Putting Challenge

Domenico Francesco Giacobbi

Lorenzo Colletta

Luca Lucioli

Giorgio Fantilli

Capitolo 1

Analisi

Si vuole realizzare un videogioco in due dimensioni ispirato al golf. Il giocatore deve cercare di fare buca in tutte le ambientazioni disponibili, cercando di evitare (o di sfruttare) gli ostacoli presenti.

1.1 Requisiti

Requisiti funzionali:

- L'applicazione deve permettere di iniziare una nuova partita dalla schermata iniziale.
- Consultare i punteggi delle partite precedenti.
- Tirare la palla con la mazza appropriata dal punto di battuta all'inizio di ogni mappa.
- Riposizionare il giocatore e la palla nel punto iniziale se quest'ultima dovesse uscire dall'ambientazione.
- Riposizionare il giocatore nel punto in cui la palla si ferma senza centrare la buca.
- Il giocatore ha a disposizione un numero limitato di possibilità (o vite) che gli vengono decurtate ad ogni tiro fallito. Per tiro fallito si intende la palla che si ferma o esce dai limiti dell'ambientazione.
- Spostarsi all'ambientazione successiva ad ogni buca effettuata.
- Tenere traccia del punteggio e dei tentativi rimasti.
- Mostrare una schermata che indichi la vittoria o la sconfitta.

Requisiti non funzionali:

- La palla deve mantenere un'animazione fluida e coerente durante lo

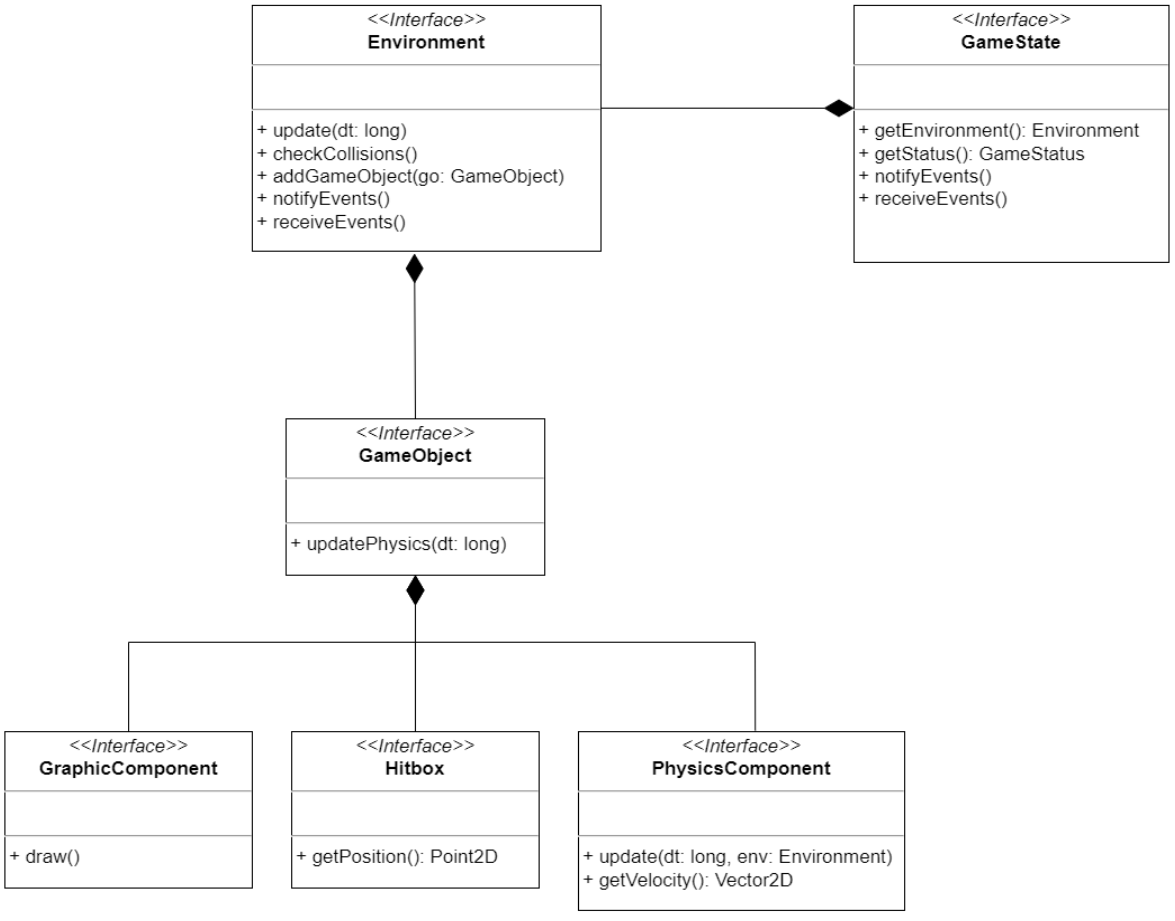
spostamento nell’ambiente e al rimbalzo con gli ostacoli.

1.2 Analisi e modello del dominio

Gli oggetti di gioco sono modellati dall’entità GameObject, la quale si compone di:

- un GraphicComponent, che ne definisce l’aspetto grafico e si occupa della sua renderizzazione;
- una Hitbox per la gestione dellle collisioni tra i GameObject;
- un PhysicsComponent, che determina il variare del suo stato fisico nel tempo.

Le ambientazioni di gioco sono incapsulate nell’entità Environment, che si occupa di mantenere ed aggiornare i GameObject allo scorrere del tempo.
L’entità GameState rappresenta un determinato stato dell’applicazione e tiene traccia delle informazioni ad esso relative.
Uno degli aspetti più complessi del dominio sarà la gestione delle collisioni tra i GameObject, in particolar modo tra la palla e gli ostacoli.

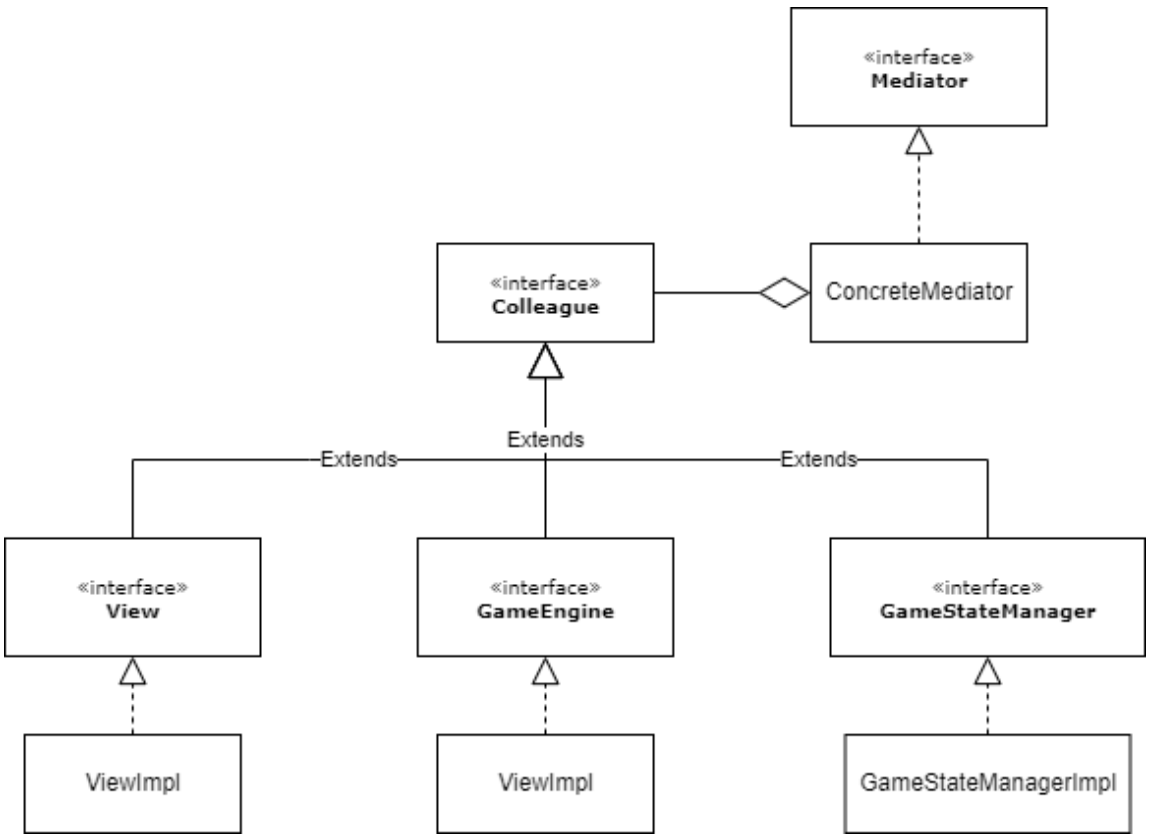


Capitolo 2

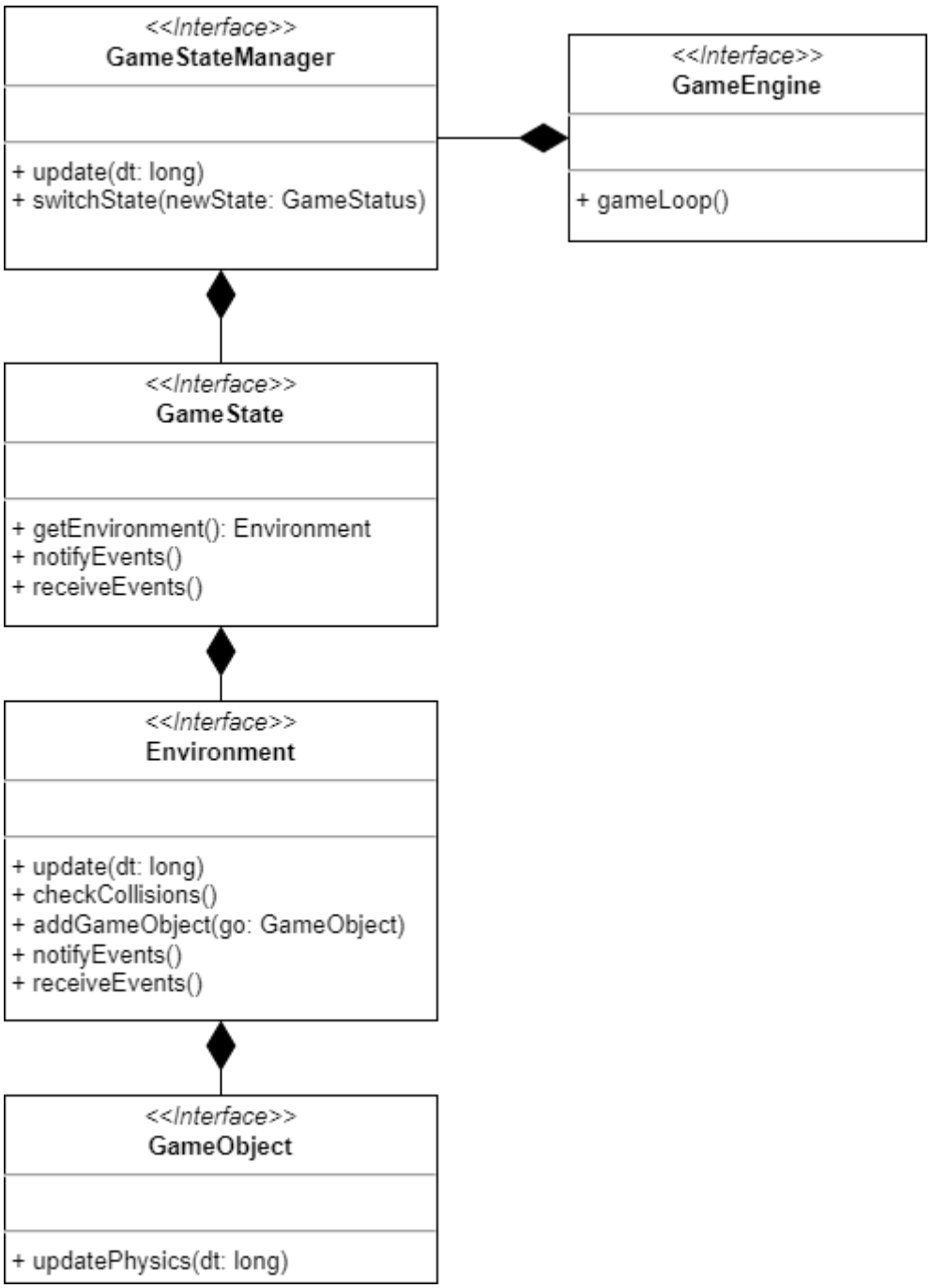
Design

2.1 Architettura

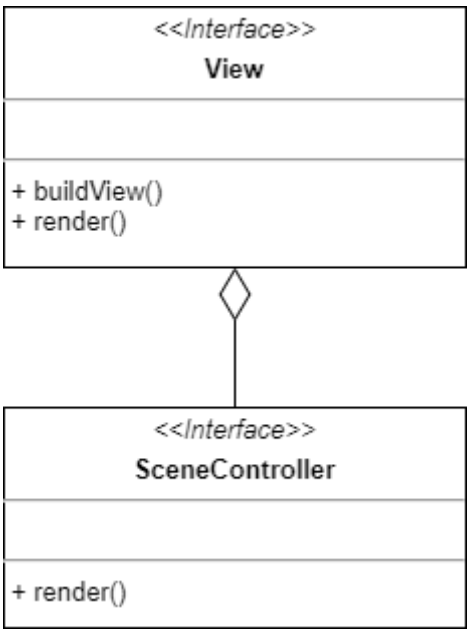
Abbiamo deciso di dividere la componente grafica dalla logica dell'applicazione creando delle apposite interfacce e le relative implementazioni.
L'interfaccia GameStateManager incapsula il corrente stato di esecuzione e le diverse ambientazioni.
View incapsula la scena grafica e si occupa della sua renderizzazione.
Durante la fase di sviluppo si è scelto di utilizzare il pattern Mediator, che permette una comunicazione in broadcast tra i "colleghi". In questo modo il model e la view possono notificarsi degli eventi attraverso un "mediatore", evitando così una dipendenza diretta tra i due attori.
Il GameEngine si occupa di impostare la comunicazione tramite il suddetto Mediator e di implementare il game loop.



Il model si compone delle entità riportate nello schema sottostante. In particolare il GameStateManager si occupa del cambiamento degli stati dell'applicazione e di comandare appropriatamente la view. Lo stato corrente dell'applicazione (GameState) incapsula il relativo Environment, cioè la rappresentazione lato model della scena, che a sua volta può essere corredata di GameObject.



La View si occupa di caricare e mantenere la scena corrente.
Essa offre la possibilità di cambiare tale scena, ricevendo eventi dal model dell'applicazione, e si compone di un controller della scena. Quest'ultimo incapsula la Scene di JavaFX ed il suo compito è quello di aggiornarla frame by frame e di fornire gli handler degli eventi lato view.



2.2 Design dettagliato

Luca Lucioli

Stati di gioco

Problema

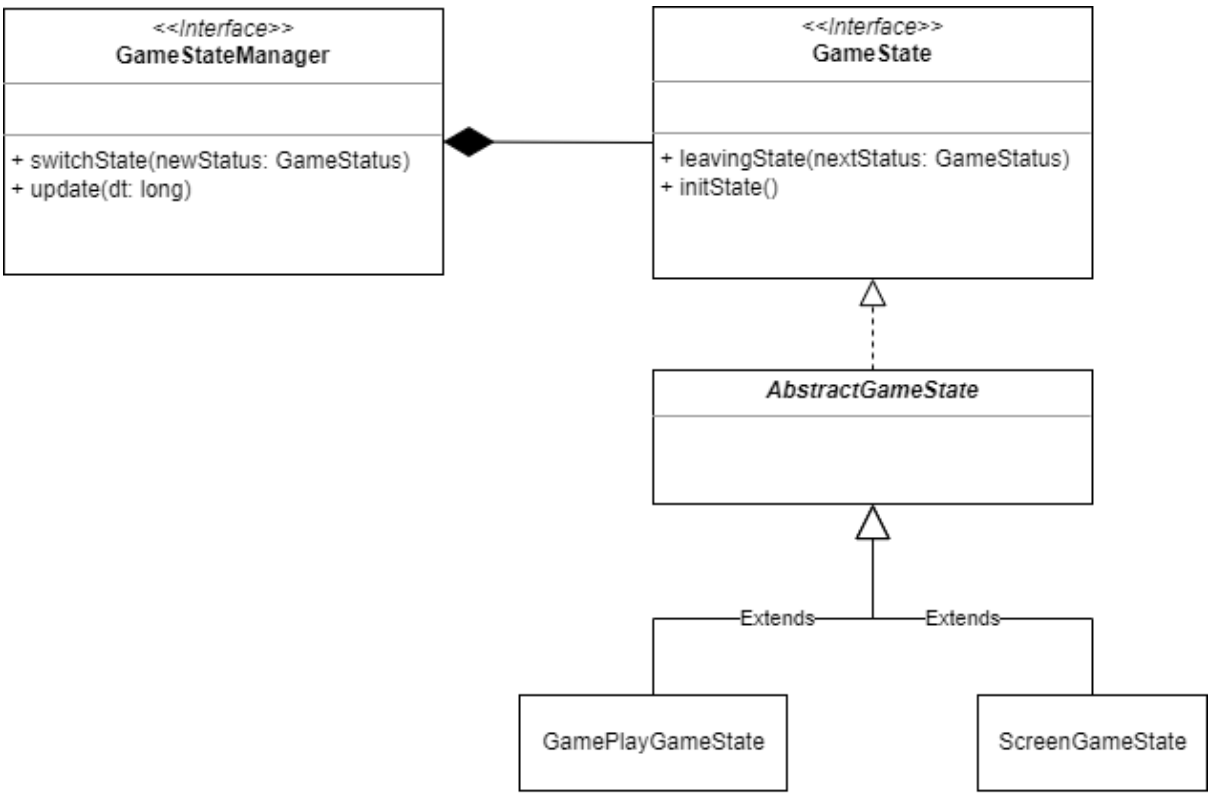
L’applicazione deve comportarsi diversamente a seconda dello stato in cui si trova.

Soluzione

Implementazione del pattern comportamentale “state”.

In una prima approssimazione di questo aspetto avevo individuato una solo entità che tenesse traccia dello stato dell’applicazione e delle sue logiche. Durante la progettazione mi sono reso conto di stare dando vita ad una “god class”, con molteplici responsabilità, spesso anche concettualmente distanti tra loro. Ho quindi deciso di utilizzare il pattern “state”, che mi ha permesso di dividere lo stato di gioco (inteso come lo stato dell’applicazione “in partita”) dagli altri stati di esecuzione. In questa implementazione del pattern “state” ogni schermata è associata ad uno stato e, se necessario, ad un environment. Nei termini di questo progetto l’unico stato

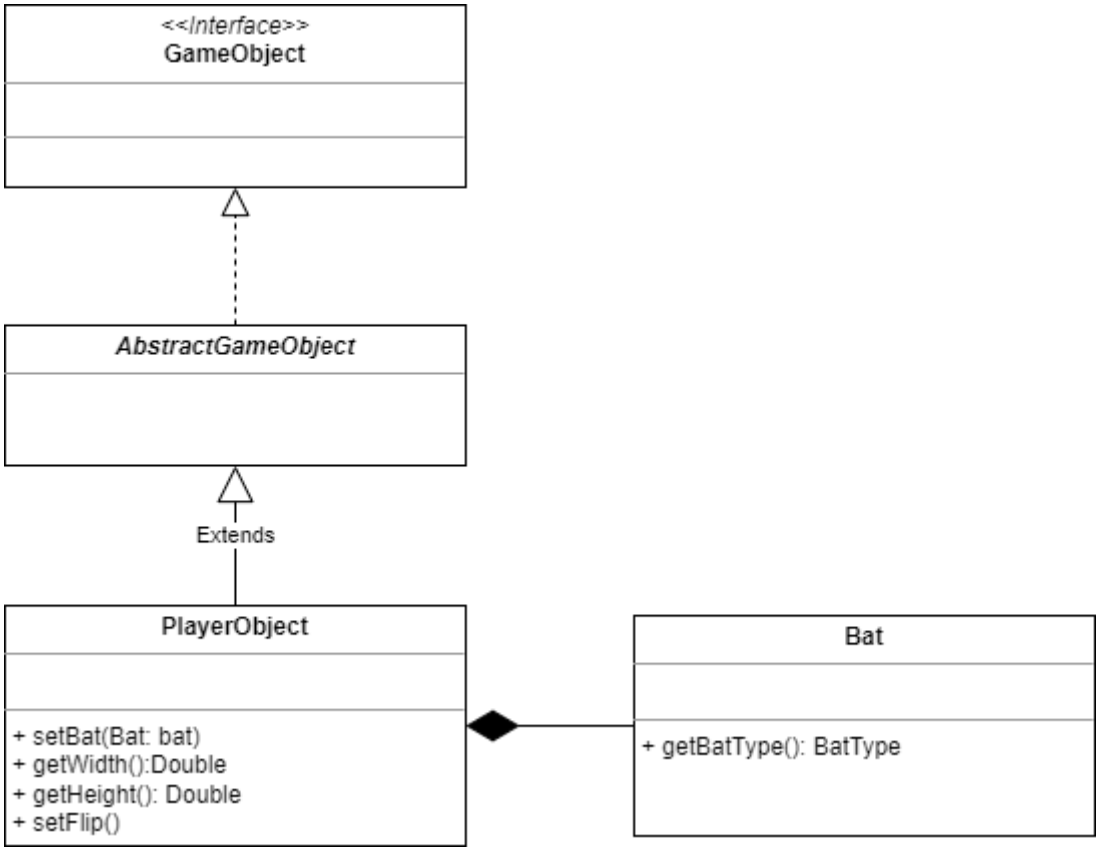
corredato di environement che siamo riusciti ad implementare è proprio lo stato di gioco.
Il “context” in questa reificazione del pattern è il “GameStateManager”, che si occupa di cambiare lo stato corrente. Nel caso di questo progetto possono essere anche gli stessi stati a definire il momento in cui “uscire di scena” e chi dovrà essere lo stato successivo. Ho preso questa decisione perché volevo incapsulare totalmente la logica di gioco nel relativo stato. In questo modo, però, il context non è a conoscenza delle logiche di gioco e quindi non può sapere se e quando è necessario cambiare stato. Ad esempio, non conoscendo le regole non potrà mai attivare lo stato di “game over” o di “game win” autonomamente, ma dovrà attendere che sia lo stato di gioco a chiederne l’attivazione.
L’alternativa che avevo considerato e intrapreso all’inizio della progettazione, come detto all’inizio, era di una classe che gestiva gli eventi in arrivo con uno switch (basato sullo stato corrente) che sarebbe potuto diventare anche parecchio complesso.
Inoltre, utilizzando una sola entità avrei compromesso l’espandibilità dell’applicazione, cosa che invece credo possibile con la soluzione implementata. Ad esempio per aggiungere funzionalità come l’editor delle mappe sarà sufficiente creare un nuovo stato a partire dalla classe astratta (e relativo environment) e un nuovo caso nello switch del context.



Player object

All'inizio della progettazione l'entità player non era stata modellata: si era pensato di considerare il giocatore come un qualsiasi altro game object statico. Durante la fase di sviluppo, però, ci siamo resi contro che gli oggetti di gioco hanno comportamenti e caratteristiche spesso troppo diversi per poter essere modellati dalla stessa classe. Si è scelto quindi di applicare il pattern "template method". Potendo assegnare al game object astratto un comportamento particolare ho potuto modellare la presenza di mazze diverse che il giocatore può utilizzare nelle diverse fasi di gioco in base al tipo di tiro necessario per raggiungere la buca. Un altro aspetto che è stato possibile modellare grazie a questa strategia è il cambiamento della skin del personaggio, magari in base all'ambientazione o, in un futuro sviluppo dell'applicazione, a piacimento dell'utente.
Un'alternativa che ho valutato è quella di gestire la mazza come un ulteriore (e quindi indipendente) oggetto di gioco. Durante le fasi finali del progetto ho realizzato che questa soluzione potrebbe essere la più adatta, tuttavia la tardività con la quale sono giunto a questa conclusione non mi ha permesso di implementarla.

In uno sviluppo fuori dai limiti di questo corso credo sarà necessario attuare questa soluzione, soprattutto per poter offrire un'esperienza di gioco più completa (scelta di una mazza ad ogni tiro, personalizzazione della mazza, animazione della mazza…). Nonostante ciò credo che l'implementazione corrente sia comunque sufficiente per la realtà che avevamo in mente di modellare all'inizio della progettazione.

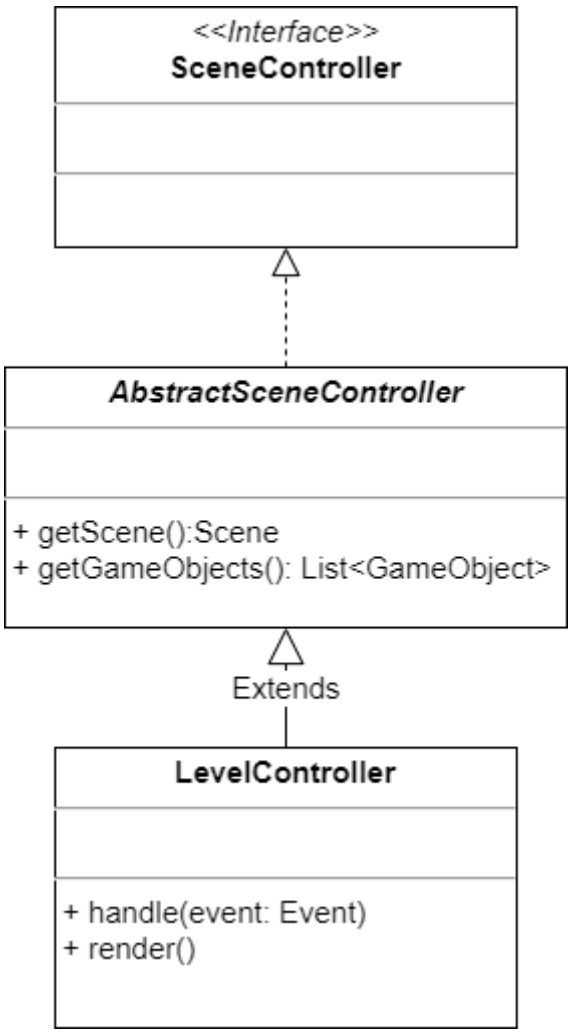


Level controller

La gestione degli eventi di input (azione su un pulsante, pressione del mouse…) è stata incapsulata nei vari SceneController. Io, dovendo gestire la meccanica di tiro, mi sono occupato maggiormente del LevelController, cioè il controller della scena di gioco. Per scena di gioco si intende la componente che si interfaccia direttamente con l'utente.

Anche per questo aspetto, come per gli oggetti di gioco, abbiamo attuato il pattern "template method". Grazie ad esso abbiamo potuto modellare scene di esecuzione diverse, ognuna con delle specifiche azioni disponibili. Infatti il LevelController

gestisce input dall'utente che gli altri controller non hanno necessità di gestire (pressione e rilascio del mouse).



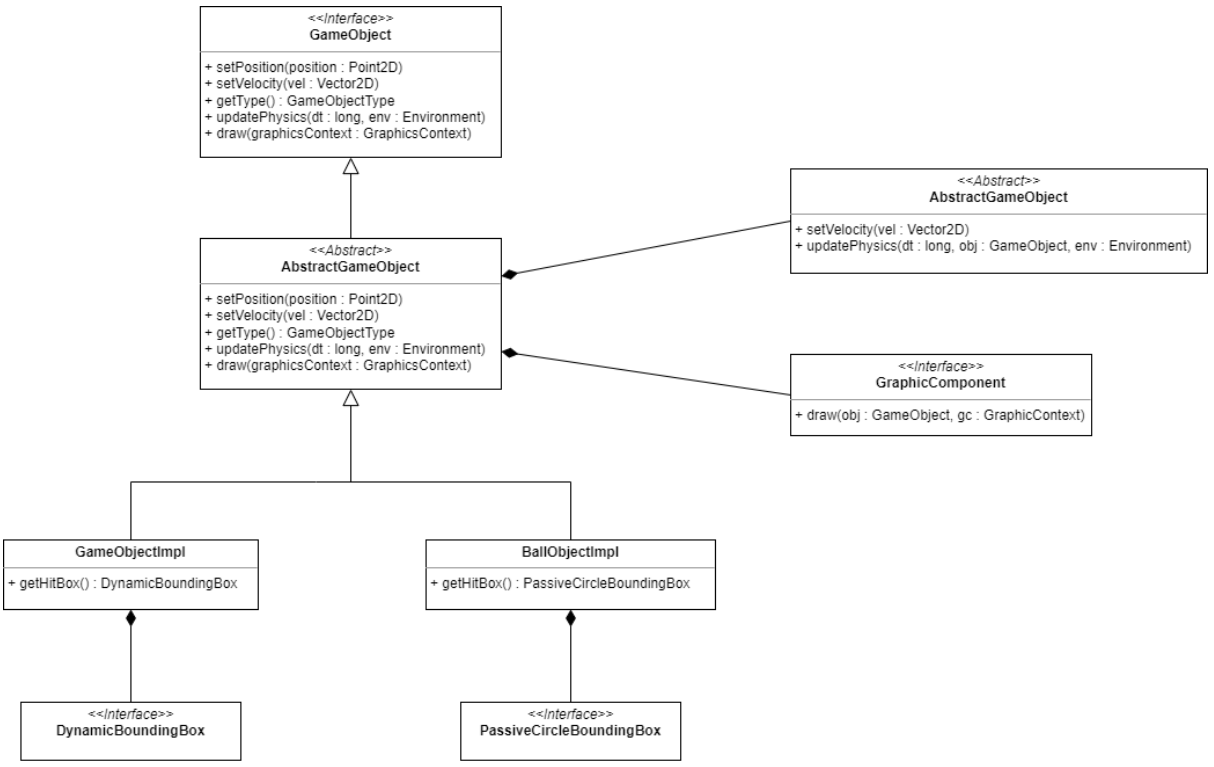
Giorgio Fantilli

Game Object

Per la definizione degli oggetti di gioco, intesi come la palla con la quale fare buca, il player e tutti gli ostacoli dell'ambientazione, avevo la necessità di creare una entità che li rappresentasse tutti, pur avendo ognuno di essi un diverso comportamento fisico e grafico, oltre ad una specifica hitbox per forma e tipo di interazione con le altre.

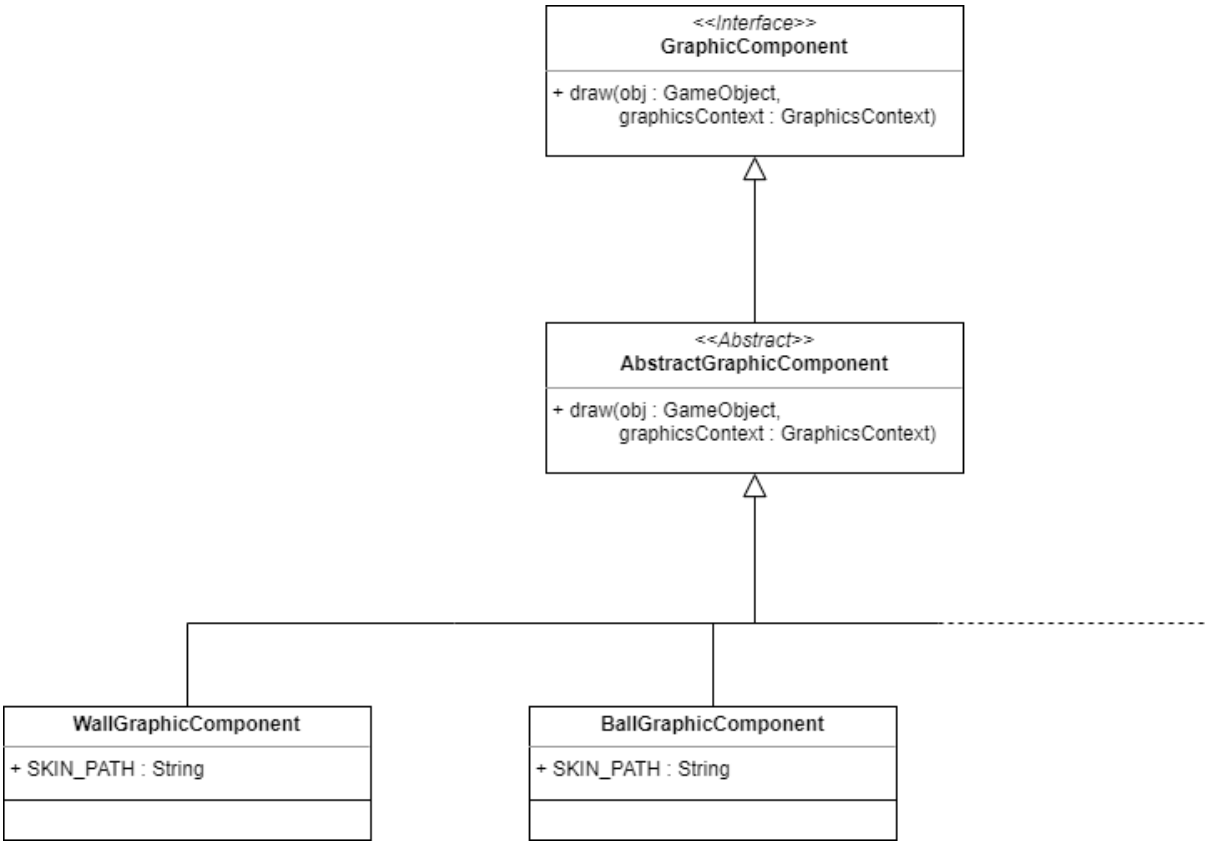
Ho quindi optato per l'adozione del pattern **Proxy**, andando ad implementare un'interfaccia comune a tutti i game object, e fornendo ad ognuno di essi un componente fisico ed uno grafico, delegando al primo l'update dello stato fisico

dell’oggetto nel tempo, mentre al secondo il render grafico per frame: ciò consente un’elevata scalabilità poichè è possibile implementare molteplici nuovi componenti (che rispettino il relativo contratto pubblico) per aggiungere nuove skin, con eventuali specifici effetti grafici, e quindi nuovi ostacoli al gioco, ad ognuno dei quali è possibile fornire diversi comportamenti fisici personalizzabili (es. nel nostro caso abbiamo definito solo ostacoli statici ma è facilmente possibile definirne di dinamici).
Durante lo sviluppo, confrontandomi con il mio compagno responsabile delle collisioni, ci siamo resi conto che la palla, essendo colei che colliderà con gli ostacoli in gioco, avesse necessità di utilizzare un’apposita hitbox, diversa da quelle degli altri oggetti. Ho quindi deciso di definire due classi game object, distinte per il tipo di hitbox da utilizzare, ma accomunate da un comportamento base che ho implementato all’interno di un’apposita classe astratta, applicando il **Template** pattern ed aprendo la possibilità a definizioni di oggetti di gioco custom.
La possibilità di implementare e assegnare ai game object diversi tipi di hitbox, supportata dal relativo adattamento della fisica della palla, permette anche la possibilità di definire ostacoli che abbiano un effetto sulla palla e sul suo stato del tutto personalizzabile.



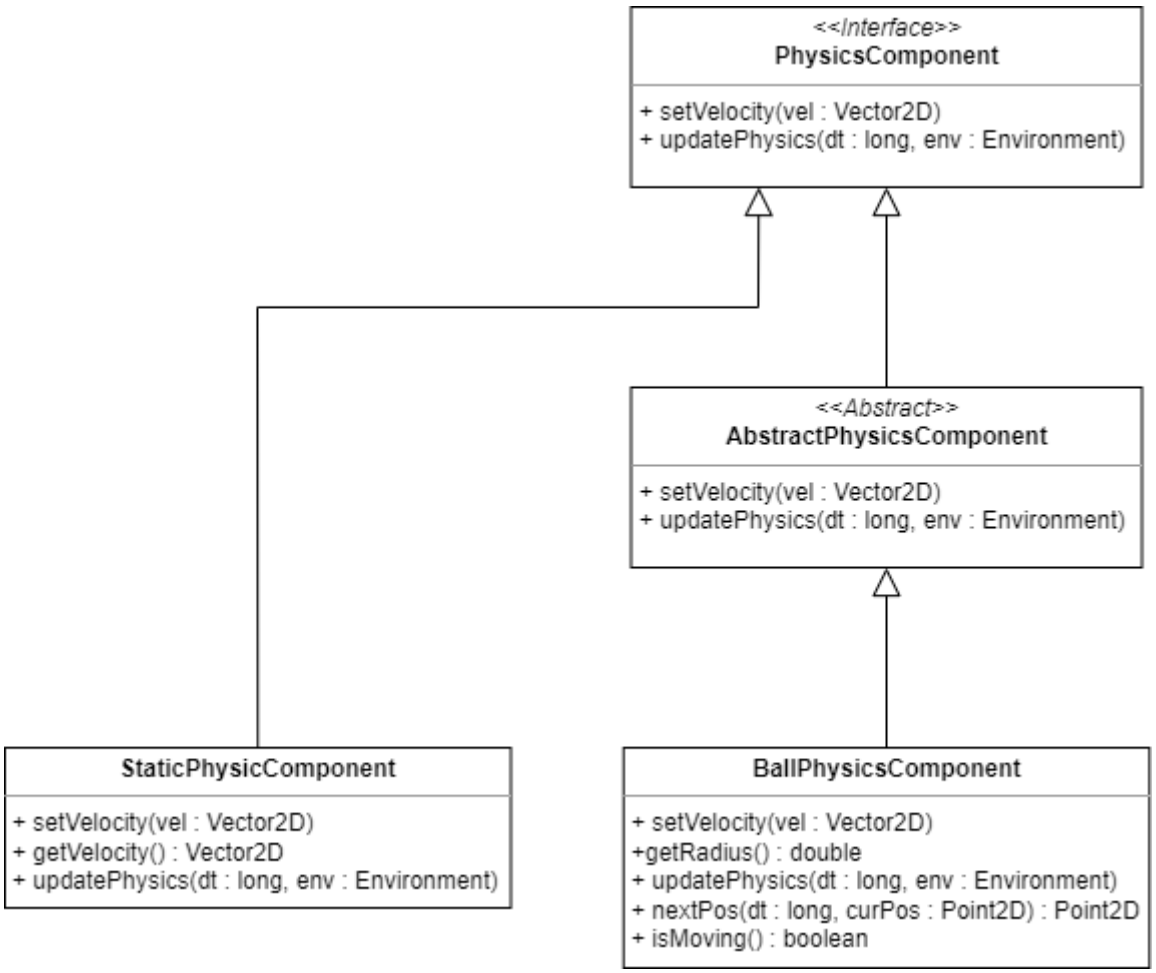
Graphic Component

Le informazioni grafiche (dimensione e skin) e il relativo comportamento di renderizzazione ad ogni frame sono quindi definite dal graphic component, uno per ogni game object. Anche in questo caso ho deciso di adottare il **Template** pattern, creando un'interfaccia comune e la relativa classe astratta che ne definisce il comportamento di base: precisamente mette a disposizione il metodo *draw()* il quale si occupa di renderizzare l'oggetto associato nella scena corrente.
Ciò permette ad ogni membro del gruppo di estendere ed implementare un componente grafico per ogni oggetto di gioco da aggiungere, che ne definisca le specifiche informazioni grafiche e lasciando a lui la scelta se adottare il comportamento definito dalla classe astratta o se sovrascrivere il metodo *draw()* ed alterarlo a piacimento.



Physics

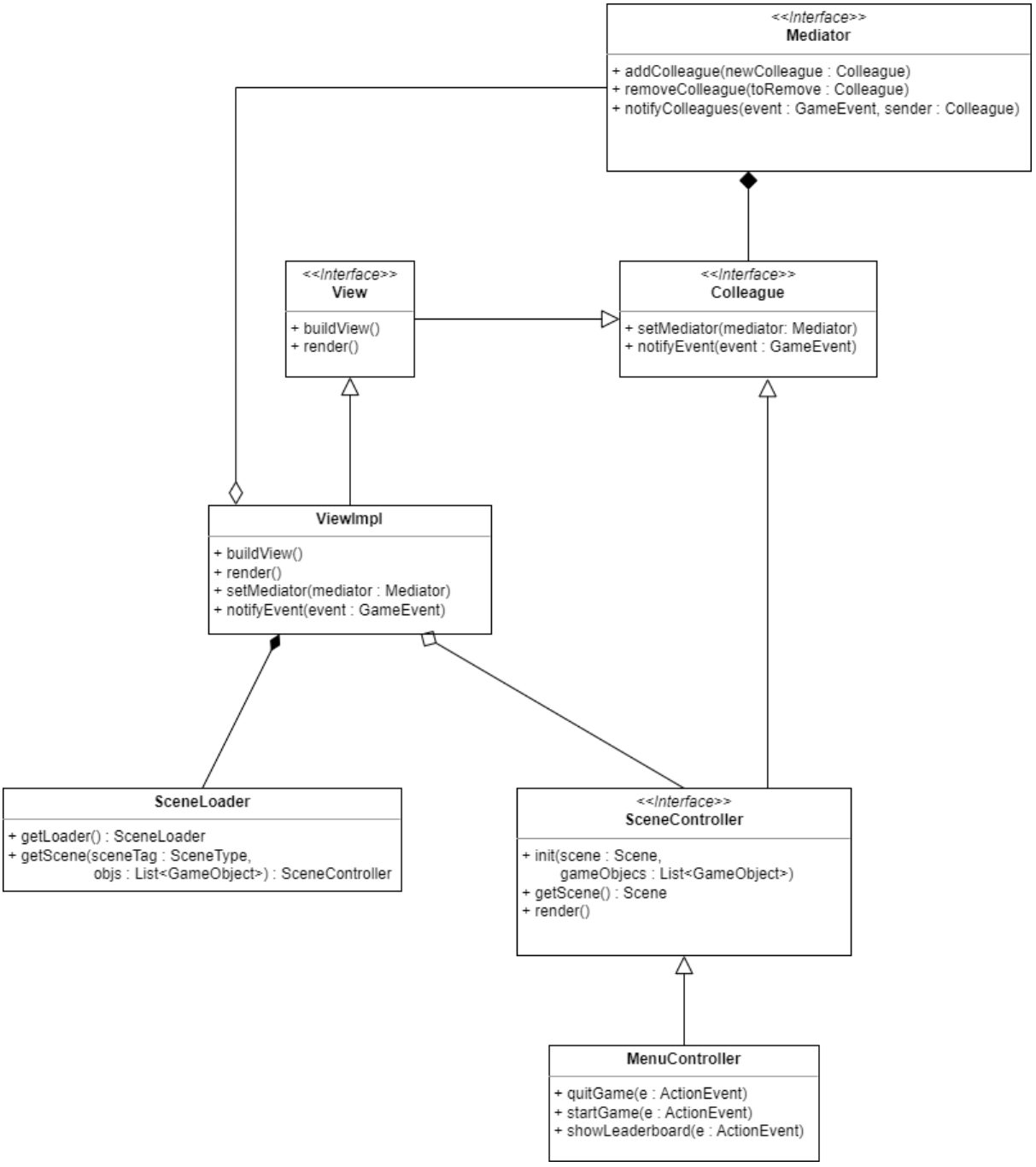
Per l’implementazione dei componenti fisici dei vari game object ho deciso di seguire la medesima struttura adottata per il design dei graphic component, andando perciò a definire un'interfaccia generale ed una classe astratta che descriva le operazioni e gli attributi di base (questo approccio consente come detto sopra di aggiungere ulteriori componenti che implementino comportamenti specifici di altri game object). In questo caso ho però optato per una diversa gerarchia, definendo una classe specifica per i game object statici, che avesse un suo comportamento ben definito e indipendente dalla classe astratta.
Quest’ultima infatti ricopre il ruolo di struttura base ma solo per i componenti fisici “in movimento”, cioè quelli che effettivamente necessitano di un aggiornamento del proprio stato in base al tempo trascorso.
Nello specifico la nostra applicazione fa uso del *BallPhysicsComponet*, che è il componente fisico della palla, il quale all’interno del metodo *update()* chiede di controllare eventuali future collisioni con gli ostacoli di gioco, e, in base al verificarsi o meno delle stesse, calcola la successiva posizione della palla ed adatta di conseguenza direzione, verso e modulo della velocità risultante. La suddetta classe mette a disposizione anche il metodo *nextPos()* il quale, basandosi sul tempo trascorso e sulla precedente posizione dell’oggetto, implementa le leggi orarie del moto del proiettile e restituisce la prossima posizione attesa, senza tener conto delle possibili collisioni: tale metodo è di fondamentale importanza poiché permette la rilevazione delle collisioni future e di ricavare un realistico punto di rimbalzo.



View

La view dell'applicazione viene implementata attraverso l'omonimo contratto pubblico e la relativa classe ViewImpl, adottando il pattern **Mediator**: l'interfaccia estende *Colleague* permettendo lo scambio di eventi ed informazioni con il controller e quindi con il model, senza avere però riferimenti diretti l'uno all'altro. Precisamente il metodo *notifyEvent()* si occupa di catturare e gestire tali eventi, che nel caso della view si limitano alla sola notifica di cambiare e caricare una diversa scena (una specifica mappa, il menu, il game over, ecc.).
La scena richiesta viene letta da file e restituita dallo *SceneLoader*, mentre la view si occupa di inizializzarla e mostrarla a schermo.
Nella realtà la view non si compone della vera scena della libreria JavaFX, ma si interfaccia con essa attraverso uno *SceneController*, che per l'appunto incapsula la

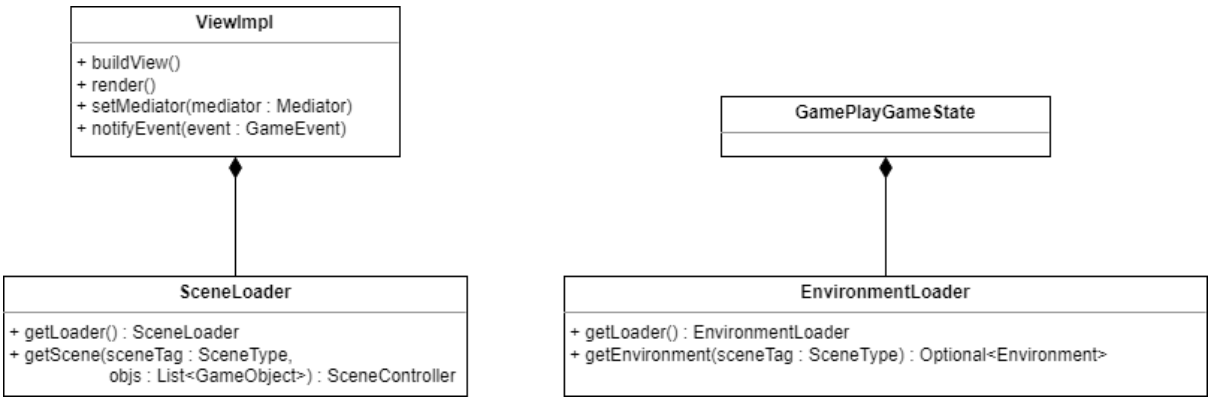
scena ed i relativi handler. Personalmente mi sono occupato di implementare il controller della schermata di menù, il quale come detto fornisce tre handler associati agli altrettanti pulsanti presenti nella schermata.
Essendo anche lo scene controller un *Colleague* (viene aggiunto alla comunicazione dalla view all'instanziazione della scena, e rimosso al cambio della stessa), gli handler dei pulsanti si occupano di inviare, attraverso il mediatore, uno specifico evento di gioco, che verrà poi ricevuto e gestito dall'entità competente.



Loader

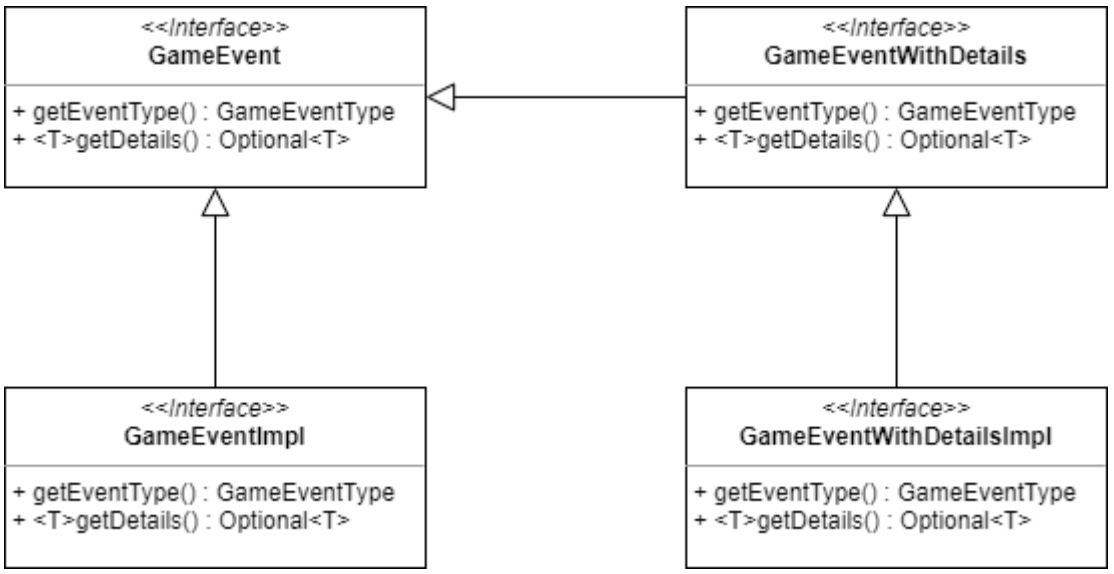
Per rappresentare e memorizzare le mappe di gioco e le schermate dell'applicazione abbiamo optato per il loro caricamento da file, attraverso due classi specializzate: l'*EvironmentLoader* e lo *SceneLoader*. Per entrambe ho deciso di adottare il pattern **Singleton**, poichè ho ritenuto fosse più intuitivo e di facile utilizzo evitare una nuova istanziazione dei loader ad ogni cambio di scena, escludendo anche la possibilità di creare una serie infinita di stream da file.
Entrambe le classi forniscono un metodo getter che ritorna la scena o l'environment indicato dall'argomento, attraverso una specifica enumerazione.
Il loro comportamento interno è però leggermente differente:

- Il loader dell'environment si occupa di caricare il model delle sole mappe di gioco (le schermate dell'applicazione non hanno un environment associato, in questo caso il loader tornerebbe un opzionale vuoto). Quest'ultime sono modellate attraverso file json, i quali contengono le informazioni su posizione, dimensione, e skin della palla, del player, della buca, e degli eventuali e multipli ostacoli di gioco.
  La classe si occupa quindi di leggere i tag json, attraverso la libreria esterna *org.json*, e passare tali informazioni ad una istanza interna dell'*EnvironmentBuilder*, il quale istanzia i vari game object letti ed infine ne costruisce il relativo environmet.
- il loader della scena invece si occupa di caricare sia le mappe di gioco lato grafica e sia di caricare le schermate ed i menù dell'applicazione.
  Nel primo caso il suo comportamento è molto simile a quello del loader dell'environmet, ma la differenza sta nel fatto che si limita alla sola lettura del path del background della mappa e delle scale di dimensione della stessa.
  Nel secondo caso invece le schermate sono memorizzate come file FXML, lasciando quindi alla libreria JavaFX il compito di costruire la gerarchia di classi che costituiscono la scena.



Game Events

Per la comunicazione in broadcast attraverso il *Mediator* ho deciso che ogni notifica avrebbe dovuto consegnare ai vari colleghi uno specifico evento: *GameEvent* è l'interfaccia che incapsula tutta la gerarchia e consente a tutti di poter accedervi in maniera standardizzata.
L'implementazione diretta di tale interfaccia rappresenta un evento senza dettagli, che perciò dichiara, attraverso il metodo *getEventType()*, di che tipo di evento si sta parlando (game over, win, ecc.) ma non contiene ulteriori informazioni (infatti il metodo *getDetails()* in questo caso restituisce sempre un opzionale vuoto).
Ho poi deciso di estendere il sopracitato contratto pubblico con un'ulteriore interfaccia *GameEventWithDetails*, la quale rappresenta invece un evento che possiede, oltre al suo tipo, anche del contenuto informativo utile alla gestione dello stesso. Quest'ultima interfaccia e la relativa classe fanno uso di generici, per fornire piena libertà sull'informazione da incapsulare nell'evento; anche il getter di tali dettagli fa uso di generici, ed al suo interno implementa perciò un controllo sul possibile mismatch tra il tipo fornito al metodo, ed il tipo effettivo del dettaglio a runtime.



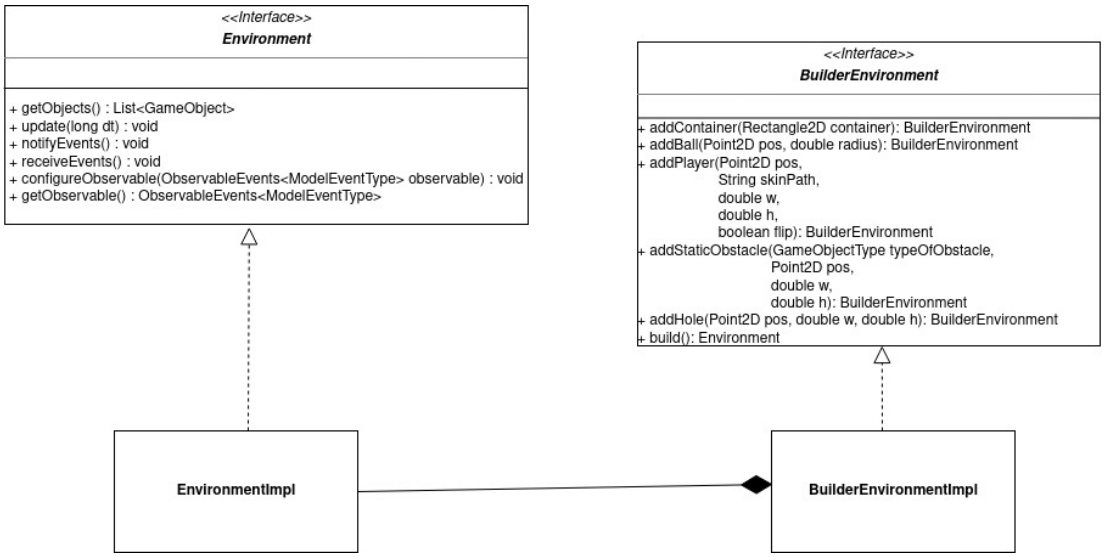
Domenico Francesco Giacobbi

Ambientazioni di gioco:

Per modellare le ambientazioni di gioco ho deciso di dividere il processo di creazione delle mappe dalla loro gestione.
La creazione delle ambientazioni è stata realizzata mediante l'interfaccia BuilderEnvironment, la quale sfrutta il pattern creazionale builder.
Tale scelta progettuale è stata intrapresa in quanto le mappe sono ricostruite a partire dalle informazioni contenute in appositi file json. In relazione a ciò è stata riscontrata la necessità di definire una strategia separata che permettesse di creare step-by-step l'oggetto che si occupa della gestione delle mappe partendo dalle informazioni estrapolate da ogni tag.
Abbiamo realizzato la gestione delle ambientazioni mediante l'interfaccia Environment, la cui implementazione (EnvironmentImpl): svolge le seguenti operazioni:

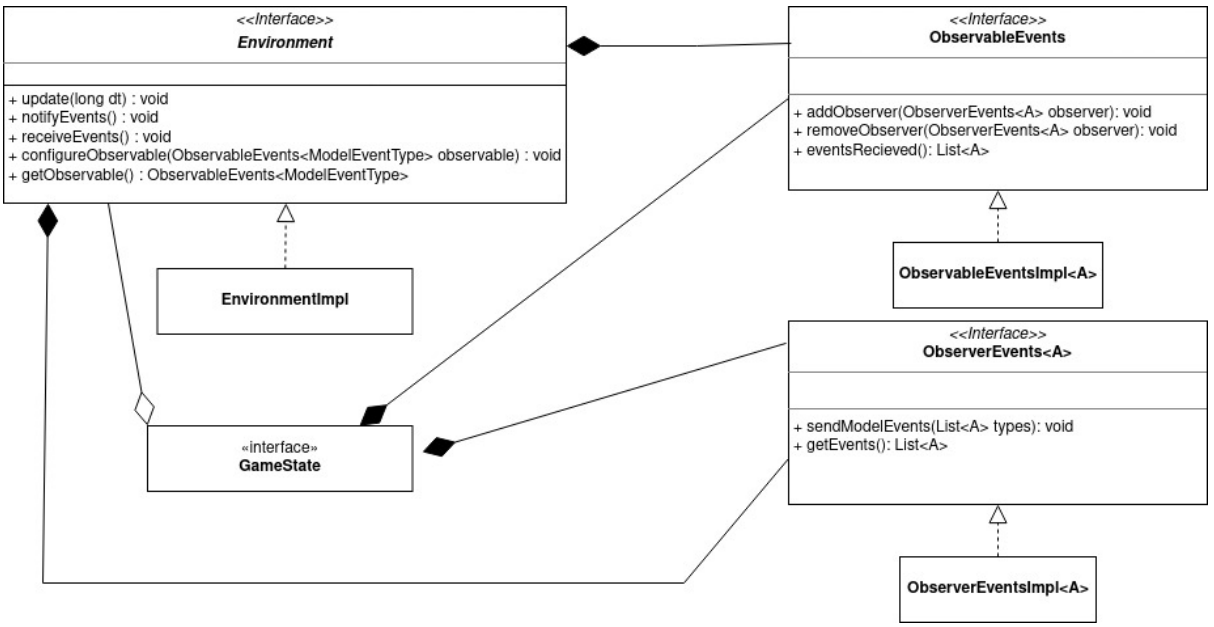
- incapsulare i GameObject presenti all'interno della mappa gestita
- richiamare l'aggiornamento dello stato fisico dei GameObject
- rilevare gli eventi avvenuti nell'attuale configurazione dei GameObject(palla ferma,palla fuori dai bordi,palla in buca) e notificarli al GameState
- ricevere gli eventi inviati dal Gamestate(giocatore in fase di tiro, giocatore da spostare), in base ai quali effettua delle apposite operazioni sui GameObject

L’Environment inoltre si occupa della gestione dello spostamento del personaggio quando la palla si è fermata su una posizione differente dalla buca e il numero di vite è maggiore di 0. Tale operazione viene effettuata alla ricezione del relativo evento.
Inizialmente questa funzionalità doveva essere gestita da Luca Lucioli e io dovevo implementare la gestione delle condizioni di vittoria e sconfitta.
Successivamente ci siamo accorti che la mia parte era inclusa nelle funzioni del GameState, e la sua invece nelle competenze dell’Environment.
In relazione a ciò abbiamo deciso di effettuare uno scambio dei ruoli.



Lo scambio di eventi che avviene tra GameState ed Environment è gestito mediante il pattern comportamentale Observer.
In particolare nell’Environment e nel GameState vengono istanziati due oggetti, uno Observer e un Observable.
L’Observer si occupa dell’invio degli eventi, mentre l’Observable si occupa della loro ricezione.
Ogni Observable viene passato all’altro oggetto, inserendo in esso il proprio Observer.
In questo modo ogni classe è in grado di notificare eventi richiamando il metodo SendModelEvents del proprio Observer e allo stesso tempo di ricevere gli eventi dell’altra classe utilizzando il metodo eventsReceived dell’ Observable fornito.
Inizialmente è il GameState a inviare il proprio Observable e a prelevare quello dell’Environment.

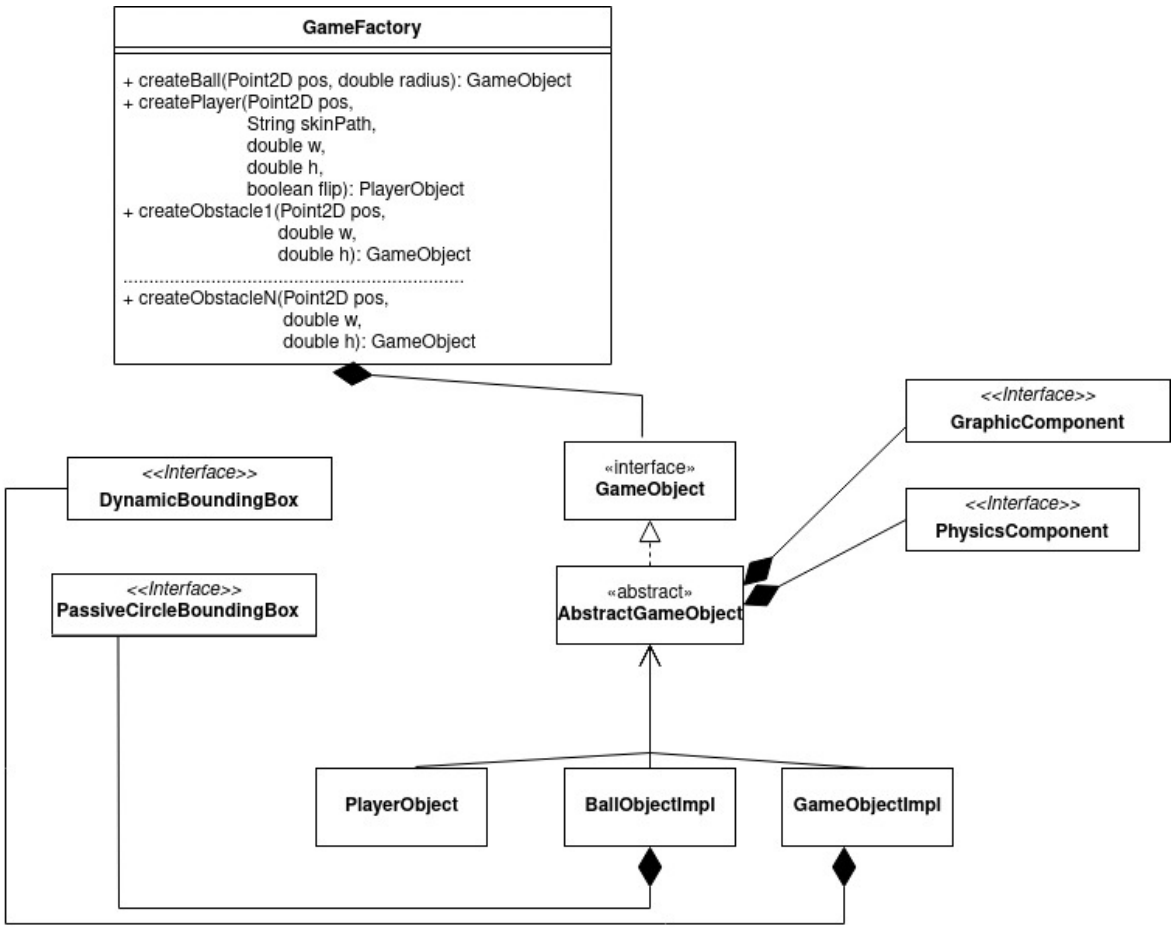
Tale scelta progettuale è stata effettuata per non inserire il GameState all'interno dell' Environment, è il GameState a gestire le mappe e non il contrario.



Gestione e creazione degli elementi di gioco

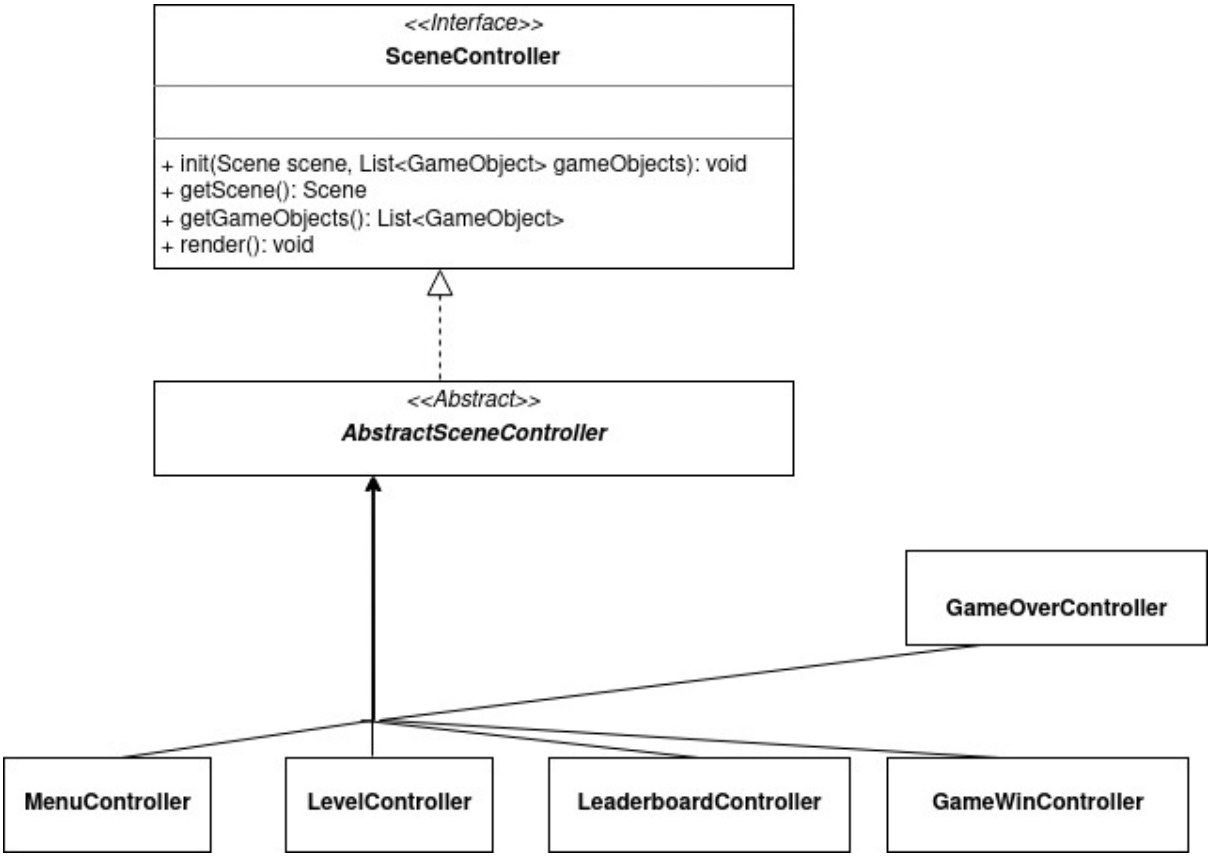
Per la creazione dei GameObject, presenti all'interno di ogni ambientazione di gioco, ho utilizzato il pattern creazionale Factory Method.

Nella classe GameFactory è presente, per ogni tipo di GameObject, un apposito metodo che ne permette la creazione.
In ogni metodo viene restituita l'interfaccia GameObject e non una sua implementazione, fatta eccezione per il giocatore (player).
Tale scelta progettuale è stata effettuata in quanto ciò rende possibile l'accesso alle dimensioni del giocatore, da parte dell'Environment, senza utilizzare il suo componente grafico e quindi non violando il pattern M.V.C.
I tipi di GameObject sono inseriti all'interno di un'apposita enumerazione.
Per ogni elemento di gioco vengono istanziati e assegnati un componente grafico, un componente fisico e un hitbox appropriati al suo tipo.
Tale procedura viene effettuata anche per gli ostacoli, assegnando loro una fisica senza movimento (StaticPhysicsComponent).



Controller della view:

In JavaFX la view si interfaccia alla scena mediante un controller, il quale incapsula la scena e gli handler associati.
Tale controller è stato realizzato mediante l'interfaccia SceneController.
Inizialmente volevo realizzare il controller in una sola classe che gestisse tutte le scene.
Successivamente mi sono accorto che tale scelta progettuale era poco scalabile e non funzionale al nostro utilizzo.
Di conseguenza ho deciso di utilizzare il pattern template method, creando un classe per ogni scena che ne realizzasse il relativo controller.
Tutte le classi create estendono una classe astratta che contiene i metodi in comune ai vari controller.



Realizzazione della view delle ambientazioni di gioco:

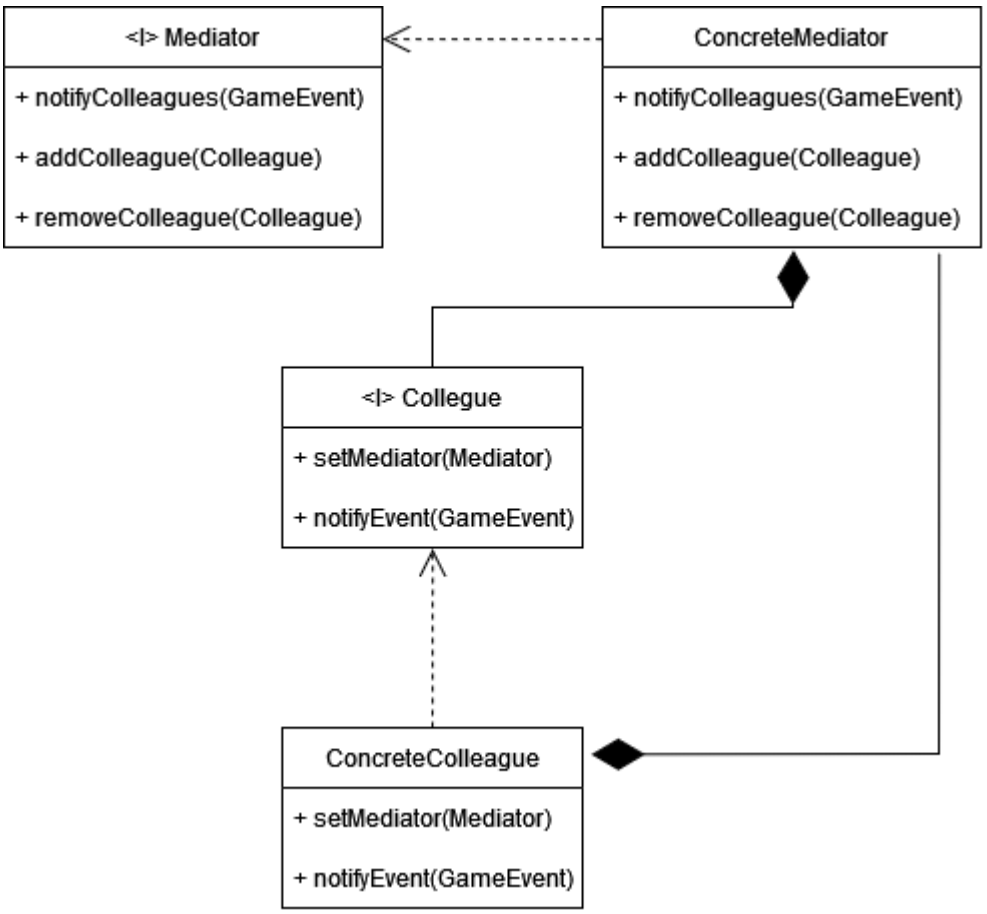
Ho realizzato la view delle varie ambientazioni di gioco, inserendola in un apposita scena realizzata nella classe EnvironmentScene.

Lorenzo Colletta

Notifica

A seguito di un’attenta analisi, volta a rispettare il modello dettato dal pattern M.V.C., abbiamo stabilito come il verificarsi di determinate condizioni, rispettivamente nella suddivisione di Model e in quella di View, corrispondano alla generazione di eventi che comportano azioni in tutte e tre le suddivisioni.

La necessità di notificare il verificarsi di un determinato evento ad una o più delle componenti nel rispetto delle interrelazioni previste dal pattern M.V.C. mi ha portato all'uso del pattern Mediator.
Nell'implementazione adottata, le componenti individuate, vista la loro natura strettamente legata al ruolo definito dal pattern Mediator, mantengono i nomi previsti dal pattern.
L'interfaccia Colleague identifica una componente in grado di generare eventi (e dunque notificarli) e di riceverne da altri Colleague. Una classe che abbia la necessità di notificare e/o ricevere gli eventi deve implementare l'interfaccia Colleague.
L'interfaccia Mediator identifica l'oggetto a cui viene delegato il compito di notificare un evento a tutti i Colleague.
La classe ConcreteMediator implementa l'interfaccia Mediator e si occupa di mantenere traccia di tutti i Colleague coinvolti e di notificarli alla ricezione di un evento.
Ogni qual volta un Colleague abbia la necessità di notificare altri Colleague, esso delega al ConcreteMediator la notifica dell'evento generato. È poi compito del Colleague stesso riconoscere la natura dell'evento ricevuto e di eseguire le dovute azioni.



Collisioni

Alla base delle scelte implementative delle collisioni risiede la necessità di rilevare l'intersezione della pallina con gli ostacoli che popolano l'ambiente di gioco.
Questa prima considerazione mi ha permesso di ridurre l'intero sistema delle collisioni alla sola verifica di intersezione tra coppie di oggetti di cui una è sempre la pallina.
Su tale osservazione ho individuato due categorie di contratti: quello descritto dall'interfaccia ActiveBoundingBox, che consiste in un'entità con una data rappresentazione geometrica in grado di effettuare dei test di intersezione con delle figure geometriche; quello descritto dall'interfaccia PassiveBoundingBox, che consiste nella rappresentazione geometrica utilizzata per identificare l'area occupata dalla palla.
I test di intersezione effettuati da un'implementazione di ActiveBoundingBox sono:

- Verifica di intersezione con un PassiveBoundingBox

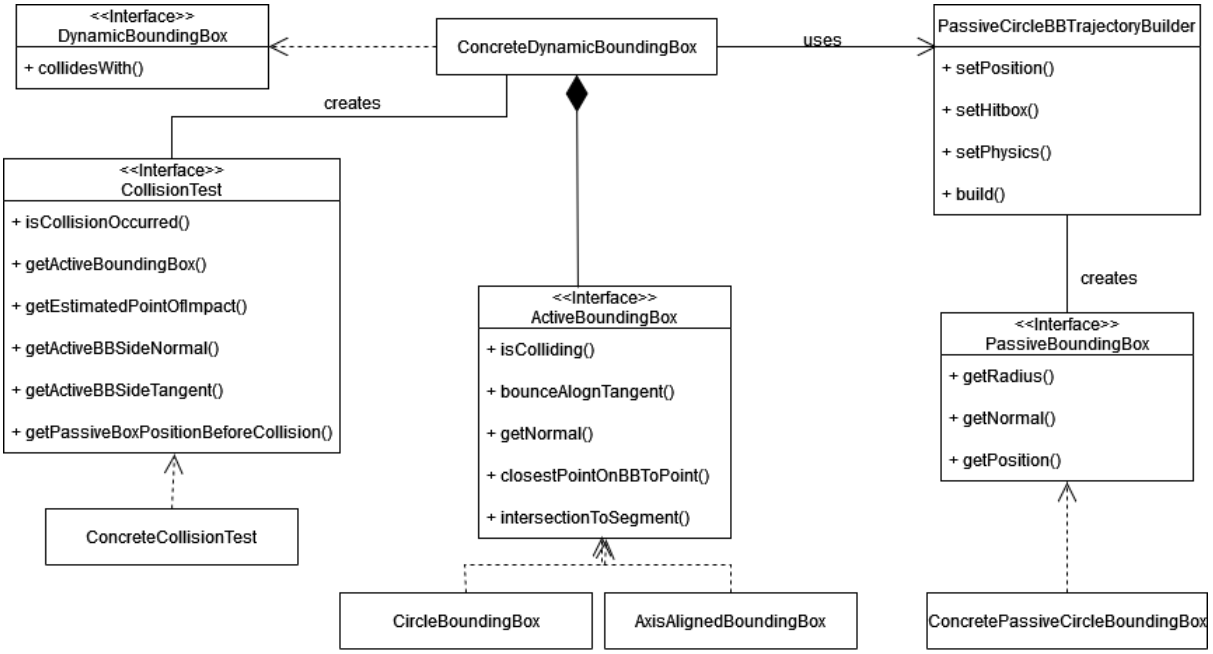
- Rilevamento del punto di intersezione con un segmento
- Rilevamento del punto appartenente alla propria area geometrica più vicino ad un dato punto
- Ricavo del versore perpendicolare ad un punto appartenente alla propria area geometrica

Le implementazioni CircleBoundingBox e AxisAlignedBoundidngBox descrivono rispettivamente un'ActiveBoundingBox la cui rappresentazione geometrica consiste in una circonferenza ed un'ActiveBoundingBox con rappresentazione geometrica di un rettangolo con lati paralleli agli assi di riferimento.
Questo primo design è in grado di verificare le intersezioni tra entità statiche: vale a dire tra un ostacolo e la pallina ad un tempo preciso. Un problema con i test statici è che quando il movimento della pallina aumenta tra un tempo e il successivo, aumenta anche la probabilità che la pallina salti semplicemente oltre l'ostacolo.
La soluzione adottata prevede di campionare il percorso della pallina e di effettuare molteplici test statici con l'ostacolo.
Al fine di riusare l'implementazione fin qui descritta ho definito l'interfaccia DynamicBoundingBox, la quale rappresenta un wrapper per un'ActiveBoundingBox, preferendo la composizione all'ereditarietà per non dover riscrivere lo stesso codice applicato a implementazioni diverse di ActiveBoundingBox.
DynamicBoundingBox ad ogni verifica di intersezione crea un oggetto definito dall'interfaccia DynamicBoundingBox.CollisionTest, la quale rappresenta un oggetto che mantiene le informazioni sul test effettuato, quali:

- Il verificarsi o meno della collisione
- L'oggetto ActiveBoundingBox con cui è avvenuta la collisione, se verificatasi
- Il primo punto di intersezione, in caso di avvenuta collisione
- Il versore perpendicolare al punto di intersezione, in caso di avvenuta collisione
- L'ultima posizione della pallina registrata

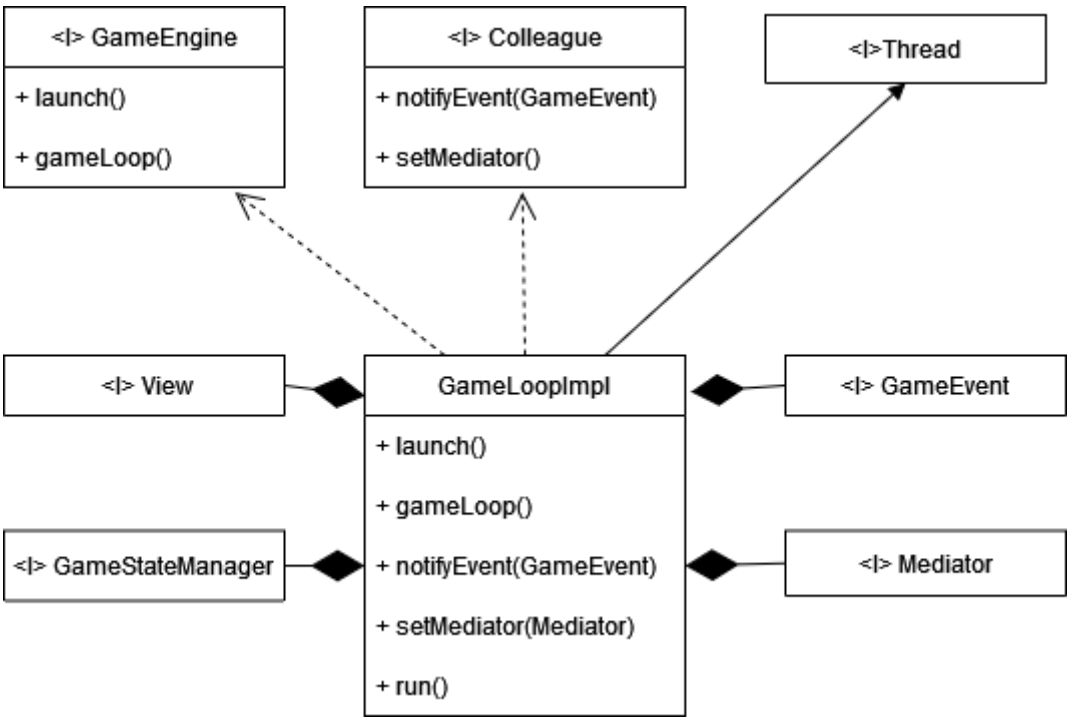
Per i test effettuati da un DynamicBoundingBox si vede la necessità di spostare la PassiveBoundingBox utilizzata per rappresentare l'area occupata dalla pallina. Data questa necessità ho adottato il pattern Builder per avere un modo flessibile di creare

un oggetto che rappresenti l’hitbox della pallina in una data posizione lungo la traiettoria del suo movimento.



GameLoop

L’intera applicazione può essere riassunta in due principali operazioni: aggiornamento dello stato e aggiornamento dell’interfaccia grafica. GameEngineImpl esegue queste operazioni in loop, seguendo il pattern GameLoop. GameEngineImpl implementa anche l’interfaccia Colleague per ricevere la notifica di chiusura dell’applicazione.



Capitolo 3

Sviluppo

3.1 Testing automatizzato

I testi automatizzati che abbiamo deciso di effettuare riguardano principalmente quattro aspetti del progetto: l' Environment, lo stato di gioco, la palla e le collisioni. Per quanto riguarda i primi due ci siamo limitati a testare la corretta inizializzazione delle componenti. Nel test dell'environment abbiamo inoltre verificato il corretto funzionamento del suo builder.

Il testing della palla ci ha permesso di controllarne lo stato (ferma o in movimento) e le posizioni assunte nello spazio di riferimento al variare della velocità e del tempo.

L'ultimo test, forse il più significativo, è quello sulle collisioni: date alla palla e ad un ostacolo statico due posizioni volutamente sovrapposte abbiamo verificato l'avvenuta collisione. Il medesimo test è stato effettuato anche per controllare il contatto della palla con la buca.

3.2 Metodologia di lavoro

Nella prima fase di analisi del progetto abbiamo riscontrato difficoltà nell'individuare dei ruoli distinti per i membri del gruppo, in quanto non riuscivamo ad avere una visione concreta della futura struttura dell'applicazione e in particolare delle sue componenti. In seguito a numerose sessioni di brainstorming siamo convenuti sulla progettazione pubblica degli attori principali del pattern MVC. In particolare abbiamo definito le seguenti interfacce: lo stato di gioco, il loop, l'environment e la view.
A questo punto è stato possibile assegnare i ruoli ai singoli componenti, in modo che potessero portare avanti lo sviluppo delle proprie porzioni di progetto senza particolari dipendenze reciproche.
Si è deciso di optare per un avanzamento in parallelo, permettendo così ai membri di effettuare autonomamente le scelte implementative considerate più opportune.
Nella fase di integrazione tra le varie porzioni è stata ovviamente necessaria una maggiore collaborazione tra due o più membri.
Durante le fasi finali, soprattutto quella del bugfix, è stato necessario "rompere" la suddivisione definita per rendere il processo più semplice e rapido.
Il DVCS è stato sfruttato assegnando ad ogni membro del gruppo vari feature branch da gestire individualmente. Raggiunta una condizione stabile dell'implementazione, il branch veniva unito a master in modo da rendere disponibili le funzionalità sviluppate al resto del gruppo.
Nelle fasi più critiche (bugfix) ci siamo trovati a lavorare spesso su pochi branch comuni. Sebbene questa pratica sia poco indicata, nel suddetto contesto ci è sembrato opportuno e più efficiente operare congiuntamente.

Domenico Francesco Giacobbi

Sviluppo della GameFactory e degli ostacoli di gioco.
Collaborazione alla creazione del classe GameObject.
Sviluppo delle ambientazioni di gioco(Environment) e del builder che si occupa della loro creazione.
Sviluppo del pattern observer per lo scambio di eventi tra l' Environment e il GameState.
Gestione del riposizionamento del personaggio.

Creazione dei controller della view e sviluppo della scena che si occupa della visualizzazione delle ambientazioni di gioco.
Creazione della fisica di un oggetto fermo (StaticPhysicsComponent)
Sviluppo di test.

Lorenzo Colletta

Implementazione pattern “mediator”, file manager, game loop, gestione collisioni (intero package model.collisions), salvataggio e lettura delle statistiche e relativa interfaccia grafica. Ho collaborato alla gestione dei rimbalzi nella classe BallPhysicsComponent.

Luca Lucioli

Ho sviluppato GameState (interfaccia, classe astratta ed estensioni), GameStateManager (interfaccia ed implementazioni), PlayerObject (e Bat), LevelController, GameOverController, GameWinController.
Ho collaborato alla classe GameFactory (istanziazione del player) e allo sviluppo dei test.

Giorgio Fantilli

Ho collaborato alla classe GameFactory (istanziazione della palla).
Ho sviluppato EnvironmentLoader, PassiveCircleBBTrajectoryBuilder (su direttive di Lorenzo Colletta), GameEvent (ed estensioni), GameObject (interfaccia, classe astratta e BallObjectImpl), package model.physics (tranne StaticPhysicsComponent), SceneLoader, View, ViewImpl e MenuController.

3.3 Note di sviluppo

Domenico Francesco Giacobbi

Feature avanzate

- Utilizzo degli Stream
- Utilizzo di Lambda Expressions
- Utilizzo di Optional
- JavaFX, in particolare per lo sviluppo della scena dell’ambiente di gioco(realizzata via codice senza l’utilizzo di FXML per adattarla alla dimensione dello schermo e per effettuare una migliore gestione degli eventi).

- Utilizzo di classi generiche per la gestione del pattern Observer che permette lo scambio di eventi tra Environment e GameState

Lorenzo Colletta

- Utilizzo di Optional
- Algoritmo di Intersezione tra segmenti
- Algoritmo di intersezione tra segmento e circonferenza
- Algoritmo di calcolo del punto di una Axis-Aligned Bounding Box più vicino ad un punto dato
- Algoritmo di calcolo del punto di una circonferenza più vicino ad un punto dato

Luca Lucioli

Feature avanzate

- Stream, per rendere più leggibile la lettura della lista di eventi di tipo model (classe GamePlayGameState);
- JavaFX (in particolare FXML), per la creazione delle interfacce e la gestione dell'interazione dell'utente (con il mouse o con gli oggetti grafici).

Giorgio Fantilli

- Utilizzo di generici nel design dei GameEvent, in particolare dei dettagli informativi;
- Gestione dei file FXML attraverso classi della libreria JavaFX per la costruzione delle schermate dell'applicazione, definite su file;
- Utilizzo di Optional nei casi in cui non era possibile prevedere la restituzione di un oggetto concreto, o nel caso di utilizzo di attributi possibilmente vuoti;
- Sfruttamento della libreria esterna org.json per la lettura automatizzata dei file json che definiscono gli oggetti presenti nelle varie mappe;
- Sviluppo autonomo dell'algoritmo che implementa il moto del proiettile della palla, adattato alle dimensioni e al piano di gioco, con gestione delle collisioni contro oggetti rettangolari e rilevamento automatico condizione di stop della palla.

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

Domenico Francesco Giacobbi

La realizzazione del progetto è stata una sfida difficile da affrontare in quanto rappresenta il primo lavoro informatico di grandi dimensioni che ho sviluppato. Inoltre generalmente sono abituato a lavorare singolarmente, di conseguenza ho dovuto anche superare lo scoglio del lavoro di gruppo.
Questa esperienza mi ha permesso di consolidare le conoscenze acquisite durante il corso di OOP e di integrarne di nuove.
Devo ammettere che inizialmente non sapevo da dove iniziare, come effettuare la progettazione e se mai saremmo stati capaci di unire in una singola applicazione le parti sviluppate da ogni singolo membro del gruppo.
Nell'utilizzo del DVCS, soprattutto nell prime fasi, ho riscontrato alcune difficoltà dovute al fatto che questa era la mia prima esperienza in cui ne ho potuto fare uso. Tuttavia, man mano che lo utilizzavo, sono riuscito ad arrivare ad utilizzarlo con fluidità ed immediatezza.
La cosa che mi è piaciuta di più di questo progetto è la possibilità di lavorare in gruppo, una dinamica tipica del mondo del lavoro. Sono sicuro che, in quest'ottica, troverò molto utile l'esperienza appena conclusa.

Lorenzo Colletta

Il progetto si è presentato come una sfida non indifferente su molteplici fronti: da un lato nel lavorare in gruppo cercando il più possibile di rendere il mio lavoro di facile interpretazione per gli altri membri, in modo da poter essere integrato al meglio; dall'altro nel realizzare un software dalla struttura ben più complessa degli ordinari esercizi di applicazione di un paradigma di programmazione.
Attraverso il progetto ho avuto inoltre occasione di trattare argomenti che da tempo mi destavano interesse ma soprattutto di rielaborarli in modo da rispettare le nozioni assunte attraverso il corso.
Sebbene l'impegno, la scelta di cimentarmi ad una mia implementazione delle collisioni la posso facilmente vedere come una scelta non ottimale. Può infatti aver gravato sul risultato finale e nelle prestazioni in generale. La reputo tuttavia una esperienza formativa in quanto mi ha messo alla prova.
Per uno sviluppo futuro prevedo un'ulteriore estensione del sistema di rilevamento delle collisioni, con l'obiettivo di offrire una maggiore possibilità di scelta tra i possibili ostacoli e dei loro comportamenti.

Luca Lucioli

La realizzazione di questo progetto è stata sicuramente la prova più dura del mio percorso universitario, sia per la quantità tempo necessaria che per la maturità che richiede: credo infatti che per poter sviluppare correttamente e professionalmente un lavoro di questo tipo sia indispensabile un bagaglio di conoscenze in ambito di programmazione non indifferente. Ho trovato altrettanto essenziale la capacità di saper valutare criticamente le proprie idee e quelle degli altri, specialmente quando in contrasto con le proprie. Durante tutte le fasi di questo progetto mi sono trovato a dover discutere le mie scelte e le mie posizioni, confrontarle con quelle dei miei compagni e anche accettare spesso dei compromessi. Avendo lavorato in passato su progetti più piccoli, sviluppati al massimo in due persone, non ero molto abituato al confronto con il gruppo. In questo aspetto sento di essere molto migliorato, e sono certo che mi sarà utile nei progetti che mi coinvolgeranno in futuro.
Grazie alle esperienze passate, però, avevo già una conoscenza basilare circa l'utilizzo del DVCS che mi ha permesso di entrare subito nel meccanismo di sviluppo di gruppo.
Mi sarebbe anche piaciuto sperimentare uno stile di progettazione di tipo test driven; purtroppo in questo progetto abbiamo sviluppato i test solo alla fine.
Per quanto riguarda il mio apporto in questo progetto posso dire di essere piuttosto soddisfatto, in quanto mi sono cimentato autonomamente nella progettazione di una delle entità principali dell'applicativo. Sebbene abbia dei dubbi circa la soluzione che ho deciso di adottare sono comunque fiero delle mie scelte, soprattutto pensando alle incertezze e alla paura che avevo all'inizio della progettazione. Credo inoltre che il fatto che nutra dei dubbi nei confronti delle mie decisioni sia sintomo della possibiltà di migliorare in futuro.

Giorgio Fantilli

Realizzare un progetto di tali dimensioni, lavorando in un team di modeste dimensioni, è stata per me sin da subito una sfida importante. L'idea di cimentarmi nello sviluppo il più possibile professionale e rigoroso mi entusiasmava molto ma al contempo mi portava a domandarmi su quanto fossi effettivamente pronto e competente.
Il risultato ha superato le mie aspettative e sono molto contento sia del lavoro di tutto il gruppo che delle parti che ho personalmente progettato ed implementato.
Sicuramente non sarà stato svolto tutto nella maniera più corretta, ma posso affermare di aver imparato molto dalla realizzazione di questo progetto, ed anzi di essermi appositamente spinto ad una forte partecipazione, soprattutto nella fase di design, proprio per cercare di mettermi alla prova e di realizzare, ricercare, e quindi imparare, come strutturare al meglio future applicazioni simili, o almeno estrapolare uno corretto modo di lavorare.
Quanto detto prima vale anche per l'utilizzo del DVCS, che già utilizzavo personalmente, ritenendolo uno strumento estremamente funzionale, ma questa

esperienza mi ha fatto capire come senza di esso la programmazione, soprattutto se fatta in gruppo, sia notevolmente più difficoltosa, e permette inoltre di sviluppare “senza pensieri”, con la consapevolezza che niente di quello precedentemente fatto potrà mai andar perso ma anzi sia facilmente recuperabile e confrontabile con le modifiche successivamente apportate.

4.2 Difficoltà incontrate e commenti per i docenti

Domenico Francesco Giacobbi

Le difficoltà maggiori che ho riscontrato durante questa esperienza sono, a mio parere, imputabili all’inesperienza. Spero che lo sforzo fatto in questo ultimo periodo sia utile ad arricchire il mio bagaglio.

Luca Lucioli

L’unico commento che sento di fare in questa sezione riguarda la complessità della fase iniziale del progetto. Durante l’ideazione e la progettazione sentivo di non avere bene a fuoco la “big picture”, di non riuscire a comprendere come i vari attori potessero interagire nel modo corretto per offrire le funzionalità che ci prefissavamo, o se fosse effettivamente possibile implementarle nell’architettura che stavamo progettando. Anche dopo dopo le esperienze del laboratorio e dei progetti realizzati extra-corso l’impatto con un progetto di questo portata mi è sembrato davvero duro. Mi sarebbe piacuto essere più pronto ad affrontare un lavoro di questo tipo, sia a livello di mera programmazione che, soprattutto, a livello progettuale. Tuttavia, arrivato a questo punto mi sento sicuramente più pronto ad affrontare un progetto di stampo professionale come questo.

Giorgio Fantilli

La difficoltà più grande è stata quella di comprendere concretamente come strutturare e modellare l’applicazione che ci siamo posti, ed onestamente, senza la possibilità di visualizzare i progetti ben riusciti degli anni precedenti o le soluzioni a problemi implementativi simili viste in rete, sarebbe a mio avviso stato impossibile per me arrivare al risultato che abbiamo ottenuto, con non poca fatica.
Il tutto aggravato da una totale mancanza di formazione ben specifica e sul campo dello sviluppo in gruppo, dell’utilizzo estensivo del DVCS, dell’applicazione rigorosa e frequente dei pattern, a partire dalla struttura MVC.

Appendice A

Guida utente

L'applicazione all'avvio presenta una schermata di menù piuttosto intuitiva.
Una volta avviata la partita l'utente può tirare la pallina. Per farlo dovrà premere il tasto sinistro del mouse in un punto qualsiasi della mappa, trascinarlo per accumulare forza e rilasciarlo quando la crede sufficiente.
Si avvisa che uscendo dalla partita in corso il punteggio accumulato verrà perso, e non sarà consultabile nella sezione "stats".

Appendice B

Esercitazioni di laboratorio

Lorenzo Colletta

- Lab. 06 : https://virtuale.unibo.it/mod/forum/discuss.php?d=87880#p136396
- Lab. 07 : https://virtuale.unibo.it/mod/forum/discuss.php?d=88829#p137110
- Lab. 08 : https://virtuale.unibo.it/mod/forum/discuss.php?d=89272#p138874
- Lab. 09 : https://virtuale.unibo.it/mod/forum/discuss.php?d=90125#p139216
- Lab. 10 : https://virtuale.unibo.it/mod/forum/discuss.php?d=91128#p140267

Giorgio Fantilli

- Lab. 06 : https://virtuale.unibo.it/mod/forum/discuss.php?d=87880#p136916
- Lab. 07 : https://virtuale.unibo.it/mod/forum/discuss.php?d=88829#p140068
- Lab. 08 : https://virtuale.unibo.it/mod/forum/discuss.php?d=89272#p140486
- Lab. 09 : https://virtuale.unibo.it/mod/forum/discuss.php?d=90125#p140780
- Lab. 10 : https://virtuale.unibo.it/mod/forum/discuss.php?d=91128#p142008

Domenico Francesco Giacobbi

- Lab. 05 : https://virtuale.unibo.it/mod/forum/discuss.php?d=87881#p135459
- Lab. 06 : https://virtuale.unibo.it/mod/forum/discuss.php?d=87880#p136235
- Lab. 07 : https://virtuale.unibo.it/mod/forum/discuss.php?d=88829#p136821
- Lab. 08 : https://virtuale.unibo.it/mod/forum/discuss.php?d=89272#p138018
- Lab. 09 : https://virtuale.unibo.it/mod/forum/discuss.php?d=90125#p139340
- Lab. 10 : https://virtuale.unibo.it/mod/forum/discuss.php?d=91128#p140224