StarCastle - Esempio di applicazione dei design pattern nella modellazione di un videogioco

Nicola Corti - 454413
Corso di Laurea Magistrale in Informatica - Università di Pisa

22 Aprile 2014

Sommario

Questa relazione ha lo scopo di illustrare come i design pattern possono essere utilizzati nella modellazione di un software complesso, quale può essere un videogioco. Gli aspetti da tenere in considerazione sono svariati, dal mantenimento di uno stato consistente, alla gestione degli eventi, delle collisioni, etc...

Il codice sorgente allegato è da considerarsi parte integrante di questa relazione, al fine di permettere una comprensione più organica di tutti gli aspetti di design ed implementativi.

Indice

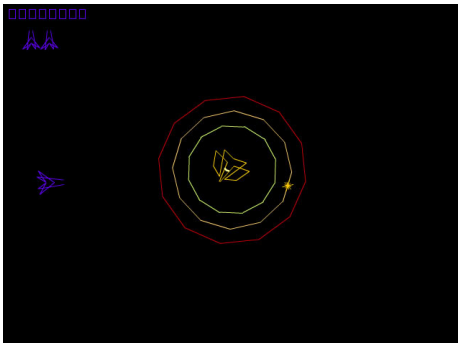

Immagine 1: Una schermata di gioco di *StarCastle*

Introduzione

I videogiochi rappresentano forse la categoria di software più complessi da realizzare, coinvolgendo in un unico progetto svariati aspetti dell'informatica (si pensi al networking, all'intelligenza artificiale o alla gestione dei database).

La realizzazione di un videogioco, se non supportata da un'accurata fase di analisi e di progettazione, può risultare veramente complessa o addirittura impossibile. Tralasciare la fase di progettazione di un videogioco potrebbe portare a notevoli perdite di tempo o addirittura al fallimento del progetto stesso.

Per questo motivo risulta fondamentale analizzare sotto ogni aspetto il videogioco che si intende realizzare, e progettare ogni classe/interfaccia con la massima accuratezza. In questa fase dello sviluppo i *design pattern* possono essere un strumento valido per risolvere alcune delle problematiche più comuni e ricorrenti nel campo dello sviluppo di videogiochi.

In particolare in questa relazione mostreremo come i *design pattern* possono essere di supporto nella progettazione del videogioco *StarCastle*[1], un videogioco vettoriale del 1980 prodotto da Cinematronics[2].

In *StarCastle* il giocatore si troverà a comandare un'astronave spaziale, e dovrà attaccare un cannone posto al centro dello schermo (vedi immagine 1). Il cannone è protetto da 3 annelli formate da barre di energie ed è in grado di attaccare il giocatore tramite mine (che inseguiranno l'astronave del giocatore) e sfere al plasma (che verranno sparate direttamente dal cannone verso l'astronave).

Tralasciamo una discussione più dettagliata di tutte le meccaniche di gioco, in quanto verranno analizzate singolarmente nelle varie sezioni seguenti.

Preferiamo piuttosto soffermarci su un pattern che ha guidato la fase di progettazione di tutto il software e che permea tutti gli aspetti che verranno analizzati nelle sezioni seguenti, il *Model-View-Controller*.

[1] `http://en.wikipedia.org/wiki/Star_Castle`

[2] È possibile testare una demo online del gioco all'indirizzo seguente
`http://www.download-free-games.com/online/game/star_castle/`

Il pattern *Model-View-Controller*

Il *Model-View-Controller* è un pattern architetturale che ha come scopo principale quello di mettere il focus sulla separazione fra la rappresentazione dei dati e la loro rappresentazione.

I componenti che interagiscono nel pattern *Model-View-Controller*, come suggerisce il nome, sono 3 (vedi immagine 2):

Model Mantiene la rappresentazione dei dati e si occupa della *business logic*,

View Interagisce con il Model e offre all'utente una sua rappresentazione visuale,

Controller Raccoglie l'input dell'utente e modifica il Model di conseguenza.

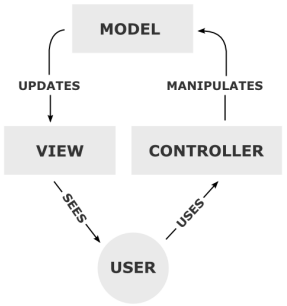

Immagine 2: Diagramma del pattern *Model-View-Controller*

In particolare in *StarCastle* sono stati i componenti del *Model-View-Controller* sono stati istanziati come segue:

Model Rappresenta lo stato di tutte le entità di gioco, mantenendone stato attuale, posizione, velocità ed eventuali interazioni con altre entità di gioco

View Rappresenta il contesto grafico sul quale verranno disegnate tutte le primitive grafiche per rappresentare i vari componenti del gioco, secondo le policy grafiche previste dal sistema

Controller Rappresenta il meccanismo che si occupa di raccogliere gli input dell'utente, e di convertirli in comandi di giochi, che verranno elaborati al fine di far evolvere le entità in gioco (nella fattispecie quindi muovere o far sparare una navicella).

1 Modellazione della realtà

Il primo passo effettuato nella fase di progettazione di dettaglio consiste nell'individuare quali saranno le classi che andranno a rappresentare la nostra realtà.

Analizzando il nostro caso ci rendiamo subito conto che:

- Le entità da modellare sono fondamentalmente 5: l'astronave, il cannone, le mine, i missili e le barre di energia.
- Esse condividono alcuni attributi comuni, quali ad esempio la posizione.
- Esse condividono dei comportamenti comuni, quali ad esempio la necessità di modificare la propria posizione.
- Le entità necessitano di mantenere uno stato comune, che muta quando avviene una collisione fra due entità distinte.
- Alcune entità (cannone e mine) hanno la necessità di essere notificate quando un utente interagisce con il gioco, al fine di poter decidere quale navicella spaziale seguire.

Per questi motivi si è deciso di progettare una gerarchia di classi che ha come padre la classe astratta GameEntity e di utilizzare i seguenti pattern:

State Per modellare gli stati delle entità in gioco e le loro transizioni di stato,

Factory method Per delegare la creazione dello stato iniziale alle sottoclassi di GameEntity,

Observer Per modellare il meccanismo di notifica di interazione del giocatore alle mine e al cannone.

La gerarchia di classi è rappresentata nell'immagine 3[3].

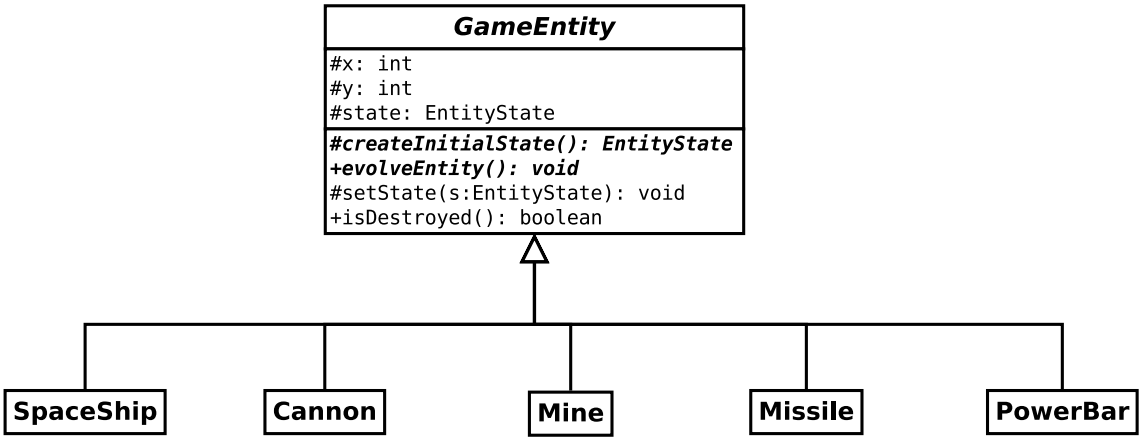

Immagine 3: Diagramma UML della classe GameEntity

Poniamo attenzione al metodo evolveEntity, esso rappresenta il metodo fondamentale per definire come l'entità si evolve nel tempo. È un metodo astratto e deve essere implementato da ogni sottoclasse definendo il proprio comportamento

[3]Si noti che in questo e nei prossimi diagrammi UML non sono rappresentati tutti gli attrubuti e i metodi della classe, ma solamente quelli significati nel contesto in cui vengono trattati, al fine di non appesantire il diagramma

(si pensi ad esempio al cannone che ruota, all'astronave che decelera linearmente o ai missili che procedono il linea retta con velocità costante).

Gli altri metodi di GameEntity verranno trattati nel dettagli nelle sezioni seguenti.

1.1 Utilizzo dal pattern *State*

Al fine di modellare al meglio la necessità di mantenere uno stato e di evolverlo da parte delle GameEntity si è utilizzato il pattern *State*: è stata definita la classe astratta EntityState che rappresenta un generico stato di un'entità.

Ogni classe provvederà a realizzare le sottoclassi di EntityState e a fornire l'implementazione dei metodi astratti della classe.

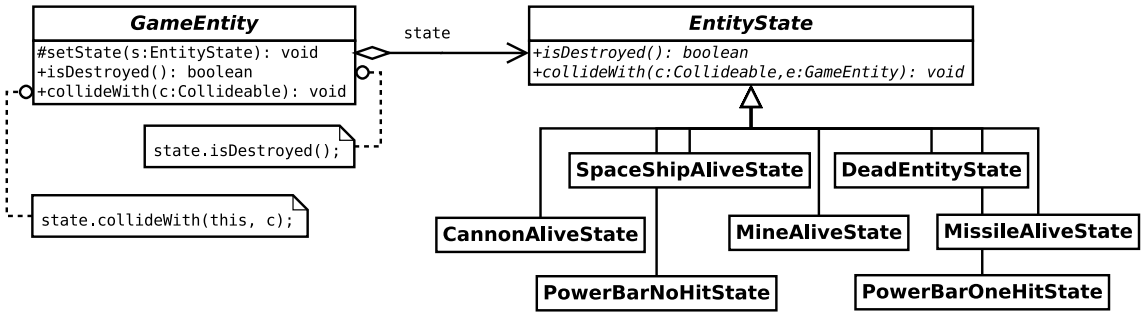

Immagine 4: Diagramma UML del pattern *State*

Nel diagramma in figura 4 si nota che la classe EntityState contiene i metodi: isDestroyed per verificare se l'astronave si trova in uno stato in cui è stata distrutta, e collideWith per gestire la collisione fra l'entità e un altro oggetto. (Per la gestione delle collisioni si rimanda alla sezione 2).

Si noti come il metodo collideWith ha due parametri, il primo rappresenta il riferimento al GameEntity stesso, mentre il secondo rappresenta un oggetto con cui si è entrati in collisione; il primo parametro è necessario in quanto lo stato invocherà il metodo setState sul GameEntity al fine di realizzare la transizione verso un nuovo stato.

Utilizzare il pattern *State* comporta la realizzazione di tante piccole classi che rappresentano i singoli stati delle entità, come si può notare dalle sottoclassi di EntityState presenti nel diagramma. Questo sembrerebbe introdurre complessità nella progettazione, ma porta invece notevoli benefici, permettendo di evitare grossi statement switch e permettendo di semplificare notevolmente la fase di rendering grafico (sezione 3).

Si noti infine che è stata realizzata una singola classe DeadEntityState al fine di rappresentare lo stato in cui termina una generica entità che è stata distrutta; invece di realizzare svariate classi del tipo SpaceShipDeadState, CannonDeadState, etc... riducendo così la replicazione del codice.

1.2 Utilizzo del pattern *Factory Method*

Per la creazione del primo stato in cui nasce una GameEntity si è deciso di utilizzare il pattern *Factory Method*: nella classe GameEntity è presente il metodo astratto createInitialState, ogni classe deve quindi implementare questo metodo, costruendo quello che è l'oggetto che rappresenta il loro stato iniziale. Per cui ad esempio la classe SpaceShip ritornerà un nuovo oggetto di tipo SpaceShipAliveState.

Nel diagramma in figura 5 riportiamo le classi relative a SpaceShip, i diagrammi per le altre sottoclassi di GameEntity sono pressoché analoghi.

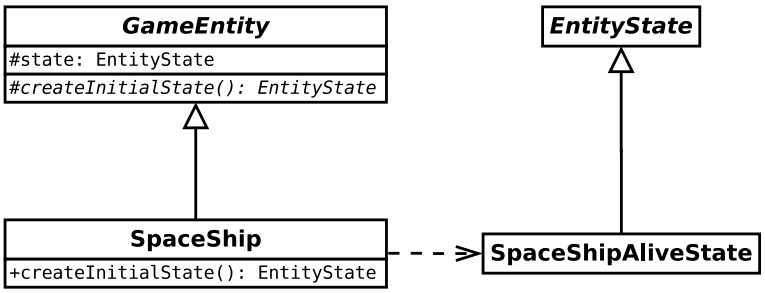

Immagine 5: Diagramma UML del pattern *Factory Method*

1.3 Utilizzo del pattern *Observer*

Per gestire la logica dello spostamento di mine e del cannone, che devono inseguire una delle astronavi in gioco, si è deciso di utilizzare il pattern *Observer*.

In particolare, ogni volta che un'astronave riceve un comando di rotazione, accelerazione o di sparo, questa notifica a tutte le mine e al cannone in gioco.

Per implementare il pattern sono stati utilizzati le classi e le interfacce delle Java API: le classi Mine e Cannon implementano l'interfaccia java.util.Observer, fornendo il metodo update che verrà invocato ogni volta che ricevono una notifica.

Nell'implementazione attuale le classi Mine e Cannon inseguono l'ultima SpaceShip da cui hanno ricevuto una notifica, ma si potrebbe ampliare la logica prevedendo di seguire l'astronave che ha la distanza minore.

Per quanto riguarda la classe SpaceShip si è deciso mantenere un riferimento ad un oggetto di tipo DelegatedObservable, in quanto SpaceShip era già in una gerarchia di classi. Si noti che la classe DelegatedObservable (sottoclasse di java.util.Observable) è una classe necessaria per il meccanismo della delega, in quanto estende la visibilità dei metodi di Observable.

Le classi coinvolte sono riassunte nel diagramma in figura 6.

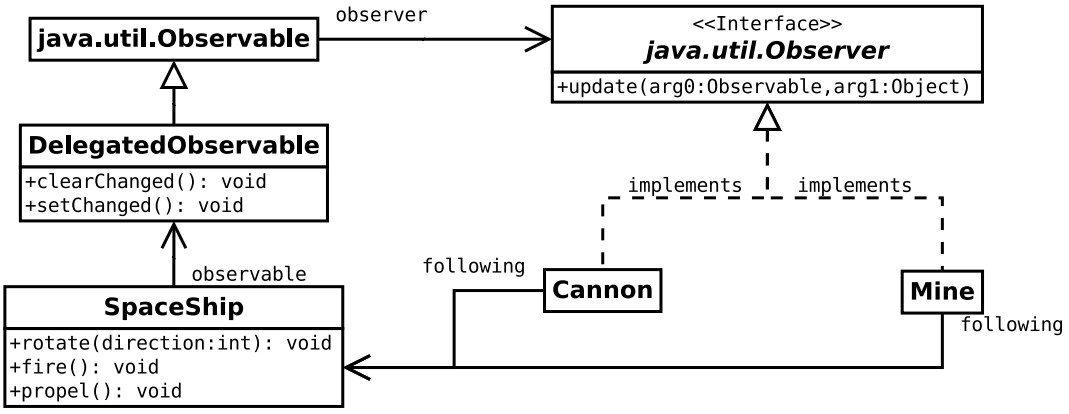

Immagine 6: Diagramma UML del pattern *Observer*

2 Gestione delle collisioni

3 Gestione del rendering grafico

4 Gestione dei comandi

5 Avvio e terminazione del gioco

6 Gestione del multiplayer

7 User guide

La simulazione è stata realizzata in Java utilizzando il simulatore Peersim ed è stata corredata di un file **ant** (il file `build.xml`) che offre dei target per automatizzare il processo di compilazione e di esecuzione del software.

Per funzionare, la simulazione ha bisogno di un file di configurazione in input da cui poter caricare la configurazione della rete (numero di nodi, numero di link, protocolli da utilizzare, etc...). Alcuni esempi di file di configurazione possono essere trovati all'interno della cartella `example/`.

Per compilare la simulazione è necessario posizionarsi all'interno della directory dove è contenuto il software ed invocare da terminale il comando

```
ant build
```

che provvederà ad invocare il compilatore `javac` per compilare i sorgenti presenti all'interno della cartella `src/`, i file `.class` generati si troveranno all'interno della cartella `bin/`.

Per pulire la cartella `bin/` al fine di avere un ambiente pulito per poter effettuare una nuova compilazione è possibile utilizzare il target

```
ant clean
```

È infine possibile generare un file `jar` contenente tutti i file compilati e tutte le librerie necessarie all'esecuzione. Per farlo è sufficiente invocare il target

```
ant jar
```

Verrà generato un file chiamato `p2p_final.jar` all'interno della cartella principale del software. Per avviare il file `jar` è necessario invocare il comando

```
java -jar p2p_final.jar [file di configurazione]
```

7.1 Documentazione

Al fine di rendere il codice sorgente più comprensibile, il software è stato corredato di documentazione. In particolare tutte le parti del codice sorgente che potrebbero risultare di difficile comprensione sono state commentate. Inoltre ogni funzione e classe del software è stata documentata con il formato javadoc, la documentazione generata può essere visionata all'interno della cartella doc/ e può essere rigenerata utilizzando il comando

```
ant javadoc
```

Per una comprensione organica del software si consiglia la lettura della seguente relazione nella sua interezza. La presente relazione viene rilasciata in Pdf ed in LaTeX e può essere visionata all'interno della cartella `doc/tex/`.

7.2 Avvio della simulazione

Una volta compilato il software è possibile invocarlo tramite il comando

```
ant Execute [-Dfile=nomefile]
```

Con `-Dfile=nomefile` è possibile impostare il file di configurazione da usare. Per effettuare alcune computazioni di esempio è quindi sufficiente impostare `-Dfile=example/<nomefile.conf>`, eseguendo alcuni dei files di esempio già predisposti all'interno della cartella `example`.

I nomi dei files contenuti nella cartella `example` permettono di comprendere facilmente quali sono le configurazioni impostate. Sono stati predisposti calcoli solamente COUNT oppure COUNT-BEACON, con grafi random, small-world e scale-free. Inoltre sono stati predisposti esempi con debugger attivato ed altri esempi in cui si calcolano altre funzioni (massimo, somma, etc...).

Se la struttura dei file di configurazione non fosse chiara è possibile visionare il file example.conf che contiene tutti i commenti necessari a comprendere le impostazioni possibili.