(int value, String relation_east,String relation_south){
        this(value,stringToRelation(relation_east),stringToRelation(relation_south));
}

public GameCell(int value){
        this(value,Relation.NONE,Relation.NONE);
}
public GameCell(GameCell copy){
        this(copy.value,copy.east,copy.south,copy.is_default);
}

public GameCell(){this(EMPTY_CELL);}

public void setValues(int value, Relation east, Relation south){
        setValue(value);
        setEastRelation(east);
        setSouthRelation(south);
}

public void setValues(int value,String east,String south){
        setValues(value,stringToRelation(east),stringToRelation(south));
}

public void setValue(int value){
        this.value=value;
}
public void setEastRelation(Relation east){
        this.east=east;
}
public void setEastRelation(String east){
        setEastRelation(stringToRelation(east));
}
public void setSouthRelation(Relation south){
        this.south=south;
}
public void setSouthRelation(String south){
        setSouthRelation(stringToRelation(south));
}
public int getValue (){
        return value;
}
public Relation getEast(){
        return east;
}
public Relation getSouth(){
        return south;
}
public void setIsDefault(boolean is_default){
        this.is_default=is_default;
}
public boolean isDefault(){return is_default;}

public static Relation stringToRelation(String relation){
        if(relation==null)return Relation.NONE;
        relation=relation.toUpperCase().trim();
        for(Relation r:Relation.values()){
```

```
                        if(r.toString().equals(relation)) return r;
                }
                return Relation.NONE;
        }//stringToRelation

        public String toString (){
                return "value="+value+";east="+east+";south="+south+";";
        }

}
```

GameCell è una cella della matrice che ha un valore che potrebbe essere vuoto o potrebbe essere modificato e potrebbe inserire le relazioni di sud ed est che corrispondono alle relazioni di maggiore o di minore o potrebbero semplicemente essere settate a None infatti la classe Relation è un enumeration che ha tre tipi di relazione infatti:

```
public enum Relation {
        //relazione minore
        LESS,
        //nessuna relazione
        NONE,
        //relazione maggiore
        GREATER
}
```

Infine parliamo della classe principale che è Futoshiki che va a concretizzare tutti i metodi della classe astratta ma parliamo dei principali e più significativi:

package Futoshiki;

```
import java.util.*;

import javax.swing.plaf.synth.SynthSeparatorUI;

public class Futoshiki extends Problema<Cell,Integer> {

        private GameTable table;
        private List<GameTable> soluzioni=new LinkedList<GameTable>();
        public Futoshiki(int dim,int nsol){
                this(new GameTable(dim),nsol);
        }
        public Futoshiki (GameTable table,int nsol){
                super(nsol);
                this.table=new GameTable(table);
        }
```

//usiamo due attributi una GameTable e una lista di GameTable che rappresenta la lista delle soluzioni, in più usiamo due tipi di costruttori.

- uno nel quale vengono passati nsol e la dimensione della GameTable,
- uno nel quale vengono passati nsol e una GameTable,

```
        @Override
        protected Cell primoPuntoDiScelta() {
                return prossimoPuntoDiScelta(new Cell(0,-1),1);
        }
        @Override
        protected Cell prossimoPuntoDiScelta(Cell c, Integer value) {
                for(int i=c.row;i<table.getDimension();i++){
                        for(int j=(c.row==i)?c.column+1:0;j<table.getDimension();j++){
                                GameCell g= table.getCellAt(i,j);
                                if(g.getValue()==GameCell.EMPTY_CELL || ! g.isDefault())
                                        return new Cell(i,j);
                        }
```

```
		};
		return ultimoPuntoDiScelta();
}//prossimoPuntoScelta

@Override
protected Cell ultimoPuntoDiScelta() {
		int dim=table.getDimension()-1;
		Cell returned=new Cell(dim,dim);
		for(int i=dim;i>=0;i--){
			for(int j=dim;j>=0;j--){
				GameCell g= table.getCellAt(i, j);
				if(g.getValue()==GameCell.EMPTY_CELL || ! g.isDefault())
					return new Cell(i,j);
			}
		}
		return returned;
}
@Override
protected boolean assegnabile(Integer value, Cell c) {
		GameCell g=table.getCellAt(c.row, c.column);
		if(g.isDefault())return true;

		/*----------inizio controlli relazioni ----------------*/

		GameCell east,west,south,north;
		int tmp=-1;
		if(c.row>0){
			north=table.getCellAt(c.row-1, c.column);
			Relation r=north.getSouth();
			tmp=north.getValue();
			switch(r){
			case LESS : if (tmp!=GameCell.EMPTY_CELL&&value<tmp) return  false; break;
			case GREATER : if (tmp!=GameCell.EMPTY_CELL&&value>tmp) return false;break;
			default : ;
			}//switch
		}

		if(c.row<table.getDimension()-1){
			south=table.getCellAt(c.row+1, c.column);
			Relation r=g.getSouth();
			tmp=south.getValue();
			switch(r){
			case LESS : if (tmp!=GameCell.EMPTY_CELL&&value>tmp) return  false; break;
			case GREATER : if (tmp!=GameCell.EMPTY_CELL&&value<tmp) return false;break;
			default : ;
			}//switch
		}
		if(c.column>0){
			west=table.getCellAt(c.row, c.column-1);
			Relation r= west.getEast();
			tmp=west.getValue();
			switch(r){
			case LESS : if (tmp!=GameCell.EMPTY_CELL&&value<tmp) return  false; break;
			case GREATER : if (tmp!=GameCell.EMPTY_CELL&&value>tmp) return false;break;
			default : ;
			}//switch
		}

		if(c.column<table.getDimension()-1){
			east=table.getCellAt(c.row, c.column+1);
			Relation r=g.getEast();
			tmp=east.getValue();
			switch(r){
			case LESS : if(tmp!=GameCell.EMPTY_CELL&&value>tmp) return false; break;
			case GREATER : if (tmp!=GameCell.EMPTY_CELL&&value< tmp) return false; break;
```

```
                    default : ;
                    }
            }
            /*------------fine controlli su relazioni--------------*/
```

Questi infatti sono tutti i controlli per verificare sulle righe e sulle colonne se valgono le relazioni dovute ai segni maggiore e minore.

```
            /*------------inizio controllo su valori uguali--------------*/

            for(int i=0;i<table.getDimension();i++){
                    if(i==c.row) continue;
                    if(value==table.getCellValueAt(i, c.column)&&value!=GameCell.EMPTY_CELL)return false;
            }
            for(int j=0;j<table.getDimension();j++){
                    if(j==c.column) continue;
                    if(value==table.getCellValueAt(c.row, j)&&value!=GameCell.EMPTY_CELL)return false;
            }
            /*------------fine controlli su valori uguali--------------*/
            return true;
    }
```

Questi invece sono i due controlli da fare uno sulle righe e uno sulle colonne per vedere se sono presenti valori uguali.

}

L'interfaccia appare in questo modo: