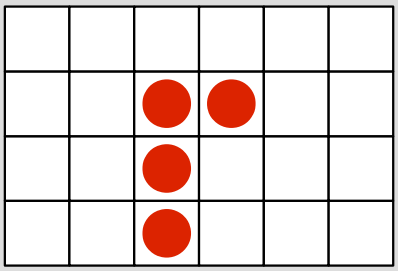
Laboratorio di informatica – classe 4ª – data 17/10/2015

Progetto “Life game”

Si desidera implementare il “motore” del gioco “Life”.

Definizione del gioco

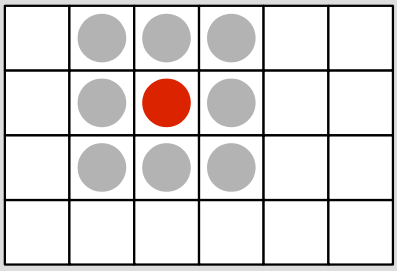
Il gioco simula un mondo che obbedisce ad un semplice sistema di regole. Il mondo può essere rappresentato mediante una griglia bidimensionale di celle; ogni cella può trovarsi negli stati **viva** o **morta**. (In figura ci sono 4 celle vive e 20 morte.)



Evoluzione nel mondo di L*ife*

L'evoluzione è determinata dalla nascita e dalla morte delle celle.

Lo stato futuro di una cella viene determinato da quante celle vive le sono vicine. (Ogni cella è adiacente ad altre 8, eccetto quelle situate sui bordi della griglia.)



Le regole che determinano l'evoluzione di una cella sono:

1. se le celle adiacenti vive sono due, la cella permane nel suo stato (viva o morta che sia);
2. se le celle adiacenti vive sono tre, la cella nasce (o resta viva);
3. in tutti gli altri casi la cella muore (o resta morta).

Evoluzione “simultanea” delle celle

Nel determinate lo stato futuro di una cella, si considera lo stato presente delle celle adiacenti e cioè lo stato che hanno prima di evolversi.

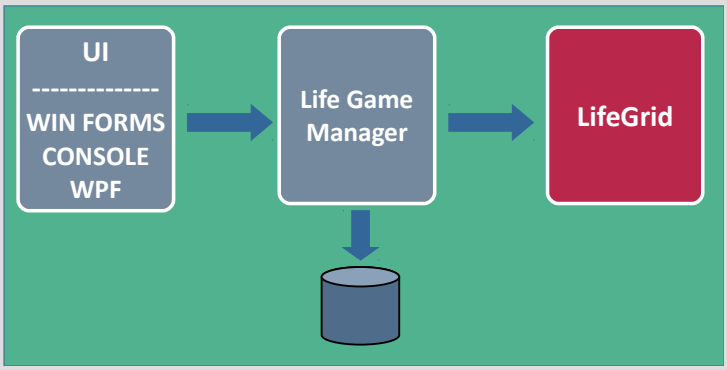
Dunque: l'evoluzione non viene influenzata dall'ordine con il quale sono esaminate le celle.

Stato iniziale di Life ed evoluzione

In *Life* l'evoluzione si misura in numero di **generazioni**.

Lo stato delle celle della "generazione zero" viene stabilito dall'esterno. Dopodiché, ad ogni generazione lo stato delle celle cambia in accordo alle regole descritte in precedenza.

Obiettivo



Si desidera realizzare il "motore del gioco", e cioè una classe che implementi le funzionalità:

1. Impostazione della griglia.
2. Reset della griglia.
3. Impostazione della generazione zero.
4. Evoluzione alla successiva generazione.
5. Comunicazione delle "celle mutate" nel passaggio da una generazione alla successiva.

Note sull'implementazione

Nella sua versione minimale, la classe potrebbe avere la seguente interfaccia pubblica:

```
public class LifeGrid
{
    public  LifeGrid(int rowCount, int colCount) {}
    public void Reset(int[,] generationZeroList) {}
    public CellInfo[] NextGeneration() {}
}
```

Il metodo **Reset()** imposta tutte le celle nello stato "morta". Inoltre imposta su "viva" le celle posizionate alle coordinate specificate nella matrice (generazione zero).

Il tipo **CellInfo** definisce le informazioni su una cella (coordinate e stato).

Il metodo **NextGeneration()** fa "evolvere" le celle in base alle regole del gioco; inoltre, ritorna l'elenco di celle che sono mutate. Questo elenco sarà utilizzato dall'applicazione per aggiornare la UI.

Nota bene: quella proposta è soltanto un'ipotesi. L'interfaccia pubblica può essere anche profondamente diversa; l'importante è che la classe assolva il compito per il quale è progettata (punti 1 – 5).

Unit test

Poiché non esiste una UI con la quale testare il codice, è necessario realizzare uno Unit test.

Nella versione di base ci si può limitare a testare il funzionamento del metodo **NextGeneration()**.

Esempi di scenari di testing

In grigio: a sinistra sono rappresentante le celle che nasceranno, a destra le celle che sono morte nel passaggio alla generazione successiva.