Relazione BubbleBobble

Mattia Maiorano, progetto svolto individualmente
Matricola: 2009650, canale M-Z

sommario

La relazione si struttura in macro aree che affrontano i vari temi richiesti per il progetto:

- *MVC* (implementazione contestualizzata al progetto)
- *Interfacce* (breve resoconto su quelle importate)
- *Observer-Observable* (implementazione contestualizzata al progetto)
- *Game Loop* (struttura del gioco)
- *Stream* (luoghi d'applicazione)
- *Leaderboard e utente* (come viene gestito il salvataggio delle informazioni)
- *Classe Sound* (gestione comparto sonoro)
- *Altri Design Pattern* (singleton)

Queste sono poi, se ritenuto necessario, suddivise a loro volta in sezioni più piccole al fine della semplicità di lettura.

mvc

Come da richiesta il progetto è stato strutturato seguendo il design pattern MVC, al fine di garantirne la modularità, suddividendo quindi come da norma il progetto nei tre package Model, View, Controller.
È stata poi posta attenzione alla completa riusabilità del Model attraverso l'interfaccia Observer-Observable, la quale rappresenta l'unico legame fra il model e la view.
Di seguito in maniera più approfondita i vari moduli.

Controller

Il controller contiene al suo interno tre classi:

- *Main*
- *GamePanel*
- *KeyHandler*

Il Main è una delle classi più snelle del gioco e si occupa sostanzialmente di creare un oggetto GamePanel.

Il GamePanel è il cuore del programma ed è infatti la sezione di codice dove, oltre che venire generate le classi Model e View (interne ai rispettivi package e instanziate entrambe come **singleton**) viene dato il tempo ai frame di aggiornamento del gioco, nello specifico 60 FPS.

L'ultima classe è KeyHandler dove vengono gestiti gli input da tastiera, tasti premuti e rilasciati.

Come da norma del design pattern dunque, il controller agisce da ponte fra Model e View senza svolgere ruoli diversi da quello amministrativo e di trasferimento dati in input al Model.

Model

Il Model è per sua natura il package con più righe di codice in quanto contiene tutta la logica del gioco, anche se il numero di classi, anch'esse abbastanza numerose, è inferiore a quello della View.

Model e View contengono quasi lo stesso numero di classi
poiché ogni classe nel Model mantiene un corrispettivo visivo nella View,
ad eccezione della classe sound presente solo nel secondo

La classe principale è chiamata anch'essa Model e si occupa sia di generare alcuni oggetti principali come il player e l'oggetto stage (implementata attraverso **singleton** e generatrice a sua volta degli oggetti propri del livello) che di mantenere un ArrayList di tutte le entità e chiamarne l'update.
Fornisce inoltre dei metodi di controllo delle collisioni ed è effettivamente l'"**osservabile**" del progetto, ovvero la classe incaricata di estendere la classe Observable e dunque essere osservata da coloro iscritti come observer.

Il protocollo di dialogo fra Model e View è affrontato in dettaglio
nella sezione Observer-Observable

Continuando all'interno del package troviamo poi un'altra classe fondamentale che è ConceptModel, da cui tutte le altre classi (ad eccezione di Model) ereditano variabili di base e metodi da implementare, è infatti una **classe astratta.**

Oltre queste classi più generali, troviamo poi tutte quelle riguardanti i singoli nemici, i power-up, il player e così via.

Al fine di rispettare le norme del design pattern, è importante specificare che le classi sopra elencate si occupano esclusivamente di svolgere la logica degli oggetti da esse rappresentati, non vi è presente infatti alcun caricamento di immagini o disposizione di draw, i quali verranno svolti dalla View.

View

L'ultimo package è quello della View, che, nonostante l'elevato numero di classi, rimane comunque abbastanza leggero poiché esente dalla logica, in esso troviamo infatti solamente il caricamento e il disegno delle classi.

Anche nel package View troviamo un'omonima classe, anch'essa instanziata come **singleton**, incaricata della generazione del pannello, attraverso **javax.Swing** e di alcuni oggetti principali.
Inoltre questa classe fornisce dei metodi di aggiunta e rimozione di **observer** nella lista mantenuta dal model, ma non implementa essa stessa l'interfaccia (questi metodi sono infatti usati dalle altre classi effettivamente interessate nel disegno a schermo).

la scelta della GUI è ricaduta su swing poiché più semplice e stabile

Anche qui troviamo una **classe astratta** chiamata ConceptView, con variabili di base e metodi da implementare, da cui ereditano tutte le altre classi ad eccezione della View (quindi le classi del player, dei nemici, ma anche del menu e dello stage).
È proprio la classe ConceptView, e dunque le sue classi figlie, a implementare Observer e ad iscriversi alla lista notificata dal model.

interfacce

Runnable

È implementata all'interno del Controller dalla classe **GamePanel** e si occupa del thread di gioco, attraverso l'oggetto Thread thread. La chiamata a thread.start() invoca infatti il metodo run dell'interfaccia runnable.

KeyListener

È implementata all'interno del Controller dalla classe **KeyHandler**, si occupa di ricevere gli input da tastiera, sia al momento di premuta che di rilascio di un tasto.

Observer

È implementata all'interno della View dalla classe **ConceptView** e da tutte le classi che ereditano da questa, è necessaria per rendere un oggetto sottoscrivibile alla lista di observer matenuta da un oggetto observable.

Observer-Observable

Uno dei design pattern più famosi e più usati, è stato implementato come da richiesta all'interno del progetto in coesione col pattern MVC.
L'Observable si trova infatti nel Model ed è l'unico collegamento fra i dati della logica e la rappresentazione a schermo della View, la quale li riceve attraverso i suoi Observer.

Observable

La classe che estende Observable è la **classe Model**, in questa è infatti contenuta una map nella quale ogni entità scrive, in corrispondenza al suo nome che funge da chiave univoca, i suoi valori all'interno di un int array. Questi possono rappresentare la posizione a schermo, se l'entità è stata colpita o meno, se va rimossa o il verso in cui si muove.
La notifica avviene attraverso il metodo **notifyObservers(Object arg)** ereditato da Observable alla fine della chiamata ad update di tutte le entità.

Observers

Le classi che implementano Observer si trovano invece nella View e sono **tutte le classi figlie di ConceptView**, le quali come obbligato dall'interfaccia implementano tutte il metodo **update(Object arg)**.

Questo, una volta chiamato notifyObservers, riceve la mappa con all'interno i dati della logica, i quali sono solamente da interpretare per la rappresentazione a schermo delle immagini.

game loop

Con game loop si vuole intendere, in maniera un po' impropria, l'alternarsi del menu, dei livelli e in generale delle varie fasi di gioco.
Di seguito verrà infatti analizzato il flusso di gioco dall'avvio del programma alla sua chiusura.

Insert Name e Opening

Il gioco prima di iniziare richiede da parte dell'utente la selezione del nome, attraverso un metodo di selezione di tre lettere che ricorda la modalità arcade (ovvero si scorre su e giù per muoversi fra le lettere in ordine alfabetico).
Una volta scelto e premuto enter si avvia la sequenza di introduzione che in maniera automatica ci porta al **menu principale**.

Questa sequenza è gestita, per quanto riguarda la logica indipendente del computer, dall'oggetto stage, mentre per quanto riguarda la gestione dell'input utente dall'oggetto menu (quindi ad esempio il premere i tasti per selezionare il nome).

Menu e Leaderboard

Arrivati al menu principale abbiamo due opzioni, premere lo start del gioco oppure immettere una password, in entrambi i casi si inizia a giocare e l'oggetto stage genera le entità. Nel caso della password i dati inseriti vengono convertiti in un livello e in un punteggio.

Se all'interno del menu si preme il tasto "l" invece, otteniamo la **leaderboard** degli utenti che si siano mai loggati nel gioco sul computer.
Per tornare al menu basterà premere il tasto escape.

Livello

Selezionato il livello l'oggetto stage carica una matrice di interi indicanti quale entità generare e prepara il quadro. Questo può essere concluso attraverso tre modi:
- la premuta del tasto "esc", la quale ci riporta al menu principale
- vincendo il livello
- terminando le vite, quest'ultima situazione ci porta al game over

Alla vittoria del livello scatta una sequenza sonora e visiva
utile per mascherare il caricamento del quadro successivo

GameOver

Una volta terminate le vite l'oggetto stage si occuperà di rimuovere tutte le entità dalle varie liste e dagli observers e mostrerà la schermata di game over, con annessa password generata criptando (in maniera inversa a quella del menu) il livello e il punteggio ottenuti. In seguito alla pressione di enter sarà possibile ritentare il livello, in seguito alla pressione di escape si tornerà al menu principale.

Nemici

I nemici incontrabili sono di tre tipi: **zen-chan**, **mighta** e **monsta**.
Il primo è un robottino che salta se non incontra piattaforme davanti a lui, il secondo segue lo stesso tipo di movimento ma genera delle rocce alla scadenza di un timer e il terzo è un fantasma viola con movimento diagonale (tutti e tre provocano danno da contatto).

Bolle Speciali

Le bolle speciali sono di due tipi: **Lightning** e **Water**.
Entrambe sono spawnate randomicamente dallo stage in posizione centrale rispetto l'asse x e sull'apice della finestra, sono oggetti Bubble fino a che non vengono colpiti, momento nel quale generano oggetti Special Power al loro posto in base alla loro categoria.

La bolla Lightning rilascia un fulmine con movimento orizzontale, la bolla Water rilascia dell'acqua soggetta a gravità, entrambi i poteri eliminano i nemici a contatto.

Power-Up

I power-up sono dieci in totale e si differenziano fra power-up relativi all'accumulo di punteggio e quelli relativi alla statistiche delle bolle e del player.

Per quanto riguarda quelli relativi il punteggio troviamo tre tipi diversi di **anelli**: blu, rosso e rosa. I quali forniscono rispettivamente: punti per spostamento, per bolla scoppiata e per salto.

Per quanto riguarda quelli relativi alle statistiche troviamo: **scarpe** della velocità, **caramelle** rosa, caramelle blu e caramelle gialle. Le quali forniscono rispettivamente: distanza di bolla maggiore, velocità di bolla maggiore e frequenza di bolla maggiore.

Poi troviamo power up con effetti combinati, che sono: **lanterne** blu, lanterne rosse e lanterne gialle. Le quali conferiscono rispettivamente, tutti i poteri di punteggio, sia i poteri di punteggio che di statistiche di bolle e solo le statistiche delle bolle.

stream

Come da richiesta progettuale è stato implementato l'uso degli stream dove ritenuto efficiente, in particolare ne troviamo nella classe View e nella classe Model.

Nella classe View è implementato uno stream classico attraverso un ArrayList, atto a controllare l'effettivo caricamento delle immagini da parte di tutte le entità, in modo da evitare un'apparizione asincrona all'inizio di un nuovo livello.
Utilizza un semplice anyMatch() per verificare la condizione sia sempre verificata (al primo risultato negativo lo stream riporterà true evitando di scorrere tutta la lista).

```
if (!imagesLoaded) {
    if (!viewsArray.stream().anyMatch(view -> view.imagesLoaded == false)) {
        imagesLoaded= true;
    }
}
```

Nella classe Model invece si è utilizzato un intStream (poiché più efficiente rispetto allo stream riguardo gli interi) per verificare un particolare tipo di collisioni richiedente più precisione.

```
boolean orizontal= false, vertical= false;
if (IntStream.rangeClosed(x +1, x+width -1).anyMatch(value -> model.coordinates[0] <= value && value <= model.coordinates[0]+model.width)) {
    orizontal= true;
}
if (orizontal && IntStream.rangeClosed(y, y+heigth).anyMatch(value -> model.coordinates[1] <= value && value <= model.coordinates[1]+model.heigth)) {
    vertical= true;
}
return orizontal && vertical;
```

In entrambi i casi si è valutato l'uso di parallel stream, in quanto anche essendo un anyMatch i thread paralleli verrebbero interrotti una volta ottenuto un risultato, ma poiché gli array in questione non sono molto grandi il tempo necessario alla sincronizzazione dei thread sarebbe più di quello necessario a svolgere sequenzialmente l'operazione.

leaderboard e utente

Il gioco mantiene i dati salvati fra le partite, anche chiudendo il gioco, questo è possibile grazie alla creazione di due appositi file: **saves.txt** e **leaderboard.txt**.

Il **primo** contiene ogni nome con accanto il numero di partite vinte, perse e l'highscore dell'utente, questi dati vengono caricati al momento dell'inserimento del nome e appaiono nel menu e nell'HUD (in caso di nuovo nome verrà creata una nuova riga, altrimenti verrà caricato lo storico).

Il **secondo** salva solamente il nome utente abbinato al punteggio ottenuto in una singola partita. Ad ogni partita conclusa vengono caricati i dati ordinati e riscritti solamente i primi cinque.

classe Sound

La classe Sound è stata posizionata nella View nonostante non sia una componente visiva, poiché, intendendo la View come modulo di feedback utente, questa risulta essere la miglior opzione.

Nonostante si trovi nel package View però, questa classe è l'unica che rappresenta la controparte visiva di una classe logica nel Model. Infatti la sua logica non è gestita da una classe apposita.
L'anomalia è dovuta al fatto che non c'è una logica propria relativa ai suoni, ma solamente dei momenti, scanditi da altri eventi, in cui attivarli, che sono però gestiti da altre classi del Model, motivo per il quale la riga sound può comunque essere trovata all'interno dell'oggetto map spedito come notifica dall'Observable.

altri Design Pattern

Singleton

Uno dei design pattern più utilizzati nel progetto è stato il singleton, infatti molti degli oggetti instanziati per loro logica era coerente potessero essere generati una sola volta. Fra questi troviamo la classe Model e la classe View, la classe Bub (ovvero il player), la classe Menu, Stage e HUD, tutte oggetti di gestione sempre presenti e da non ripetere.