

Sistemi per l'elaborazione dell'informazione 2

Life 2: la vendetta

Forse qualcuno di voi conosce già Life, l'automa cellulare inventato dal matematico John Conway. Dovete scrivere un'applicazione in grado di visualizzare il comportamento di un'istanza di Life tramite viste multiple che rappresentano parti diverse e indipendenti dell'automa con ingrandimenti diversi.

Ricordiamo innanzitutto come funziona Life. L'automa è composto da una matrice $N \times M$ di elementi, dette *cellule*, che possono essere in due stati: vive o morte. Attorno a ogni cellula ci sono otto cellule adiacenti (nord, sud, est, ovest, nord-ovest, nord-est, sud-ovest e sud-est): si noti che la matrice è pensata come un toro, e quindi il bordo destro è adiacente a quello sinistro, così come il bordo superiore è adiacente a quello inferiore. Per esempio, la cellula di coordinate $(0, 0)$ ha a nord la cellula $(0, M - 1)$ e a ovest la cellula $(N - 1, 0)$.

Le regole di evoluzione della vita in Life sono molto semplici: se una cellula è viva, sopravvive all'istante di tempo successivo se nel suo intorno ci sono due o tre cellule vive; altrimenti muore per solitudine o sovraffollamento. Se una cellula è morta e nel suo intorno ci sono esattamente tre cellule vive diventa viva all'istante successivo.

Implementazione

Prima di tutto, *niente panico*. Il disegno MVC permette di dividere la costruzione dell'applicazione in fasi ben definite. Le istruzioni che seguono vi danno le dritte giuste per organizzare l'applicazione, e un po' di lettura attenta dei Javadoc farà il resto.

Dovete scrivere un modello che implementi Life (e che, ovviamente, estenda `Observable`). Il modello prende nel costruttore il numero di righe e di colonne e un `double` tra 0 e 1 che viene utilizzato per inizializzare la matrice (usate la funzione `Math.random()` in modo che ogni cella, indipendentemente dalle altre, risulti viva con probabilità pari al `double` sopra menzionato), e espone un metodo pubblico `nextRound()` che calcola il prossimo stato della matrice utilizzando le regole summenzionate. Il metodo deve inviare una notifica agli osservatori, dato che cambia il contenuto del modello.

Per semplicità ed efficienza, la matrice di stato corrente ha visibilità di pacchetto e viene acceduta direttamente. La classe espone anche due variabili pubbliche finali `rows` e `columns`, di ovvio significato.

Suggerimenti: state attenti all'uso del modulo con i segni negativi, e leggete la documentazione di `System.arraycopy()`, che potrà risultarvi utile nell'implementazione di `nextRound()`.

Se non vi sentite sicuri, aggiungete un metodo `main()` che crea un'istanza e ne stampa l'evoluzione per un numero finito di passi.

Creazione di controlli

Vogliamo avere un'interfaccia utente che presenta i seguenti elementi:

- Uno `Slider` (con valori variabili da 1 a 50) che definisce quanti pixel vengono occupati da ogni cellula.

- Uno **Slider** (con valori variabili da 3 a 1000) che definisce il ritardo in millisecondi con cui viene rallentata l'evoluzione dell'automa (ritardi più piccoli possono dare problemi: provare per credere).
- Un pulsante che apre una nuova vista sullo stesso automa.
- Una zona grafica in cui viene visualizzato l'automa (vedi sotto).

In virtù di queste scelte, il controllo deve osservare i **SelectionEvent** prodotti dai pulsanti e dagli slider. Dato che ci sono diversi controlli con una semantica molto differenziata, conviene che il controllo non implementi direttamente le interfacce, ma piuttosto contenga dei campi del tipo appropriato (**SelectionListener**) che vengono riempiti con un'istanza creata al volo. Per esempio

```
SelectionListener speedListener = new SelectionAdapter() {
    public void widgetSelected(SelectionEvent e) {
        timeInterval = ((Slider)e.getSource()).getSelection();
    }
};
```

timeInterval è l'unico campo (oltre, naturalmente, a un riferimento al modello) del controllo, e contiene un ritardo in millisecondi utilizzato per rallentare l'animazione. È inoltre necessario che il controllo osservi anche gli eventi di chiusura (*dispose*) delle finestre (prendendosi cura di rimuovere il widget dall'elenco degli osservatori registrati sul modello; per i dettagli implementativi leggete la documentazione di **addDisposeListener()** della classe **Shell**). Notate che lo **Slider** che modifica il numero di pixel per cellula va gestito interamente all'interno della vista, dato che non riguarda in alcuno modo il controllo e il modello.

Il metodo **main()** del controller deve istanziare un modello leggendo da riga di comando i tre parametri, istanziare un controllo passandogli il modello e lanciare il metodo **go()** del controllo (analogo a quello visto a lezione).

Canvas

Un **Canvas** è un widget generale in cui potete disegnare, e lo utilizzeremo per visualizzare Life. In generale, per disegnare in un qualunque widget è necessario creare un *contesto grafico*, cioè un'istanza della classe **GC**, passando al costruttore il widget. Dopo l'uso, occorre invocare **dispose()** sul contesto grafico. Il contesto grafico fornisce primitive in grado di disegnare righe, rettangoli, scrivere testo e così via.

Nel vostro caso, non sarà necessario creare un contesto grafico, perché vi verrà fornito automaticamente. Vediamo come.

Per visualizzare un **Canvas** di grandi dimensioni lo si immerge in uno **ScrolledComposite**, che è un widget in grado di mostrare parti di un widget più grande. Lo **ScrolledComposite** deve essere l'ultimo widget aggiunto alla vostra finestra, e la deve occupare interamente. Ecco un modo per ottenere questo effetto (posto che stiate utilizzando un **GridLayout**):

```
ScrolledComposite scrolledcomposite =
    new ScrolledComposite(this, SWT.H_SCROLL | SWT.V_SCROLL);
scrolledcomposite.setLayoutData(new GridData(GridData.FILL_BOTH));
canvas = new Canvas(scrolledcomposite, SWT.NONE);
scrolledcomposite.setContents(canvas);
```

A questo punto vi occorre dimensionare il **Canvas**. Per farlo dovete sapere quanti pixel occupa una cellula, valore che terrete in un campo della vista di nome **size**. Sapendo quante sono le righe e le colonne del modello (che potete recuperare dal controllo), potete dimensionare il **Canvas**:

```
canvas.setSize(size * controller.model.columns, size * controller.model.rows);
```

Quando dovrete disegnare i rettangoli corrispondenti alle cellule? Ovviamente, a ogni cambiamento dello stato del modello, ovvero ogni volta che viene invocato il metodo `update()` della vista, che come vedremo tra poco conviene implementare come sottoclasse di `Shell`. Per ridisegnare una parte del `Canvas` è necessario invocare il metodo `redraw()`. L'effetto è la generazione di un evento di tipo `PaintEvent`, che viene gestito da un `PaintListener` (registrato sul `Canvas`). Il `PaintEvent` contiene un campo `gc` che vi fornisce un contesto grafico pronto per l'uso. In prima istanza, potete creare un `PaintListener` nel controllo che ridisegna tutte le cellule. Il contesto grafico si occuperà di disegnare veramente solo la parte visibile. Per disegnare un rettangolo usate `gc.fillRect()`, che vuole le coordinate e l'ampiezza/altezza del rettangolo. Nota bene: il rettangolo è dipinto col colore di sfondo, che viene modificato da `gc.setBackground()`.

Ovviamente, per disegnare i rettangoli vi serve sapere qual è la vista che ha chiesto di essere ridipinta: non è difficile, perché l'espressione `((Canvas)e.getSource()).getShell()` vi dà la `Shell` a cui appartiene il `Canvas` che vuole essere ridipinto. Siccome le viste sono sottoclassi di `Shell`, a questo punto sarà sufficiente un casting per ottenere la vista da ridipingere. Nota la vista, potete recuperare la dimensione delle cellule (che varia da vista a vista) e disegnare i rettangoli.

Notate che nella gestione dello `Slider` che modifica la dimensione di una cellula (gestione che è interamente interna alla vista) occorrerà aggiornare `size`, ridimensionare il `Canvas` e invocare il metodo `redraw()`, in modo che venga ridipinto alla prima occasione.

Eventi temporizzati

Per ottenere un effetto di animazione, è necessario che ogni `timeInterval` millisecondi il modello venga aggiornato. Per ottenere questo effetto si usa il metodo `timerExec()` di `Display`, che ha come argomento un oggetto la cui classe implementa `Runnable` (quest'ultima interfaccia ha un solo metodo pubblico `run()`) e un ritardo in millisecondi: il metodo `run()` del `Runnable` verrà eseguito dopo il numero di millisecondi dato. Nel vostro caso, per esempio, basta definire nel controllo una variabile di istanza.

```
Runnable timer = new Runnable() {
    public void run() {
        model.nextStep();
        display.timerExec(timeInterval, this);
    }
};
```

Una volta che il modello è creato e inizializzato, dovete lanciare per la prima volta il `Runnable` (`timer.run()`). La prima esecuzione registra se stessa per una esecuzione dopo `timeInterval` millisecondi, e così via all'infinito.

Varianti

La prima modifica importante è l'ottimizzazione del gestore di `PaintEvent`. L'evento contiene le coordinate e le dimensioni del rettangolo che ha bisogno di essere ridipinto (in coordinate interne al `Canvas`). Potete quindi evitare di ridisegnare le cellule che sono fuori dal rettangolo.

Potete poi aggiungere numerosi controlli: per esempio, uno che arresti e faccia ripartire l'animazione (ma attenti alla gestione dell'esecuzione temporizzata) o uno che modifichi il colore delle cellule vive e morte. Potete combinare un controllo di testo in cui l'utente deve inserire un numero tra 0 e 1 con un pulsante di reimpostazione che fa ripartire l'automa da uno stato generato utilizzando la probabilità contenuta nel controllo di testo.

Un'altra variante consiste nel mettere un po' di spaziatura (magari regolabile) tra una cellula e l'altra.