

Relazione  
“Geometry-Bash”

Elena Montalti, Luca Latini e Paolo De Mori

6 febbraio 2026

## **Sommario**

Questo documento è la relazione del progetto del corso di "Programmazione a Oggetti" a.a 24/25. Realizzato da Elena Montalti, Luca Latini e Paolo De Mori. Il gruppo si è prodigato a sviluppare un clone del celebre gioco 2d a scorrimento orizzontale Geometry Dash, ma con una ispirazione al terminale bash Linux. La relazione esamina le varie fasi di sviluppo del progetto, la suddivisione del lavoro e le tecnologie utilizzate; mantenendo particolare attenzione all'utilizzo delle principali metodologie di sviluppo OO aderendo il più possibile alle buone tecniche di design e sviluppo analizzate nel corso.

# Indice

<b>1</b>	<b>Analisi</b>	<b>2</b>
1.1	Descrizione e requisiti . . . . .	2
1.2	Modello del Dominio . . . . .	3
<b>2</b>	<b>Design</b>	<b>6</b>
2.1	Architettura . . . . .	6
2.2	Design dettagliato . . . . .	8
<b>3</b>	<b>Sviluppo</b>	<b>36</b>
3.1	Testing automatizzato . . . . .	36
3.2	Note di sviluppo . . . . .	39
<b>4</b>	<b>Commenti finali</b>	<b>42</b>
4.1	Autovalutazione e lavori futuri . . . . .	42
<b>A</b>	<b>Guida utente</b>	<b>44</b>

# **Capitolo 1**

## **Analisi**

Il gruppo si pone l'obiettivo di realizzare un videogioco 2D a scorrimento orizzontale ispirato fortemente a "Geometry-Dash", che venne sviluppato nell'agosto del 2013 e da allora ha guadagnato una grande notorietà e fama tra gli amanti della categoria, sia per il suo gameplay scorrevole, sia per la sua natura frenetica e le meccaniche ritmiche. Il giocatore ha come obiettivo quello di completare interamente un livello, schivando gli ostacoli presenti e interagendo con oggetti presenti all'interno del percorso. Ogni contatto del personaggio con un ostacolo, provoca la terminazione della partita corrente e l'inizio di una nuova. Il gruppo vuole creare una variante ispirata al design dei terminali bash, fornendo le meccaniche di base del celebre gioco con una diversa veste grafica, ma senza perderne le ispirazioni principali.

### **1.1 Descrizione e requisiti**

#### **Requisiti funzionali**

- Sviluppo di un livello base, giocabile, in cui il personaggio principale, scorre da sinistra verso destra nello schermo.
- Personaggio principale reattivo all'input dell'utente.
- Interazione del personaggio con elementi di gioco: diverse tipologie di ostacoli e elementi interagibili o collezionabili dal personaggio.
- Possibilità di personalizzazione del personaggio, tramite apposito menù.
- Gestione di una valuta cumulativa, ottenibile durante il game-play.

## **Requisiti non funzionali**

- Possibilità di giocare a diversi livelli di risoluzione.
- Creazione di un gameplay fluido e godibile dall’utente.
- Possibilità di giocare un livello salvato su file.
- Realizzazione di una interfaccia di interazione che simula un terminale bash, con cui interagire tramite comandi fittizzi.
- Implementazione di un software usufruibile su Windows, MacOs e Linux.

## **1.2 Modello del Dominio**

### **Game Object**

Qualsiasi elemento presente nel mondo di gioco. Un’entità è in grado di fornire la propria posizione e le proprie informazioni geometriche. Rappresenta la base comune a tutti gli oggetti collocati nella mappa.

### **Personaggio Principale**

Il personaggio è l’unico elemento controllabile e dinamico del gioco, costituisce l’interfaccia diretta tra utente e ambiente. Possiede attributi relativi alla fisica e alla geometria che determinano la modalità di interazione con gli altri elementi. Reagisce agli input dell’utente gestendo la meccanica del salto, che viene eseguito quando richiesto e se le condizioni lo permettono. Durante il livello il personaggio può raccogliere monete, interagire con entità che ne alterano lo stato, urtare ostacoli oppure raggiungere il traguardo completando con successo il livello.

### **Ostacoli**

Un ostacolo è un oggetto statico e inamovibile. Gli ostacoli, insieme ai power up, costituiscono il mondo di gioco. Ogni ostacolo oltre ad esporre le informazioni fornite in quanto Game Object, quali la propria posizione e le informazioni geometriche, fornisce informazioni utili in caso di contatto con il giocatore. Il livello contiene ostacoli di varie tipologie, ciascuno dei quali può causare la morte del personaggio se urtato in un determinato modo.

## **Spike**

Ostacolo di forma triangolare che provoca la morte del protagonista in caso di contatto, indipendentemente dalla direzione dell'urto.

## **Block**

Ostacolo standard di forma quadrata. Il personaggio può salirvi sopra e camminarci. Un urto laterale causa il fallimento del livello.

## **PowerUp**

Un power up è un oggetto statico che, al contatto con il personaggio, può modificarne lo stato oppure alterare la logica di gioco o la sessione corrente. Fornisce informazioni su posizione, geometria e comportamento in caso di collisione con il giocatore.

## **Moneta**

Si configura come l'elemento collezionabile principale. La sua raccolta non altera la fisica del movimento, ma influenza direttamente le statistiche di completamento mostrate al termine del livello.

## **Speed Boost**

Questo potenziamento interviene sulla cinematica del player, determinando un incremento temporaneo della velocità. L'effetto richiede al giocatore una maggiore reattività, alterando il ritmo della sfida per superare segmenti specifici del percorso.

## **Shield**

Lo scudo fornisce una protezione strategica contro i pericoli ambientali. Una volta raccolto, garantisce l'immunità rispetto a una singola collisione mortale con le punte (*spike*); dopo aver assorbito l'impatto, il potenziamento si esaurisce, riportando il personaggio al suo stato di vulnerabilità standard.

## **Livello**

Il livello costituisce l'ambiente di gioco vero e proprio. Esso è costituito dall'insieme degli ostacoli, dei power up e del personaggio. Contiene le informazioni relative alla posizione e alla disposizione spaziale di tutte le entità nel mondo di gioco.

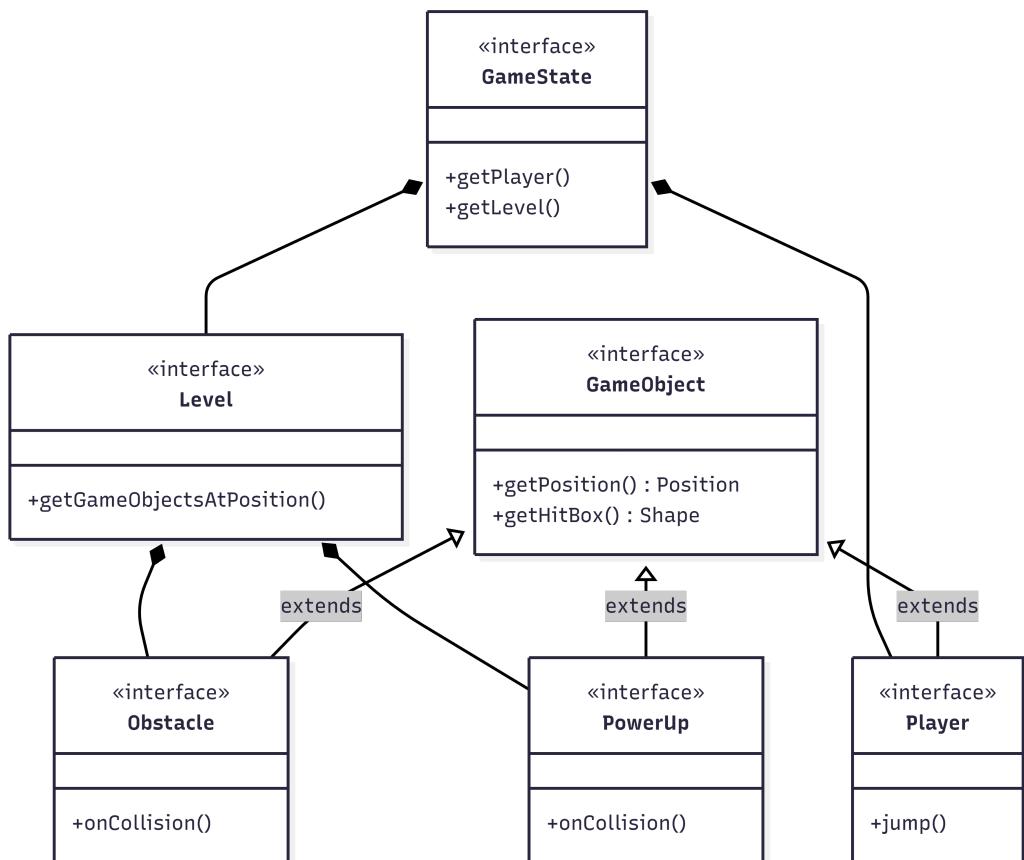


Figura 1.1: Schema UML dell’analisi del problema, con rappresentate le entità principali ed i rapporti fra loro

# Capitolo 2

## Design

### 2.1 Architettura

L’architettura dell’applicazione segue il pattern MVC, realizzato tramite il pattern Observer per gestire la comunicazione asincrona tra i componenti principali. Il Model, una volta inizializzato dal Controller, gestisce l’evoluzione del mondo di gioco tramite il metodo **update**. Inoltre, il Model, restituisce una rappresentazione del mondo di gioco sottoforma di una classe comune chiamata **updateInfoDTO**. La rappresentazione a schermo del mondo di gioco avviene tramite il metodo **update** della View. Il Controller funge da intermediario e orchestratore, gestendo il game loop che ad ogni sua iterazione si occuperà di:

- Chiamare l’update del Model
- Recuperare la rappresentazione aggiornata del Model tramite apposito metodo
- Aggiornare la View con la nuova rappresentazione

Questo approccio garantisce che la View possa accedere esclusivamente a una rappresentazione non modificabile dei dati, assicurando l’integrità dello stato interno. La View si occupa della presentazione grafica, integrando sia l’interfaccia terminale che il rendering della partita, e notifica il Controller sugli input utente tramite eventi tipizzati. Grazie a questa struttura, l’implementazione della View risulta facilmente intercambiabile, poiché la logica di dominio e di controllo non dipendono dalle tecnologie di visualizzazione adottate. Oltre alla View anche il Model utilizza il pattern Observer, insieme al Controller, per notificare eventi a cui il controller può immediatamente reagire.

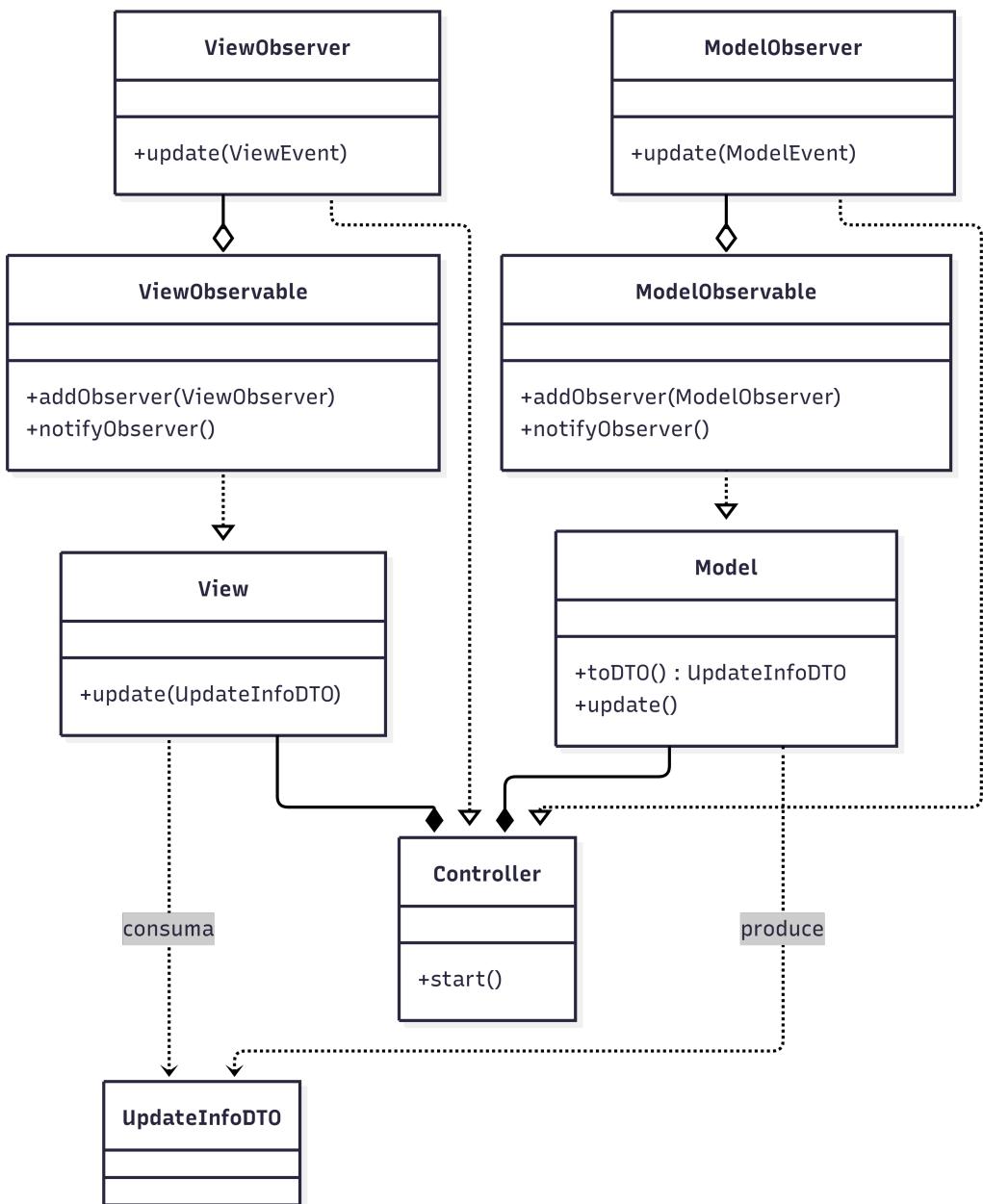


Figura 2.1: Architettura MVC

## 2.2 Design dettagliato

De Mori Paolo

Creazione infrastruttura della struttura e della logica di update del GameModel

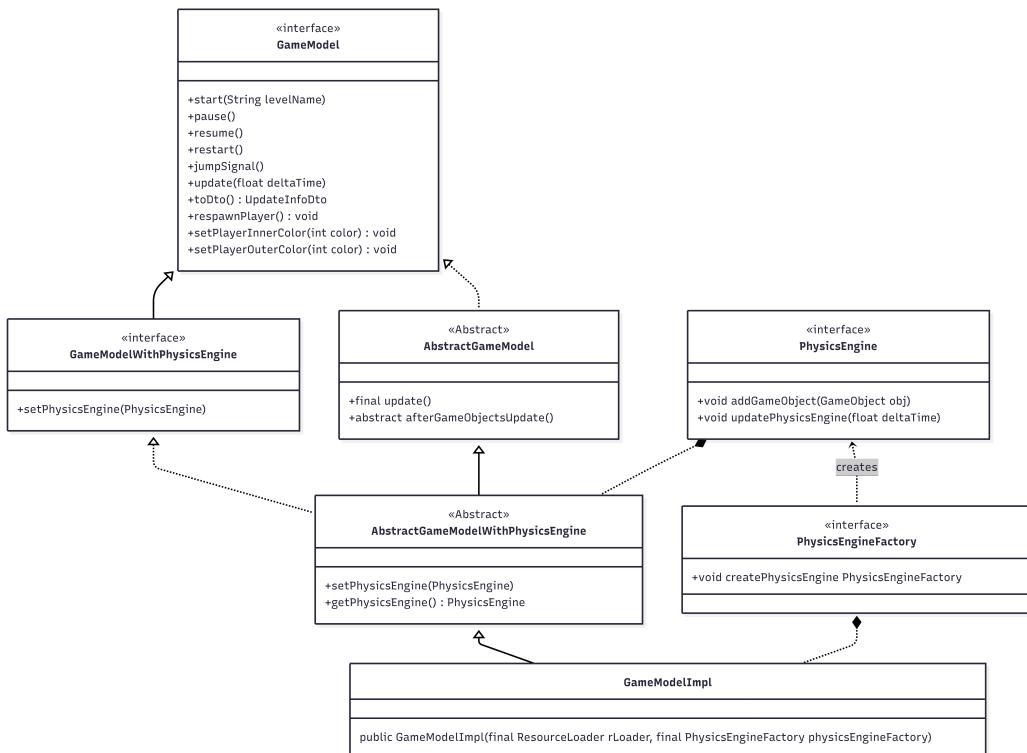


Figura 2.2: Struttura generale del GameModel.

### Problema

Realizzazione del game model, più specificatamente dell'orchestrazione delle entità e della loro evoluzione durante il gioco. Creazione di una struttura che permetta l'update del mondo di gioco tramite apposito metodo e allo stesso tempo gestisca le varie entità in maniera coerente allo stato del sistema in quel momento. Predisposizione del sistema all'utilizzo di un motore fisico, non vincolando però tutto il sistema a questa scelta implementativa, garantendo classi riusabili.

### Soluzione

Per realizzare il gamemodel, particolare attenzione è stata posta alla riusabi-

lità del codice, il codice è stato realizzato in maniera tale da permettere ad un eventuale cliente di scegliere il punto di ingresso nella logica implementata. Il cliente in questo modo può :

- Utilizzare solamente l’interfaccia GameModel, implementando i metodi forniti offrendo le funzionalità necessarie al funzionamento del programma.
- Estendere la classe AbstractGameModel per sfruttare lo scheletro dell’algoritmo fornito tramite Template pattern utilizzando la logica di update fornita tramite Strategy Pattern.
- Realizzare un GameModel che preveda l’utilizzo di un motore fisico tramite implementazione di AbstractGameModelWithPhysicsEngine.

La classe AbstractGameModel nasce con l’obiettivo di realizzare una logica di update del livello che possa essere utilizzata in un gioco che sfrutta un meccanismo di update costante e che possa essere riutilizzato in maniera flessibile.

Per garantire la riusabilità del codice la classe sfrutta il Template pattern fornendo due metodi

- il metodo final update
- il metodo astratto afterGameObjectsUpdate

Il metodo update fornisce in maniera non modificabile una logica in cui in primo luogo: La classe delega a una serie di oggetti, che implementano la functional interface updatable, l’evoluzione del mondo di gioco. Mentre il metodo afterGameObjectsUpdate, implementato da classi figlie, attua in un secondo momento tutte le operazioni generali di update non legate agli oggetti di tipo updatable.

La delegazione agli updatable avviene tramite strategy pattern, queste classi vengono notificate dell’avvenuto aggiornamento ricevuto e reagiscono chiamando la funzione update fornita dall’interfaccia.

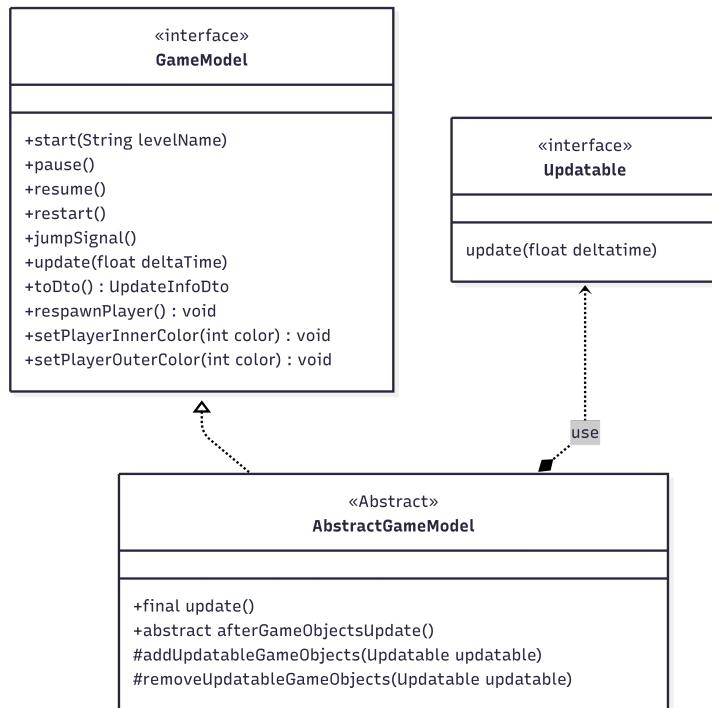


Figura 2.3: Utilizzo degli updatable con strategy pattern.

Nella nostra implementazione abbiamo utilizzato la libreria esterna jbox2d [Mac25], un porting in Java del noto motore fisico per c++ Box2D [Cat25]. Utilizzando questa infrastruttura, se correttamente implementato con le interfacce esistenti, è possibile cambiare motore fisico nel GameModelImpl semplicemente passandogli da costruttore una nuova implementazione di PhysicsEngineFactory, ossia una classe che sfrutta il Factory Method pattern per creare una istanza di una classe che implementa PhysicsEngine.

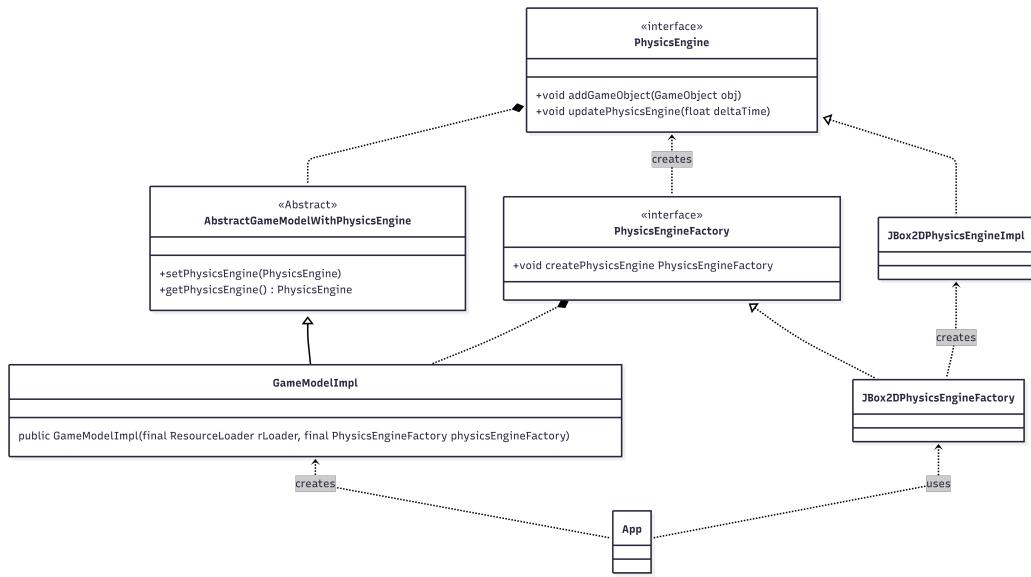


Figura 2.4: Utilizzo di jbox2d nel gamemodel.

## Creazione di un GameLoop

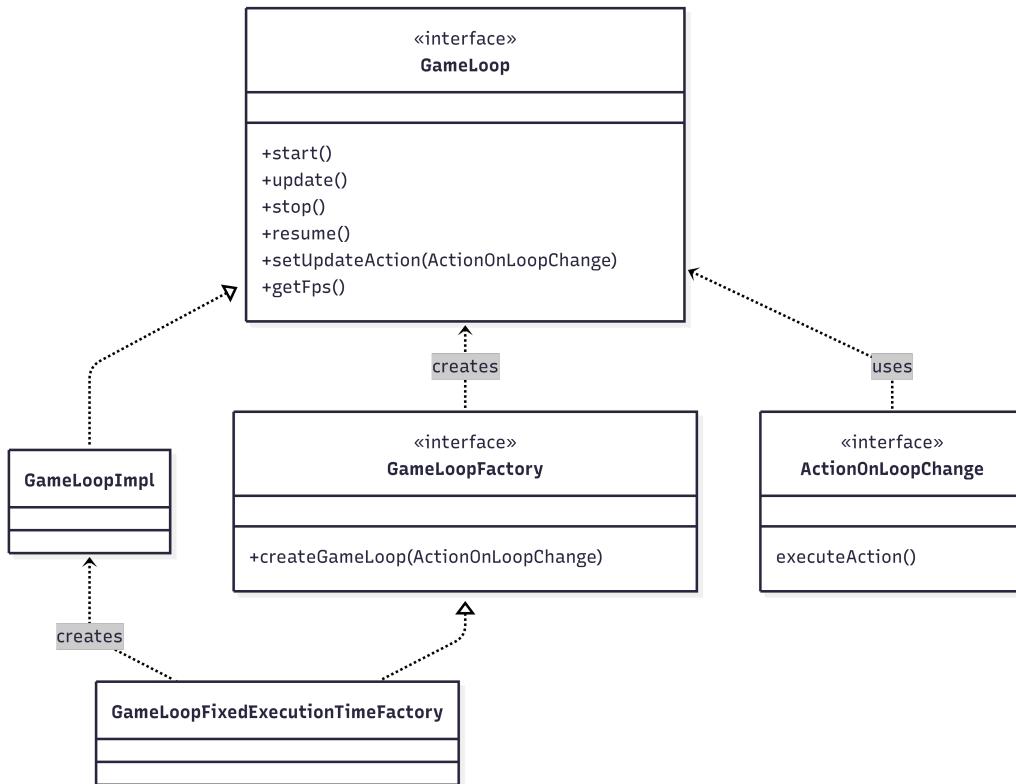


Figura 2.5: Struttura generale del GameModel.

### Problema

Realizzazione del gameloop, creando interfacce riusabili e una implementazione che esegua una azione predefinita circa sessanta volta al secondo.

### Soluzione

Per realizzare il gameloop è stata data particolare cura nella creazione di interfacce che possano permettere un possibile riuso di gran parte dell'infrastruttura. Per fare ciò sono state realizzate le interfacce `GameLoop` e `GameLoopFactory` che rappresentano rispettivamente:

- Un thread che esegue una azione ripetutamente, offrendo la possibilità di: mettersi in pausa, ricominciare ad eseguire l'azione, interrompersi definitivamente, recuperare il numero di iterazioni eseguite nell'ultimo secondo.

- Una interfaccia che offre un metodo per la creazione di una istanza di una classe che implementa GameLoop.

Per realizzare il gameloop che esegue una determinata azione circa sessanta volte al secondo, la classe GameLoopImpl implementa uno strategy pattern utilizzando l' interfaccia funzionale ActionOnLoopChange per settare le operazioni da eseguire ad ogni ciclo. L'interfaccia GameLoopFixedExecutionTimeFactory utilizza il Factory Method pattern per instanziare una istanza di GameLoopImpl.

### Creazione infrastruttura Observer Pattern

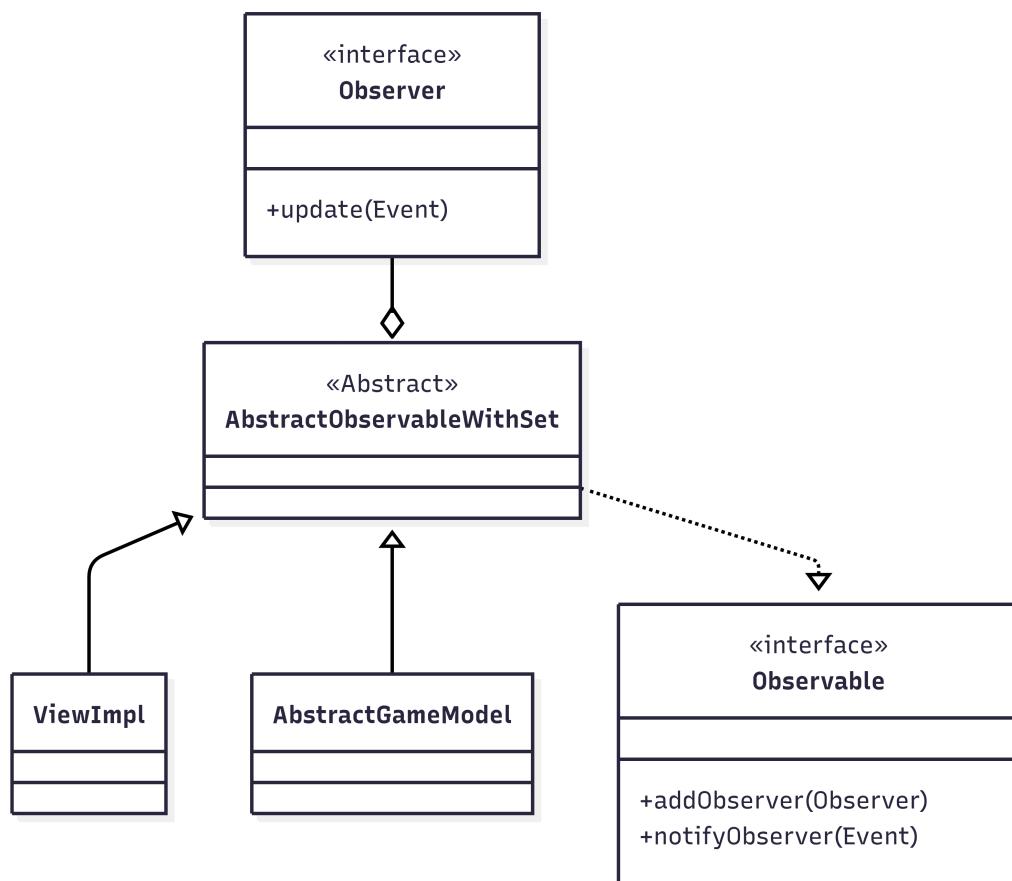


Figura 2.6: Struttura generale Observer Pattern.

### Problema

Realizzazione delle classi necessarie all'utilizzo dell'observer pattern nel progetto garantendo il riuso.

## Soluzione

Per realizzare le classi necessarie all'observer pattern, per prima cosa sono state realizzate due interfacce, Observable e Observer. Rispettivamente rappresentanti una classe che notifica un determinato evento e una classe che permette di essere notificata ricevendo una classe rappresentante l'evento. Successivamente è stata realizzata una classe astratta AbstractObservableWithSet che implementa i meccanismi tipici delle classi che permettono di ricevere una notifica, così al bisogno non è necessario ricostruire la struttura ma basterà che la classe che ne ha bisogno estenda AbstractObservableWithSet.

Queste interfacce di base sono state specializzate a loro volta per permettere lo scambio di messaggi tra view e controller e tra model e controller, garantendo però una suddivisione totale.

Per il Model sono state realizzate le classi ModelObserver e ModelObservable che si scambiano messaggi di tipo ModelEvent. La classe model event contiene al suo interno l'enumerativo ModelType che permette al model di comunicare alle classi che lo osservano, nel nostro caso il controller, quando avvengono la morte del giocatore o la vittoria della partita.

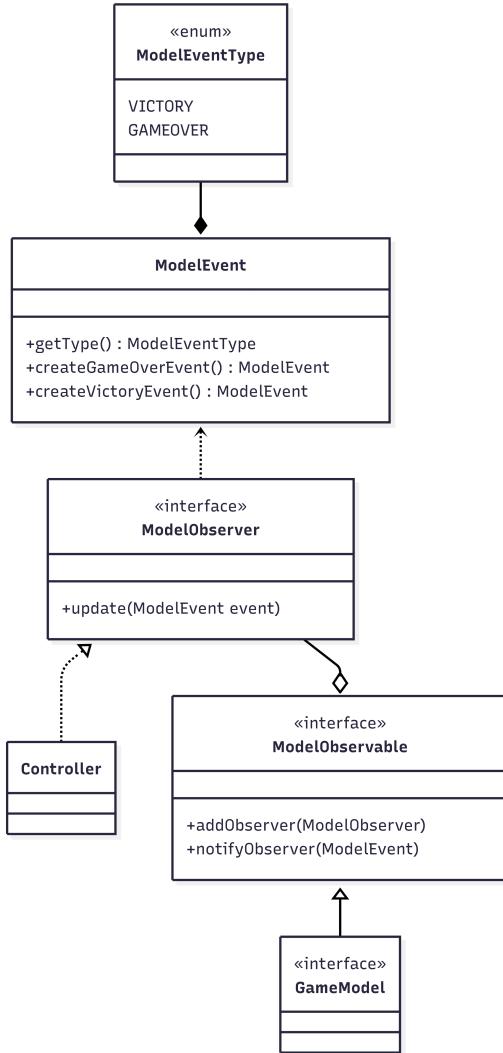


Figura 2.7: Struttura Observer Pattern del Model.

Per quanto riguarda la view sono state realizzate le classi ViewObserver e ViewObservable che si scambiano messaggi di tipo ViewEvent. ViewEvent tramite la classe ViewEventType, comunica alle classi che ricevono la notifica, nel nostro caso il controller, quando avvengono da view eventi da segnalare.

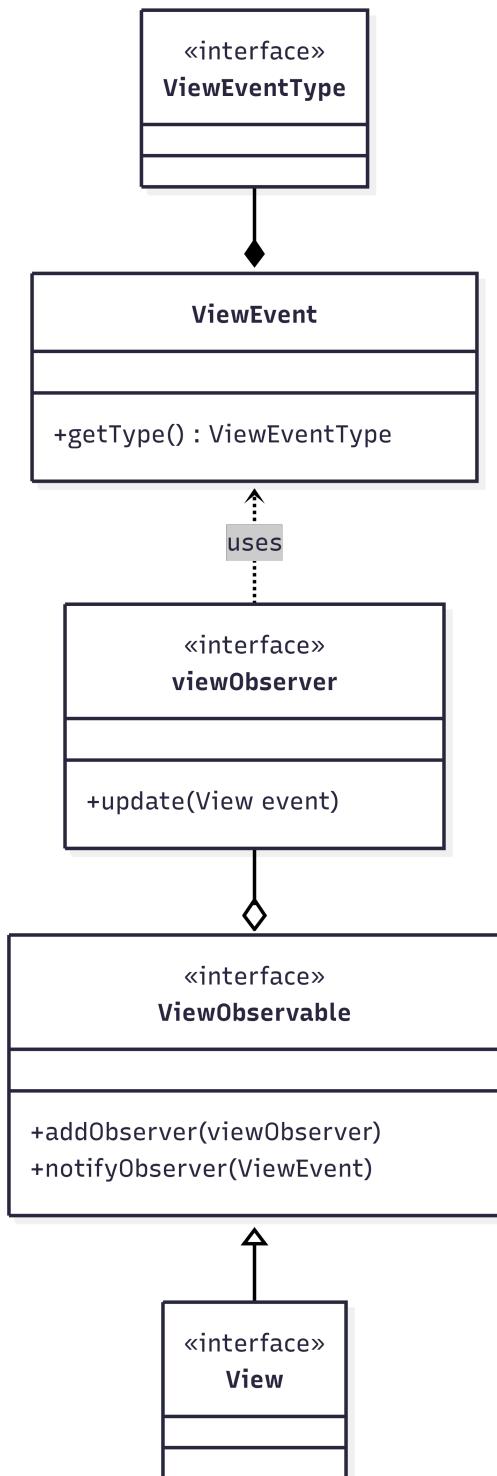


Figura 2.8: Struttura Observer Pattern della View.

## Latini Luca

### Gestione degli eventi di input

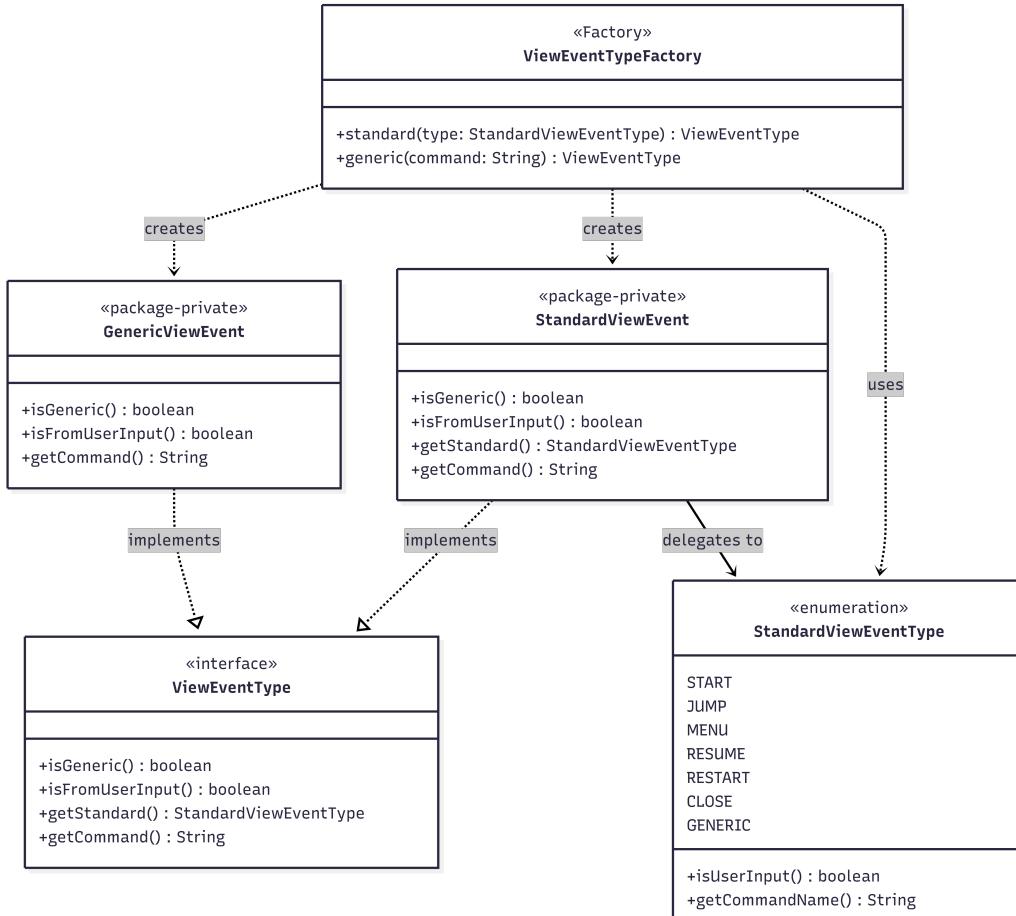


Figura 2.9: UML gestione eventi di input.

### Problema

Una sfida nella progettazione ha riguardato la creazione di un sistema di gestione eventi capace di distinguere tra input di gioco standard, come JUMP o START, e comandi testuali complessi provenienti dal terminale. La necessità era fornire sia un insieme di comandi necessari alla funzione del gioco eliminando l'uso di "stringhe magiche" per l'identificazione degli eventi, che andrebbero a comprometterebbe la manutenibilità del sistema. Sia la possibilità di fornire comandi personalizzati.

### Soluzione

La risposta al problema precedente è stata individuata nell'adozione del pattern *Factory Method*, che ha come obiettivo quello di uniformare la gestione dei diversi tipi di input. L'architettura si articola attorno all'interfaccia `ViewEventType`, che definisce il contratto astratto per ogni evento, e alla classe `ViewEventTypeFactory`, incaricata di centralizzare la logica di creazione.

Le implementazioni concrete, `StandardViewEvent` e `GenericViewEvent`, sono dichiarate *package-private*, rendendo la factory l'unico punto di accesso autorizzato. In questo modo si garantisce che ogni evento sia validato correttamente prima dell'istanziazione. A supporto della *type safety*, l'enum `StandardViewEventType` definisce le azioni di sistema, includendo la logica necessaria a distinguere tra interazioni di gioco e comandi di interfaccia.

La struttura favorisce l'estensibilità del sistema, infatti per aggiungere nuovi comandi è sufficiente aggiornare l'enum, mantenendo del tutto invariata la logica di gestione e comunicazione esistente.

## Gestione dei diversi tipi di input tramite controller

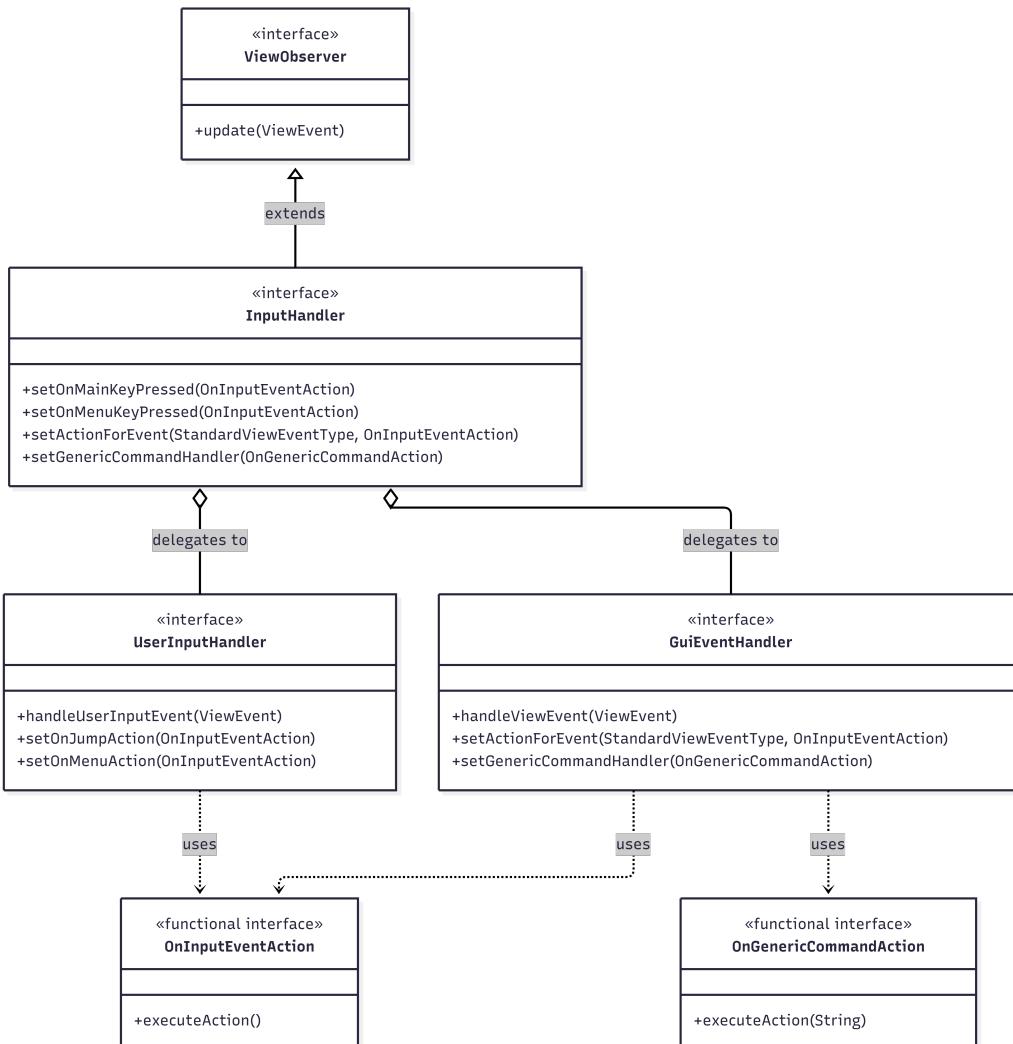


Figura 2.10: Architettura del composite input handler. Vengono mostrate le interfacce e le loro relazioni per una pulizia maggiore nel UML.

### Problema

Il Controller deve gestire due tipi di input molto diversi come comandi testuali dal menu (come "start", "setcolor -inner red") e azioni di gioco immediate (JUMP da barra spaziatrice, MENU da tasto ESC). I comandi testuali richiedono parsing, mentre le azioni di gioco devono essere eseguite immediatamente. Serviva un sistema che permettesse di configurare dinamicamente le

azioni da eseguire per ogni evento, mantenendo separate le logiche di gestione dei due flussi di input.

### Soluzione

Il sistema di gestione dell'input è stato progettato integrando diversi pattern. L'ingresso dei dati nel sistema avviene tramite il pattern *Observer*, dove il CompositeInputHandler ricopre il ruolo di *Observer* della View. Al verificarsi di un evento, la View notifica il sistema tramite il metodo update, garantendo che il Controller non abbia informazioni riguardo alla periferica di input utilizzata.

Per gestire in modo ottimale la doppia natura degli input è stato implementato un meccanismo di delegazione. La classe CompositeInputHandler coordina due handler specializzati: UserInputHandlerImpl per gli eventi di gameplay (salto e pausa) e GuiEventHandlerImpl per i comandi da interfaccia (start, close, comandi custom). Il metodo update interroga la natura dell'evento tramite isFromUserInput() e delega automaticamente all'handler appropriato.

Per incapsulamento, la creazione del CompositeInputHandler avviene tramite una factory che permette al Controller di ottenere istanze indipendenti senza esporre dipendenze, dunque di inserire oggetti mutabili nel suo costruttore.

Infine, l'esecuzione effettiva delle azioni è affidata al pattern *Strategy*, implementato attraverso interfacce funzionali quali: OnInputEventAction e OnGenericCommandAction. Queste interfacce rappresentano le *Strategie* intercambiabili che il Controller può iniettare negli handler. Grazie a questo approccio, il Controller può definire il comportamento del sistema tramite Lambda expressions (o method references). La soluzione apportata permette di modificare la logica di un comando senza alterare le classi che gestiscono materialmente l'input, rispettando pienamente i requisiti di manutenibilità e garantendo una buona flessibilità.

## Gestione degli ostacoli e PowerUp

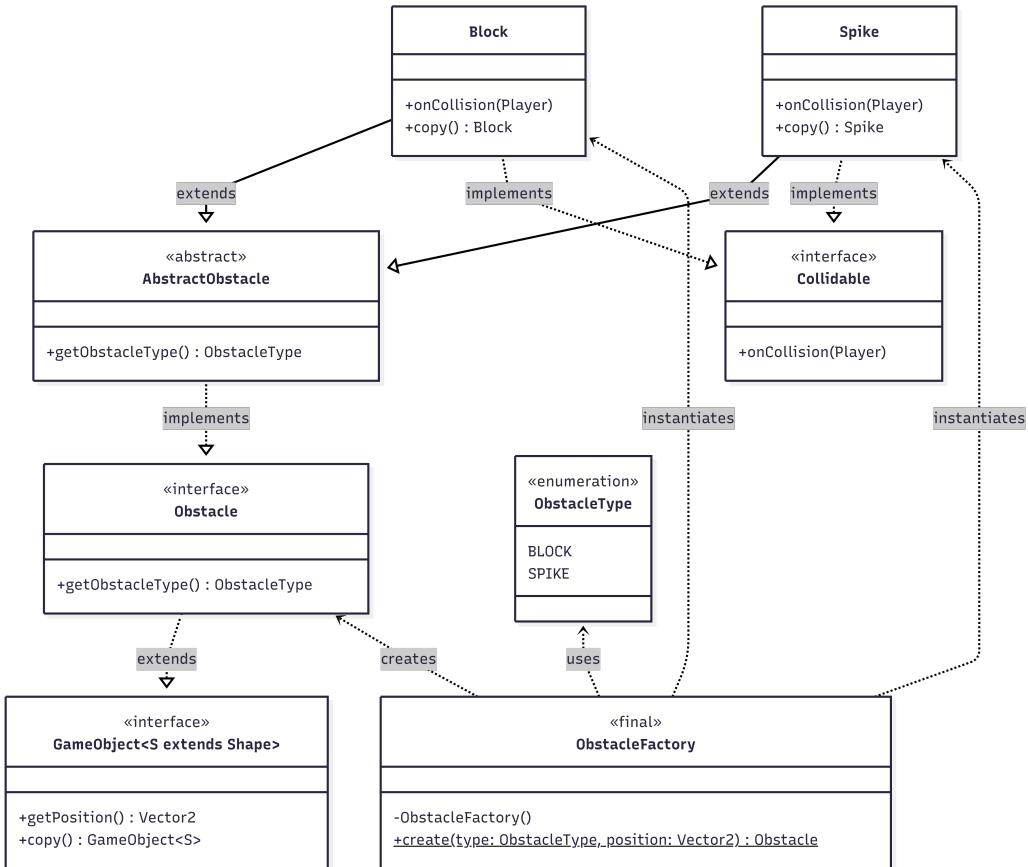


Figura 2.11: Architettura degli ostacoli.

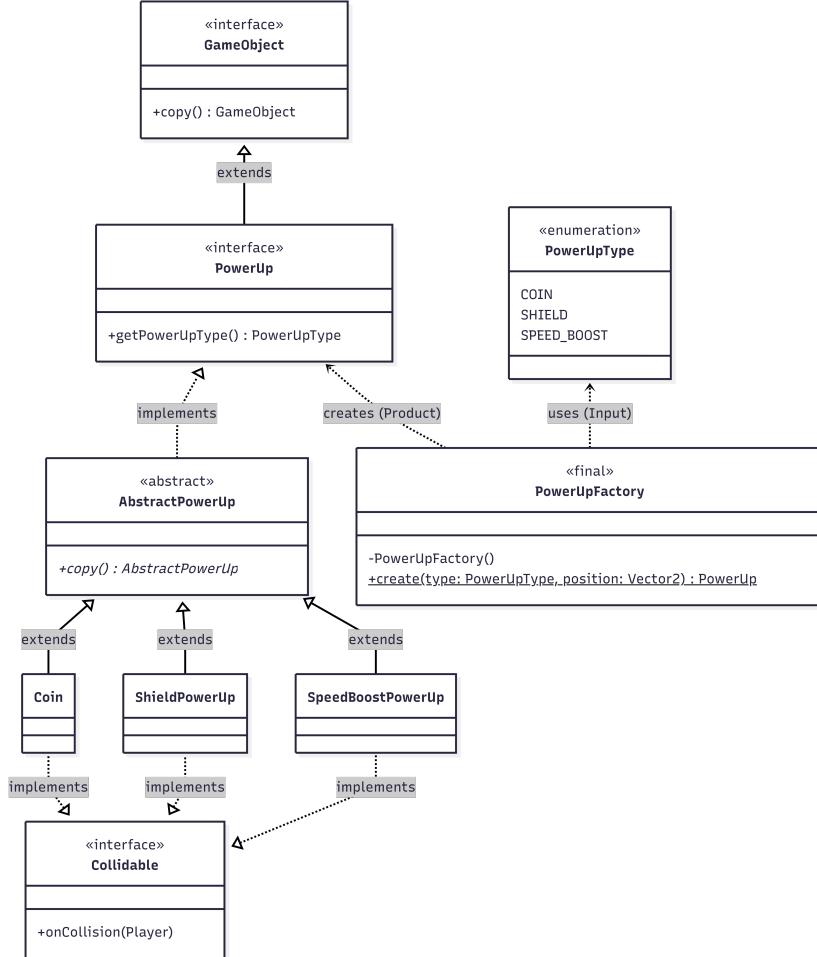


Figura 2.12: Architettura dei PowerUp.

### Problema

Il gioco presenta altre entità di gioco oltre al player come ostacoli e PowerUp. Nella progettazione ho dovuto gestirle in modo uniforme cercando di estrarre le proprietà comuni, come la posizione, e proprietà geometriche. L'obiettivo era quindi definire un'architettura estensibile che evitasse la dispersione della logica, garantendo prestazioni e facilità di manutenzione.

### Soluzione

La soluzione adottata prevede l'impiego di gerarchie basate su classi astratte, che permettono di racchiudere il codice comune e delegare alle sottoclassi esclusivamente le specificità.

Le risposte alle collisioni sono modellate diversamente per ogni ostacolo e PowerUp. Ognuno di essi incapsula al suo interno la propria logica all'interno di una classe specifica.

no dell’interfaccia Collidable. Il blocco, ad esempio, distingue tra atterraggio letale o sicuro, mentre l’ostacolo spike delega la strategia al player. Questo approccio garantisce che l’aggiunta di nuovi comportamenti non richieda modifiche al loop principale, rispettando il principio *Open/Closed*.

La gestione dei potenziamenti temporanei è affidata al PowerUpManager. Questa scelta riflette rigorosamente il principio di *separazione delle responsabilità*. Le classi dei potenziamenti fungono da semplici attivatori, mentre la logica di gestione dei timer e il ripristino degli stati sono centralizzati nel manager.

La creazione dei due tipi di entità viene in entrambi i casi gestita tramite il pattern *Factory*. Le classi ObstacleFactory e PowerUpFactory centralizzano la logica di istanziazione, permettendo di creare ostacoli e power-up a partire dal tipo enumerativo (ObstacleType, PowerUpType) e dalla posizione. Questo approccio facilita l’estensione del sistema con nuovi tipi senza modificare il codice che utilizza le factory.

## Organizzazione spaziale e gestione dei livelli

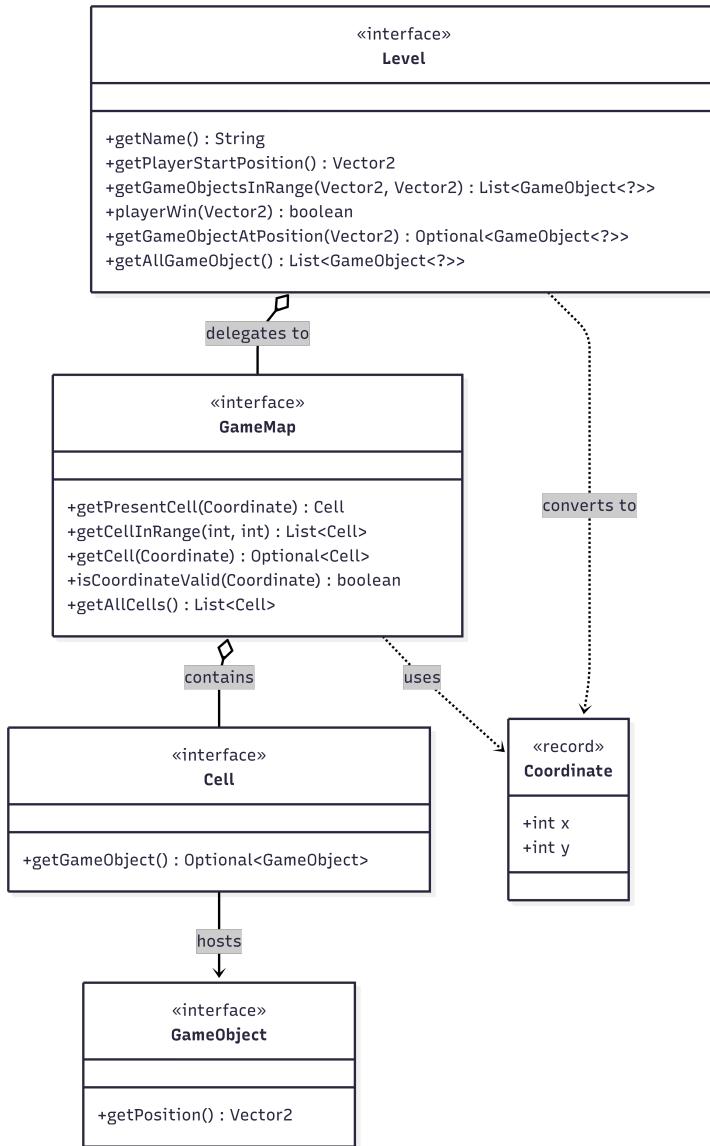


Figura 2.13: Gestione dei livelli e l'organizzazione spaziale.

### Problema

In un platform, un livello può contenere centinaia di oggetti, ma il giocatore ne vede solo una piccola parte alla volta. Elaborare tutti gli oggetti contemporaneamente, anche quelli fuori dallo schermo, causerebbe rallentamenti inutili e spreco di risorse. Esistono due tipi di coordinate: le coordinate

fisiche (numeri decimali) per il movimento fluido del player e le coordinate della griglia (numeri interi) per posizionare gli oggetti nella mappa. Usare lo stesso tipo di dato per entrambi aumenta il rischio di errori logici e rende il codice difficile da leggere. Senza una separazione chiara, il Controller sarebbe costretto ad eseguire abbondanti calcoli matematici diventando difficile da mantenere.

### Soluzione

Per semplificare l’interazione con il mondo di gioco, l’architettura introduce l’interfaccia Level come unico punto di accesso per le operazioni di alto livello. Questa interfaccia agisce nascondendo la complessità dei calcoli spaziali e fornendo al Controller metodi diretti per verificare la vittoria o ottenere gli oggetti necessari al rendering.

La mappa è stata strutturata come una griglia gestita dall’interfaccia GameMap. Per risolvere il problema dei diversi sistemi di coordinate, è stato introdotto un tipo di dato dedicato, Coordinate, che incapsula una posizione intera sulla griglia. Questa scelta fa sì che il compilatore impedisca di passare una coordinata fisica laddove il sistema si aspetta una cella della griglia, rendendo il codice più robusto e chiaro.

Il view culling viene gestito tramite il metodo `getGameObjectsInRange`, che traduce l’intervallo visibile in coordinate della griglia e recupera solo le celle interessate. Per garantire che questa operazione avvenga in tempo costante, l’implementazione della mappa organizza le celle in una Nested Map basata sulle loro posizioni  $X$  e  $Y$ . Questo design permette un recupero immediato delle entità, eliminando la necessità di iterare sull’intero set di oggetti del livello e garantendo prestazioni ottimali anche in livelli con molte entità di gioco.

La struttura della soluzione proposta per questo problema determina una netta separazione delle responsabilità e della logica. GameMap si occupa esclusivamente dell’organizzazione spaziale, l’interfaccia Level centralizza le regole del dominio, come il calcolo del completamento o il superamento dei traguardi.

trasformazione dell'input fisico in comandi di gioco.

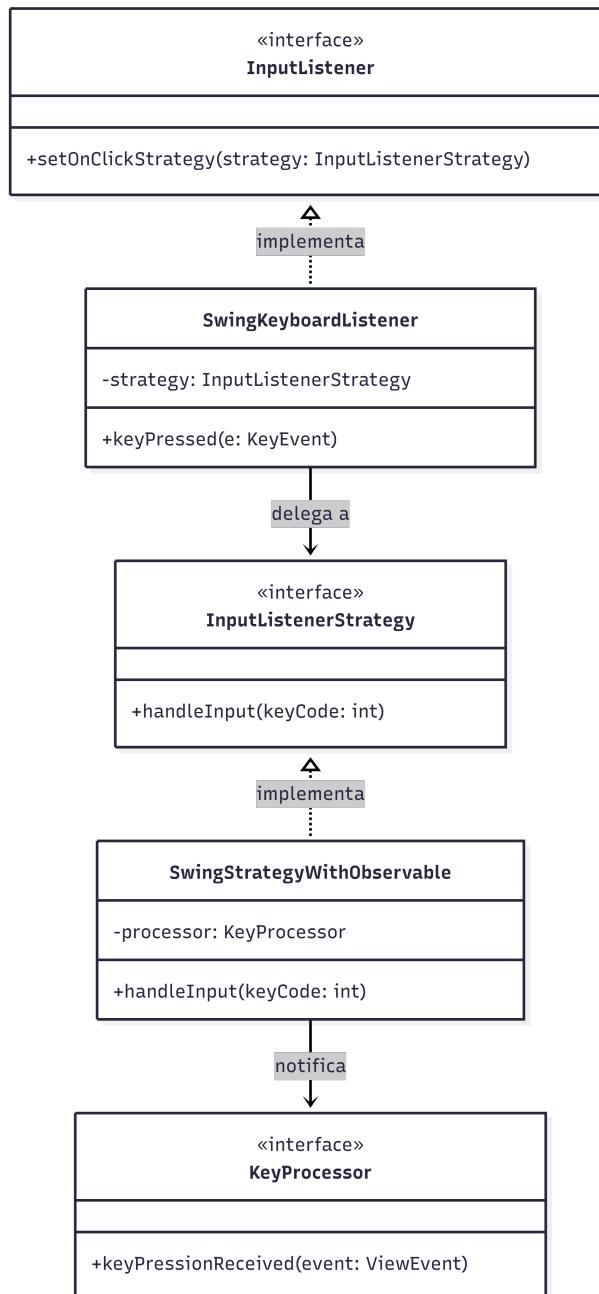


Figura 2.14: trasformazione dell'input fisico in comandi di gioco.

## Problema

Utilizzare i comandi di Swing, come i codici della tastiera, crea un legame forte tra la logica e la grafica, rendendo il codice molto rigido non permettendo, qualora lo si volesse, cambiare libreria grafica o sistema di input come un controller. Una buona progettazione prevede la separazione dell'azione fisica dal suo significato in gioco.

## Soluzione

La soluzione adotta il pattern *Strategy* per creare uno strato di traduzione tra l'input fisico e la logica di gioco suddividendolo in più componenti. L'interfaccia InputListenerStrategy ha come compito quello di non far gestire l'evento direttamente al componente grafico, delegando il compito ad una strategia intercambiabile. La classe SwingKeyboardListener funge da ponte intercettando il tasto premuto e passandone il suo codice numerico alla strategia associata, senza interpretarlo. SwingStrategyWithObservable è il fulcro della logica di input. Riceve il codice numerico e lo "traduce" in un evento di gioco, ViewEvent (es. trasforma la barra spaziatrice nell'azione JUMP) permettendo di separare i tasti fisici dai comandi logici. L'evento così creato viene inviato al sistema di notifica dell'applicazione (Pattern Observer), permettendo a qualsiasi componente interessato di reagire senza mai sapere che l'input è stato generato da una tastiera. La soluzione adottata permette, in caso di aggiunta di un nuovo sistema di input come il mouse, di creare solamente un nuova strategia garantendo la sostituibilità. Viene inoltre anche rispettato il principio di *separazione delle responsabilità* definendo il compito di ciascuna classe.(cattura dell'input e la traduzione semantica.)

## Elena Montalti

### Gestione della fisica del Player tramite Adapter

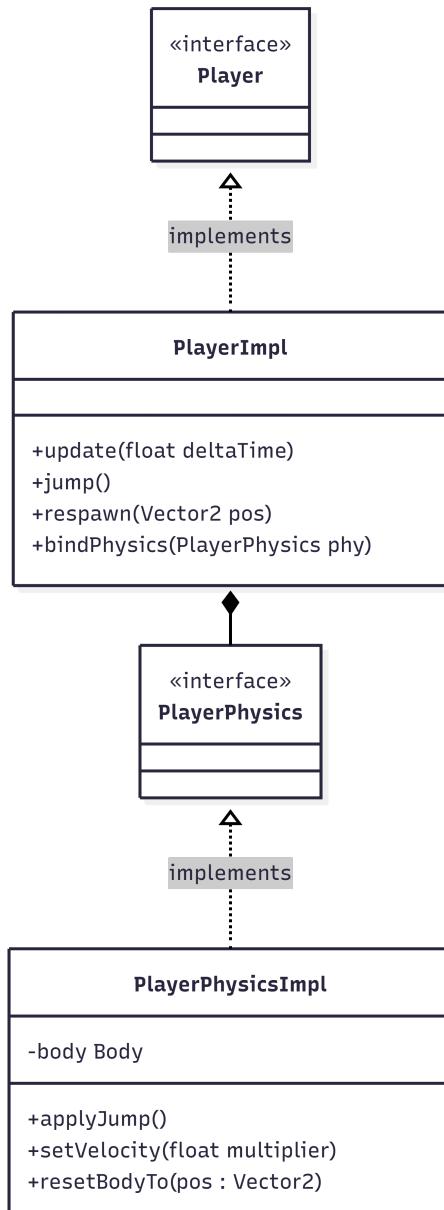


Figura 2.15: Diagramma UML del Player e della sua gestione fisica.

### Problema

Dopo aver definito il modello di dominio dell'applicazione, abbiamo valutato

l'integrazione di un motore fisico per simulare correttamente movimento, accelerazione e interazioni dinamiche. Per questo scopo è stata scelta la libreria JBox2D, che fornisce un insieme di primitive e API per la gestione semplificata di corpi rigidi, forze, impulsi.

All'interno del gioco, il Player rappresenta l'unico oggetto la cui fisica viene modificata continuamente durante la partita. Il problema principale consisteva quindi nel collegare la logica di dominio del Player con l'implementazione fisica reale fornita dalla libreria.

### Soluzione

Per risolvere il problema è stato introdotto un livello di astrazione basato sull'Adapter Pattern, separando la logica di gioco del player dalla gestione della fisica. La classe `PlayerImpl` rappresenta il player nel dominio e delega le operazioni fisiche ad un componente `PlayerPhysics`, definito tramite un'interfaccia indipendente dal motore utilizzato. L'implementazione concreta `PlayerPhysicsImpl` agisce da adapter: implementa `PlayerPhysics` e wrappa l'oggetto `Body` di JBox2D, traducendo tramite delegazione le richieste provenienti dal dominio nelle corrispondenti chiamate al motore fisico. In questo modo il Player non ha alcuna necessità di conoscere la struttura interna del motore fisico, ma interagisce tramite un'interfaccia progettata appositamente per il dominio. Questo rende l'applicazione estendibile all'utilizzo di un motore fisico differente rispetto a quello attuale (JBox2D). L'integrazione di un nuovo motore fisico richiederebbe unicamente una nuova implementazione di `PlayerPhysics`, senza modificare `PlayerImpl`.

## Gestione degli oggetti di gioco

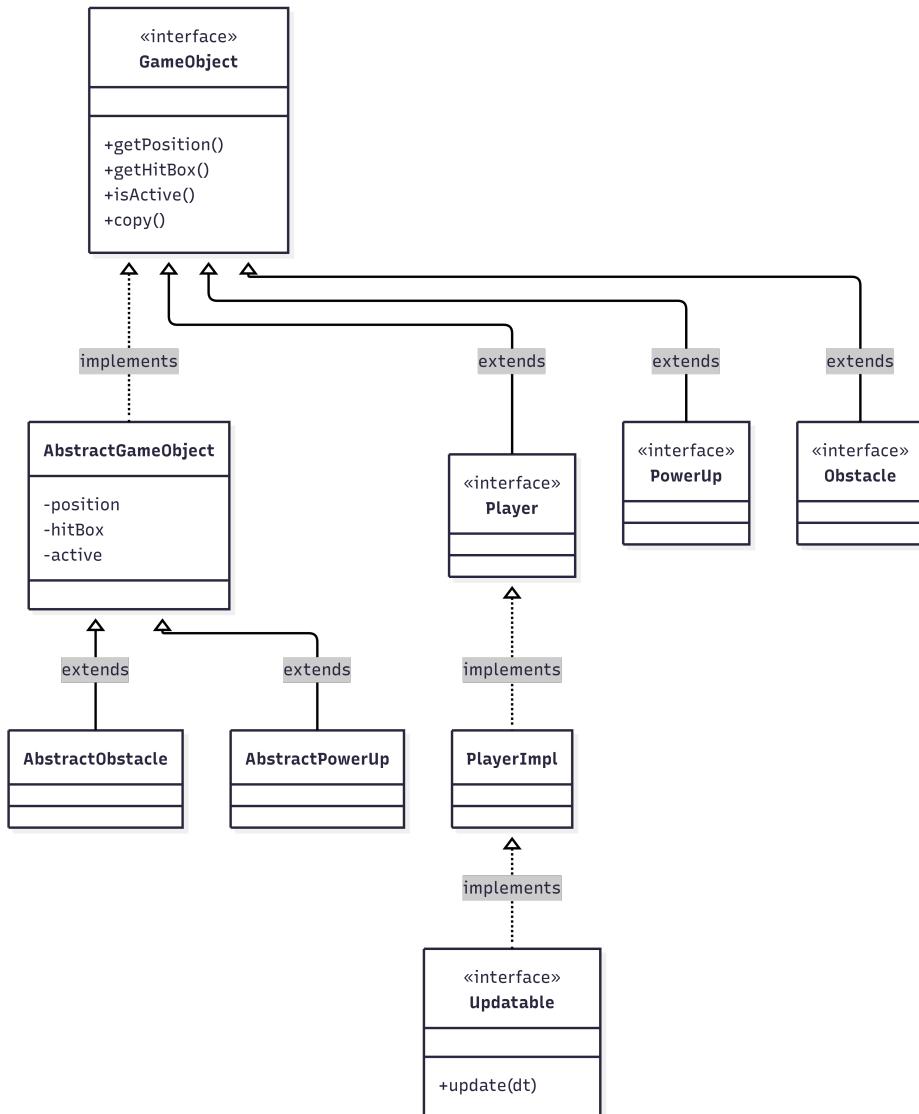


Figura 2.16: Diagramma UML della gerarchia degli oggetti di gioco.

### Problema

Nell'applicazione sono diverse tipologie di oggetti di gioco tra loro eterogenee per comportamento e ruolo, ma accomunate dal fatto di essere tutte parte dello stesso mondo di gioco. I vari oggetti condividono alcune proprietà fondamentali: una posizione nel mondo, una rappresentazione geometrica utile per consentire la gestione delle collisioni, e uno stato che influenza la loro par-

tecipazione al gameplay. Il problema principale consiste quindi nel progettare una struttura che permetta di gestire tali oggetti in modo uniforme, evitando duplicazione di codice e garantendo un modello facilmente estendibile.

### Soluzione

La soluzione adottata si basa sulla definizione di un contratto comune tramite l’interfaccia `GameObject`, che rappresenta l’astrazione condivisa tra tutte le entità del gioco. Le funzionalità e gli attributi realmente comuni sono centralizzati nella classe astratta `AbstractGameObject`, che implementa l’interfaccia e fornisce una base riusabile per tutte le classi derivate, riducendo la duplicazione e migliorando coerenza e manutenibilità del codice. Inoltre, poiché non tutti gli oggetti presenti nel mondo di gioco necessitano di una logica temporale, è stata introdotta una separazione tra oggetti statici e oggetti dinamici. Tale distinzione è modellata tramite la functional interface `Updatable`, che rappresenta la capacità di essere aggiornati durante il game loop.

## Gestione delle collisioni

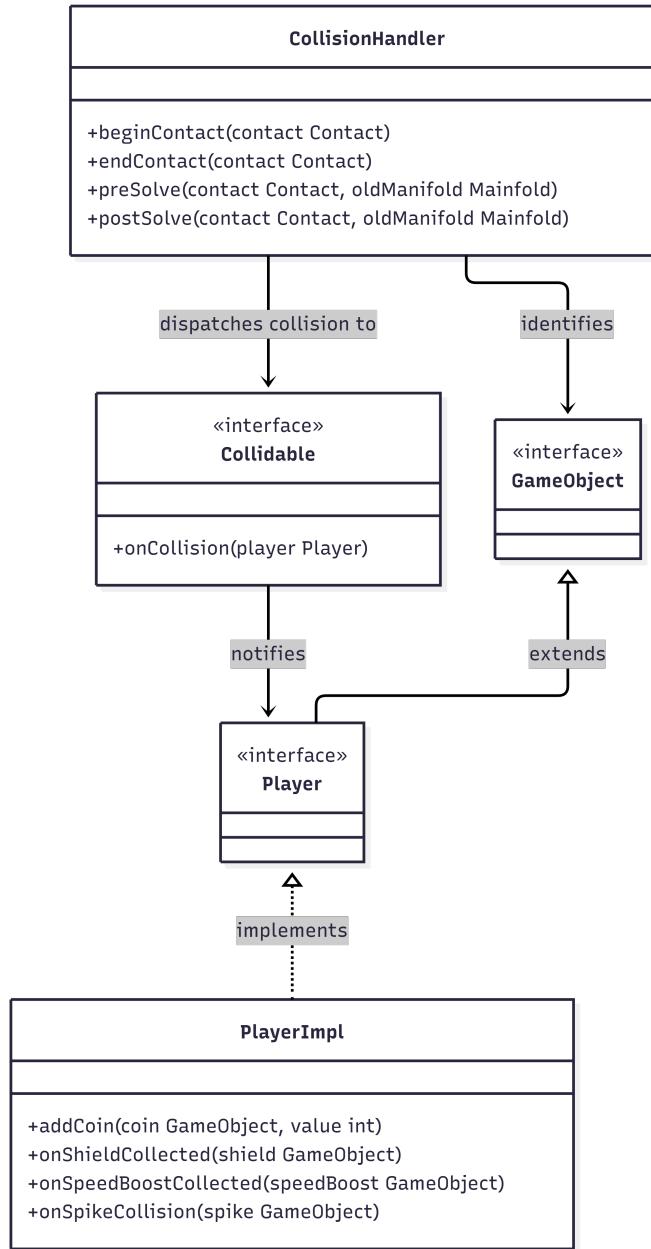


Figura 2.17: Diagramma UML delle componenti del sistema di collisione.

### Problema

- Collegare il sistema di collision detection della libreria con il modello di gioco senza introdurre un accoppiamento diretto tra la logica applicativa e la libreria.

va e i dettagli dell'engine fisico. La libreria JBox2D fornisce un sistema efficiente per la rilevazione delle collisioni, tuttavia si vuole mantenere l'indipendenza del dominio e della logica di gioco da librerie esterne.

- Progettare un sistema che consenta di gestire collisioni tra oggetti eterogenei garantendo una chiara separazione delle responsabilità, mantenendo la logica di gameplay nel dominio applicativo e minimizzando la dipendenza tra componenti.

## Soluzione

- Per risolvere questo problema è stato introdotto un componente dedicato alla gestione delle collisioni **CollisionHandler**, modellato come un adattatore tra la libreria fisica e il dominio dell'applicazione. La classe implementa l'interfaccia richiesta da JBox2D per ricevere le notifiche di contatto e si occupa di tradurre questi eventi in interazioni significative tra oggetti di gioco. Questo garantisce che la logica di gioco non dipenda direttamente da strutture proprie del motore fisico.
- Il **CollisionHandler** ha il solo compito di rilevare la collisione e identificare le entità coinvolte. Esso non contiene logica specifica di gameplay e non necessita di conoscere le diverse tipologie concrete di oggetti presenti nel gioco. Una volta determinati gli oggetti coinvolti, il gestore di collisioni delega la gestione dell'evento all'oggetto che ha colliso, invocando il metodo comune agli oggetti che implementano **Collidable**. Tale oggetto, a sua volta, non modifica direttamente lo stato del Player, ma si limita a comunicare a quest'ultimo che la collisione è avvenuta e qual è la natura dell'interazione. Il Player è quindi l'unico responsabile della gestione del proprio stato.

## Riattivazione degli elementi di gioco tramite Strategy

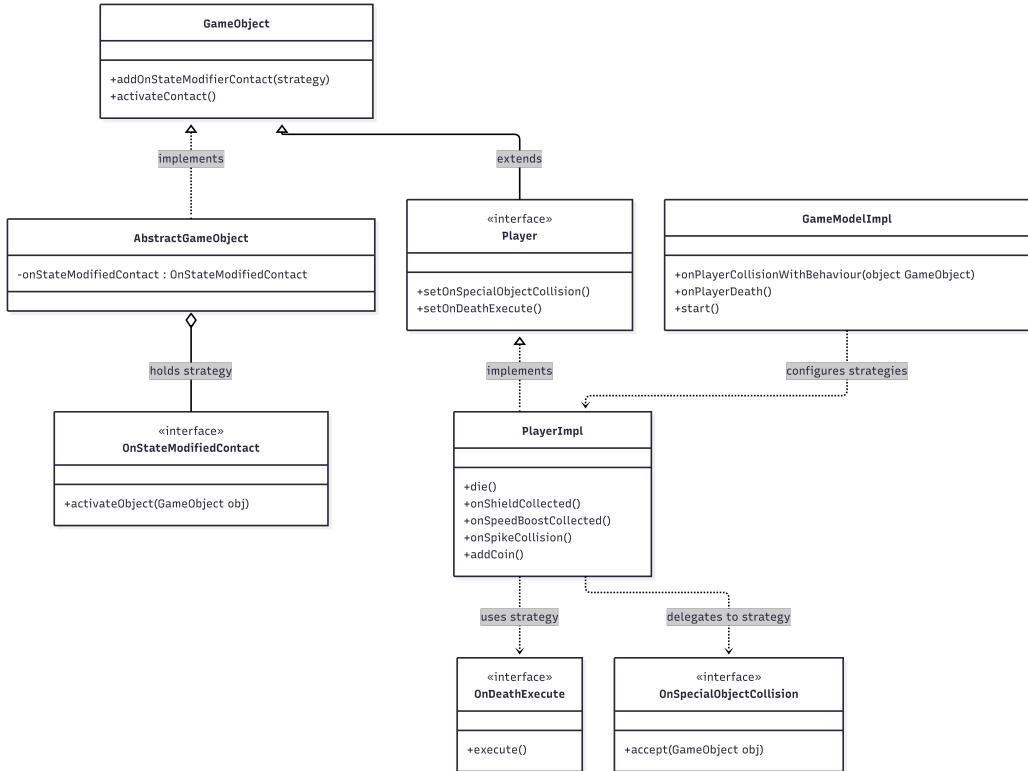


Figura 2.18: Diagramma UML della gestione dello stato del livello tra run successive.

### Problema

Il gioco adotta una dinamica di restart immediato: alla morte del player la run ricomincia.

Per motivi di efficienza il livello viene caricato una sola volta e mantenuto in memoria, evitando la ricostruzione completa della scena ad ogni restart. Durante una run, tuttavia, alcuni oggetti vengono "consumati" dall'interazione con il player e il loro stato viene modificato in modo persistente. Di conseguenza, al restart della partita, questi elementi rimangono nello stato alterato generando un livello non coerente con l'idea di ripartenza da zero.

### Soluzione

Per gestire il ripristino dello stato senza ricaricare l'intero livello, è stata adottata un'architettura basata sullo Strategy Pattern, implementato tramite interfacce funzionali. Il player non gestisce direttamente il reset della

scena. Quando una collisione comporta la disattivazione o modifica persistente di un oggetto del livello, il player invoca una strategia configurabile `onSpecialObjectCollision` che notifica l'evento al Model. Il Model mantiene quindi traccia degli oggetti modificati durante la run. Alla morte del player viene eseguita una seconda strategia `OnDeathExecute`, incaricata di ripristinare lo stato degli oggetti registrati, garantendo un restart coerente e mantenendo separata la logica di gameplay dalla gestione del livello tra una run e la successiva. Questo riduce l'accoppiamento, evita che Player dipenda dal Model o da dettagli del livello, e rende il reset estendibile.

*Nota:* attualmente i metodi legati alla gestione dei contatti sono stati mantenuti all'interno della gerarchia comune dei `GameObject`. Formalmente, tali responsabilità andrebbero gestite tramite interfaccia dedicata e applicate solo agli oggetti effettivamente collidabili. Nel dominio attuale, tuttavia, questa scelta rimane giustificata perché tutti gli oggetti di gioco presenti partecipano alle collisioni.

# Capitolo 3

## Sviluppo

### 3.1 Testing automatizzato

Per la verifica del corretto funzionamento del software sono stati creati test automatizzati usando JUnit. Essi vogliono garantire il corretto funzionamento del Model e delle classi di utility. Inoltre abbiamo utilizzato tutti i plugin presenti all'interno del template di progetto presentato a lezione che sono: Checkstyle, Pmd, SpotsBugs e Jacoco affinché potessimo realizzare codice di buona qualità.

- `TestAbstractControllerImpl`: Verifica il corretto funzionamento dei metodi pubblici della classe `AbstractControllerImpl`
- `GenericCommandsTest`: Controlla il corretto funzionamento dei metodi che eseguono il parsing e il controllo dei `GenericCommands` che il controller riceve dalla view.
- `ViewEventTypeFactoryTest`: Verifica la creazione corretta di eventi standard e generici, e che vengano lanciate eccezioni per argomenti invalidi (null, stringa vuota, tipo `GENERIC` usato come standard).
- `UserInputHandlerImplTest`: Testa la gestione di eventi non di tipo user-input e di eventi senza azioni registrate.
- `GuiEventHandlerImplTest`: Verifica la gestione di eventi GUI senza handler associato, il lancio di eccezioni per eventi generici e per input null.
- `CompositeInputHandlerTest`: Testa la corretta delega degli eventi ai sotto-handler: JUMP, START, MENU, RESUME e comandi generici. Verifica il lancio di eccezione per eventi null

- `TestInvalidViewEventArgsException`: Verifica i messaggi di errore dell'eccezione con costruttore di default e con messaggio aggiuntivo.
- `TestGameLoopImpl`: Esegue il test dei principali metodi del gameloop eccetto il resume.
- `GameModelImplTest`: Esegue il test della maggior parte dei metodi pubblici della classe GameModel impl.
- `AbstractGameModelTest`: Esegue il test della logica implementata tramite template method della classe AbstractGameModel.
- `PlayerImplTest`: Verifica il binding della fisica, la rotazione in aria, lo snap della rotazione al vertice più vicino al contatto col suolo, la normalizzazione dell'angolo, il respawn dopo la morte, il reset di monete e power-up alla morte, e i callback `onDeath` e `onSpecialObjectCollision`.
- `CoinTest`: Verifica creazione, copia, e che la collisione col player incrementi le monete e disattivi la coin.
- `ShieldPowerUpTest`: Verifica creazione, copia, e che la collisione col player attivi lo scudo e disattivi il power-up.
- `SpeedBoostPowerUpTest`: Verifica creazione, copia, e che la collisione col player applichi il moltiplicatore di velocità e disattivi il power-up.
- `PowerUpManagerTest`: Testa lo stato iniziale (no scudo, moltiplicatore 1.0), attivazione/consumo scudo, scadenza del boost di velocità nel tempo, e reset completo
- `PowerUpFactoryTest`: Verifica che la factory crei istanze corrette di Coin, ShieldPowerUp e SpeedBoostPowerUp in base al tipo richiesto.
- `SpikeTest`: Verifica creazione, hitbox triangolare, dimensioni, copia, e collisione (morte del player senza scudo, distruzione dello spike con scudo).
- `BlockTest`: Verifica creazione, hitbox quadrata, dimensioni, copia, e collisione (morte del player solo se arriva lateralmente, non dall'alto).
- `HitBoxTest`: Verifica calcolo di larghezza e altezza, immutabilità della lista di vertici e verifica della coerenza dei vertici con la definizione di poligono concavo.

- LevelTest: Verifica le impostazioni del livello, condizione di vittoria, recupero degli oggetti in un range, recupero di tutti gli oggetti, ricerca per posizione, conteggio totale monete, e coordinata di vittoria.
- GameMapTest: Testa il recupero di celle presenti, il filtraggio per range, la validazione delle coordinate, e il recupero opzionale di una cella
- CoordinateTest: Verifica i valori x/y e l'uguaglianza tra coordinate.
- CellTest: Verifica che una cella contenga correttamente il suo GameObject.
- CollisionHandlerTest: Testa il rilevamento delle collisioni nel mondo fisico che avviene tramite i metodi begin e end contact. Verifica il corretto cambiamento dello stato del player a seguito della collisione con un oggetto speciale di gioco.
- PlayerPhysicsTest: Verifica il salto, il reset della posizione del body, la conversione da coordinate mondo a coordinate gioco, e il set/get di userData.
- JBox2DPhysicsEngineImplTest: Esegue il test di alcuni metodi della classe, per garantire le principali funzionalità.
- TestAbstractObservableWithSet: Testa il corretto funzionamento tipico dell'observer pattern.
- ModelEventTest: Test dei metodi principali della classe.
- ModelEventTypeTest: Test dei metodi di utility dell'enumerativo.
- StandardViewEventTypeTest: Testa i nomi dei comandi, la distinzione tra eventi GUI e user-input, e la completezza dell'enum.
- StandardViewEventTest: Verifica la creazione di un evento standard con tipo valido e il lancio di eccezione per tipo null.
- SwingStrategyWithObservableTest: Verifica il mapping dei tasti e che quelli non mappati non producano eventi.
- SwingKeyboardListenerTest: Testa la delega del keyCode alla strategy, il cambio dinamico di strategy,

## 3.2 Note di sviluppo

### Latinì Luca

#### Uso di stream

Permalink: <https://github.com/PaoloDeMori/OOP24-GB/blob/3d60d329ee1f0a76f8f18be9acsrc/main/java/it/unibo/geometrybash/model/level/map/GameMapImpl.java#L97C5-L105C6>

#### Uso di generici

Permalink: <https://github.com/PaoloDeMori/OOP24-GB/blob/3d60d329ee1f0a76f8f18be9acsrc/main/java/it/unibo/geometrybash/model/powerup/AbstractPowerUp.java#L16>

#### Uso di Optional

Permalink: <https://github.com/PaoloDeMori/OOP24-GB/blob/3d60d329ee1f0a76f8f18be9acsrc/main/java/it/unibo/geometrybash/model/level/LevelImpl.java#L71-L75>

Permalink: <https://github.com/PaoloDeMori/OOP24-GB/blob/3d60d329ee1f0a76f8f18be9acsrc/main/java/it/unibo/geometrybash/model/level/map/CellImpl.java#L12-L31>

#### Uso di Wildcards

Permalink: <https://github.com/PaoloDeMori/OOP24-GB/blob/3d60d329ee1f0a76f8f18be9acsrc/main/java/it/unibo/geometrybash/model/powerup/PowerUpFactory.java#L28>

#### Uso di Lambda expressions

Permalink: <https://github.com/PaoloDeMori/OOP24-GB/blob/3d60d329ee1f0a76f8f18be9acsrc/main/java/it/unibo/geometrybash/model/level/map/GameMapImpl.java#L24-L29>

## **De Mori Paolo**

### **Uso di stream**

<https://github.com/PaoloDeMori/OOP24-GB/blob/3d60d329ee1f0a76f8f18be9ade1732a834src/main/java/it/unibo/geometrybash/model/physicsengine/impl/jbox2d/BodyFactoryImpl.java#L106-L109>

### **Uso di generici**

<https://github.com/PaoloDeMori/OOP24-GB/blob/3d60d329ee1f0a76f8f18be9ade1732a834src/main/java/it/unibo/geometrybash/model/physicsengine/GameModelWithPhysicsEngine.java#L11-L21>

### **Uso di Optional**

Permalink: <https://github.com/PaoloDeMori/OOP24-GB/blob/3d60d329ee1f0a76f8f18be9ade1732a834src/main/java/it/unibo/geometrybash/controller/GenericCommands.java#L75-L81>

### **Uso di Wildcards**

<https://github.com/PaoloDeMori/OOP24-GB/blob/3d60d329ee1f0a76f8f18be9ade1732a834src/main/java/it/unibo/geometrybash/commons/pattern/observerpattern/Observable.java#L18>

### **Uso di Lambda expressions**

<https://github.com/PaoloDeMori/OOP24-GB/blob/3d60d329ee1f0a76f8f18be9ade1732a834src/main/java/it/unibo/geometrybash/controller/AbstractControllerImpl.java#L101-L109>

### **Uso di libreria di terze parti (Jbox2D)**

<https://github.com/PaoloDeMori/OOP24-GB/blob/3d60d329ee1f0a76f8f18be9ade1732a834src/main/java/it/unibo/geometrybash/model/physicsengine/impl/jbox2d/Jbox2DPhysicsEngineImpl.java#L92-L96>

### **Uso di libreria di terze parti (Json)**

<https://github.com/PaoloDeMori/OOP24-GB/blob/3d60d329ee1f0a76f8f18be9ade1732a834src/main/java/it/unibo/geometrybash/model/level/loader/LevelLoaderImpl.java#L160-L172>

## **Montalti Elena**

### **Uso di stream**

<https://github.com/PaoloDeMori/OOP24-GB/blob/3d60d329ee1f0a76f8f18be9ade1732a834/src/main/java/it/unibo/geometrybash/model/geometry/HitBox.java#L91-L117>

### **Uso di generici**

<https://github.com/PaoloDeMori/OOP24-GB/blob/3d60d329ee1f0a76f8f18be9ade1732a834/src/main/java/it/unibo/geometrybash/model/player/Player.java#L22>

### **Uso di libreria di terze parti (Jbox2D)**

<https://github.com/PaoloDeMori/OOP24-GB/blob/3d60d329ee1f0a76f8f18be9ade1732a834/src/main/java/it/unibo/geometrybash/model/physicsengine/impl/jbox2d/PlayerPhysicsImpl.java#L48-L57>

# Capitolo 4

## Commenti finali

### 4.1 Autovalutazione e lavori futuri

#### Latinì Luca

Lavorare a questo progetto è stata un'esperienza decisamente formativa. Nonostante le difficoltà iniziali dovute alla mia limitata esperienza con progetti di queste dimensioni, ho trovato l'intero percorso stimolante.

Ho dedicato particolare attenzione alla fase progettuale, cercando di applicare i design pattern analizzati a lezione per far sì che il codice fosse manutenibile e pronto in caso di refactoring. Tuttavia, nonostante l'attenta pianificazione iniziale, la struttura ha subito alcune modifiche in corso d'opera.

Un momento critico è stato l'uscita di un componente dal gruppo di lavoro a ridosso della consegna. Questo imprevisto ha influito negativamente sullo sviluppo della MainMenuView, essendo parte del lavoro del componente uscito, rallentando i tempi e costringendomi a una riorganizzazione rapida del lavoro. Sono consapevole che questa parte del progetto presenta ancora dei margini di miglioramento, specialmente per quanto riguarda la verbosità del codice in alcuni punti, che avrei voluto snellire con più tempo a disposizione.

Nonostante queste sfide, sono molto soddisfatto del risultato ottenuto e della collaborazione instaurata con i miei compagni, che mi ha permesso di consolidare le mie conoscenze in un contesto di lavoro collettivo reale.

#### De Mori Paolo

La realizzazione di questo progetto è stata particolarmente utile per comprendere al meglio i principali concetti della programmazione Object Oriented. Nonostante le difficoltà che questo progetto comporta non sono poche sento

di aver aumentato di molto, grazie ad esso, il mio bagaglio di conoscenze, non solo del linguaggio java, ma in generale mi ha permesso di comprendere a pieno come si deve procedere per la stesura di un buon software e di quanto, un buon design può impattare sulla semplicità di manutenzione e creazione del codice.

Una grande difficoltà è stata sicuramente comprendere bene come strutturare un progetto di questo tipo, mantenendo una struttura solida, il che ha richiesto a me e ai miei compagni grande impegno e studio. Inoltre durante il testing delle classi più complesse mi è risultato difficile ottenere una copertura totale delle casistiche, nonostante un tool come jacoco, abbia comunque aiutato molto.

### **Montalti Elena**

Nel complesso, sono soddisfatta del risultato. Questo progetto è stato fondamentale per mettere in pratica quanto appreso durante il corso, aiutandomi a dare concretezza alla teoria, che all'inizio non era sempre facile da interpretare. Sento di aver acquisito una buona base sui principi teorici, ma ho capito che devo ancora lavorare sulla gestione degli imprevisti e sulla flessibilità. Mi sono accorta che, davanti a un bug, la mia prima reazione è cercare la soluzione più rapida per risolvere il problema nell'immediato; questo però mi porta a volte a perdere di vista la strategia generale e l'impatto che quella scelta ha sull'intero progetto.

Infine, lavorare con i miei compagni è stato fantastico: si è creato un clima di confronto sincero e di grande collaborazione che mi ha aiutata tantissimo.

# Appendice A

## Guida utente

### Menù

All'avvio dell'applicazione, l'utente viene accolto da una schermata che simula un terminale in stile bash, coerente con il tema del progetto. Questa interfaccia è completamente interattiva e permette la navigazione attraverso l'inserimento di specifici comandi testuali. Seguendo le indicazioni riportate sotto il logo principale, è possibile richiamare la lista dei comandi disponibili per configurare la sessione di gioco. Nello specifico, l'utente può utilizzare il comando `man resolution` per consultare le tre risoluzioni video supportate, mentre i comandi `levels` e `colors` consentono di visualizzare, rispettivamente, l'elenco dei livelli affrontabili e le opzioni cromatiche disponibili per la personalizzazione estetica del proprio player.

### Game

La partita ha inizio una volta digitato il comando `start`. Durante la sessione è possibile controllare il player tramite la barra spaziatrice o la freccia direzionale superiore. Entrambi i tasti permettono di eseguire i salti necessari per evitare gli ostacoli mortali o per raggiungere le piattaforme elevate dove sono collocate le monete collezionabili. In qualsiasi momento è possibile sospendere l'azione premendo il tasto Esc. Questa operazione rimanda a un menù di pausa testuale in cui l'utente può decidere se riprendere la partita dal punto esatto dell'interruzione tramite il comando `resume`, oppure se ricominciare il livello dall'inizio utilizzando il comando `restart`.

### Completamento

Il sistema di gioco prevede il riavvio immediato del livello in caso di collisione con gli ostacoli, garantendo un'esperienza continua fino al raggiungimento del traguardo. Una volta completato il percorso, il menù mostrerà un riepilogo delle statistiche finali, indicando in particolare il numero di monete raccolte

rispetto al totale presente nel livello e offrendo la possibilità di utilizzare tutti i comandi precedentemente citati come disponibili nel menu iniziale.

# Bibliografia

- [Cat25] Erin Catto. Box2d documentation. <https://box2d.org/documentation/>, 2025. Novembre 2025.
- [Mac25] Daniel Murphy and altri contributori. Jbox2d: a 2d java physics engine (repository). <https://github.com/jbox2d/jbox2d>, 2025. Novembre 2025.