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

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

10 Novembre 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

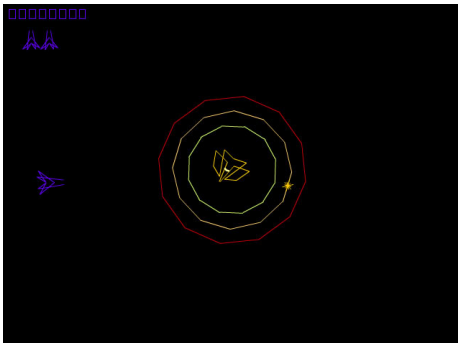

Immagine 1: Una schermata di gioco di *StarCastle*

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.

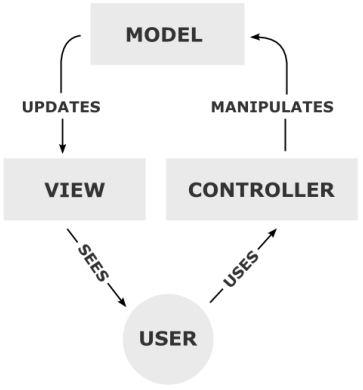

Immagine 2: Diagramma del pattern *Model-View-Controller*

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].

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 (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.

[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

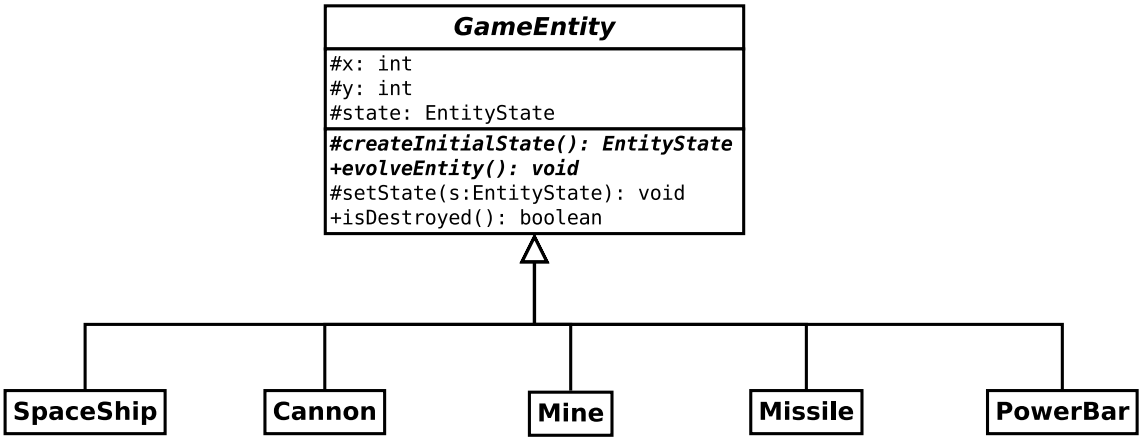

Immagine 3: Diagramma UML della classe GameEntity

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.

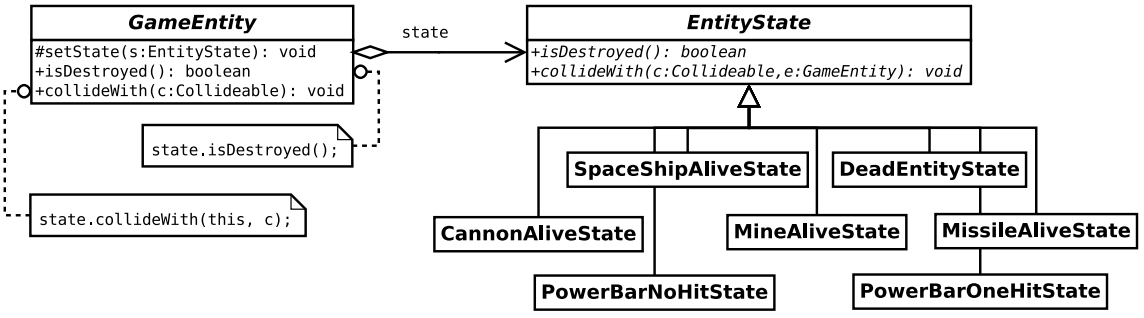

Immagine 4: Diagramma UML del pattern *State*

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.

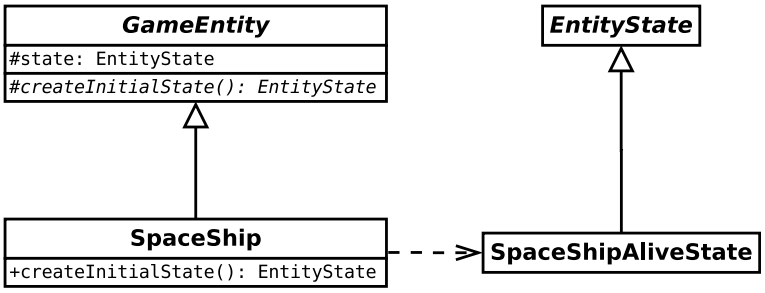

Immagine 5: Diagramma UML del pattern *Factory Method*

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.

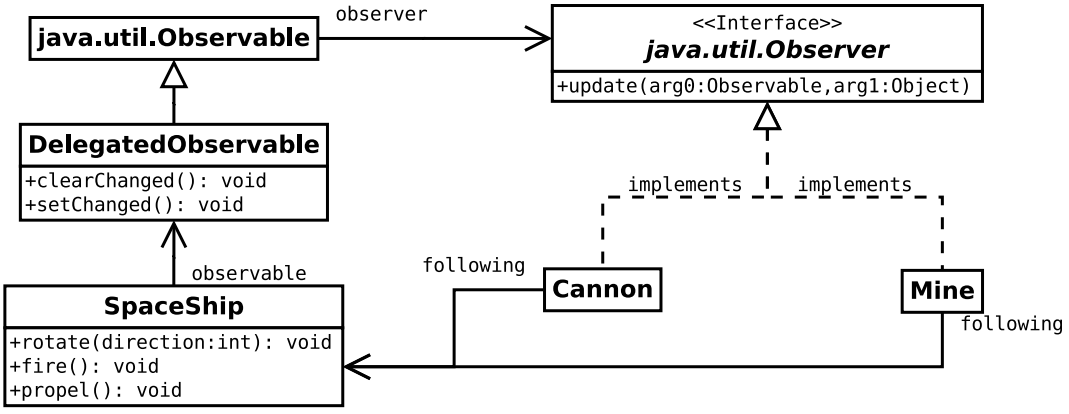

Immagine 6: Diagramma UML del pattern *Observer*

2 Gestione delle collisioni

Un altro aspetto fondamentale, legato al concetto di stato, è quello della gestione delle collisioni. Una gestione delle collisioni mal implementata può ridurre drasticamente il gameplay di un videogioco, e portare frustrazioni al giocatore con conseguente degrado dell'esperienza utente.

Abbiamo intanto definito un'area circolare, centrata sull'entità e con un determinato raggio, in cui si registrano le collisioni; consideriamo una collisione nel momento in cui queste aree circolari si intersecano in almeno un punto. Quest'area è rappresentata dalla classe BoundCircle.

Concettualmente ogni GameEntity dovrebbe controllare ogni altra entità in gioco e verificare se sono presenti delle collisioni, ottenendo il BoundCircle e verificando se si interseca con il proprio.

Invece di obbligare ogni GameEntity ad interagire con ogni altra GameEntity presente, abbiamo preferito un approccio più centralizzato, utilizzando il pattern *Mediator*.

In particolare abbiamo realizzato:

- L'interfaccia Collideable che deve essere implementata da ogni entità, in particolare Collideable è implementata da GameEntity che fornisce direttamente l'implementazione dei due metodi dell'interfaccia:

  - getBoundCircle che ritorna il BoundCircle di una dimensione standard (le sottoclassi faranno l'override di questo metodo se il bound circle è differente).
  - collideWith che rappresenta il metodo invocato quando GameEntity entra in collisione con un altro Collideable, implementato invocando il metodo collideWith dello stato attuale (sezione 1.1).

- L'interfaccia CollisionMediator, che deve essere implementata dai mediator concreti, con il metodo checkCollision che deve controllare, data una lista di Collideable, se sono presenti collisioni, e notificare i soggetti coinvolti nella collisione
- La classe CollisionConcreteMediator che rappresenta un'implementazione concreta dell'interfaccia CollisionMediator.

Si noti come questa implementazione del pattern *Mediator* risulta leggermente differente da quella prevista dalla *GoF*: i soggetti *Colleague* previsti dal *GoF* dovrebbero mantenere il riferimento al *Mediator*, mentre nel nostro caso non abbiamo previsto che i GameEntity potessero interagire direttamente con il CollisionMediator, ma che sia il CollisionMediator a richiedere le informazioni per computare le collisioni alle singole GameEntity.

Le classi coinvolte nella gestione delle collisioni sono presentate nel diagramma in figura 7.

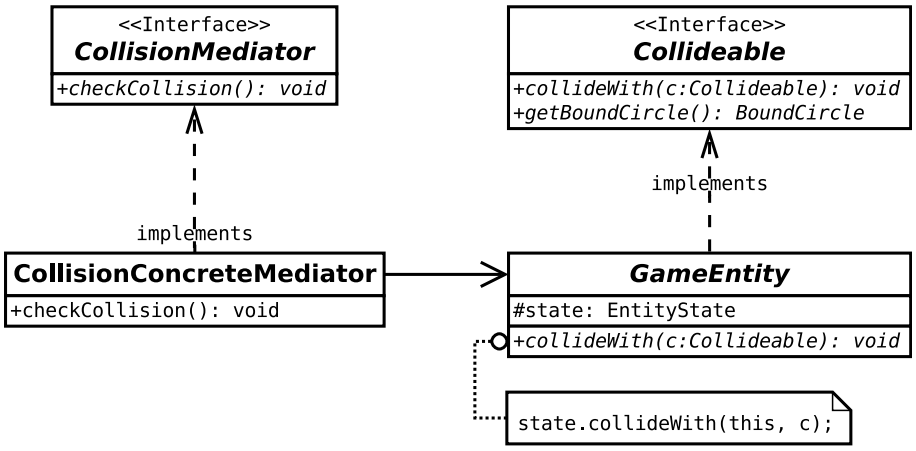

Immagine 7: Diagramma UML del pattern *Mediator*

3 Gestione del rendering grafico

Una volta definita la componente *Model* del nostro sistema *Model-View-Controller*, passiamo a definire la parte *View*.

Generalmente i videogiochi permettono di essere visualizzati mostrando una serie di primitive grafiche per ogni entità in gioco. Abbiamo dunque modellato le primitive grafiche nel modo seguente:

- Un'interfaccia GraphicEntity che rappresenta una generica primitiva grafica.
  - La classe GraphicSprite che implementa GraphicEntity e permette di creare una *sprite* indicando il nome del file da disegnare,
  - La classe GraphicVector che implementa GraphicEntity e permette di creare un vettore grafico, indicando le coordinate di inizio, di fine e il colore.

La gerarchia di classi delle primitive grafiche è mostrata in figura 8.

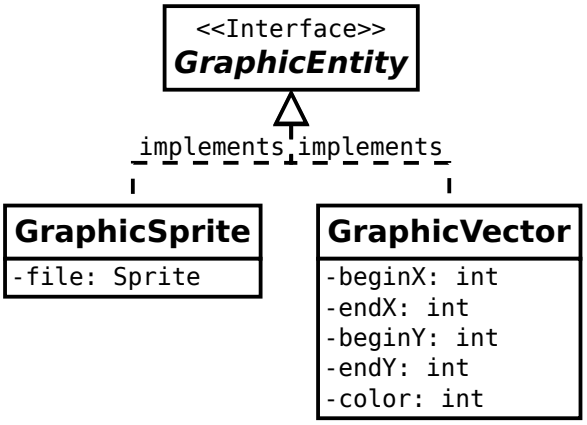

Immagine 8: Diagramma UML delle primitive grafiche

Abbiamo successivamente realizzato l'interfaccia Drawable (immagine 9), che deve essere implementata dai soggetti che desiderano essere disegnati a scherma. L'interfaccia prevede un metodo draw con un parametro di tipo GraphicEnvironment (il contesto grafico), si realizzano quindi tutte le operazioni necessarie a mostrare su schermo l'oggetto.

Nel nostro caso abbiamo deciso di far implementare l'interfaccia Drawable alla classe GameEntity e di lasciare l'onere di dover fornire le primitive grafiche ad ogni singolo stato sottoclasse di EntityState: ogni stato fornisce il metodo getGraphicEntities che ritorna una lista di primitive grafiche (List<GraphicEntity>) che permettono di renderizzare la singola entità.

Chiariamo meglio questo concetto con un esempio: una PowerBar che non è stata mai colpita[4] invocherà il metodo getGraphicEntities sul suo stato, ottenendo una lista di GraphicEntity, in particolare un solo GraphicVector che permette di disegnare l'oggetto PowerBar. Nel momento in cui avviene una collisione con un Missile l'oggetto PowerBar transisce in un nuovo stato[5] ed invocando il metodo getGraphicEntities si otterranno le primitive grafiche per ridisegnare la PowerBar che è stata colpita.

[4]La PowerBar si trova dunque nello stato PowerBarNoHitState
[5]Lo stato PowerBarOneHitState

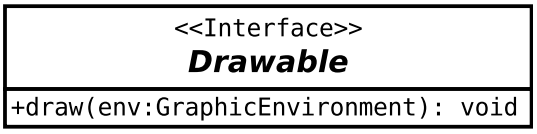

Immagine 9: Diagramma UML dell'interfaccia Drawable

3.1 Uso del pattern *Strategy*

Consideriamo inoltre che dobbiamo prevedere inoltre un meccanismo per animare le primitive grafiche di una GameEntity, per gestire le animazioni abbiamo deciso di implementare il pattern *Strategy* nei GameEntity.

Abbiamo realizzato la classe astratta DrawStrategy che contiene il metodo drawEntity: questo metodo accetta due parametri, un GraphicEnvironment dove poter disegnare e una lista di GraphicEntity. Abbiamo inoltre fornito due strategie concrete:

- DrawVectors che disegna semplicemente l'elenco dei vettori forniti sull'ambiente grafico
- DrawSprites che anima le sprites fornite, disegnano una delle sprites nella lista a turno.

La DrawStrategy iniziale viene impostata tramite il metodo astratto createInitialStrategy della classe GameEntity, utilizzando il pattern *Factory Method*, in un modo analogo a quanto avviene con lo stato (sezione 1.2).

È quindi possibile cambiare strategia a runtime usando il metodo setStrategy, oppure realizzare strategie più complesse, che prevedano animazioni più fluide, unendo sprite e vettori assieme.

Le classi coinvolte nell'utilizzo del pattern *Strategy* sono mostrate nel diagramma in figura 10

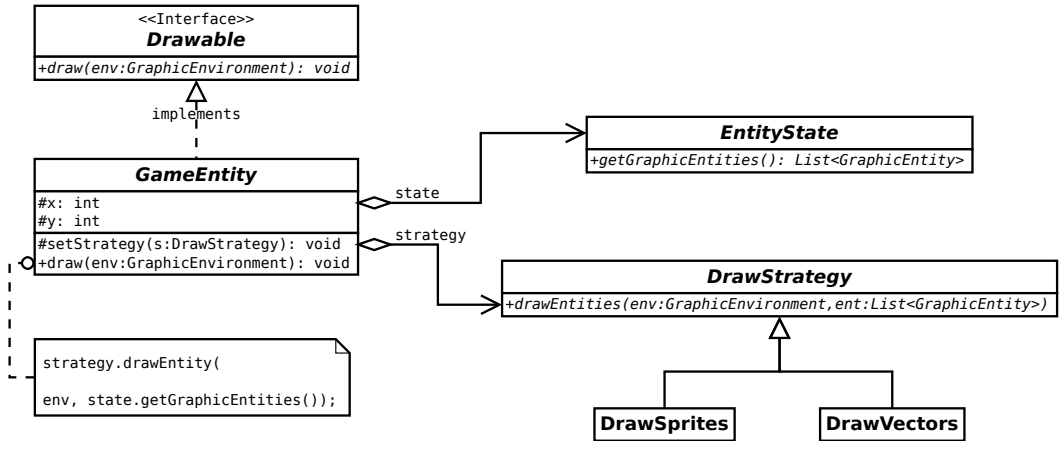

Immagine 10: Diagramma UML del pattern *Strategy*

4 Gestione dell'engine di gioco

Tutte le GameEntity devono essere mantenute da una classe che si occuperà di gestire il ciclo di rendering. Questa classe è GameEngine, che mantiene i riferimenti a

- La lista delle GameEntity in gioco
- Il CollisionMediator
- I contesti grafici dove effettuare il rendering (*back buffer* e *display buffer*)

Inoltre GameEngine deve offrire i metodi per

- Inizializzare lo stato di gioco,
- Avviare il ciclo di gioco,
- Interrompere il ciclo di gioco,
- Aggiungere un'entità al gioco,
- Rimuovere un'entità dal gioco,
- Interagire con un'astronave (farla ruotare, sparare, etc...),
- Ricevere un comando (sezione 5).

Il ciclo di gioco è schematizzato nell'algoritmo seguente:

1. Controllo se sono nello stato di gameover,
2. Rimuovo le entità che si sono distrutte (isDestroyed),
3. Eseguo comandi che ho ricevuto dall'utente,
4. Faccio evolvere le entità in gioco (evolveEntity),
5. Controllo se ci sono state collisioni,
   (a) Se sì, notifico i soggetti coinvolti,
6. Renderizzo sul *back buffer*,
7. Disegno il *back buffer* su schermo.

4.1 Utilizzo del pattern *Singleton*

L'oggetto GameEngine è fondamentale per il funzionamento del gioco, e non vogliamo che siano presenti due istanze concrete della classe, abbiamo dunque deciso di utilizzare il pattern *Singleton*.

Abbiamo un campo statico privato di tipo GameEngine dentro la classe stessa, che mantiene il riferimento all'istanza concreta; abbiamo reso il costruttore privato e realizzato il metodo statico getInstance per permettere di recuperare l'istanza di GameEngine.

Utilizzando questo pattern possiamo inoltre evitare di dover passare i riferimenti al GameEngine concreto, ma recuperarli invocato il metodo statico GameEngine.getInstance().

L'implementazione del pattern *Singleton* è rappresentata dal diagramma in figura 11

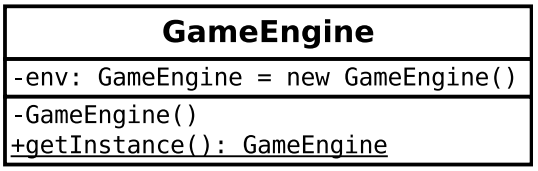

Immagine 11: Diagramma UML del pattern *Singleton*

Si noti che si sta utilizzando la versione *eager* del pattern *Singleton* in quanto si inizializza subito l'istanza concreta, questo per evitare eventuali problemi di concorrenza; l'impatto in termini di performance è trascurabile.

4.2 Utilizzo del pattern *Decorator*

Il videogioco deve prevedere anche la modalità *windowed*. Aggiungere la seguente modalità consiste fondamentalmente in aggiungere del comportamento alla fase di rendering iniziale del videogioco.

All'avvio il gioco determina le dimensioni dell'area che può utilizzare per effettuare le operazioni di rendering grafico. Aggiungere la modalità *windowed* consiste in ridimensionare quest'area e aggiungere gli elementi grafici della finestra (bordi, bottoni, menù, etc...).

Trovandoci in una situazioni in cui si sta estendendo una funzionalità, si è deciso di utilizzare il pattern *Decorator*, il GameEngine rappresenta la nostra classe base, la classe astratta GameDisplay rappresenta la radice della gerarchia che definisce le operazioni comuni a GameEngine e ai decorator.

Abbiamo inoltre realizzato la classe WindowDecorator che rappresenta l'unico *decorator* concreto, abbiamo omesso la classe astratta padre della gerarchia dei *decorator* in quanto non vi erano altri *decorator* concreti.

WindowDecorator mantiene un riferimento component a un oggetto di tipo GameDisplay (nella fattispecie il GameEngine), di cui estende il comportamento. In particolare il metodo renderWindow esegue dapprima lo stesso metodo su component ed aggiunge la logica della finestra.

Le classi coinvolte nell'implementazione del pattern *Decorator* sono raffigurate nel diagramma in figura 12.

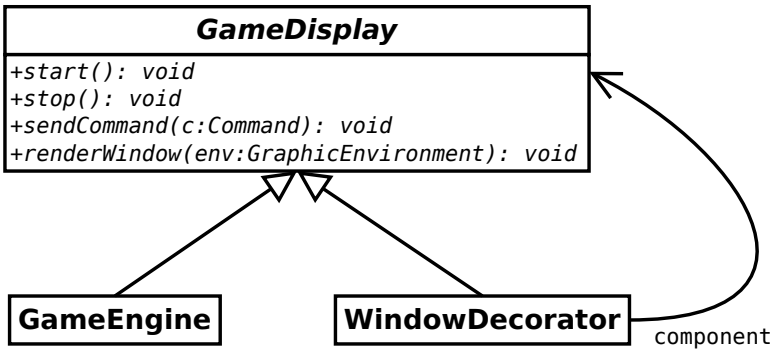

Immagine 12: Diagramma UML del pattern *Decorator*

5 Gestione dei comandi

La gestione dei comandi rappresenta il componente *Controller* del sistema *Model-View-Controller*. È bene definire quale sarà il flusso che dovrà percorrere l'input utente al fine di arrivare al modello e poterlo mutare.

Si è deciso in particolare di utilizzare i pattern *Command* e *Chain of responsability* per modellare la gestione dei comandi.

5.1 Utilizzo del pattern *Command*

La prima classe che presentiamo è la classe KeyEventManager, che implementa l'interfaccia java.swing.event.KeyListner, è si occupa di raccogliere gli eventi da tastiera dell'utente e li trasforma in comandi.

I comandi vengono modellati dalle classi che implementano l'interfaccia Command: CommandRotate, CommandFire e CommandPropel che rappresentano gli omonimi comandi da realizzare sull'astronave.

La classe che si occupa della gestione dei comandi è GameEngine che mantiene una lista di comandi da eseguire e si occupa di eseguirli ad ogni ciclo di rendering. Si è deciso di gestire i comandi in questo modo per non far interferire la frequenza di arrivo dei comandi con la frequenza di esecuzione del ciclo di rendering.

La classe GameEngine svolge il ruolo sia di *Invoker*, in quanto è lei che invoca il metodo execute sui comandi, sia il ruolo di *Receiver*, in quanto i comandi eseguono metodi di GameEngine per far evolvere lo stato (rotateSpaceShip, etc...).

La classe KeyEventManager svolge invece il ruolo di *Client* del pattern, esso crea una volta singola tutti i comandi (al fine di guadagnarne in performance) e li recapita all'*Invoker* appena riceve un input.

Le classi coinvolte nell'implementazione del pattern *Command* sono raffigurate nel diagramma in figura 13.

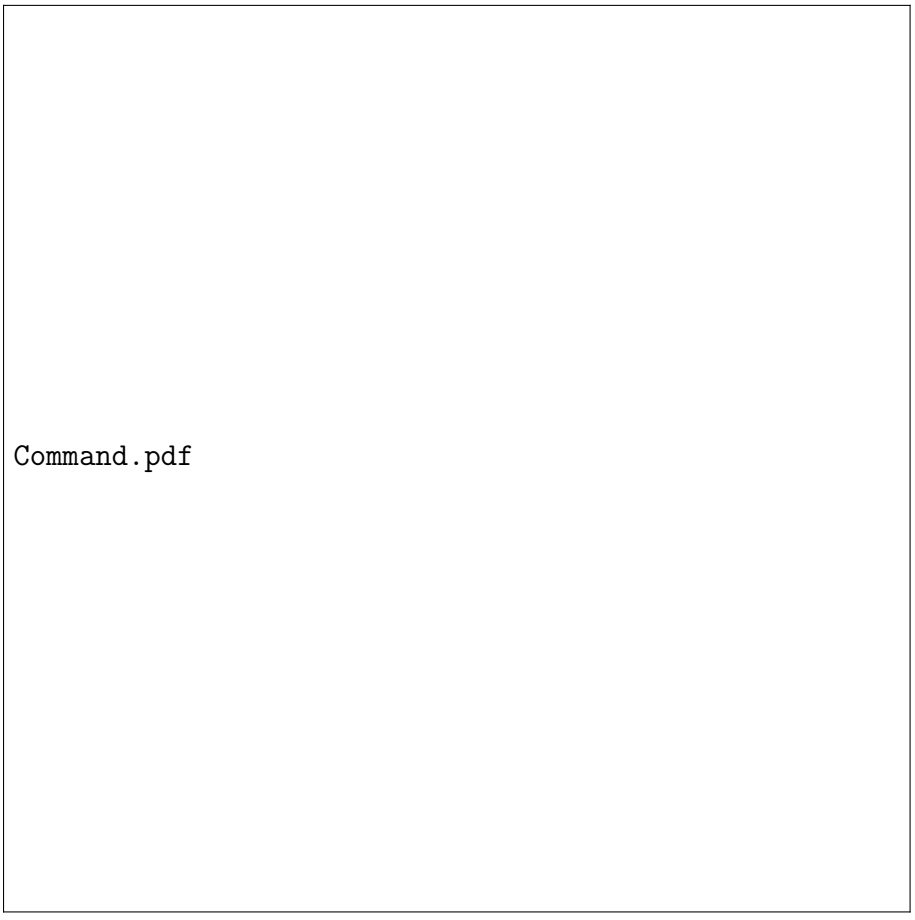

Immagine 13: Diagramma UML del pattern *Command*

5.2 Utilizzo del pattern *Chain of reponsability*

Si è deciso di utilizzare la struttura realizzata con il pattern *Decorator* per implementare il pattern *Chain of reponsability*: aggiungendo il metodo receiveCommand a GameDisplay prevediamo che ogni oggetto della gerarchia dei *decorator* abbia questo metodo.

Quando il KeyEventManager invierà un comando al GameDisplay, il *decorator* più esterno[6] potrà decidere se gestire il comando e non propagarlo ulteriormente, oppure passare il comando al proprio component invocando il solito comando receiveCommand.

Si noti come, malgaro le classi siano le stesse, il flusso di esecuzione è invertito: nel caso del metodo renderWindow, il decorator più esterno chiedere prima l'esecuzione al component ed estende successivamente con il proprio comportamento; nel caso del metodo receiveCommand si esegue prima la logica del *decorator* verificando se è possibile gestire il comando, e successivamente lo si propaga al component.

Le classi coinvolte sono le stesse del diagramma in figura 12.

6 Avvio e terminazione del gioco

6.1 Utilizzo del pattern *Façade*

7 Gestione del multiplayer

7.1 Utilizzo del pattern *Remote Proxy*

8 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

[6] Nel nostro caso abbiamo solamente un *decorator* concreto (WindowDecorator), ma nulla vieta di poter scrivere altri *decorator*

```
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]
```

8.1 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/`.

8.2 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.