Mastropaolo Marco matr:9715749
Gruppo n° ______

Laboratorio di :

Linguaggi di Programmazione : Paradigmi di Programmazione (Sperimentazioni)

Progetto JBomb

E-Mail di riferimento
marco.mastropaolo@libero.it
st971422@educ.di.unito.it

Descrizione del progetto

Il progetto proposto è un clone del noto gioco "Bomberman", commercializzato su più piattaforme e versioni dalla Hudsonsoft.
In questo gioco il giocatore (qui rappresentato da un pinguino) deve superare numerosi livelli eliminando tutta una serie di mostri che cercheranno di bloccargli la via verso la vittoria. L'unica arma a disposizione del giocatore sono delle bombe che elimineranno tutto ciò che si trova lungo la linea di fuoco (e all'interno della gittata) della bomba che il giocatore deposita. Le bombe sono "a orologeria", vale a dire esplodono dopo un certo numero di secondi dopo che il giocatore l'ha posata a terra, e creano due fiammate - una orizzontale e una verticale - che distruggono tutto ciò che si trova sul percorso.

La plancia di gioco è definita da due tipi di muri, quelli in pietra che sono distruggibili dalle bombe e quelli in marmo che viceversa sono indistruttibili e che non possono essere mai valicati. Essi delimitano il bordo della plancia e i confini interni a quest'ultima. Il pinguino è guidato dai tasti della tastiera. I tasti delle frecce guidano il pinguino, la barra spaziatrice provoca lo sganciamento della bomba (se disponibile) e il tasto P mette in pausa il gioco. Per il resto l'interazione dell'utente col gioco avviene tramite un comodo menù a tendina. L'interfaccia utente dal punto di vista di uso di controlli AWT/Swing è limitata a poche finestre. Ad ogni modo si è scelto per una questione di compatibilità di usare completamente controlli AWT. Questo per soddisfare uno degli obbiettivi che era

mantenere il più possibile la compatibilità con la modalità di funzionamento come applet (obiettivo non pienamente raggiunto a causa dell'uso di collection).

Obiettivi

Gli obiettivi alla base del progetto sono i seguenti :

- Struttura basata su classi con una semplice e logica gerarchia
- Creazione di un package di classi di utilità generica
- Creazione di un package che consenta una semplice implementazione di giochi basati su concetti simili (es. PacMan o labirinti o simili).
- Implementare le varie classi in modo da rendere semplice la creazione di alcune semplici varianti
- Consentire l'uso del gioco come applet e mantenere il più possibile la compatibilità con Java 1.1 (obiettivo raggiunto solo in parte dato l'uso delle collection di Java 2 all'interno del programma - comunque facilmente sostituibili con un vettore allocato dinamicamente)

Gerarchia delle classi e package

Nella gerarchia delle classi si è preferito mantenere sempre un'impostazione più generica possibile. Quando si dovevano condividere parti di codice tra varie classi si è spesso ricorso a una classe astratta. Infine in alcuni casi si è ereditata un'interfaccia pur non essendo necessaria al semplice fine di mantenere una compatibilità con futuri rinnovi della struttura del programma. Ad esempio tutte le JBombCell implementano Tickable - un'interfaccia che definisce un metodo di nome timeTick() che viene chiamata da un Thread separato. Questo non è necessario in quanto le timeTick() delle JBombCell vengono richiamate dal codice del metodo timeTick() di JBombGame. Tuttavia questo lascia la porta aperta a una (seppur dal lato pratico impensabile) soluzione con un thread per ogni JBombCell o ad altre soluzioni che richiedano un'interfaccia Tickable. Essendo la funzione del metodo timeTick() delle JBombCell molto simile a quello dell'interfaccia Tickable la si è implementata anche se non necessario.

Per quanto riguarda i package sono stati generati due package separati : **Utils** che contiene classi di generica utilità utilizzabili in altri progetti e **CellBasedGame** che contiene classi fondamentali per ogni gioco basato su celle.

Gerarchia delle classi

Java.awt.Dialog
Java.awt.Component
Java.lang.
Runnable
InputBox
MsgBox
KeybComponent
TimeController
Java.awt.event.
ActionListener
Java.awt.event.
FocusListener
Tickable
KeybHandler
Java.awt.event.
KeyListener
KeybController
Package Utils

Explodable
CellBasedGame
DynamicObject
Cell
Package CellBasedGame
Bomb
java.awt.Applet
JBombGame
JBombDynamic
JBPlayer
JBombApplet
Monster
MonsterAlien
MonsterWoman
JBombCell
MonsterEnergy
MonsterTrash
JBombWallCell
MonsterFish
MonsterTooth
MonsterGhost
MonsterSpider
JBombBorderCell
MonsterMan
MonsterSkull
JBombGrassCell
JBombBonusFlameWalkCell
java.awt.Frame
Java.awt.event.
ActionListener
JBombBonusCell
Java.awt.event.
WindowListener
JBombBonusBombWalkCell
JBomb
JBombBonusLifeCell
JBombBonusFlameCell
JBombBonusBombCell

Interfaccia
Classe
Classe astratta

Descrizione delle classi

Questa è una breve descrizione delle classi nei vari package. Per una descrizione più approfondita si rimanda alla documentazione Javadoc.

Package Utils

MsgBox : Questa classe implementa una finestra modale che restituisce un messaggio all'utente.
InputBox : Questa classe implementa una finestra modale che chiede all'utente di immettere una String.
KeybController, KeybComponent, *KeybHandler* : La classe KeybController implementa la gestione della tastiera. Essa crea un componente nascosto (KeybComponent) nel Container che riceve come parametro e tramite la KeybComponent intercetta tutti gli eventi di tastiera e ne notifica il KeybHandler. Inoltre gestisce automaticamente la perdita del focus dal KeybComponent, forzandolo nuovamente.
TimeController, *Tickable* : La classe TimeController implementa semplici funzioni per la gestione di un timer. In pratica il metodo timeTick() del Tickable passato come argomento viene richiamato a intervalli di un certo numero di millisecondi. L'intervallo di tempo è approssimativo non essendo la JVM un sistema RealTime.

Package CellBasedGame

CellBasedGame : Questa classe costituisce la base di un gioco basato su celle. Essa gestisce l'array di Cell costituenti il mondo e molti altri aspetti fondamentali come il Double Buffering per la grafica o il paint degli sprites.
Cell : Questa classe astratta è la base per le celle del gioco
DynamicObject : Gestisce oggetti in movimento nel campo di celle. Implementa anche alcune funzioni utili per gli oggetti in movimento dotati di intelligenza artificiale.

Altre classi principali

JBomb : E' la classe che gestisce il frame su cui si disegna il gioco, i menu a tendina, la costruzione e la distruzione degli oggetti JBombGame.
JBombApplet : E' la classe che implementa il funzionamento come Applet
JBombGame : E' la classe principale del gioco. Essa gestisce molti degli aspetti di gioco e contiene parte della logica del gioco.
***Explodable* :** E' un'interfaccia che definisce metodi comuni a tutti gli oggetti che possono esplodere al contatto con la fiamma.
Bomb : E' la classe che gestisce le bombe gettate dal giocatore
JBombCell : Questa classe astratta contiene la definizione di base di tutte le cell del gioco.
JBombWallCell, JBombBorderCell, JBombGrassCell : sono le classi che gestiscono rispettivamente celle di muro, muro di bordo (indistruttibile) e di erba (dove il giocatore può camminare).

JBombBonusCell e classi figlie : JBombBonusCell è una classe che deriva da JBombGrassCell. Essa implementa celle di erba contenenti un bonus che il giocatore può prendere. Se il giocatore prende il bonus o se la cella esplode o viene posata una bomba essa si trasforma immediatamente in una JBombGrassCell e passa l'esecuzione a questa nuova cella. Le classi figlie implementano i vari bonus disponibili al giocatore.
JBombDynamic : Classe astratta che definisce il comportamento comune di tutti gli oggetti dinamici del gioco (mostri e giocatore).
JBPlayer : Classe che contiene la gestione del pinguino rappresentante il giocatore.
Monster e classi figlie : Monster è la classe che definisce l'implementazione base dei mostri nel gioco. Le classi figlie implementano i vari tipi di mostri presenti.

Eventi gestiti

Gli eventi (intesi come eventi AWT) gestiti riguardano principalmente i menù a tendina (gestiti nella classe JBomb) e la chiusura delle MsgBox e InputBox (nelle rispettive classi). Inoltre nella classe KeybController vengono gestiti gli eventi di Focus e di Tastiera al fine di passare al *KeybHandler* gli eventi riguardanti la tastiera.
In particolare vengono gestiti :

- in JBomb
  - windowClosing per terminare correttamente i thread e gli oggetti JBombGame attivi
  - windowDeactivated e windowIconified per mettere il gioco automaticamente in pausa
  - actionPerformed per i menu a tendina
- in InputBox e MsgBox
  - actionPerformed per la chiusura della finestra
- in KeybController
  - keyPressed, keyReleased e keyTyped per la comunicazione degli eventi di tastiera
  - focusLost per la riacquisizione del focus

Thread

Come già accennato e come si vedrà dopo nella parte di "Descrizione del funzionamento" esiste un secondo thread parallelo a quello principale che gestisce degli eventi a tempo grazie al TimeController. Questo thread era peraltro evitabile (il codice poteva essere tutto in una procedura di paint con un Repaint finale) ma la soluzione del codice nella paint non era soddisfacente sia dal punto di vista dell'organizzazione sia di quello dell'efficienza. Alcune parti di codice di gestione del gioco sono tuttora nelle paint, in quanto codice che era necessario eseguire dal thread principale.

Librerie utilizzate

Sono state - come già detto - utilizzate le classi AWT (java.awt e java.awt.event), le classi delle Collections (in particolare LinkedList e Iterator) e ovviamente le classi standard java.Lang.

Descrizione del funzionamento

L'esecuzione del programma parte dalla classe JBomb. Essa crea e gestisce un frame di dimensioni (area client) di 512x416 (tale da essere contenuta sulla totalità dei dispositivi video). Questo frame contiene una MenuBar che offre un comodo menù a tendina con le funzioni principali. Inoltre essa crea (su richiesta da menù e alla partenza) un oggetto JBombGame su cui focalizzeremo la nostra attenzione. Essa non viene più utilizzata se non per il repaint (ma si limita a chiamare la JBombGame.repaint()) e per l'uso dei menù a tendina.

JBombGame crea quindi il CellBasedGame (costruttore super), il Player, una LinkedList per i mostri e i dati del livello (celle con le loro posizioni etc.).
CellBasedGame crea a sua volta il KeybController e il TimeController con cui si gestisce il tutto. Oltre a fornire funzioni base il CellBasedGame chiama a ogni timeTick un metodo (eachFrame o eachPausedFrame - dipendentemente dallo stato del gioco) in JBombGame.

Ad ogni tick del gioco (erroneamente chiamata eachFrame in quanto avviene a ogni tick e non a ogni repaint) JBombGame esegue la timeTick di ogni cella, muove il giocatore e i vari mostri, e controlla la morte del giocatore per contatto con un mostro. Ad ogni repaint invece controlla se deve ricostruire il mondo (e lo ricostruisce se necessario) e disegna - sopra le Cell già ridisegnate da CellBasedGame - il giocatore e i vari mostri. Nel caso il gioco sia in uno stato di game over (partita vinta o persa) provvede a disegnare due scritte scorrevoli che mostrano lo stato del gioco e il nome del creatore.

Il resto della logica di gioco è dovuto "solo" alle interazioni fra i vari oggetti. Per esempio quando una bomba esplode provvede a chiedere a JBombGame un'array di oggetti che possono esplodere e a chiamare opportuni metodi su questi oggetti. Oppure qualunque DynamicObject che passi su una JBombCell chiama il metodo walkOver() di quella JBombCell. Interessante è l'implementazione dei DynamicObject. DynamicObject offre ai sui eredi metodi con cui muoversi lungo una data direzione e alcuni metodi che offrono decisioni automatiche sulla direzione. Questo consente di implementare oggetti dinamici anche molto diversi fra loro con poco codice. Infatti ogni erede deve implementare un metodo chooseDirection() che restituisce in un intero la direzione prescelta da quell'oggetto per quel ciclo. Un oggetto che reagisce alla pressione di tasti (il JBPlayer nel nostro caso) implementerà chooseDirection con un codice simile al seguente :

```
protected int chooseDirection ()
{
        if (m_bKeyDown) return DIRECTION_DOWN;
        if (m_bKeyRight) return DIRECTION_RIGHT;
        if (m_bKeyLeft) return DIRECTION_LEFT;
        if (m_bKeyUp)  return DIRECTION_UP;
        return DIRECTION_NONE;
}
```

Un oggetto con intelligenza potrebbe invece eseguire codice come :

```
protected int chooseDirection ()
{
        return chase(m_jbGame.getPlayer());
}
```

dove chase è un metodo di DynamicObject che offre un discreto algoritmo di cammino minimo tra due DynamicObject.

In particolare le interazioni di maggiore importanza gestite dai vari oggetti sono le seguenti (le interazioni di una classe padre vale ovviamente anche per tutte le classi figlie):

Bomb : La classe Bomb contiene un metodo statico per la creazione di una nuova bomba (dropBomb). Questo codice è stato messo nella classe Bomb e non in JBombGame per una questione di possibili modifiche future. Questa funzione usa RTTI per capire se la JBombCell in uso è effettivamente una cella di tipo JBombGrassCell (l'unica che può ospitare bombe) e in caso positivo crea una nuova bomba. Alla creazione di un oggetto Bomb viene decrementato il contatore d'uso delle bombe. Al momento dell'esplosione questo contatore viene reincrementato e vengono fatte esplodere le celle opportune. Poiché l'esplosione è ricorsiva una variabile booleana ci assicura di non creare un ciclo infinito (o, meglio, finito ma che causa uno Stack Fault). La Bomb è a tempo e usa un contatore che viene decrementato alla chiamata del suo timeTick().

JBombWallCell : JBombWallCell è invece in grado grazie al metodo changeSelf() di Cell di cambiare se stessa in una JBombGrassCell quando esplode. Essa si trasforma in una Cell diversa a seconda di un numero casuale secondo una tabella di probabilità che segue :

Probabilità	Classe
70%	JBombGrassCell
10%	JBombBonusFlameCell
10%	JBombBonusBombCell
3%	JBombBonusLifeCell
3%	JBombBonusBombWalkCell
4%	JBombBonusFlameWalkCell

JBombGrassCell : JBombGrassCell mantiene sia la gestione della cella vuota che quella del fuoco e della bomba. Alla richiesta se JBombGrassCell blocca o meno un DynamicObject la JBombGrassCell interroga direttamente il JBombDynamic e controlla se ha o meno bombe o fiamme, rivelando quindi se è da considerarsi una cella bloccante o meno. JBombGrassCell inoltre passa il timeTick() alla Bomb che contiene (nel caso ne contenga una) e dopo un certo periodo estingue il fuoco creatosi da un'esplosione. In risposta a un'esplosione "si incendia" e in caso vi sia il fuoco uccide ogni DynamicObject che ne effettuano il walkOver (chiamata al metodo DynamicObject.dye()).

JBombBonusCell : JBombBonusCell deriva direttamente da JBombGrassCell. Essa non ha una vera implementazione di una JBombGrassCell, ma si limita nel caso di esplosioni, lancio di bombe o walkOver del giocatore a usare changeSelf per trasformarsi in una JBombGrassCell regolare. Essa stessa provvede a chiamare i metodi della nuova JBombGrassCell. Per riconoscere il JBPlayer da un altro DynamicObject viene ancora una volta usata RTTI.

JBombDynamic : JBombDynamic definisce oltre ai metodi già presenti in DynamicObject altri 4 metodi che rendono possibile l'uso di particolari privilegi (immunità al fuoco ad

esempio) ad alcuni oggetti dinamici. Oltre a questo ridefinisce il metodo move() con una nuova versione che dopo aver chiamato la versione precedente (tramite super.move() ovviamente) chiama anche il walkOver della cella appropriata.

JBPlayer : JBPlayer non è nulla di più che un'implementazione di un JBombDynamic che usa i tasti della tastiera come metodo di scelta della direzione. Il suo metodo dye() richiama il metodo gameOver() di JBombGame che decrementa il contatore vite e eventualmente mette il gioco in uno stato di gameOver.

Monster : Infine Monster è una classe astratta di supporto per tutti i mostri. Esso non è nulla più che un normale JBombDynamic.

MonsterTooth : Sebbene non sia un oggetto di importanza per la logica del gioco, esso offre una caratteristica particolare. Grazie ai metodi di JBombDynamic esso si dichiara come in grado di attraversare i muri. La move() di questa classe è ridefinita per far esplodere la cella su cui cammina se questa è una JBombWallCell (identificata tramite RTTI), dando così l'impressione che questo mostro "mangi" i muri.

Problemi riscontrati nella realizzazione

Il primo problema che si è riscontrato nella realizzazione di questo progetto è stato un problema di velocità. Originalmente il ciclo principale doveva essere definito nella paint con un metodo come :

```
public void paint (Graphics gfx)
{
        eachFrame(); // esegue le azioni
        customPaint(); // esegue il paint
        repaint();
}
```

tuttavia i cicli erano intervallati da lunghe pause (lunghe per un gioco in tempo reale) e non offrivano una buona giocabilità. Pertanto si è dovuto ricorrere a un thread separato, che però non ha risolto tutti i problemi. Difatti ancora l'esecuzione a volte non è cosi fluida. In realtà si è notato come questo sia probabilmente dovuto al modo con cui si intervallano i thread tra loro. Il risultato ottimale si ottiene con un alternarsi perfetto dei thread. Inoltre sulla VM della Sun nella macchina di prova stranamente questo alternarsi ideale viene raggiunto solo alla seconda creazione di un oggetto JBombGame. In pratica eseguendo il gioco si ottiene un alternanza non ottimale, ma azionando il menù apposito e creando un nuovo gioco si ottiene l'alternanza ottimale. Sulla VM Microsoft si ottiene sempre quella ottimale. Una soluzione finale potrebbe essere quella di sincronizzare i thread a eseguirsi forzatamente in maniera alternata (con una variabile booleana per esempio) ma questo potrebbe causare rallentamenti nella logica di gioco su sistemi che effettuano il repaint più lentamente. Ad ogni modo ho introdotto all'interno del ciclo di paint una System.out.print("*") e all'interno del ciclo di timeTick una System.out.print(".") in modo da poter osservare l'alternarsi dei thread. Il risultato ottenuto (VM Sun) è il seguente :

```
Prima esecuzione :
.*....*****.***.*.*.*.*.*.*.*.*.**.*.*.*..**.*.*..*.**.*..**.*.*..**.*.*.*.*.*.*
.*.*.*.*.*.*.*..*.*.**.*..**.*..*.*.*.*.*.**.*.*.*..**.*..*.**.*..*.*.*.*.**..**
..*.*.**..*.*.**.*.*.*..**..*.*.*.**.*.*.*..*.*.**..*.*.**.*.*.*..**.*.*.*..**.*
Seconda esecuzione :
*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.
*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.
*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.
```

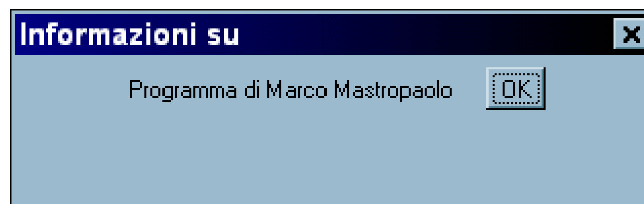
da cui si può notare il differente alternarsi dei thread, più casuale alla prima esecuzione, ben regolato alla seconda.

Un secondo problema è nel funzionamento come applet o in generale nel funzionamento su alcuni sistemi non-Windows (esempio VM Kaffe sotto Linux - mentre sempre sotto Linux funziona perfettamente la VM Sun) dovuti principalmente a un non sempre corretto caricamento del file di immagine "grass.jpg". Questo causa però delle grosse anomalie grafiche durante l'esecuzione.

Un terzo problema è la velocità di esecuzione. Non solo è sicuramente inferiore a quella di altri linguaggi - pur rimanendo accettabile - ma non è per nulla prevedibile. L'influenza della VM è troppo alta (a seconda della VM e del sistema operativo sottostante l'esecuzione cambia completamente). Questo significa anche che per ora il software

sviluppato in Java si comporta troppo differentemente sulle varie VM per poter essere considerato veramente portabile. Per esempio anche il disegno delle Dialog Box (MsgBox e InputBox) pur essendo funzionante su tutti i sistemi provati, potrebbe non offrire coeerenza di aspetto fra le varie VM. Per esempio la MsgBox del gioco si rivela con questi aspetti differenti in una VM Sun e VM Microsoft (sotto Windows 2000SP1) :

VM Sun (JRE v.1.2.2)



VM Microsoft (v 5.00.3310)

differenze grafiche di poco conto, ma che su maschere più complesse potrebbero avere il loro peso - soprattutto se si pensa che entrambe sono in ambiente Windows e che in ambienti
Apple MacOS, X-Windows, OS/2 e BeOS (solo per elencarne alcuni) potrebbero esserci differenze di peso ben maggiore.

Un ulteriore problema è dato dalle librerie. Infatti la VM Microsoft si è più volte rifiutata di far funzionare il programma a causa della presenza dell'uso delle collections (nonostante sia una JVM Java2) e lo stesso avrebbe fatto per le Swing - se fossero state usate. In questo progetto l'uso delle collections è perfettamente evitabile (è sufficiente un vettore dimensionato dinamicamente) ma per progetti di dimensioni maggiori potrebbero essere necessarie. Sebbene sia possibile scaricare i file jar di supporto per queste classi (o sia anche possibile includerli in un sito web - nel caso di applet - per renderla utilizzabile da tutte le VM) non è una soluzione accettabile : o si costringe l'utente a dover installare file aggiuntivi dalla non semplice installazione o lo si costringe a lunghe attese affinché tutte le classi necessarie siano scaricate dal sito web (questo è il motivo per cui non sono state utilizzate le Swing : mentre l'uso della collection può essere rimpiazzato in un pochi minuti con un vettore, l'uso delle Swing per le dialog avrebbe comportato il non poter utilizzare questo progetto come applet per il web se non dopo consistenti modifiche).

Ancora, Java si è rivelato incapace di prendere input da tastiera senza l'uso di un controllo AWT. Per cui si è dovuto implementare un controllo nascosto (KeybComponent) al solo scopo di ottenere l'input di tastiera.

Un ultimo problema riscontrato e risolto è l'aver dovuto inserire tutta la fase di timeTick e di move come synchronized. Sebbene non ci sia la necessità (non si hanno inconsistenze di dati o crash o altro) un esecuzione di una paint durante il ciclo principale causava alcuni artefatti grafici (dovuti allo spostamento del viewport non immediatamente notificata agli oggetti che effettuano il repaint).

Confronto con altri linguaggi

Il confronto con altri linguaggi rende giustizia a Java sulla facilità d'uso. E' veramente difficile commettere errori a cui non si riesca a risalire con semplici sessioni di debug, contrariamente ad altri linguaggi. Questo grazie sia alla restrittiva gestione degli oggetti (che crea limitazioni ma che obbliga anche a scrivere codice il più possibile "pulito") sia alla gestione della memoria. Il lato in cui Java soffre maggiormente è quello delle prestazioni, soprattutto se confrontato con il C++, linguaggio a cui assomiglia per molti aspetti.

Una particolarità in cui il C++ è inevitabilmente migliore è forse nella gestione dei distruttori. Purtroppo il metodo di garbage collection di Java non offre alternative a quella esistente di un distruttore chiamato posticipatamente, tuttavia la possibilità di avere un distruttore richiamato automaticamente - e immediatamente - alla distruzione dell'oggetto sarebbe utile. D'altronde, qualora necessario, chiamare un metodo dispose() per un certo oggetto non è differente dal chiamare *delete* su quell'oggetto. La differenza è nel fatto che mentre in C++ è "automatico" per il programmatore eseguire il delete degli oggetti non più usati, in Java solo in casi particolari serve chiamare un metodo tipo dispose o altro per cui è più facile dimenticarsene.

Per il resto non si può non dire che Java non sia migliore sotto moltissimi aspetti, soprattutto sintattici. Per esempio RTTI in Java è molto più semplice (se non sono richieste identificazioni particolari) di come è in C++. A vantaggio di Java è anche la presenza delle interfacce che permettono un'ereditarietà multipla molto più semplice e efficacie (e con minori problemi) di quella standard offerta dal C++. Sempre a vantaggio di Java il fatto che per default i metodi sono ereditabili e modificabili contrariamente al C++ dove i metodi che potranno essere ridefiniti da un erede devono essere dichiarati "virtual" in fase di definizione della classe padre.

Una curiosa mancanza invece di Java è la mancanza di un precompilatore. In molte situazioni avrebbe fatto comodo poter definire semplici macro con una #define o poter includere/escludere parti di codice con direttive di compilazione come #if o #ifdef. Questo certo avrebbe forse portato rischi di abusi delle macro con conseguente perdità di leggibilità del codice, ma sicuramente in molte situazioni avrebbe potuto essere uno strumento molto utile.

Il confronto con linguaggi procedurali come il C o il Pascal fa risaltare invece la potenza della gestione ad oggetti. Una qualunque soluzione a oggetti (sia in Java che in C++ che Object Pascal o altro) ha dei grandi vantaggi dal punto di vista della leggibilità e della semplicità del codice. Il confronto con altri linguaggi non procedurali come Scheme o Prolog non si può fare, visto che si tratta di linguaggi interpretati e che richiedono interventi dell'utente.

Il supporto della libreria standard, per quanto a volte crei dei vincoli o non preveda usi particolari del linguaggio (come nel caso dell'input di tastiera) è comunque di grande aiuto per ottenere un codice omogeneo, anche se un metodo semplice di gestione degli errori di caricamento delle Image avrebbe aiutato maggiormente.

Note sul riuso del codice

Il codice contenuto nei package Utils e CellBasedGame non è legato all'architettura particolare del gioco. Infatti si tratta di codice facilente riusabile in futuri progetti.

Il package Utils contiene classi che nulla hanno a che fare con la logica di gioco e possono essere utilizzate anche in progetti molto diversi. InputBox e MsgBox altro non sono che generiche finestre di dialogo inseribili facilmente ovunque sia necessario. TimeController può essere utilizzata in molti contesti differenti (per esempio può essere utilizzata in un modo simile al controllo Timer di Visual Basic o al TTimer di Delphi/C++ Builder). KeybContoller è invece un po' particolare e probabilmente trova meno applicazioni delle altre classi, mantenendo comunque una certa generalità.

Il package CellBasedGame invece offre codice che può essere utilizzato come base per qualunque gioco basato su celle. Offre numerose funzionalità (scrolling con effetto parallasse, clipping, back buffering, intelligenza artificiale di base per gli oggetti mobili, etc.) senza porre particolari vincoli alla logica del gioco.

Come visto questi package offrono un codice molto riusabile in nuovi progetti. Questo senza un design a oggetti, senza l'incapsulamento e l'information hiding, non sarebbe probabilmente stato possibile.

Conclusioni

In conclusione, un buon design a oggetti coadiuvato da un linguaggio che ne permetta la realizzazione è sicuramente un aiuto enorme al programmatore in termini di semplicità di codifica e modifica quanto in termini di riuso del codice. Il tempo di coding è stato relativamente basso - rispetto se non altro a quanto avrebbe richiesto con un linguaggio di programmazione più tradizionale come il C o il Pascal.