Flavio Mascetti 262277
Davide Cortellucci 260321

Progetto Ingegneria del Software
SHISEN-SHO

Docente: Prof. Edoardo Bontà

Specifica del problema

Il gioco segue le regole dello Shisen-Sho, una variante del Mahjong dal quale eredita i mattoncini (tile). Tali mattoncini sono disposti su un piano rettangolare; è possibile rimuovere due mattoncini per volta secondo le seguenti regole:

- devono essere dello stesso tipo;
- devono poter essere collegati con al massimo 3 linee orizzontali e/o verticali che passano attraverso spazi vuoti (non occupati da altre tile).

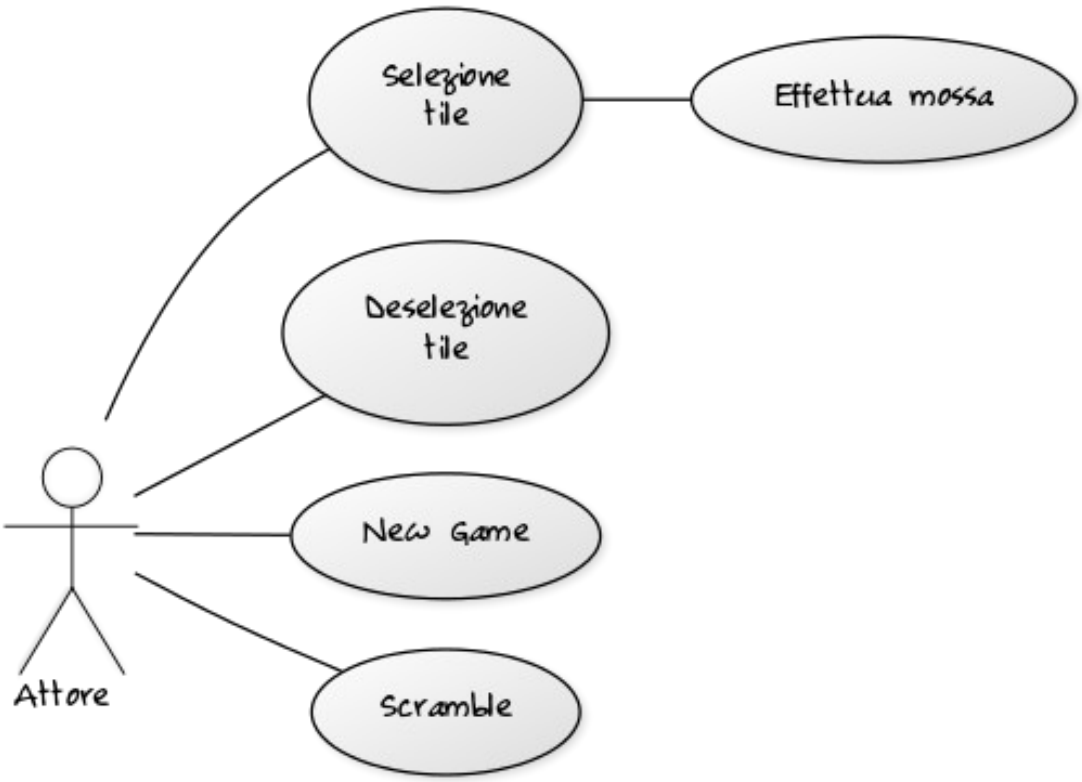
La partita termina se tutti i mattoncini sono stati rimossi (vittoria) o se ce ne sono ancora e non è possibile alcun collegamento (sconfitta).

Specifica dei requisiti

Descrizione Informale:

- Il giocatore evidenzia una tile eseguendo un click su di essa
- Il giocatore deseleziona una tile effettuando un click sulla stessa
- Il giocatore effettua una mossa selezionando 2 tile sul piano di gioco
- Viene verificata la validità della mossa (le tile devono essere uguali ed il cammino valido)
- Se la verifica fallisce, la mossa non viene eseguita e le tile selezionate vengono deselezionate
- Se la verifica ha successo, la mossa viene eseguita rimuovendo dal piano di gioco le tile
- A mossa eseguita deve essere notificata la sconfitta/vittoria

Diagrammma dei casi d'uso:



Descrizione dei casi d'uso

Caso d'uso: Selezione tile
ID: ST
Pre-condizioni: /
Corso degli eventi: l'attore esegue un click su una delle tile
Post-condizioni: la tile viene marchiata come selezionata
Alternativa: /

Caso d'uso: Deselezione tile
ID: DT
Pre-condizioni: è stata selezionata un tile (ST)
Corso degli eventi: l'attore esegue un secondo click sulla tile selezionata
Post-condizioni: la tile viene deselezionata
Alternativa: /

Caso d'uso: Effettua Mossa
ID: EM
Pre-condizioni: è stata già selezionata un altra tile (ST)
Corso degli eventi: l'attore esegue un click su un'altra delle tile
Post-condizioni: se le tile selezionate sono uguali e la mossa possibile, le tile vengono rimosse
Alternativa: se le tile selezionate sono diverse viene deselezionata la prima tile e marchiata come selezionata la seconda

Caso d'uso: New Game
ID: NG
Pre-condizioni: /
Corso degli eventi: Viene generato un nuovo tavolo di gioco
Post-condizioni: il giocatore può cominciare a selezionare tile (ST)
Alternativa: /

Caso d'uso: Scramble
ID: SC
Pre-condizioni: /
Corso degli eventi: Viene mischiato il tavolo di gioco (le tile)
Post-condizioni: il giocatore può selezionare tile (ST)
Alternativa: /

Analisi e progettazione

Scelte di progetto:

- Ambiente di sviluppo
- GUI
- Dimensioni piano di gioco
- Numero di tile diverse tra loro
- Possibilità di mischiare le tile
- Individuazione game over
- Ricerca cammino analizzando le varie combinazioni
- Ricerca cammino: dimezzamento combinazioni tramite scambio di coordinate
- Ricerca cammino delegata a più di un metodo
- Ordine di selezione tile rilevante
- Cammini non stampati a video
- New Game a fine partita

Ambiente di sviluppo

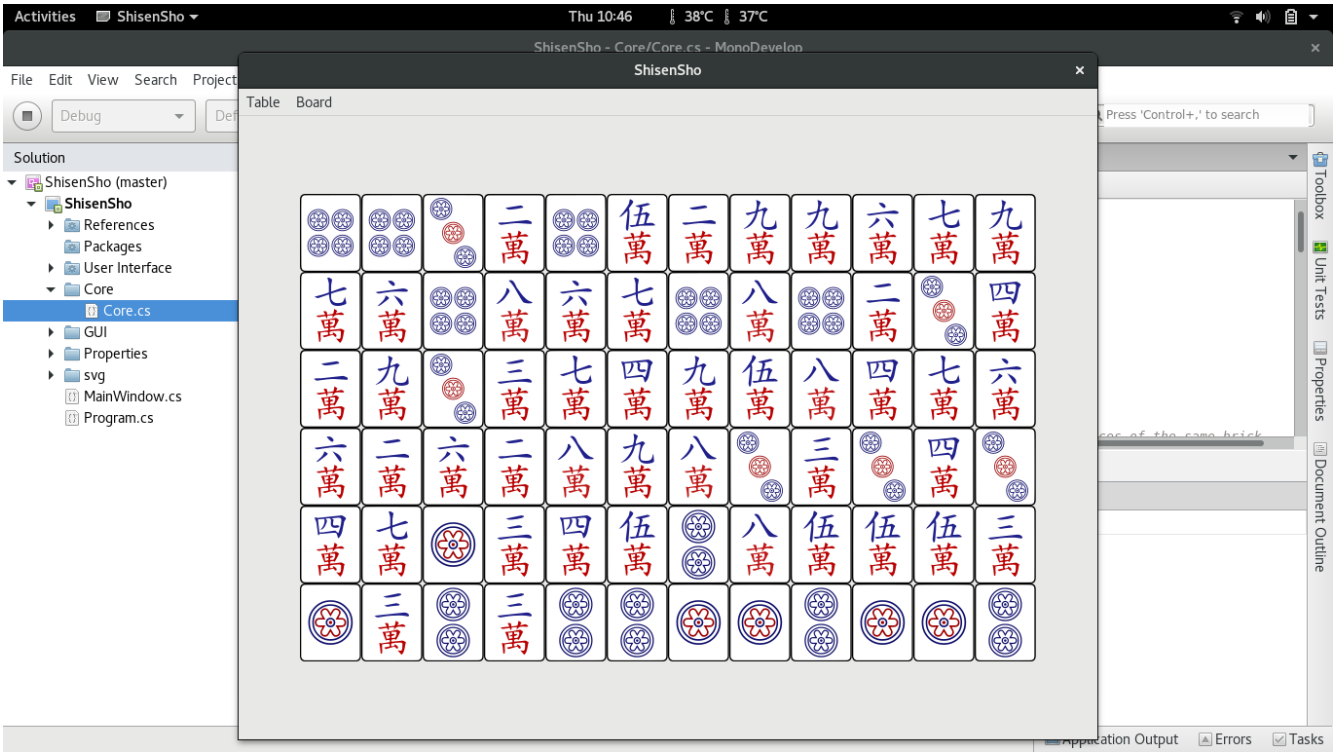
A riguardo vedere la sezione *Compilazione ed esecuzione.*

GUI

Volendo sviluppare un applicazione in C# per Gnu/Linux ed avere un tema in linea con gnome abbiamo deciso di utilizzare la libreria grafica Gtk#.
Le icone utilizzate per le tile sono state reperite dal seguente link:
https://commons.wikimedia.org/wiki/User:Shizhao/Mahjong

Dimensioni piano di gioco

Le dimensioni del piano di gioco sono prefissate (non possono essere modificate dall'utente). Tale piano consiste in una griglia di 12 colonne e 6 righe (quindi 72 tile totali), inoltre sono presenti due colonne e due righe vuote ai confini del piano per permettere la ricerca di collegamenti tra le tile.



Numero di tile diverse tra loro

Anche il numero di tile diverse tra loro è fisso e non modificabile dall'utente. In questo caso si parla di 12 tipi di tile differenti.

Scramble board

Si è scelto di implementare un metodo che mischia le tile del piano scambiando l'ordine tra di esse. L'utilizzo di tale funzionalità è a discrezione dell'utente ed è utilizzabile quante volte si vuole, ma è efficace solo se non viene utilizzata due volte consecutive (foto 2), cioè nel caso in cui l'utente dovesse invece fare ciò, la configurazione del piano sarebbe la stessa di partenza (foto 1).

Foto 1: configurazione del piano prima di scramble board

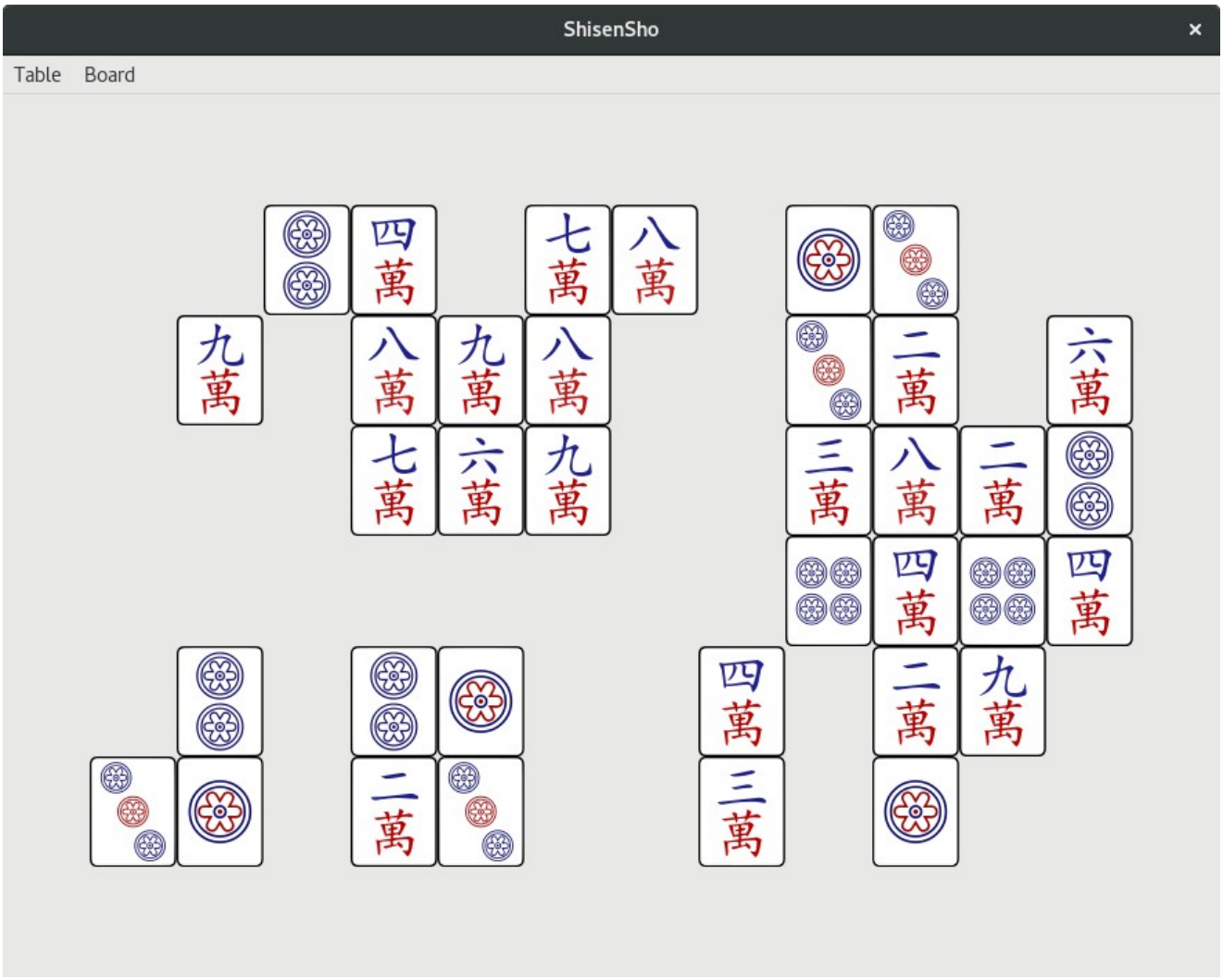
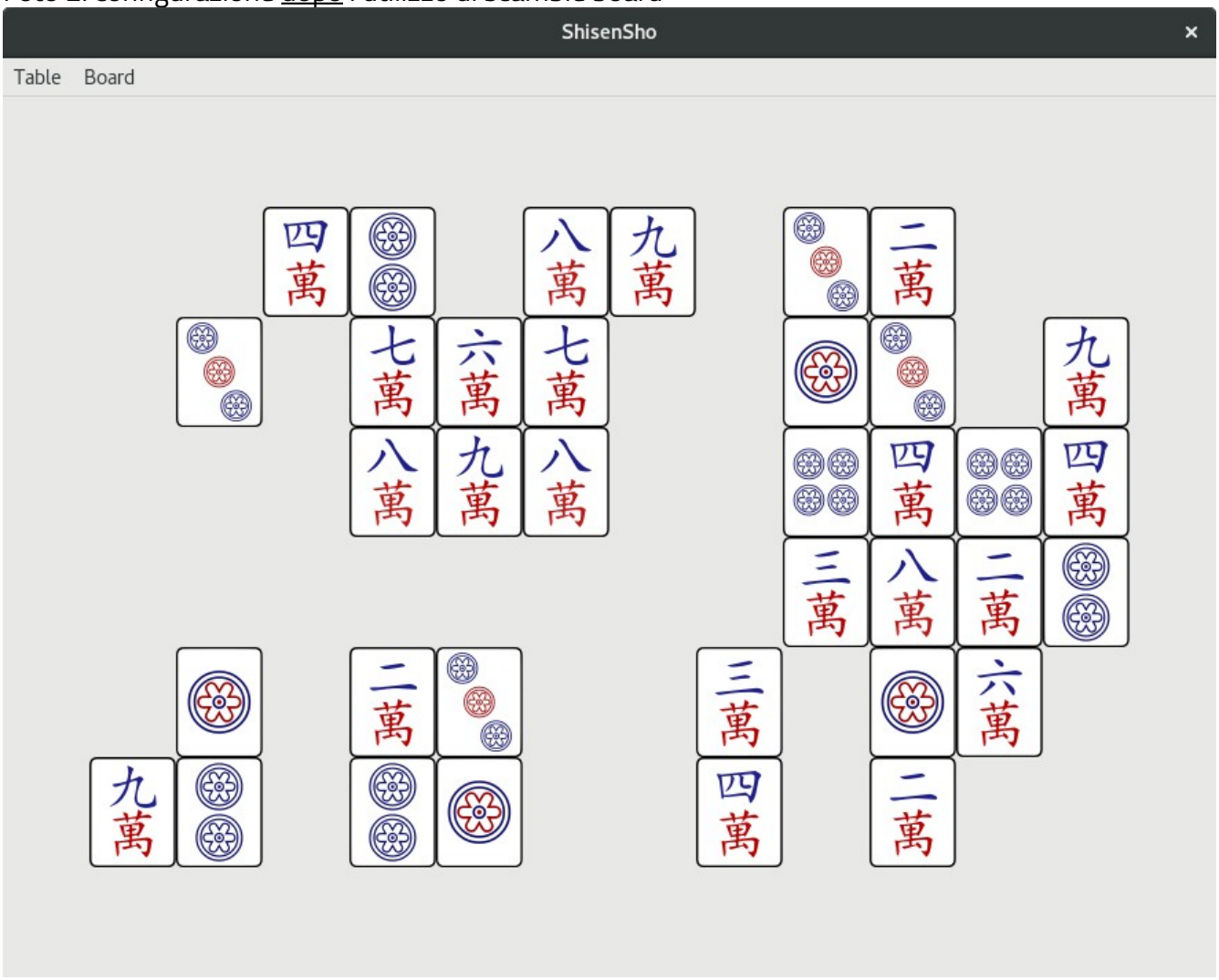


Foto 2: configurazione dopo l'utilizzo di scamble board



Individuazione game over

Il programma è in grado di individuare una situazione di game over e di stampare un messaggio a video nel caso si verifichi tale eventualità.

Ordine di selezione tile

L'ordine di selezione delle tile effettuato tramite GUI è rilevante: I metodi di ricerca dei cammini comincieranno la ricerca a partire dalla prima tile selezionata.

Ricerca cammino

La ricerca dei cammini si basa sulla suddivisione in casi di tutte le possibili configurazioni geografiche delle tile selezionate (ad esempio tile sulla stessa riga, tile sulla stessa colonna, eccetera).
Essendo l'ordine delle tile rilevante nella ricerca del cammino, si è trovato il modo di dimezzare I casi da analizzare in modo da non dover scrivere codice ulteriore (questa ottimizzazione è spiegata più in dettaglio nella descrizione delle classi con UML , classe Core, metodo pathViability).
Si è infine deciso di utilizzare più di un metodo per il controllo dei cammini, in modo da eliminare ridondanza di codice e migliorare la leggibilità di quest'ultimo.

Cammini non stampati a video

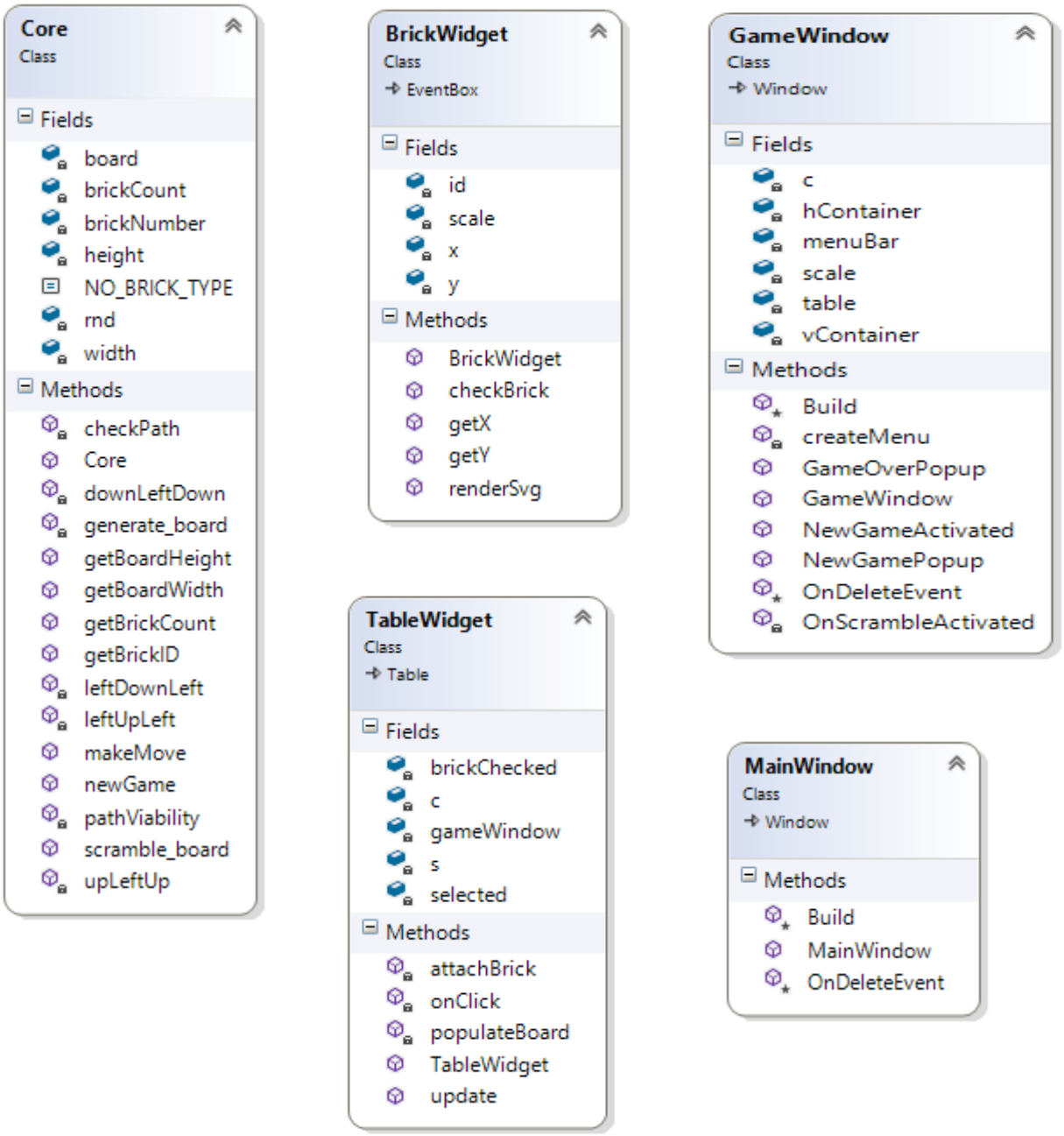
I cammini trovati non vengono stampati a video; questa decisione è stata presa per non utilizzare ulteriori risorse computazionali a fronte dell'implementazione della ricerca dei cammini.

New Game a fine partita

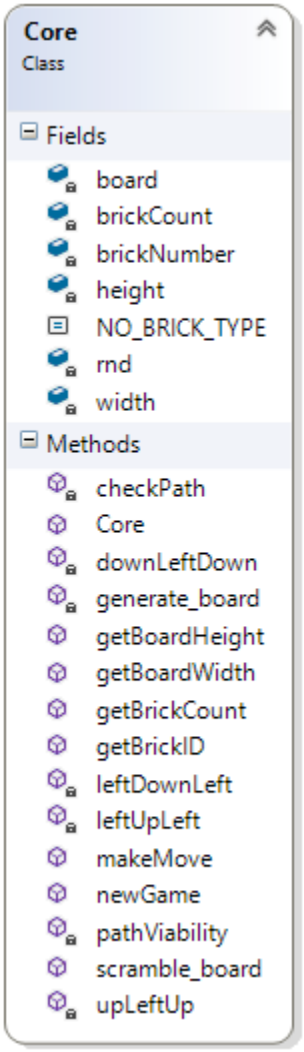
Una volta che il piano è stato completamente svuotato (partita vinta), il programma chiede al giocatore se desidera effettuare una nuova partita. In caso di risposta affermativa, il software genera un nuovo piano di gioco, altrimenti esso termina.

Descrizione delle classi tramite UML

Di seguito viene elencata una panoramica delle classi che costituiscono il programma:



- **Core**



Questa classe rappresenta la parte logica del programma; tra le mansioni più importanti che Core svolge vi è quella di generare il piano di gioco (board), popolarlo con delle tile e controllare se è possibile rimuovere le due tile selezionate dall'utente tramite la GUI.

La classe presenta I seguenti campi:

- **board:** array bidimensionale di interi che rappresenta il piano di gioco; il primo indice si riferisce al numero di colonne del piano, mentre il secondo al numero di righe.
- **brickCount:** numero totale di mattoncini presenti nel piano, esso diminuisce di 2 ogni volta che vengono rimossi 2 mattoncini.
- **brickNumber:** intero che identifica il numero di tipi di tile presenti nel piano. In questo caso vi sono 72 tile totali e 3 coppie di tile uguali tra loro (dunque 6), quindi *brickNumber = 72 / 6 = 12*.

- **height:** parametro che indica l'altezza del piano popolato dai mattoncini (sono escluse le righe vuote in cima e in fondo al piano).
- **width:** similmente a height, width denota la larghezza, escludendo le due colonne vuote a sinistra e a destra del piano.
- **rnd:** istanza della classe Random, è un numero casuale che contribuirà a popolare il piano casualmente.
- **NO_BRICK_TYPE:** costante di valore zero (indica una casella vuota).

Per quanto riguarda I metodi della classe, essi sono I seguenti:

- **Core:** costruttore della classe.
- **getBrickCount:** restituisce il numero di tile rimanenti.
- **getBrickID:** restituisce l'ID della tile .
- **getBoardHeight:** restituisce il valore memorizzato in height (altezza)
- **getBoardWidth:** restituisce il valore memorizzato in width (larghezza)
- **generateBoard:** questo metodo inizializza il piano (board) con una "cornice" vuota, cioè senza mattoncini (zero). Succesivamente popola il piano partendo dalla prima colonna a sinistra e riempiendo via via ogni casella con una tile generata in maniera pseudo-casuale.
- **newGame:** richiamato quando l'utente vuole iniziare una nuova partita, new game ripopola il piano di gioco daccapo richiamando generateBoard.
- **scramble_board:** mischia I mattoncini senza modificare la geografia del piano (le caselle vuote rimangono tali).
- **makeMove:** ha come input le coordinate dei mattoncini. Dal momento che l'ordine in cui le tile vengono selezionate dalla GUI ha ripercussioni sugli algoritmi che controllano I collegamenti, questo metodo si occupa di dimezzare I casi da controllare e di richiamare successivamente il controllo sulle coordinate in ingresso. Se il collegamento va a buon fine, brickCount viene decrementato di 2.
- **pathViability:** controlla se esiste un cammino per collegare I 2 mattoncini selezionati (le coordinate vengono ricevute in input); se ne è possibile almeno uno il metodo restituisce vero, altrimenti falso. In base alla posizione delle tile, pathViability delega la ricerca del cammino ad altri metodi. Le combinazioni possibili sono 8, ridotte a 4 da makeMove scambiando tra loro le coordinate dellle tile quando serve. Tenendo conto che x1, y1 sono le coordinate della prima tile selezionata e x2, y2 quelle della seconda, I casi possono essere riassunti nella seguente tabella:

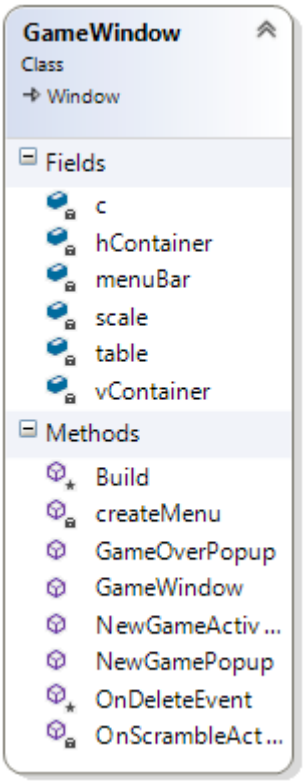
X1, x2	Y1, y2	Implementazione
>	>	Ricerca cammino
>	<	Ricerca cammino
>	=	Ricerca cammino
<	>	Scambio coordinate
<	<	Scambio coordinate
<	=	Scambio coordinate
=	>	Ricerca cammino
=	<	Scambio coordinate

Lo scambio di coordinate, ove effettuato, permette di ricondurre tali casi a quelli che vengono analizzati dai metodi che si occupano di cercare un collegamento: ad esempio il quarto caso viene ricondotto al secondo, il quinto al primo, eccetera.
A questo punto la ricerca del cammino viene affidata ai successivi 5 metodi.

- **checkPath:** ha come input le coordinate delle tile e il caso da analizzare, quest'ultimo computato in pathViability. Questo metodo è in grado di analizzare tutti I casi elencati in tabella, tranne dei cammini particolari, la cui verifica viene a sua volta delegata ai rimanenti 4 metodi. Esso restituisce vero se un cammito esiste, falso altrimenti.
- **downLeftDown:** ha come input le coordinate delle tile; questo metodo controlla un cammino che prevede almeno:
  uno spostamento verso il basso, uno a sinistra ed uno di nuovo in basso.
- **leftDownLeft:** ha come input le coordinate delle tile; questo metodo controlla un cammino che prevede almeno:
  uno spostamento verso sinistra, uno in basso ed uno di nuovo a sinistra.
- **leftUpLeft:** ha come input le coordinate delle tile; questo metodo controlla un cammino che prevede almeno:
  uno spostamento verso sinistra, uno in alto ed uno di nuovo a sinistra.
- **upLeftUp:** ha come input le coordinate delle tile; questo metodo controlla un cammino che prevede almeno:
  uno spostamento verso l'alto, uno a sinistra ed uno di nuovo in alto.

Passiamo ora all'interfaccia grafica:

- **GameWindow**



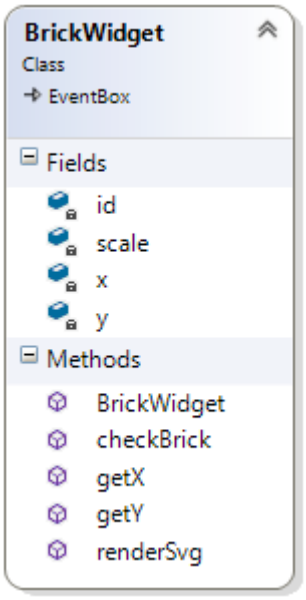
Classe principale della GUI, ha I seguenti campi:

- c
  Oggetto di tipo core (parte logica del programma)
- scale
  Attributo intero utilizzato nel costruttore per adattare la dimensione della GUI rispetto l'altezza dello schermo
- table
  Oggetto per il tavolo di gioco
- vContainer
  Istanza della classe Vbox della libreria Gtk#
- hContainer
  Istanza della classe Hbox della libreria Gtk#
- menuBar
  Istanza della classe MenuBar della libreria Gtk#

Ed i seguenti metodi:

- Build
  Metodo virtuale e protetto che inizializza e mostra la GUI.
- createMenu
  Metodo privato che inizializza il menu e lo appende alla GUI.
- GameOverPopup
  Metodo pubblico che notifica il gameover al giocatore tramite popup
- GameWindow
- NewGameActivated
  Metodo pubblico che crea una nuova partita e richiede un refresh al tavolo di gioco (table)
- NewGamePopup
  Metodo pubblico che richiede all'utente se si vuole effettuare una nuova partita
- OnDeleteEvent
  Metodo protetto per terminare il programma
- OnScrambleActivated
  Metodo privato che mischia le pedine sul tavolo di gioco

- **BrickWidget**



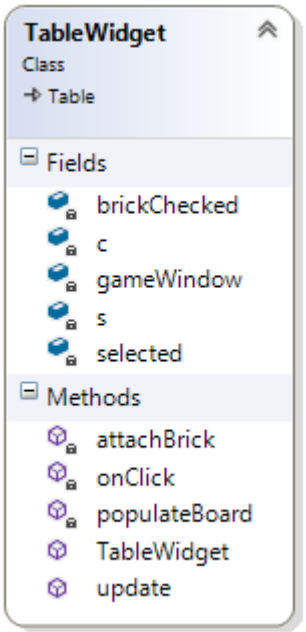
Classe Brick, ha I seguenti campi:

- id
  Attributo stringa privato rappresentante l'id del mattoncino/tile
- scale
  Attributo intero privato utilizzato per rapportare la grandezza della tile rispetto lo schermo (viene passato come parametro al costruttore)
- x
  Attributo privato intero che rappresenta l'indice x della tile nel tavolo di gioco
- y
  Attributo privato intero che rappresenta l'indice y della tile nel tavolo di gioco

Ed i seguenti metodi:

- BrickWidget
  Metodo costruttore, prende i parametri di input, imposta gli attributi, richiama la funzione per renderizzare un svg e se lo associa
- checkBrick
  Metodo publico utile a marcare la tile come selezionata
- getX
  Metodo pubblico che restituisce l'indice x della tile
- getY
  Metodo pubblico che restituisce l'indice x della tile
- renderSvg
  Metodo pubblico che renderizza un svg e ne restituisce l'immagine risultante

- **TableWidget**



Classe Tavolo di gioco:

- brickChecked
  Attributo privato booleano (vero in caso un mattoncino sia selezionato)
- c:
  Oggetto di tipo core (parte logica del programma)

- gameWindow
  Oggetto di tipo GameWindow
- s
  Attributo intero privato per la scala del tavolo di gioco
- selected
  Attributo di tipo brickWidget (e' il riferimento alla tile selezionata)

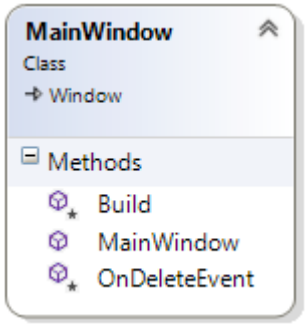
Ed i seguenti metodi:

- TableWidget
  Metodo costruttore, prende i parametri di input, imposta gli attributi ed imposta l'attributo brickChacked a falso
- attachBrick
  Metodo pubblico che aggiunge il brickWidget passato come parametro al tavolo di gioco (TableWidget)
- onClick
  Metodo privato che gestisce l'evento onClick e seleziona/deseleziona/effettua una mossa a seconda del caso che si è verificato
- populateBoard
  Metodo privato che popola il tavolo di gioco in base alle tile presenti in gioco (le direttive vengono prese da core)
- update
  Metodo pubblico che effettua il refresh del tavolo di gioco

- Program

Classe principale del programma, invoca il metodo init, crea un istanza dell'oggetto Core con 3 parametri (6 tile in verticale, 12 in orizzontale e 12 tipologie di tile diverse), un instanza dell'oggetto GameWindow (passando il core come parametro) ed avvia l'applicazione.

- **MainWindow**



Metodi Classe MainWindow:

- Build
  Metodo della classe base Gtk.Window
- MainWindow
  Costruttore classe MainWindow
- OnDeleteEvent
  Metodo protetto per terminare il programma

Implementazione

- **Classe Core**

```
using System;
using Gtk;

namespace ShisenSho
{
    public class Core
    {
        public const int NO_BRICK_TYPE = 0;
        private int height;
        private int width;
        private int brickNumber;    // Number of different type of bricks
        private int [,] board;
        private int brickCount;    // Number of bricks with value != NO_BRICK_TYPE
        private Random rnd;

        public Core (int height, int width, int brickNumber)
        {
            this.height = height;
            this.width = width;
            this.brickNumber = brickNumber;
            newGame ();
        }

        public void newGame ()
        {
            this.brickCount = height * width;
            board = new int[this.width + 2,this.height + 2];
            this.rnd = new Random ();
            generate_board ();
        }

        // Generating bricks for the board
        private void generate_board ()
        {
            int i, j, k;
            int[] numBrick = new int[brickNumber];    // Each entry is for a different brick

            // Initializing the array
            for (i = 0; i < brickNumber; i++)
                numBrick [i] = (this.width * this.height) / brickNumber;    // Number of
                                                    // instances of the same brick
            // Initializing the edges of the board
            for (i = 0; i <= this.height + 1; i++)
                board [0, i] = 0;
            for (j = 0; j <= this.width + 1; j++)
                board [j, 0] = 0;
            // Placing bricks
            for (i = 1; i <= this.height; i++)    // Rows
                for (j = 1; j <= this.width; j++) {    // Column
                    do
```

```
            {
                k = (rnd.Next () % brickNumber);
                if (k != (brickNumber - 1) && numBrick [k] == 0 && numBrick [k + 1] != 0)
                    k++;
                else if (k != 0 && numBrick [k] == 0 && numBrick [k - 1] != 0)
                    k--;
            }
            while (numBrick [k] == 0);
            board [j, i] = k + 1;
            numBrick [k]--;
        }
}

public int getBoardHeight ()
{
    return this.height;
}

public int getBoardWidth ()
{
    return this.width;
}

public int getBrickID (int x, int y)
{
    return this.board [x,y];
}

public int getBrickCount ()
{
    return brickCount;
}

public bool makeMove (int x1, int y1, int x2, int y2)
{
    if (x1 < x2 || (x1 == x2 && y1 < y2))
    {
        int temp = x1;
        x1 = x2;
        x2 = temp;
        temp = y1;
        y1 = y2;
        y2 = temp;
    }

    bool res = pathViability (x1, y1, x2, y2); // Checks if there is a path

    if (res)
    {
        board [x1, y1] = board [x2, y2] = NO_BRICK_TYPE;
        brickCount -= 2;
    }
    return res;
}

public void scramble_board ()
{
    int temp_x = 0;
```

```
    int temp_y = 0;
    int temp_type = NO_BRICK_TYPE;

    // Scrambling board's bricks
    for (int i = 1; i <= this.width; i++)
        for (int j = 1; j <= this.height; j++)
            if (this.board [i,j] > NO_BRICK_TYPE)
            if (temp_type == NO_BRICK_TYPE) {
                temp_x = i;
                temp_y = j;
                temp_type = this.board [i,j];
            } else {
                this.board [temp_x,temp_y] = this.board [i,j];
                this.board [i,j] = temp_type;
                temp_type = NO_BRICK_TYPE;
            }
}

private bool pathViability (int x1, int y1, int x2, int y2)
{
    if (board [x1, y1] != board [x2, y2])
        return false;
    /*** Checking all the cases ***/
    else if (x1 == x2 && y1 > y2) {        // Selected tiles on the same column
            if (checkPath (x1, y1, x2, y2, 1))
                return true;
            else {
                if (board [x1 - 1, y1] == 0      // Try starting from the left
                    && checkPath (x1, y1, x2, y2, 3)) {
                        return true;
                }
                else if (board [x1 + 1, y1] == 0)    // Otherwise try from the right
                    return checkPath (x1, y1, x2, y2, 4);
                else
                    return false;
            }
    }
    else if (y1 == y2 && x1 > x2) {        // Selected tiles on the same row
            if (checkPath (x1, y1, x2, y2, 2))
                return true;
            else {
                if (board [x1, y1 - 1] == 0      // Try starting from the top
                    && checkPath (x1, y1, x2, y2, 5)) {
                    return true;
                }
                else if (board [x1, y1 + 1] == 0)    // Otherwise try from the bottom
                    return checkPath (x1, y1, x2, y2, 6);
                else
                    return false;
            }
    }
    else if (x1 > x2 && y1 > y2) {    // Second tile is top-left
        if (board [x1 - 1, y1] == 0 && checkPath (x1, y1, x2, y2, 7))    // Try starting
                                                                // from the left
            return true;
        else if (board [x1, y1 - 1] == 0 && checkPath (x1, y1, x2, y2, 8))    // Try
                                                        // starting from the top
            return true;
```

```
            else if (board [x1 + 1, y1] == 0 && checkPath (x1, y1, x2, y2, 4))    // Try
                                                    // starting from the right
                return true;
            else if (board [x1, y1 + 1] == 0 && checkPath (x1, y1, x2, y2, 6))    // Try
                                                   // starting from the bottom
                return true;
            else
                return false;
        }
        else if (x1 > x2 && y1 < y2) {    // Second tile is bottom-left
            if (board [x1 - 1, y1] == 0 && checkPath (x1, y1, x2, y2, 9))    // Try
                                                    //  starting from the left
                return true;
            else if (board [x1, y1 - 1] == 0 && checkPath (x1, y1, x2, y2, 5))    // Try
                                                      // starting from the top
                return true;
            else if (board [x1 + 1, y1] == 0 && checkPath (x1, y1, x2, y2, 11))    // Try
                                                    //  starting from the right
                return true;
            else if (board [x1, y1 + 1] == 0 && checkPath (x1, y1, x2, y2, 10))    // Try
                                                    // starting from the bottom
                return true;
            else
                return false;
        }
        else    // Specified for the compiler; end of al the cases
            return false;
    }

private bool checkPath (int x1, int y1, int x2, int y2, int example)
    {
        int i, j;

        switch (example) {
        case 1:    // Tiles on the same column, 0 turns, go up
            for (j = y1 - 1; (j != y2 && board [x1, j] == 0 && j >= 0); j--)
                ;
            if (j != y2)    // Case one
                return false;
            else
                return true;
        case 2:    // Tiles on the same row, 0 turns, go left
            for (i = x1 - 1; (i != x2 && board [i, y1] == 0 && i >= 0); i--)
                ;
            if (i != x2)    // Case one
                return false;
            else
                return true;
        case 3:    // Tiles on the same column, 2 turns, start by going left
            i = x1 - 1;
            do {    // Then go up
                for (j = y1; (j != y2 && board [i, j - 1] == 0 && j >= 0); j--)
                    ;
                i--;
            } while (j != y2 && i >= 0 && board [i, y1] == 0);
            if (j == y2) {    // Go right (last turn)
                for (i = i + 1; (i < x2 && board [i, j] == 0); i++)
                    ;
```

```
            if (i == x2)
                return true;    // End of case three (left-up-right)
            else
                return false;
        }
        else
            return false;
    case 4:    // Tiles on the same column, 2 turns, start by going right
        i = x1 + 1;
        do {    // Then go up
            for (j = y1; (j != y2 && board [i, j - 1] == 0 && j >= 0); j--)
                ;
            i++;
        } while (j != y2 && i <= this.width + 1 && board [i, y1] == 0);
        if (j == y2) {    // Go left (last turn)
            for (i = i - 1; (i > x2 && board [i, j] == 0); i--)
                ;
            if (i == x2)
                return true;    // End of case four (right-up-left)
            else
                return false;
        }
        else
            return false;
    case 5:    // Tiles on the same row, start by going up
        j = y1 - 1;
        do {    // Then go left
            for (i = x1; (i != x2 && board [i - 1, j] == 0 && i >= 0); i--)
                ;
            j--;
        } while (i != x2 && j >= 0 && board [x1, j] == 0);
        if (i == x2) {    // Go down (last turn)
            for (j = j + 1; (j < y2 && board [i, j] == 0); j++)
                ;
            if (j == y2)
                return true;    // End of case three (up-left-down)
            else
                return false;
        }
        else
            return false;
    case 6:
        j = y1 + 1;
        do {    // Then go left
            for (i = x1; (i != x2 && board [i - 1, j] == 0 && i >= 0); i--)
                ;
            j++;
        } while (i != x2 && j <= this.height + 1 && board [x1, j] == 0);
        if (i == x2) {    // Go up (last turn)
            for (j = j - 1; (j > y2 && board [i, j] == 0); j--)
                ;
            if (j == y2)
                return true;    // End of case three (down-left-up)
            else
                return false;
        }
        else
            return false;
```

```
case 7:    // First tile bottom-right; all paths starting from left
    // First try the shortest paths
    for (i = x1; (i != x2 && board [i - 1, y1] == 0 && i >= 0); i--)    // Go left
        ;
    if (i == x2) {    // Try going up
        for (j = y1 - 1; (j != y2 && board [i, j] == 0 && j >= 0); j--)
            ;
        if (j == y2)
            return true;
        else if (board [x2 - 1, y2] == 0) {    // Try left-up-right
            i = i - 1;
            if (board [i, y1] == 0) {
                do {    // Then go up
                    for (j = y1 - 1; (j != y2 && board [i, j] == 0 && j >= 0); j--)
                        ;
                    i--;
                } while (j != y2 && i >= 0 && board [i, y1] == 0);
            }
            if (j == y2) {    // Go right (last turn)
                for (i = i + 1; (i < x2 && board [i, j] == 0); i++)
                    ;
                if (i == x2)
                    return true;    // End of case three (left-up-right)
                else    // Try left-up-left
                    return leftUpLeft (x1, y1, x2, y2);
            } else
                return leftUpLeft (x1, y1, x2, y2);
        } else
            return leftUpLeft (x1, y1, x2, y2);
    } // End if (i == x2)
    else
        return leftUpLeft (x1, y1, x2, y2);
case 8:    // First tile bottom-right; all paths starting from the top
    // First try the shortest paths
    for (j = y1; (j != y2 && board [x1, j - 1] == 0 && j >= 0); j--)    // Go up
        ;
    if (j == y2) {    // Try going left
        for (i = x1 - 1; (i != x2 && board [i, j] == 0 && i >= 0); i--)
            ;
        if (i == x2)
            return true;
        else if (board [x2, y2 - 1] == 0) {    // Try up-left-down
            j = j - 1;
            if (board [x1, j] == 0) {
                do {    // Then go left
                    for (i = x1 - 1; (i != x2 && board [i, j] == 0 && i >= 0); i--)
                        ;
                    j--;
                } while (i != x2 && j >= 0 && board [x1, j] == 0);
            }
            if (i == x2) {    // Go down (last turn)
                for (j = j + 1; (j < y2 && board [i, j] == 0); j++)
                    ;
                if (j == y2)
                    return true;    // End of case three (up-left-down)
                else
                    return upLeftUp (x1, y1, x2, y2);
            } else
```

```
                        return upLeftUp (x1, y1, x2, y2);
                } // End up-left-down
                else
                    return upLeftUp (x1, y1, x2, y2);
            } // End if (j == y2)
            else
                return upLeftUp (x1, y1, x2, y2);
        case 9:    // First tile top-right; all paths starting from the left
            // First try the shortest paths
            for (i = x1; (i != x2 && board [i - 1, y1] == 0 && i >= 0); i--)    // Go left
                ;
            if (i == x2) {    // Try going down
                for (j = y1 + 1; (j != y2 && board [i, j] == 0 && j <= this.height + 1); j++)
                    ;
                if (j == y2)
                    return true;
                else if (board [x2 - 1, y2] == 0) {    // Try left-down-right
                    i = i - 1;    // Go left
                    if (board [i, y1] == 0) {
                        do {    // Then go down
                            for (j = y1 + 1; (j != y2 && board [i, j] == 0 && j <=
                                                    this.height + 1); j++)
                                ;
                            i--;    // Go left until you can go down
                        } while (j != y2 && j <= this.height + 1 && board [i, y1] == 0);
                    }
                    if (j == y2) {    // Go right (last turn)
                        for (i = i + 1; (i < x2 && board [i, j] == 0); i++)
                            ;
                        if (i == x2)
                            return true;
                        else
                            return leftDownLeft (x1, y1, x2, y2);
                    } else
                        return leftDownLeft (x1, y1, x2, y2);
                }    // End left-down-right
                else
                    return leftDownLeft (x1, y1, x2, y2);
            }    // End if (i == x2)
            else
                return leftDownLeft (x1, y1, x2, y2);
        case 10:    // First tile top-right; all paths starting from the bottom
            // First try the shortest paths
            for (j = y1; (j != y2 && board [x1, j + 1] == 0 && j <= this.height + 1); j++)
// Go down
                ;
            if (j == y2) {    // Try going left
                for (i = x1 - 1; (i != x2 && board [i, j] == 0 && i >= 0); i--)
                    ;
                if (i == x2)
                    return true;
                else if (board [x2, y2 + 1] == 0) {    // Try down-left-up
                    j = j + 1;    // Go down
                    if (board [x1, j] == 0) {
                        do {    // Then go left
                            for (i = x1 - 1; (i != x2 && board [i, j] == 0 && i >= 0); i--)
                                ;
                            j++;    // Go down until you can't go left
```

```
                    } while (i != x2 && j <= this.height + 1 && board [x1, j] == 0);
                }
                if (i == x2) {    // Go up (last turn)
                    for (j = j - 1; (j > y2 && board [i, j] == 0); j--)
                        ;
                    if (j == y2)
                        return true;    // End of case three (down-left-up)
                    else
                        return downLeftDown (x1, y1, x2, y2);
                } else
                    return downLeftDown (x1, y1, x2, y2);
            }    // End down-left-up
            else
                return downLeftDown (x1, y1, x2, y2);
        }    // End if (j == y2)
        else
            return downLeftDown (x1, y1, x2, y2);
    case 11:    // First tile top-right; path starting from the right
        i = x1 + 1;
        do {    // Then go down
            for (j = y1; (j != y2 && board [i, j + 1] == 0 && j <= this.height + 1); j++)
                ;
            i++;
        } while (j != y2 && i <= this.width + 1 && board [i, y1] == 0);
        if (j == y2) {    // Go left (last turn)
            for (i = i - 1; (i > x2 && board [i, j] == 0); i--)
                ;
            if (i == x2)
                return true;    // End of case right-down-left
            else
                return false;
        }
        else
            return false;
    default:
        return false;
    }
}

private bool leftUpLeft (int x1, int y1, int x2, int y2)    // Checks a left-up-left path
{
    int i, k,    // X axis
        j;        // Y axis

    if (board [x2 + 1, y2] == 0) {
        k = x1;    // Go left
        do {
            k--;
            do {    // Then go up
                for (j = y1; (j != y2 && board [k, j - 1] == 0 && j > 0); j--)
                    ;
                k--;
            } while (j != y2 && k > 0 && board [k, y1] == 0);
            // Go left (final step)
            for (i = k; (i != x2 && board [i, j] == 0 && i > 0); i--)
                ;
        } while ((i != x2 || j != y2) && i > x2 && board [k, y1] == 0 &&
                                                   board [k - 1, y1] == 0);
```

```
            if (i == x2 && j == y2)
                return true;
            else
                return false;
        } else
            return false;
    }

    private bool upLeftUp (int x1, int y1, int x2, int y2)    // Checks an up-left-up path
    {
        int i,        // X axis
            j, k;    // Y axis

        if (board [x2, y2 + 1] == 0) {
            k = y1;    // Go up
            do {
                k--;
                do {    // Then go left
                    for (i = x1; (i != x2 && board [i - 1, k] == 0 && i > 0); i--)
                        ;
                    k--;
                } while (i != x2 && k > 0 && board [x1, k] == 0);
                // Go up (final step)
                for (j = k; (j != y2 && board [i, j] == 0 && j > 0); j--)
                    ;
            } while ((i != x2 || j != y2) && j > y2 && board [x1, k] == 0 &&
                                                       board [k, y1 + 1] == 0);
            if (i == x2 && j == y2)
                return true;
            else
                return false;
        } else
            return false;
    }

    private bool leftDownLeft (int x1, int y1, int x2, int y2)    // Checks a left-down-left
// path
    {
        int i, k,    // X axis
            j;        // Y axis

        if (board [x2 + 1, y2] == 0) {
            k = x1;    // Go left
            do {
                k--;
                do {    // Then go down
                    for (j = y1; (j != y2 && board [k, j + 1] == 0 && j < this.height + 1);
                                 j++)
                        ;
                    k--;
                } while (j != y2 && k > 0 && board [k, y1] == 0);
                // Go left (final step)
                for (i = k; (i != x2 && board [i, j] == 0 && i > 0); i--)
                    ;
            } while ((i != x2 || j != y2) && k > x2 && board [k, y1] == 0 &&
                                                       board [k - 1, y1] == 0);
            if (i == x2 && j == y2)
                return true;
```

```
            else
                return false;
        } else
            return false;
    }

    private bool downLeftDown (int x1, int y1, int x2, int y2)    // Checks a down-left-down
                                                                  // path
    {
        int i,        // X axis
            j, k;    // Y axis

        if (board [x2, y2 - 1] == 0) {
            k = y1;    // Go down
            do {
                k++;
                do {    // Then go left
                    for (i = x1; (i != x2 && board [i - 1, k] == 0 && i > 0); i--)
                        ;
                    k++;
                } while (i != x2 && k < this.height + 1 && board [x1, k] == 0);
                // Go down (final step)
                for (j = k; (j != y2 && board [i, j] == 0 && j < this.height + 1); j++)
                    ;
            } while ((i != x2 || j != y2) && j < y2 && board [x1, k] == 0 &&
                                                       board [k, y1 - 1] == 0);
            if (i == x2 && j == y2)
                return true;
            else
                return false;
        } else
            return false;
    }
  }
}
```

Passiamo alla GUI:

- **Classe BrickWidget**

```
using System;
using System.IO;
using System.Reflection;
using System.Runtime.InteropServices;
using Gtk;

namespace ShisenSho
{
    public class BrickWidget : Gtk.EventBox
    {
        private int x;
        private int y;
        private string id;
        private int scale;

        public BrickWidget (int x, int y, int s, int brickIdentifier)
        {
            this.scale = s * 20;
            this.id = brickIdentifier.ToString ();
            this.x = x;
            this.y = y;

            Image img = renderSvg (id);
            Fixed f = new Fixed ();
            f.Add (img);
            //f.ShowAll ();
            this.Add (f);
            //this.Show ();
        }

        // This function will produce an image of the desired scale from the svg file
        public Gtk.Image renderSvg (string id)
        {
            Gdk.Pixbuf display;
            string basePath = System.IO.Path.GetDirectoryName
(System.Reflection.Assembly.GetExecutingAssembly ().GetName ().CodeBase).Substring (5);
            try {
                display = new Gdk.Pixbuf (basePath + @"/svg/" + id + ".svg",
                                                        this.scale, this.scale);
            } catch (GLib.GException e) {
                Console.WriteLine (e);
                display = null;
            }
            return (new Gtk.Image (display));
        }

        public Gtk.Image checkBrick ()
        {
            Gdk.Pixbuf display;
            string basePath = System.IO.Path.GetDirectoryName
(System.Reflection.Assembly.GetExecutingAssembly ().GetName ().CodeBase).Substring (5);
```

```
            try {
                display = new Gdk.Pixbuf (basePath + @"/svg/filter.svg",
                                           this.scale, this.scale);
            } catch (GLib.GException e) {
                Console.WriteLine (e);
                display = null;
            }
            return (new Gtk.Image (display));
        }

        public int getX ()
        {
            return this.x;
        }

        public int getY ()
        {
            return this.y;
        }
    }
}
```

- **Classe GameWindow**

```
using System;
using Gtk;

namespace ShisenSho
{
    public class GameWindow : Gtk.Window
    {
        private Core c;
        private TableWidget table;
        private VBox vContainer;            // Vertical container box
        private HBox hContainer;            // Horizontal container box

        private MenuBar menuBar;

        private int scale;


        public GameWindow (Core c) : base (Gtk.WindowType.Toplevel)
        {
            Build ();

            this.c = c;

            // Workaround needed to not exceed window dimension on HD screen or lesser
            scale = (Screen.Height < 1000) ? 4 : 5;

            table = new TableWidget (c,this , scale);
            vContainer = new VBox ();
            hContainer = new HBox ();

            menuBar = createMenu ();
```

```
        this.vContainer.PackStart(menuBar, false, false, 0);
        this.vContainer.PackStart (this.table, false, false, 0);

        hContainer.PackStart (new HBox ());
        hContainer.PackStart (vContainer, false, false, 0);
        hContainer.PackStart (new HBox ());
        hContainer.ShowAll ();

        this.Add (hContainer);

        table.update ();

        this.Show ();
    }

    // Imported from the designer generated file (necessary if you do not want to use
    // the default monodevelop designer)
    protected virtual void Build ()
    {
        global::Stetic.Gui.Initialize (this);
        // Widget MainWindow
        this.Name = "GameWindow";
        this.Title = global::Mono.Unix.Catalog.GetString ("ShisenSho");

        // 3 is the value needed to show the window at the center of the screen
        this.WindowPosition = ((global::Gtk.WindowPosition)(3));
        if ((this.Child != null)) {
            this.Child.ShowAll ();
        }
        this.DefaultWidth = 400;
        this.DefaultHeight = 300;
        this.Show ();
        this.DeleteEvent += new global::Gtk.DeleteEventHandler (this.OnDeleteEvent);
    }

    private void OnScrambleActivated (object sender, EventArgs args)
    {
        c.scramble_board ();
        table.update ();
    }

    public void NewGameActivated (object sender, EventArgs args)
    {
        c.newGame ();
        table.update ();
    }

    public void NewGamePopup ()
    {
        MessageDialog md = new MessageDialog(this,
            DialogFlags.DestroyWithParent, MessageType.Question,
            ButtonsType.YesNo, "Do you want to play another game?");

        md.Response += delegate (object o, ResponseArgs resp) {

            if (resp.ResponseId == ResponseType.No) {
                Application.Quit ();
            }
```

```
            };

            md.Run();
            md.Destroy();
        }

        public void GameOverPopup ()
        {
            MessageDialog md = new MessageDialog(this,
                DialogFlags.DestroyWithParent, MessageType.Question,
                ButtonsType.YesNo, "You lost! Do you want to play again?");

            md.Response += delegate (object o, ResponseArgs resp) {

                if (resp.ResponseId == ResponseType.No) {
                    Application.Quit ();
                }
            };

            md.Run();
            md.Destroy();
        }

        private MenuBar createMenu ()
        {
            MenuBar m = new MenuBar ();
            MenuItem entry = new MenuItem("Table");
            Menu menu = new Menu();

            MenuItem item = new MenuItem ("New Game");
            item.Activated += NewGameActivated;
            menu.Append (item);

            entry.Submenu = menu;
            m.Append(entry);

            entry = new MenuItem ("Board");
            menu = new Menu ();

            item = new MenuItem("Scramble");
            item.Activated += OnScrambleActivated;
            menu.Append(item);

            entry.Submenu = menu;
            m.Append (entry);

            return m;
        }

        protected void OnDeleteEvent (object sender, DeleteEventArgs a)
        {
            Application.Quit ();
            a.RetVal = true;
        }
    }
}
```

- **Classe TableWidget**

```
using System;
using Gtk;

namespace ShisenSho
{
    public class TableWidget : Table
    {
        private int s;
        private Core c;
        private bool brickChecked;    // True if a brick is checked
        private BrickWidget selected;
        private GameWindow gameWindow;

        public TableWidget (Core core, GameWindow gameWindow, int scale) : base
((uint)core.getBoardHeight () + 2, (uint)core.getBoardWidth () + 2, true)
// +2 is added at both values because we need empty boxes in the outline
        {
            this.c = core;
            this.s = scale;
            this.gameWindow = gameWindow;
            brickChecked = false;
        }

        private void attachBrick (Widget w, int x, int y)
        {
            this.Attach (w, (uint)x, (uint)x + 1, (uint)y, (uint)y + 1);
        }

        public void update ()
        {
            brickChecked = false;
            selected = null;

            //Destroy all childrens
            foreach (Widget child in this.Children) {
                child.Destroy ();
            }
            //add all children
            populateBoard ();
            if (c.getBrickCount () == 0) {
                // Simulation a click on NewGame
                gameWindow.NewGameActivated (null, null);
                gameWindow.NewGamePopup ();
            }
            else
            {
                if (c.getBrickCount () == 4) {
                    /*
                        In this implementation of ShisenSho (6 x 12 tiles with 12 tile
      // types), if the tiles amout to four, they are positioned one next
                        // to the other forming a square and with alternated types, we are
      // in GameOver use case.

                        Here is an example of "no more move possible" tile disposition 12
```

```
            */
            Console.WriteLine ("Checking if game is over");

            for (int x = 1; x < c.getBoardWidth () + 2; x++) {
                for (int y = 1; y < c.getBoardHeight () + 2; y++) {
                    if (this.c.getBrickID (x, y) != Core.NO_BRICK_TYPE &&
                        this.c.getBrickID (x, y) == this.c.getBrickID (x + 1,
                              y + 1) &&
                        this.c.getBrickID (x + 1, y) != Core.NO_BRICK_TYPE &&
                        this.c.getBrickID (x + 1, y) == this.c.getBrickID (x,
                               y + 1))
                    {
                        x = c.getBoardWidth () + 2;
                        y = c.getBoardHeight () + 2;
                        gameWindow.NewGameActivated (null, null);
                        gameWindow.GameOverPopup ();
                    }
                }
            }
        }
        else
            Console.WriteLine (c.getBrickCount ());
    }
}

private void populateBoard ()
{
    // Starting from one because of the outline
    for (int x = 1; x < c.getBoardWidth () + 2; x++) {
        for (int y = 1; y < c.getBoardHeight () + 2; y++) {
            if (this.c.getBrickID (x, y) != Core.NO_BRICK_TYPE)
            {
                BrickWidget b = new BrickWidget (x, y, s,
                                                 this.c.getBrickID (x, y));

                b.ButtonPressEvent += onClick;
                this.attachBrick (b, x, y);
            }
        }
    }
    this.ShowAll ();
}

private void onClick (object obj, ButtonPressEventArgs args)
{
    BrickWidget brick = (BrickWidget)obj;

    if( ((Gdk.EventButton)args.Event).Type == Gdk.EventType.ButtonPress)
    {
        if (!brickChecked) {
            brickChecked = true;
            selected = brick;
            Fixed f = (Fixed)(brick.Child);
            Image check = brick.checkBrick ();
            f.Add (check);
            f.ShowAll ();
        } else {
```

```
                    Fixed f = (Fixed)(selected.Child);
                    if (f.Children.Length > 1)
                        f.Remove (f.Children [1]);

                    if (selected == brick) {
                        brickChecked = false;
                    } else {
                        if (!c.makeMove (selected.getX (), selected.getY (),
                                brick.getX (), brick.getY ())) {
                            selected = brick;
                            f = (Fixed)(brick.Child);
                            Image check = brick.checkBrick ();
                            f.Add (check);
                            f.ShowAll ();
                        } else {
                            selected = null;
                            brickChecked = false;
                            update ();
                        }
                    }
                }
            }
        }
    }
}
```

Infine le classi principali del programma:

- **Classe MainWindow**

```
using System;
using Gtk;

public partial class MainWindow: Gtk.Window
{
    public MainWindow () : base (Gtk.WindowType.Toplevel)
    {
        Build ();
    }

    protected void OnDeleteEvent (object sender, DeleteEventArgs a)
    {
        Application.Quit ();
        a.RetVal = true;
    }
}
```

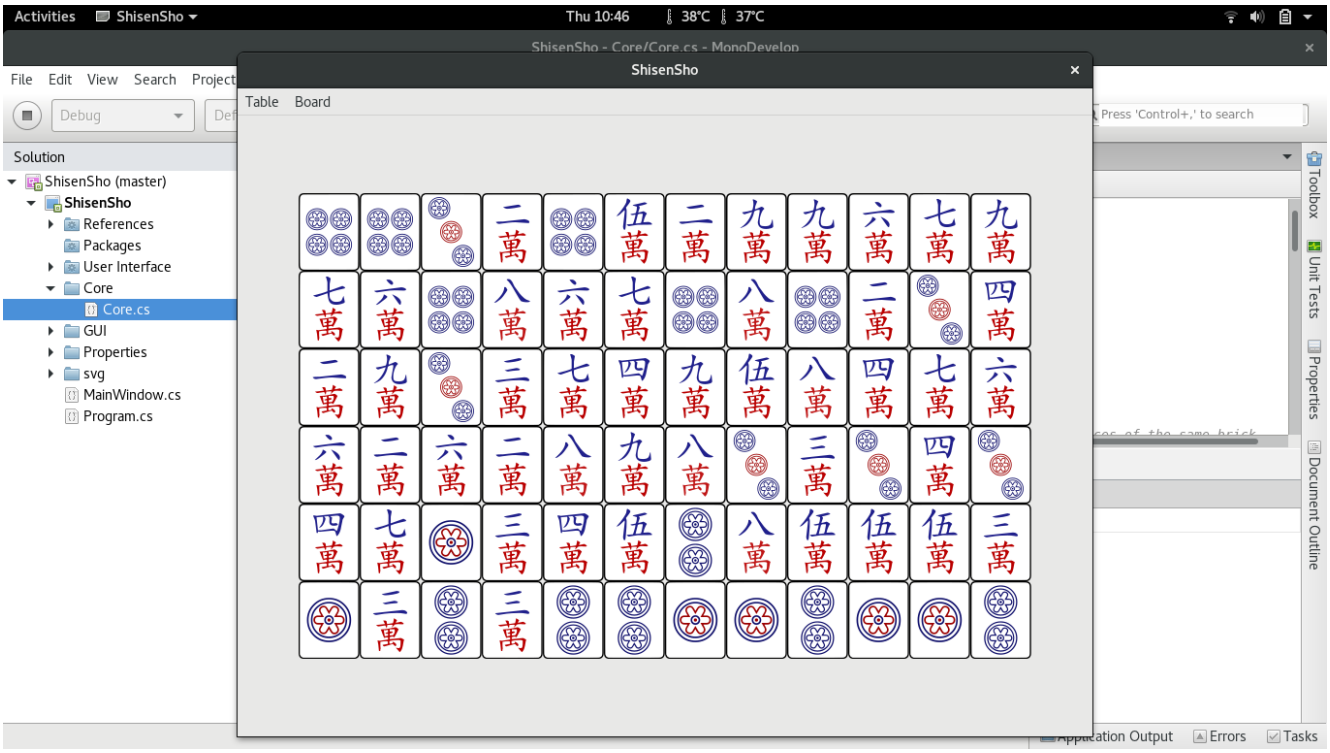
- **Classe Program**

```
using System;
using Gtk;

namespace ShisenSho
{
    class MainClass
    {
        public static void Main (string[] args)
        {
            Application.Init ();

            Core core = new Core (6,12,12);

            GameWindow win = new GameWindow (core);
            win.Show ();

            Application.Run ();
        }
    }
}
```

Test del programma

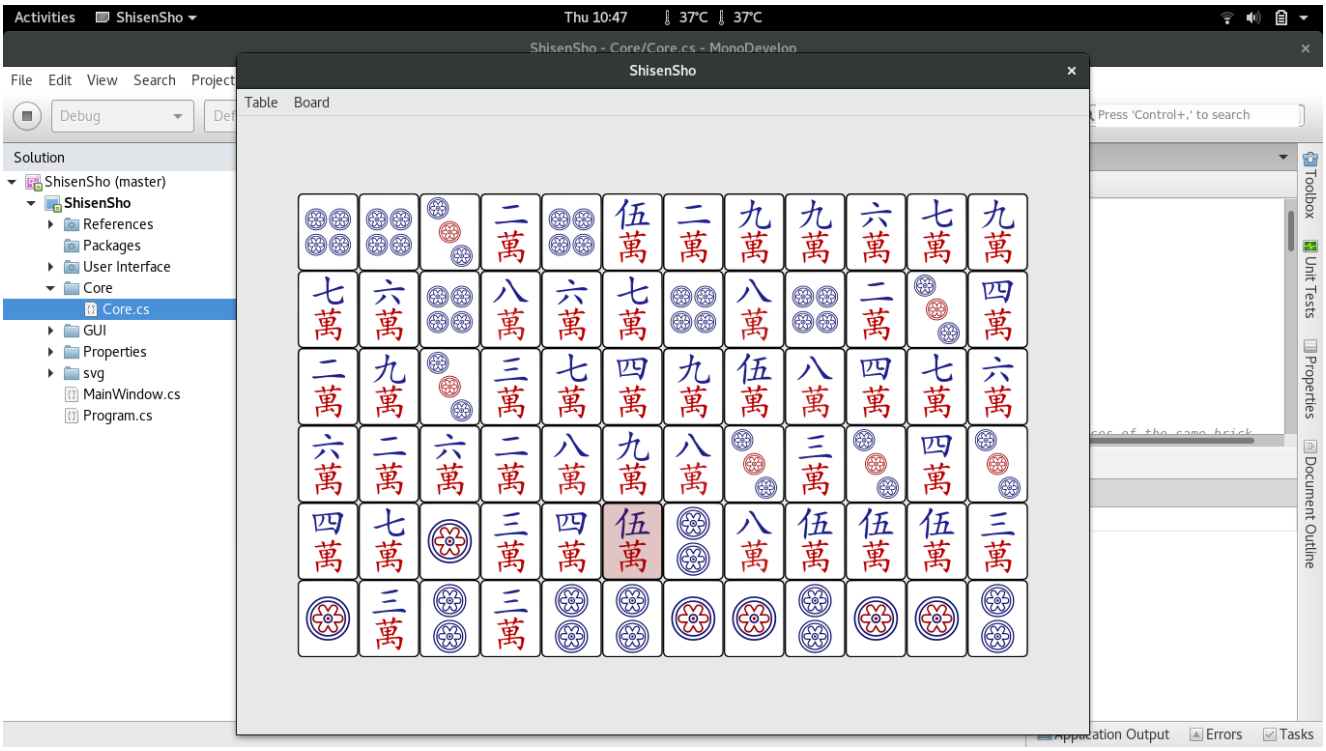
Il programma è stato continuamente testato durante la fase di sviluppo seguendo la metodologia di controllo black-box, la quale consiste nell'esaminare il funzionamento del software senza preoccuparsi di come è stato implementato. Ovvero a seconda dei casi d'uso, Il tester verifica il corretto funzionamento del programma esaminandone gli output e controllando che essi coincidano con le aspettative.

Innanzitutto, una volta lanciato, il programma si presenta come segue:



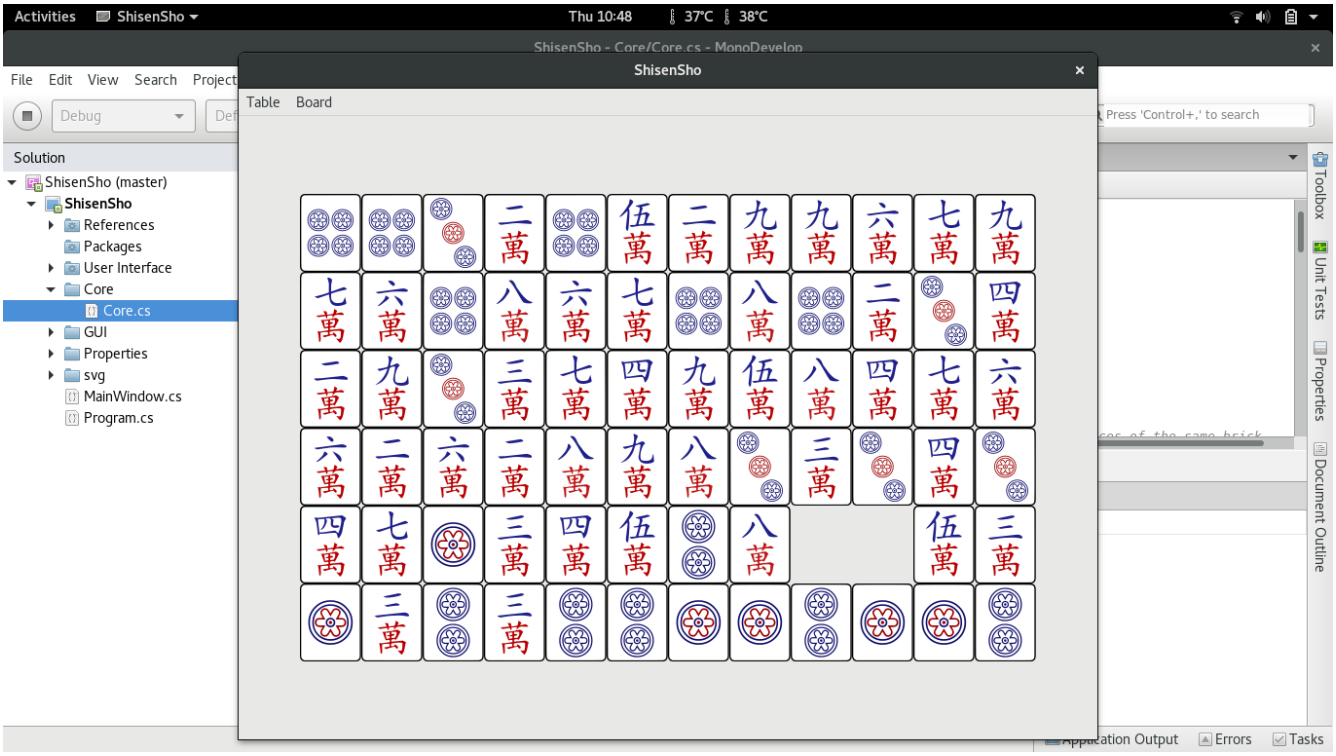
Andiamo ora ad analizzare I vari casi d'uso:

- Tile selezionata (ST), l'utente seleziona una tile ed essa viene evidenziata dalla GUI:

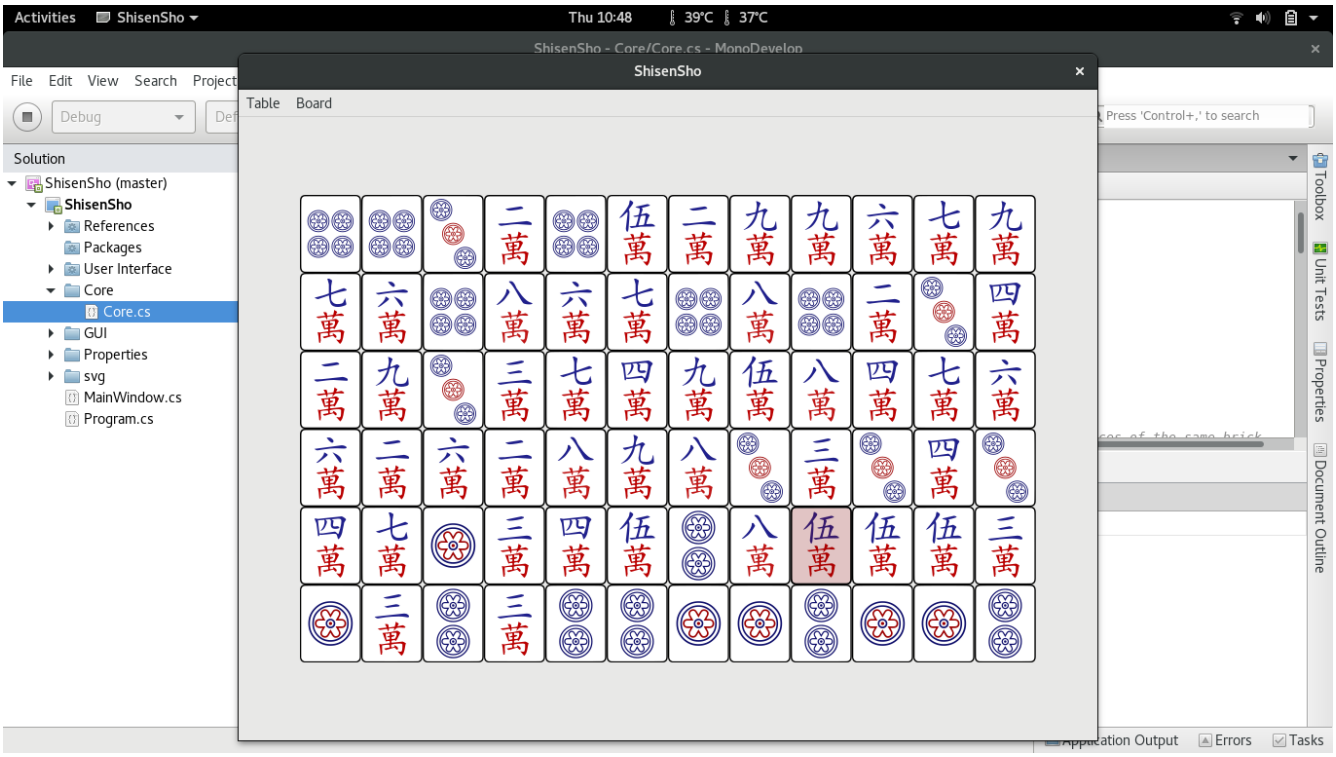


- Tile deselezionata (DT), l'utente deseleziona la tile precedentemente selezionata e quest'ultima viene deselezionata.

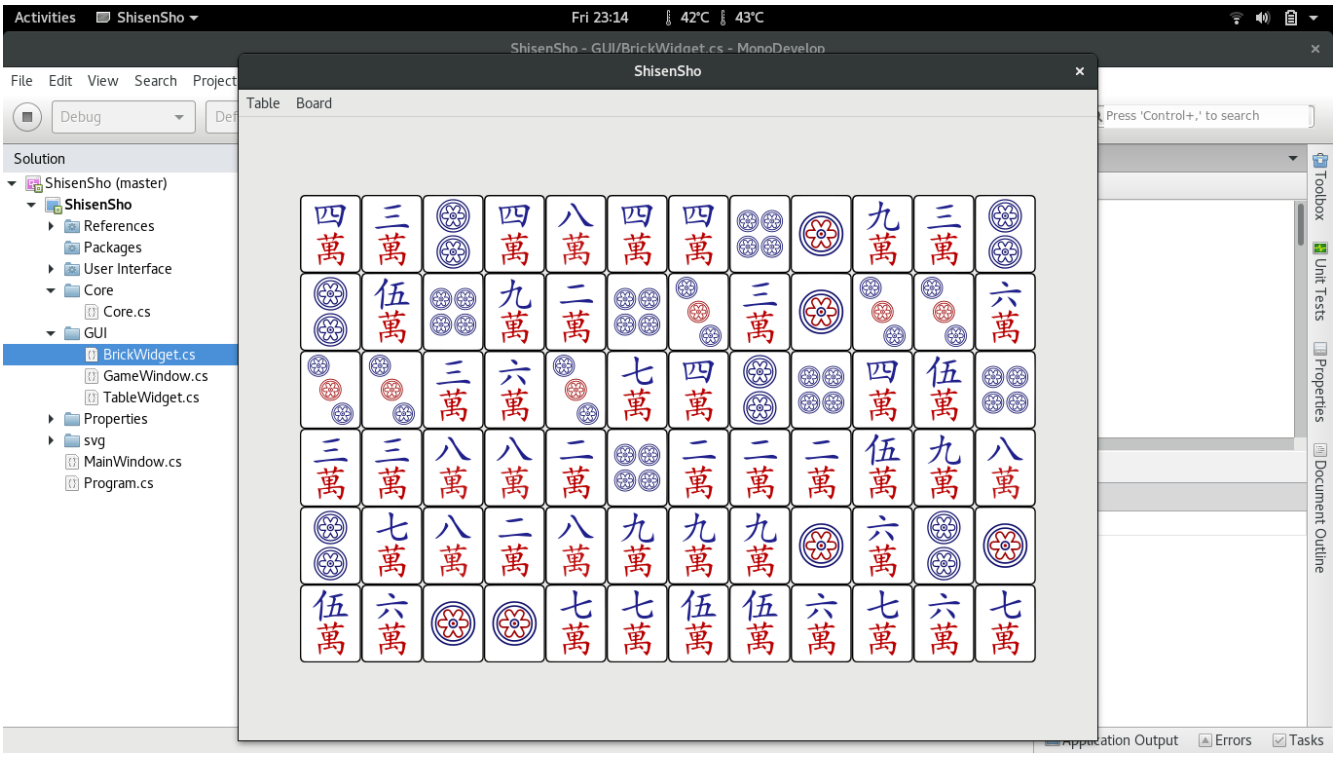
- Selezione seconda tile (EM), vi sono due possibilità: le tile sono uguali, viene eseguita la verifica della mossa ed eventualmente vengono rimosse,



oppure le tile sono diverse e viene deselezionata la prima e selezionata la seconda (di seguito lo screenshot):



- New game (NG), genera un nuovo piano di gioco.



- Scramble (SC), mischia tutte le tile non vuote tra loro, dunque non posiziona delle tile nelle caselle vuote.

Compilazione ed esecuzione

Il software è stato sviluppato e compilato utilizzando l'IDE MonoDevelop 5.10.1 con framework Mono 4.4.0. La scelta è dipesa dalla volontà di sviluppare software libero.

Per eseguire il software è necessario aprire il file di estensione .sln con MonoDevelop e premere il tasto f5 per compilare il codice e lanciare il programma.
Esso è stato compilato e testato su una macchina con sistema operativo Parabola GNU/Linux-libre 64-bit, quad core, 8 GB di RAM e risoluzione 1366x768, e su un MacBook Pro con sistema operativo ArchLinux 64-bit, dual core, 8 GB di RAM, risoluzione 1440x900.