

S.C.A.T.
Space Combat: Alien Takeover

Girolamo Ronzoni
Leonardo Lioi
Mario Lungu

February 15, 2026

Contents

1	Analisi	2
1.1	Descrizione e requisiti	2
1.2	Modello del Dominio	4
2	Design	6
2.1	Architettura	6
2.2	Design dettagliato	8
2.2.1	Girolamo Ronzoni	8
2.2.2	Leonardo Lioi	14
2.2.3	Mario Lungu	19
3	Sviluppo	24
3.1	Testing automatizzato	24
3.2	Note di sviluppo	26
3.2.1	Girolamo Ronzoni	26
3.2.2	Leonardo Lioi	27
3.2.3	Mario Lungu	28
4	Commenti finali	29
4.1	Autovalutazione e lavori futuri	29
4.1.1	Girolamo Ronzoni	29
4.1.2	Mario Lungu	30
4.1.3	Leonardo Lioi	31
4.2	Difficoltà incontrate e commenti per i docenti	32
A	Esercitazioni di laboratorio	33

Chapter 1

Analisi

1.1 Descrizione e requisiti

Il progetto Space Combat: Alien Takeover (S.C.A.T.) consiste nello sviluppo di un video-gioco che rivisita in chiave moderna il classico arcade Space Invaders. L'applicazione pone il giocatore al comando di una navicella spaziale con l'obiettivo di difendere la propria posizione da un'invasione aliena organizzata in ondate successive di difficoltà crescente. S.C.A.T. gestisce un ambiente di gioco popolato da diverse entità: il giocatore (player), gli invasori (invaders), i proiettili (shot, sia del player che degli invaders) e i bunker. Il giocatore può spostarsi orizzontalmente e fare fuoco per distruggere i nemici. Gli invaders si muovono seguendo sempre lo stesso schema coordinato, avvicinandosi sempre di più al player e possono rispondere al fuoco. A protezione della navicella sono presenti dei bunker, strutture difensive statiche che assorbono i colpi degradandosi progressivamente fino alla distruzione.

L'obiettivo è totalizzare il maggior punteggio possibile eliminando gli alieni prima che raggiungano i bunker o che distruggano la navicella del player. Il gioco include un sistema di gestione dei record per salvare e visualizzare i migliori punteggi in una classifica persistente.

Requisiti funzionali

- Possibilità di navigare nel menù, accedendo alle seguenti schermate:
 - Schermata *tutorial* con una breve storia del gioco e un tutorial di base.
 - Schermata *leaderboard* nella quale sarà possibile consultare la lista delle partite svolte in precedenza da ciascun utente, ordinate per livello raggiunto e punteggio.
 - Schermata di *gioco* che permette all'utente di iniziare effettivamente la partita.

Ciascuna schermata dovrà permettere all'utente di tornare al menù principale.

- Possibilità per l'utente di impostare un nickname e di personalizzare la propria navicella.
- Gestione del movimento delle entità e delle interazioni e collisioni tra esse.
- Incremento progressivo della difficoltà, ossia maggior velocità degli invaders e maggior velocità nella generazione dei proiettili da parte degli invaders.
- Riproduzioni degli effetti sonori di gioco.

- Possibilità di salvare e consultare la leaderboard.
- Possibilità di mettere in pausa la partita, di riprenderla, o di resettarla.

Requisiti non funzionali

- S.C.A.T. deve rispondere correttamente agli input dell'utente ed essere fluido e prestante.
- S.C.A.T. deve essere progettato in un'ottica di estendibilità futura, favorendo la separazione delle responsabilità e prevedendo la possibile aggiunta di nuove entità, schermate o regole.
- S.C.A.T. deve presentare un'interfaccia grafica accattivante e innovativa, ma senza distaccarsi eccessivamente dal gioco originale.

1.2 Modello del Dominio

Il dominio applicativo di S.C.A.T. è quello di un videogioco arcade di tipo *space shooter*, in cui una partita si svolge all'interno di un mondo di gioco popolato da diverse entità che interagiscono tra loro secondo regole ben definite.

La partita è caratterizzata da uno stato, che può essere di esecuzione, pausa o terminazione, e termina in condizioni di vittoria o sconfitta. Il mondo di gioco contiene tutte le entità coinvolte nella partita e ne governa le interazioni.

Le entità di dominio principali sono il *player*, gli *invader*, i *bunker* e gli *shot*. Il player rappresenta l'entità controllata dall'utente, mentre gli invader rappresentano i nemici presenti nel mondo di gioco. Gli invader sono suddivisi in più tipologie, tre standard e una speciale detta *bonus invader*, che si distingue per dimensioni maggiori, comparsa sporadica e un pattern di movimento diverso da quello degli altri invader. I bunker fungono da elementi difensivi, interponendosi tra player e invader. Gli shot rappresentano i proiettili generati sia dal player sia dagli invader.

Le entità possono interagire tramite collisioni: un proiettile può colpire un invader, il player o un bunker, producendo una riduzione delle vite dell'entità colpita o la sua eliminazione dal mondo di gioco; i proiettili degli invader e quelli del player viaggiano chiaramente in direzioni opposte, e possono anche scontrarsi tra di loro, eliminandosi a vicenda. Ogni entità è caratterizzata da un numero di vite, che varia in base al tipo: gli invader possiedono una sola vita, mentre player e bunker possono resistere a più colpi.

Gli invader si muovono autonomamente secondo un pattern predefinito e generano proiettili. La difficoltà della partita è incrementale e dipende dal livello corrente e dal numero di invader ancora presenti. Il player può muoversi e generare proiettili che si muovono in direzione opposta rispetto a quelli degli invader.

Il dominio include inoltre un sistema di punteggio: il player accumula punti eliminando gli invader, con un punteggio che varia in base alla tipologia dell'invader eliminato. I risultati delle partite vengono registrati in una *leaderboard*, che consente la persistenza e la consultazione dei punteggi associati agli utenti.

Gli elementi costitutivi del dominio del problema sono sintetizzati in Figure 1.1.

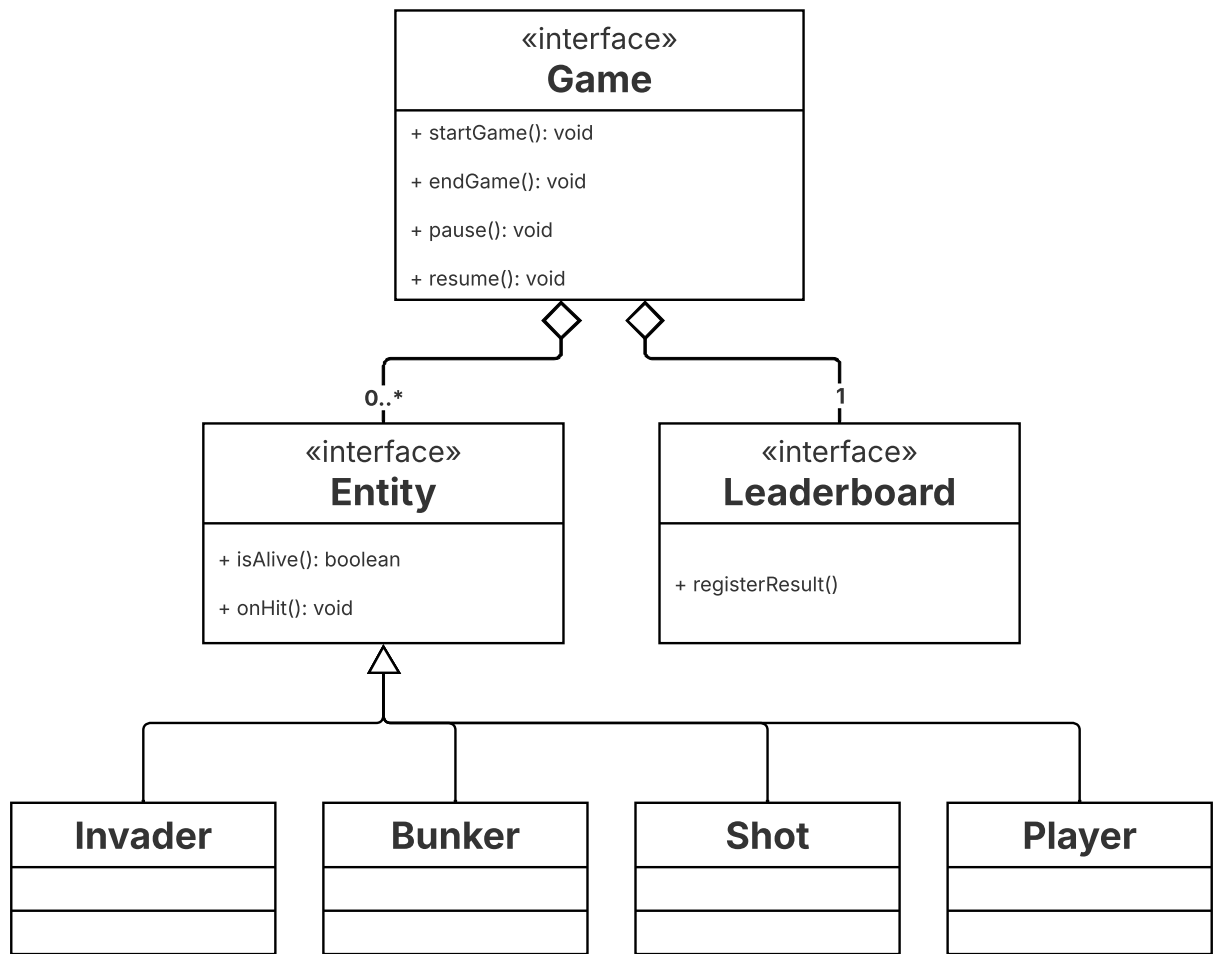


Figure 1.1: Schema UML dell'analisi del dominio, con rappresentate le entità principali ed i rapporti fra loro

Chapter 2

Design

2.1 Architettura

L'architettura dell'applicazione segue il pattern *Model-View-Controller* (MVC), scelto per separare nettamente la logica di gioco dalla rappresentazione grafica e dalla gestione degli input dell'utente.

Il **Model** rappresenta il cuore dell'applicazione e incapsula lo stato della partita e le regole del dominio. È responsabile della gestione delle entità di gioco, dell'evoluzione della partita, del calcolo del punteggio e del salvataggio e caricamento della leaderboard. Il modello non dipende dalla rappresentazione grafica né dai meccanismi di input.

La **View** si occupa esclusivamente della visualizzazione dello stato del gioco e dell'interazione con l'utente. Essa mostra le schermate di menu, la partita in corso e la leaderboard, e traduce gli input dell'utente in eventi ad alto livello, senza contenere logica di gioco.

Il **Controller** funge da mediatore tra View e Model. Riceve gli input generati dalla View, li interpreta e invoca le opportune operazioni sul Model. Allo stesso tempo coordina il flusso dell'applicazione, ad esempio gestendo l'avvio, la pausa e la terminazione della partita.

La comunicazione tra i componenti è progettata in modo da ridurre al minimo l'accoppiamento: la View accede ad alcuni dati del Model esclusivamente in lettura, mentre ogni modifica allo stato di gioco avviene tramite il Controller. Questa scelta architetturale favorisce la manutenibilità, l'estendibilità del software e la possibilità di sostituire la View senza impatti significativi sulla logica.

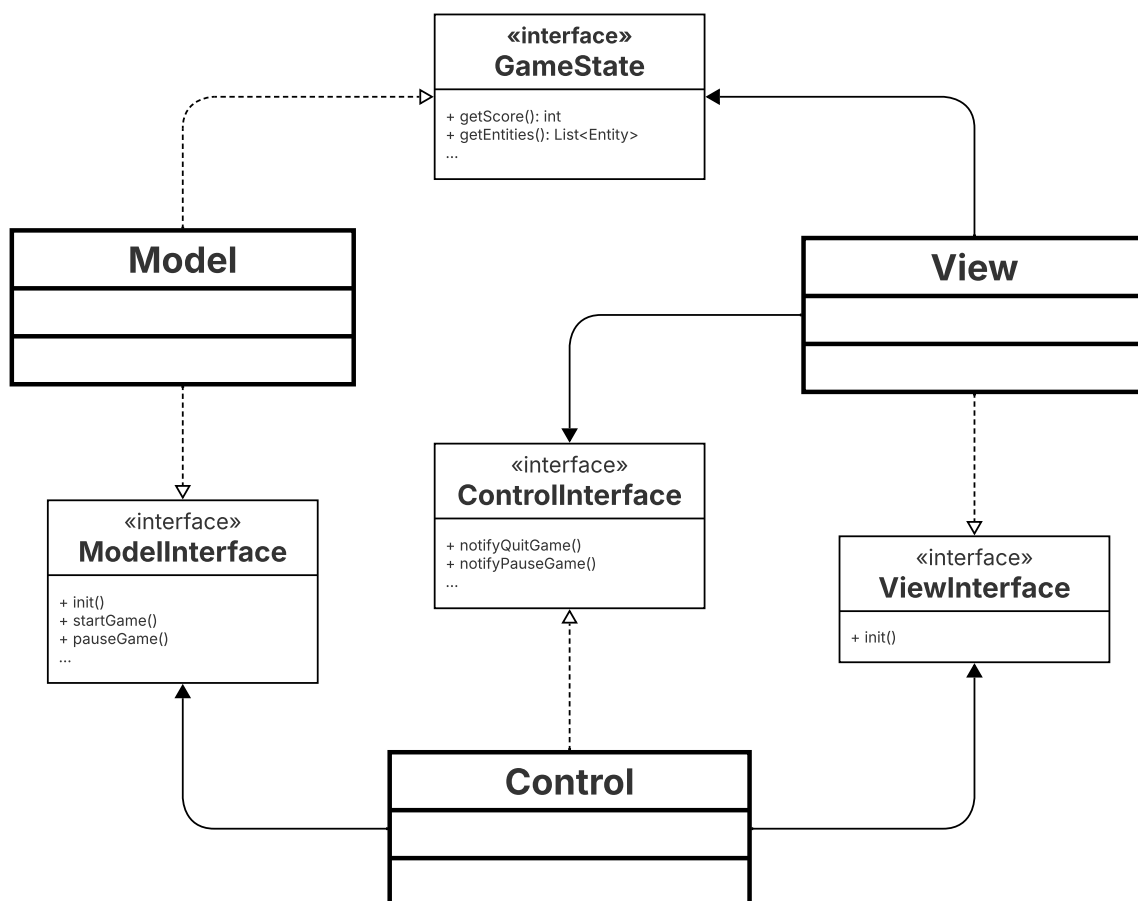


Figure 2.1: Schema UML architetturale di **S.C.A.T.**: la classe **Control** è il controller del sistema, mentre **ModelInterface** e **ViewInterface** sono le interfacce che **Model** e **View** espongono per **Control**.

Control a sua volta espone un'interfaccia **ControlInterface** per consentire alla **View** di notificargli gli input dell'utente.

Infine, **Model** espone un'interfaccia di sola lettura per permettere alla **View** di conoscere lo stato della partita e delle entità, per poterle riprodurre graficamente.

2.2 Design dettagliato

2.2.1 Girolamo Ronzoni

1) Struttura del Model: separazione dello stato e della logica

Problema: Durante lo sviluppo della parte Model, la gestione delle entità (creazioni, rimozioni, riferimenti alle varie liste di entità) e l'applicazione delle regole di gioco (movimento, collisioni, fine partita, generazione colpi, controlli vari) rischiavano di convergere in un unico componente centrale, rendendo il codice difficile da estendere e mantenere e aumentando l'accoppiamento fra responsabilità diverse.

In sostanza la classe `Game` (la classe principale del Model) stava diventando una God-class.

Soluzione: La soluzione adottata consiste nel suddividere la classe `Game` in due sotto-classi, con responsabilità diverse:

- **GameWorld:** classe che contiene esclusivamente lo *stato* del mondo di gioco, ossia i riferimenti alle collezioni di entità (player, invaders, shots ecc.), oltre alle operazioni strettamente necessarie a mantenerle consistenti (ad esempio `addEntity(...)` e `removeEntity(...)`).
- **GameLogic:** incapsula le *regole* e gli aggiornamenti della partita, operando sullo stato fornito da `GameWorld` (ad esempio `update()`, `checkGameEnd()`, `addPlayerShot()`, ecc...).

Questa soluzione permette di:

1. applicare il **Single Responsibility Principle**: una classe si occupa solo dello stato del mondo di gioco, e un'altra classe si occupa solo delle regole.
2. applicare il pattern di **Dependency Injection**, poiché `GameLogic` non crea direttamente lo stato del mondo ma riceve dall'esterno il riferimento a `GameWorld`, riducendo l'accoppiamento tra i componenti e rispettando il principio **IoC**.
3. favorire l'**estendibilità**: eventuali nuove regole aggiunte in futuro impatteranno principalmente `GameLogic` senza richiedere modifiche alla struttura del mondo di gioco.
4. rendere il codice più chiaro e leggibile, riducendo anche significativamente il rischio di God-class.

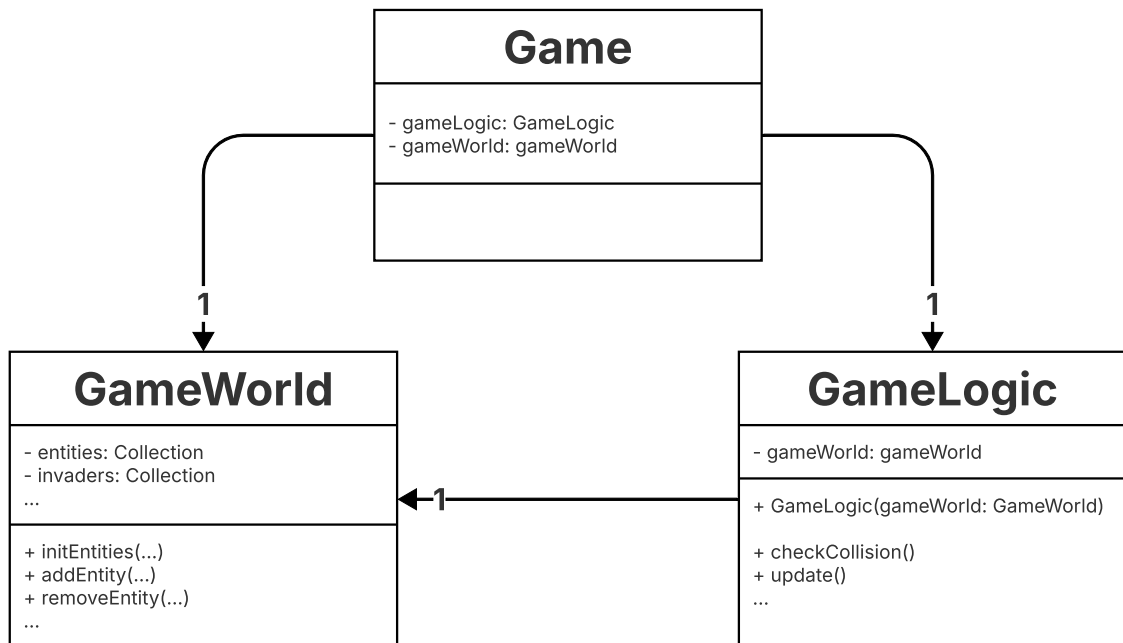


Figure 2.2: Schema UML della suddivisione di **Game** in sottoclassi

2) Isolamento della logica di creazione delle entità mediante **Factory Method**

Problema: Come già visto, **GameWorld** è il sotto-componente del Model incaricato di mantenere lo stato del mondo di gioco e di gestire le entità presenti, inclusa la loro creazione. All'interno di **GameWorld** vengono gestite diverse tipologie di entità, organizzate in strutture distinte (lista di tutte le entità, lista con soli invaders, lista con soli colpi, e così via), ciascuna con caratteristiche e modalità di creazione differenti. In particolare le entità non solo sono rappresentate da classi diverse, ma alcune entità presentano anche costruttori con parametri diversi (ad esempio gli shot, che richiedono informazioni aggiuntive).

Il problema consisteva quindi nel definire una modalità di creazione delle entità che fosse coerente con la separazione delle responsabilità, evitando di concentrare all'interno di **GameWorld** una logica di istanziazione complessa e fortemente dipendente dalle singole tipologie di entità.

Soluzione: in una prima fase è stata considerata la possibilità di gestire direttamente la creazione delle entità all'interno di **GameWorld**, tramite una serie di metodi dedicati (ad esempio `createInvader()`, `createShot()`, `createPlayer()`, ecc.). Questa soluzione, seppur semplice, avrebbe però portato ad un aumento significativo delle responsabilità di **GameWorld**, rendendolo fortemente accoppiato alle singole implementazioni concrete delle entità e più difficile da estendere in caso di introduzione di nuove tipologie; per questo motivo è stata scartata.

Per risolvere il problema è stato dunque adottato il pattern *Factory Method*: è stata introdotta un'interfaccia **EntityFactory**, con un unico metodo `createEntity(...)`, e una sua implementazione concreta **EntityFactoryImpl**, incaricata della creazione delle entità di gioco.

In questa soluzione **GameWorld** delega la responsabilità della creazione delle entità alla

factory, limitandosi a ricevere l'istanza creata e ad inserirla nella struttura dati appropriata. La factory incapsula la logica necessaria a distinguere le diverse tipologie di entità e a invocare il costruttore corretto in base ai parametri forniti. L'adozione del Factory Method ha permesso di isolare la logica di creazione delle entità migliorando la manutenibilità e l'estendibilità del sistema.

GameWorld rimane focalizzato sulla gestione dello stato del mondo di gioco, mentre la factory può essere modificata o estesa separatamente, senza impatti sulle altre componenti del Model.

Questa architettura rispetta **SRP** e **OCP**.

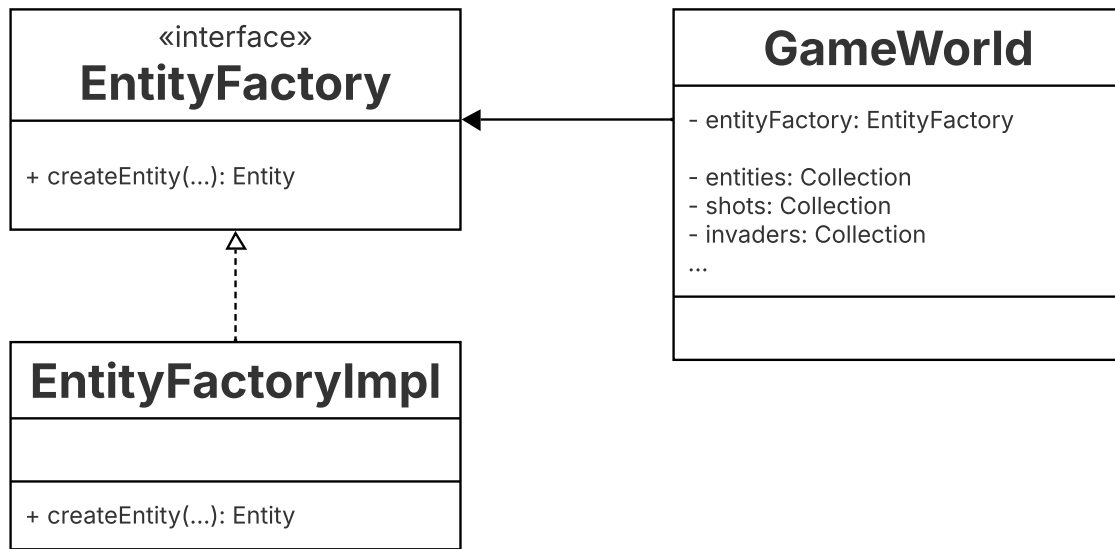


Figure 2.3: Schema UML dell'uso del Factory Method per la creazione delle entità all'interno di **GameWorld**

3) Meccanismo di aggiornamento della View basato su Observer

Problema: La View necessita di essere aggiornata in tempo reale sullo stato della partita, in particolare su informazioni quali punteggio, livello e posizione delle entità. Tuttavia, nel rispetto del pattern MVC e del principio di separazione delle responsabilità la View non deve avere accesso diretto al Model, nè il Model alla View.

Soluzione: Per risolvere questo problema è stata anzitutto introdotta un'interfaccia di sola lettura, denominata **ModelState**, implementata dal Model. Tale interfaccia espone esclusivamente metodi di accesso ai dati necessari alla View, senza consentirne la modifica (vengono restituite sempre copie difensive), garantendo così il disaccoppiamento tra i due componenti e preservando l'incapsulamento dello stato interno del Model. Successivamente è stata introdotta una funzione **update()** nella View, incaricata di aggiornare i dati tramite chiamate ai metodi di **ModelState** e di ridisegnare l'interfaccia grafica.

Per regolare gli aggiornamenti dell'interfaccia (ossia le chiamate al metodo **update()** della View) sono state valutate due opzioni:

1. La funzione **update()** della View viene chiamata dal Controller ad ogni iterazione del ciclo di gioco.

2. Implementazione del pattern *Observer* per la notifica delle variazioni di stato. In questa soluzione il Model agisce come soggetto osservabile, mentre la View si registra come osservatore. Il Model notifica la View solo quando lo stato di gioco subisce modifiche, e solo a quel punto la View esegue la funzione `update()`.

La prima soluzione, seppur funzionale e semplice da implementare, presentava alcuni limiti: introduceva una dipendenza rigida tra aggiornamento del Model e aggiornamento della View e comportava un ridisegno continuo della View anche in assenza di reali cambiamenti nello stato di gioco. La seconda soluzione, invece, consente di disaccoppiare ulteriormente il ciclo di aggiornamento del Model da quello della View, evitando che il Controller debba coordinare esplicitamente il ridisegno dell'interfaccia grafica; l'adozione del pattern Observer permette inoltre di notificare la View solo in presenza di cambiamenti effettivi dello stato di gioco, riducendo aggiornamenti ridondanti, migliorando l'efficienza complessiva del sistema e favorendo l'estendibilità del software, rendendo possibile la sostituzione della View senza alcuna modifica al Model o al Controller.

È evidente come la seconda soluzione sia superiore alla prima, per questo motivo è stato deciso di implementarla.

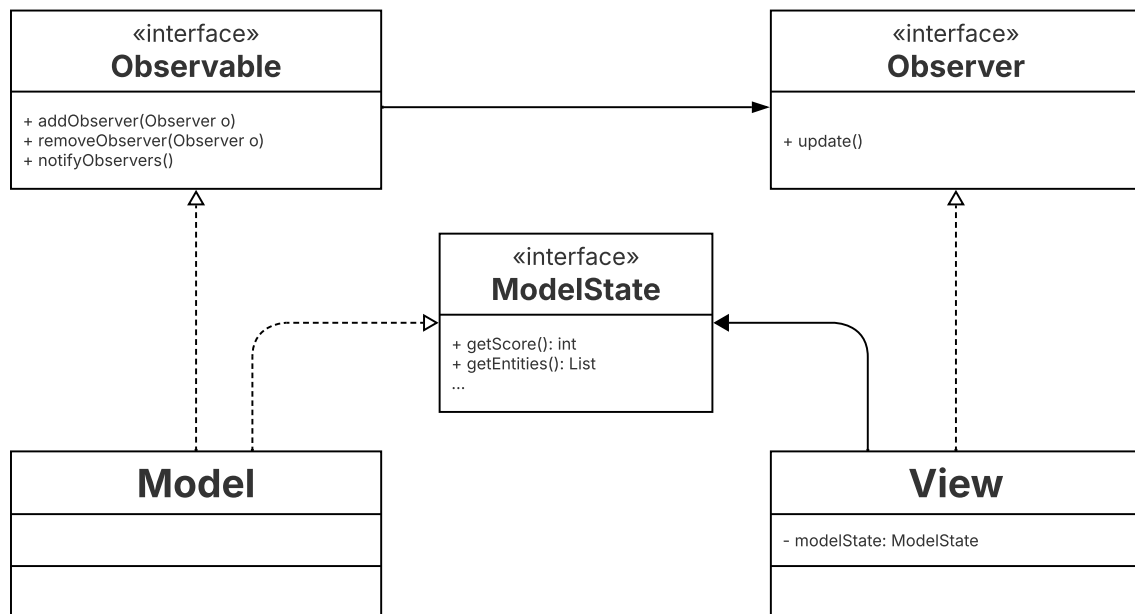


Figure 2.4: Comunicazione tra Model e View, mediante interfaccia di sola lettura **ModelState** e pattern Observer per l'aggiornamento dinamico.

4) Esposizione controllata dello stato delle entità verso la View

Problema: Ad ogni notifica di aggiornamento, la View richiede al Model i nuovi dati relativi alle entità presenti nel mondo di gioco (il meccanismo di comunicazione tra Model e View è già stato descritto nel paragrafo precedente).

Il Model restituisce sempre copie difensive dei propri dati, proteggendo lo stato interno; tuttavia, le entità di gioco espongono anche operazioni che ne modificano lo stato (ad esempio `move(...)` e `onHit()`), che non devono essere accessibili dalla View. Consentire alla View di invocare tali operazioni violerebbe la separazione delle responsabilità e l'incapsulamento della logica di gioco.

Soluzione: Per risolvere questo problema è stata introdotta un'interfaccia di sola lettura, denominata **EntityState**, implementata dalle entità. Tale interfaccia espone esclusivamente le informazioni necessarie alla rappresentazione grafica delle entità (ad esempio `getPosition()`, `getDimension()`, ecc.), nascondendo tutte le operazioni che comportano modifiche di stato. La View interagisce quindi unicamente con riferimenti di tipo **EntityState**, potendo leggere i dati senza alcuna possibilità di modificarli.

Le entità di gioco condividono inoltre una base comune modellata tramite la classe astratta **AbstractEntity** (questa scelta è approfondita in questo paragrafo) che raccoglie lo stato e il comportamento comuni alle diverse tipologie di entità (e implementa l'interfaccia **EntityState**). In questo modo il Model può operare sulle entità concrete attraverso la gerarchia astratta, mentre la View rimane confinata all'uso del solo contratto di **EntityState**. Sia la View che il Model conoscono il contratto di **EntityState**, in quanto tale interfaccia è collocata nel package condiviso `/common/api`.

Questa architettura rispetta i fondamentali principi di progettazione:

- **Single Responsibility Principle:** il Model rimane responsabile della logica di gioco e della gestione dello stato, mentre la View si occupa esclusivamente della rappresentazione grafica.
- **Open/Closed Principle:** l'uso dell'interfaccia **EntityState** consente in futuro di estendere le tipologie di entità o le informazioni esposte alla View senza richiedere modifiche al codice della View stessa.

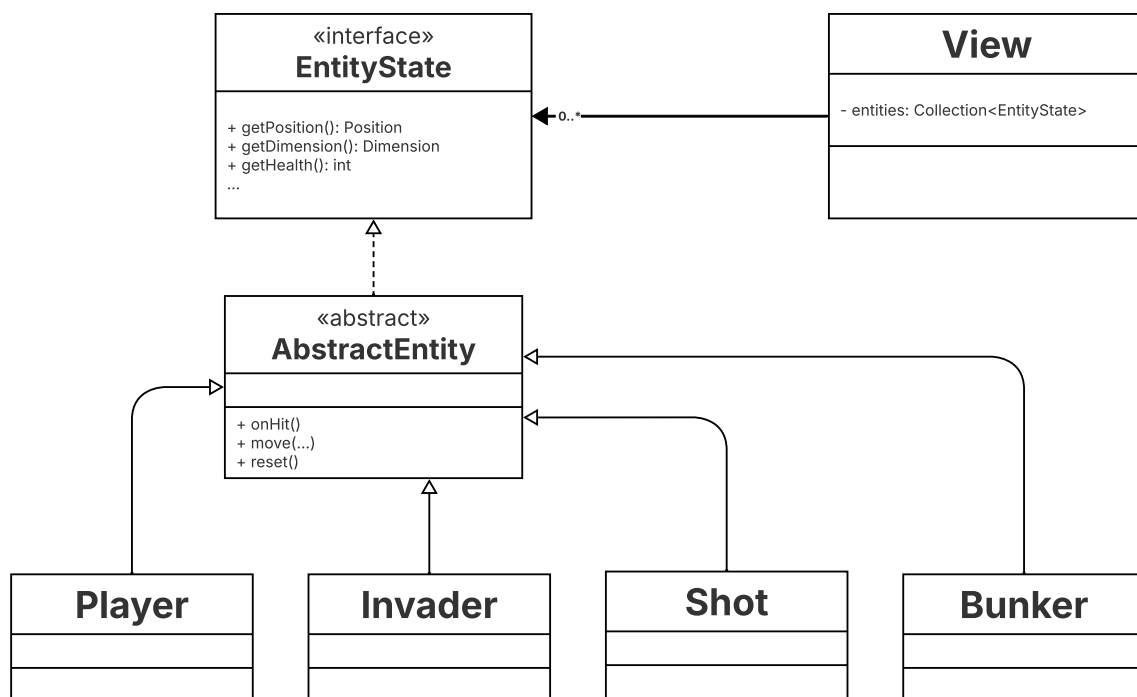


Figure 2.5: Schema UML dell'uso di **EntityState** per la lettura delle entità da parte della View

5) Gestione centralizzata degli sprite

Problema: La classe **Canvas**, è un sotto-componente della View ed è responsabile del rendering delle entità ad ogni aggiornamento del gioco. Poiché il ridisegno delle entità è invocato molto frequentemente, caricare e ridimensionare le immagini durante ogni rendering comporterebbe un notevole peggioramento delle prestazioni.

Inoltre, ogni immagine non solo deve essere caricata e ridimensionata in base ai fattori di scala del mondo grafico, ma molte entità possiedono più frame per gestire le animazioni, rendendo la gestione diretta degli sprite all'interno del componente grafico poco efficiente e poco modulare.

Soluzione: È stata introdotta l'interfaccia **SpriteManager**, con implementazione concreta **SpriteManagerImpl**, incaricata di fornire all'occorrenza le immagini degli sprite richiesti. Il caricamento, il ridimensionamento e la memorizzazione delle immagini è gestito internamente da **SpriteManagerImpl**, con metodi privati.

La classe **Canvas** prima di disegnare uno sprite dovrà limitarsi ad interrogare lo **SpriteManager** tramite il metodo **getSprite(...)** che restituirà lo sprite richiesto, sulla base di alcuni parametri forniti, garantendo efficienza, manutenibilità e separazione delle responsabilità.

Anche questa scelta architetturale è basata su **SRP** e **OCP**.

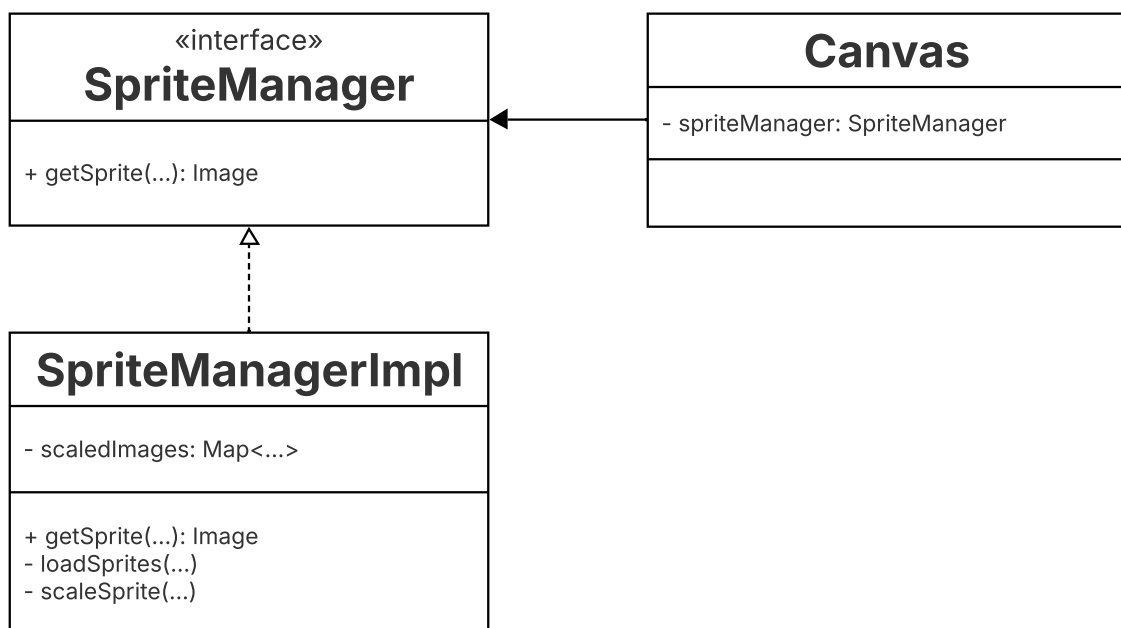


Figure 2.6: Schema UML dell'uso di **SpriteManager** per il caricamento, il ridimensionamento e il salvataggio degli sprite delle entità.

2.2.2 Leonardo Lioi

1) Gestione della Leaderboard: architettura e persistenza dei dati

Problema: La gestione dei punteggi e dello storico delle partite richiedeva un sistema per salvare e visualizzare i record in modo persistente. Senza una struttura dedicata, si rischiava di mescolare la logica del gioco con i dettagli tecnici del salvataggio su file (I/O), violando il principio SRP. Inoltre, integrare la logica di ordinamento della classifica (necessaria solo per fini di visualizzazione) all'interno del motore di gioco principale avrebbe creato un accoppiamento eccessivo. Questo avrebbe appesantito il Model principale con compiti non strettamente legati all'esecuzione della partita, andando contro la netta separazione delle responsabilità prevista dall'architettura MVC.

Soluzione: Per evitare queste criticità, la gestione della classifica è stata separata dal nucleo del gioco attraverso l'uso di componenti specializzati e interfacce:

- **L'interfaccia (LeaderboardInterface):** Per evitare che il Model dipenda da una classe specifica, è stata introdotta un'interfaccia. In questo modo, il Model non deve conoscere i dettagli tecnici del salvataggio (ovvero "come" vengono scritti i dati), ma si limita a chiamare i metodi necessari. Questo garantisce che il codice della partita non sia "incollato" a quello del sistema di archiviazione.
- **La classe concreta (Leaderboard):** Questa classe implementa l'interfaccia e si occupa della gestione pratica dei dati. Gestisce i file all'interno della cartella utente (.scat), carica i punteggi all'avvio e si occupa dell'ordinamento. Grazie a questa separazione, ogni modifica relativa ai file non influisce minimamente sullo svolgimento della partita.
- **La modellazione dei dati (GameRecord):** Per rappresentare i singoli risultati (nome, punti, livello e data), è stata creata la classe **GameRecord**. Essa funge da sottocomponente incaricato di organizzare le informazioni di ogni partita in un oggetto dedicato, rendendo il passaggio dei dati tra i moduli più ordinato e sicuro.

Vantaggi della scelta:

- **Organizzazione del codice:** il Model può concentrarsi esclusivamente sulla gestione della partita, delegando alla classifica tutto ciò che riguarda la persistenza dei dati.
- **Flessibilità:** se in futuro si decidesse di cambiare il formato di salvataggio, basterebbe modificare solo la classe **Leaderboard**, lasciando intatta la logica di gioco nel resto del progetto.
- **Chiarezza:** l'utilizzo di una classe specifica come **GameRecord** permette di gestire lo storico dei punteggi in modo strutturato, evitando la dispersione di dati all'interno del codice.

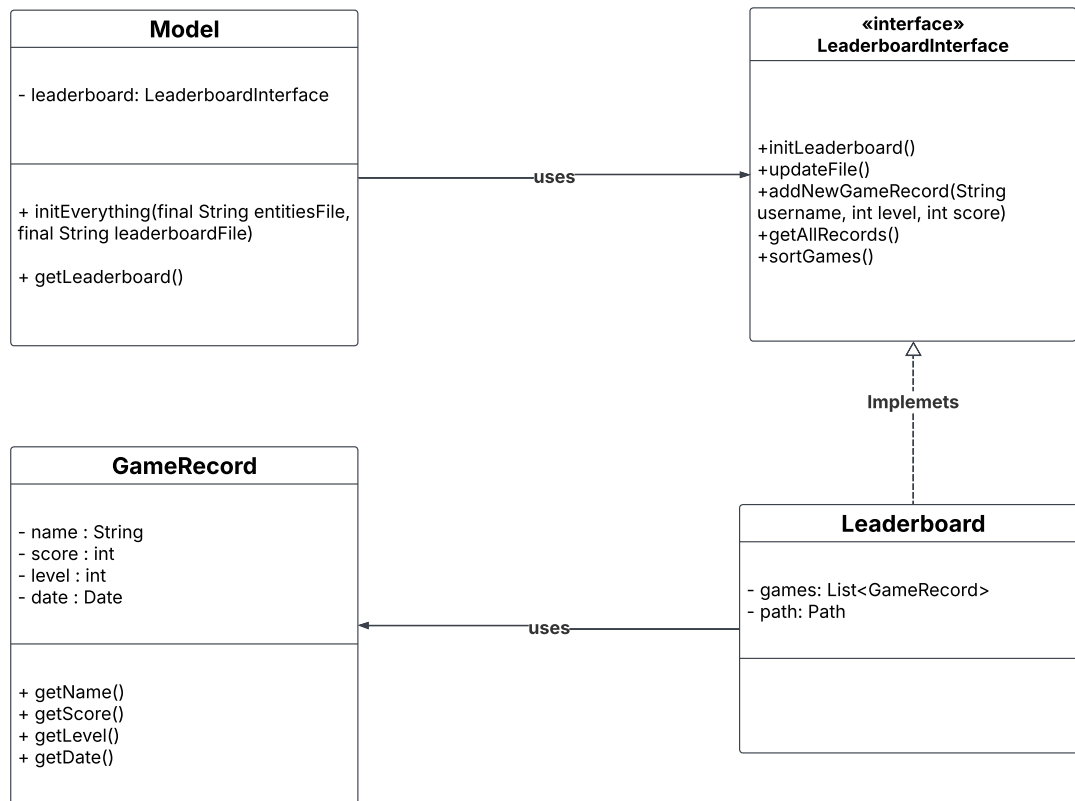


Figure 2.7: Schema UML della gestione della Leaderboard.

2) Gestione del tempo e delle entità: la classe TimeAccumulator

Problema: Una delle sfide principali nello sviluppo della logica di gioco è stata la gestione dei tempi di aggiornamento delle diverse entità, poiché ognuna richiede un "frame rate" indipendente per muoversi correttamente. Ad esempio, i proiettili (*shots*) si spostano ogni 100ms, mentre gli Invaders hanno ritmi differenti basati sulla difficoltà. Inserire variabili temporali o calcoli di latenza direttamente nei metodi di movimento avrebbe appesantito il **Model** con logiche matematiche estranee al comportamento delle entità, rendendo il codice caotico e difficile da sincronizzare con il ciclo di gioco (*game loop*).

Soluzione: Per risolvere questo problema è stata introdotta la classe **TimeAccumulator**, che funge da motore temporale per l'intero **Model**. Questa scelta di design permette di gestire lo scorrere del tempo in modo modulare e preciso:

- **Logica ad accumulatori atomici:** la classe utilizza oggetti **AtomicInteger** per gestire diversi accumulatori (uno per invasori, uno per il bonus e uno per i proiettili). Ad ogni ciclo di gioco, il metodo **incrementTimeAccumulators()** aggiunge il tempo trascorso (**GAME_STEP_MS**) a tutti i contatori.
- **Consumo del tempo e soglie:** tramite il metodo **consumeAccumulators()**, la classe verifica se il tempo accumulato ha raggiunto la soglia specifica definita nelle costanti (es. **SHOT_STEP_MS**). Se la soglia è superata, il tempo viene "consumato" e la logica di gioco riceve il via libera per eseguire il movimento.
- **Il caso del Bonus Invader:** nel metodo **handleBonusInvader(int bonusInvader)**, la logica interroga l'accumulatore dedicato. L'entità agisce solo quando è trascorso

il tempo necessario; in quel momento, il sistema decide se muovere il nemico (se già attivo) o se tentare di generarlo con una probabilità di *spawn* del 5%.

Questa architettura garantisce che ogni componente del gioco mantenga la propria velocità caratteristica indipendentemente dalla frequenza con cui il **Model** viene aggiornato. Separando il calcolo dei millisecondi dalla logica di comportamento, il codice risulta pulito e centralizzato: per modificare la reattività del gioco o la velocità di un'entità, è sufficiente intervenire sulle costanti di *step* o sulla classe **TimeAccumulator**, senza dover ricalibrare l'intero sistema di movimento.

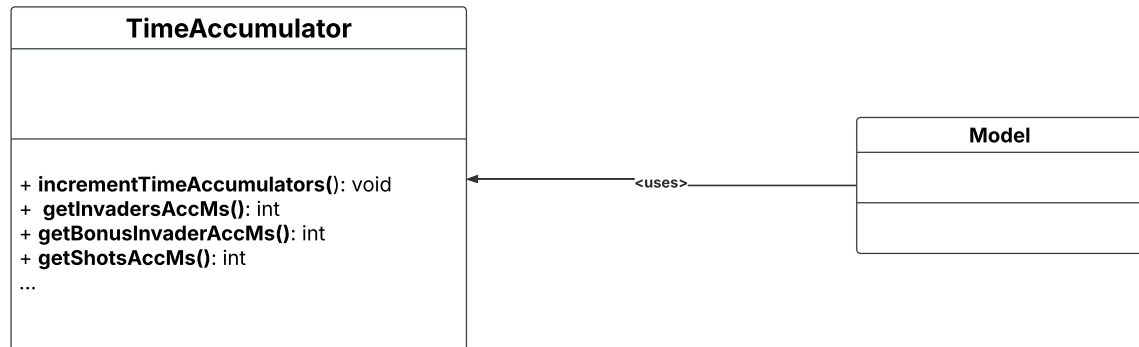


Figure 2.8: Schema UML della classe **TimeAccumulator** per la gestione sincronizzata dei tempi di gioco.

3) Gerarchia delle entità: implementazione della classe astratta **AbstractEntity**

Problema: La gestione di diverse tipologie di entità nel gioco (**Player**, **Invader**, **Shot**, **Bunker**) presentava il rischio di una massiccia duplicazione di codice. Molti attributi (**position**, **width**, **height**, **health**) e comportamenti (**onHit()**, **die()**, **reset()**) sono identici per ogni elemento presente sul campo di gioco. Utilizzare una classe concreta avrebbe permesso l'istanziamento di oggetti "generici" privi di senso logico, mentre l'uso di una semplice interfaccia avrebbe costretto a implementare manualmente la stessa logica di gestione dello stato in ogni singola sottoclasse.

Soluzione: Per risolvere questo problema, è stata introdotta la classe astratta **AbstractEntity**. La scelta dell'astrazione rispetto a un'interfaccia è stata fondamentale per i seguenti motivi tecnici:

- **Campi condivisi:** a differenza di un'interfaccia, la classe astratta permette di dichiarare campi comuni come **position**, **health** e **alive** una sola volta. Questo evita di dover ridichiarare le variabili e i relativi getter/setter in tutte le classi concrete.
- **Implementazione di default:** metodi come **onHit()** (che decrementa la salute e chiama **die()**) o **reset()** (che ripristina **startingPosition** e **startingHealth**) possiedono una logica che non cambia tra un **Invader** e un **Player**. La classe

astratta permette di scrivere questo codice una sola volta, garantendo che tutte le sottoclassi lo ereditino correttamente (principio DRY).

- **Incapsulamento:** **AbstractEntity** permette di proteggere variabili sensibili (come **startingHealth**) rendendole inaccessibili dall'esterno (**private**) ma utilizzabili dalla logica interna di **reset**, fornendo un controllo maggiore rispetto a quanto permesso da un'interfaccia.
- **Gestione selettiva dei comportamenti:** si è deciso di non utilizzare un'interfaccia con metodi di default per evitare di forzare l'implementazione di comportamenti non pertinenti a tutte le entità. Un esempio critico è il metodo **move()**: sebbene necessario per **Player**, **Invader** e **Shot**, esso non ha senso per la classe **Bunker**, che è un'entità statica. L'uso di un'interfaccia avrebbe costretto la classe **Bunker** a ereditare o implementare un metodo di movimento logicamente errato per la sua natura.

Questa architettura è ottima perché organizza il codice in modo gerarchico e manutenibile, offrendo vantaggi concreti:

- **Manutenibilità:** se si decide di cambiare il modo in cui una collisione riduce la vita, basta modificare il metodo **onHit()** in **AbstractEntity** per aggiornare istantaneamente tutto il sistema.
- **Polimorfismo:** classi come **GameLogic** possono manipolare liste generiche di **AbstractEntity**, invocando metodi comuni senza conoscerne l'effettiva sottoclasse, semplificando la gestione di collisioni e aggiornamenti.
- **Estendibilità:** l'aggiunta di nuovi tipi di nemici richiede solo l'estensione di **AbstractEntity**, ereditando automaticamente tutta l'infrastruttura di gestione vita e posizionamento già collaudata.

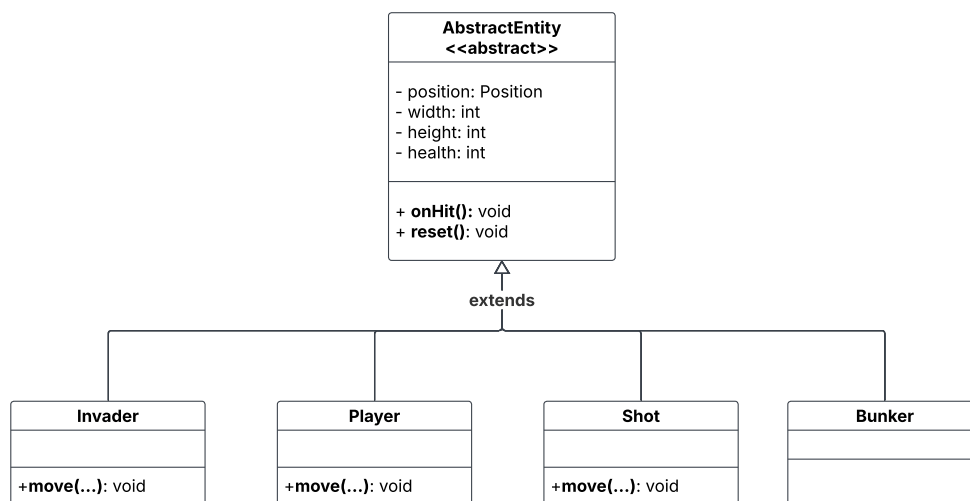


Figure 2.9: Schema UML per la gestione della gerarchia delle entità

4) Pattern Template Method: gestione dei punti tramite onHit()

Problema: Ogni entità del gioco, quando viene colpita, deve seguire una sequenza logica predefinita: subire un danno, verificare se la salute è esaurita e, in caso positivo, passare allo stato "morto". Tuttavia, solo alcune entità (come gli **Invaders**) devono restituire un punteggio specifico al termine di questa procedura. Gestire questa differenza con logiche condizionali nel Model avrebbe sporcato il codice con controlli sul tipo di entità.

Soluzione: È stato implementato il pattern **Template Method** utilizzando la classe **AbstractEntity** come "stampo" per definire la struttura dell'algoritmo di collisione:

- **Il Template Method:** All'interno di **AbstractEntity**, il metodo **onHit()** definisce i passaggi fissi: richiama **decreaseHealth()**, controlla se la salute è pari a zero per invocare **die()** e, infine, chiama il metodo **getEntityPoints()**.
- **getEntityPoints():** Nella classe astratta, questo metodo restituisce di default ZERO. Esso funge da segnaposto che le sottoclassi possono sovrascrivere.
- **L'override specifico (Invader):** La classe **Invader** implementa il proprio comportamento specifico sovrascrivendo solo **getEntityPoints()**, restituendo il punteggio corretto in base alla tipologia di alieno tramite uno **switch**.

Dal punto di vista puramente funzionale, in questo contesto il pattern offre un vantaggio limitato. Tuttavia, la scelta di utilizzare il **Template Method** è stata fatta in un'ottica di **estendibilità futura del software**. Separando nettamente la "meccanica" del colpo (fissa nella classe base) dal "valore" del premio (variabile nelle sottoclassi), il sistema risulta predisposto per l'aggiunta di nuove entità con logiche di ricompensa diverse senza il rischio di alterare o dover riscrivere la logica fondamentale di gestione della vita e della morte dell'entità.

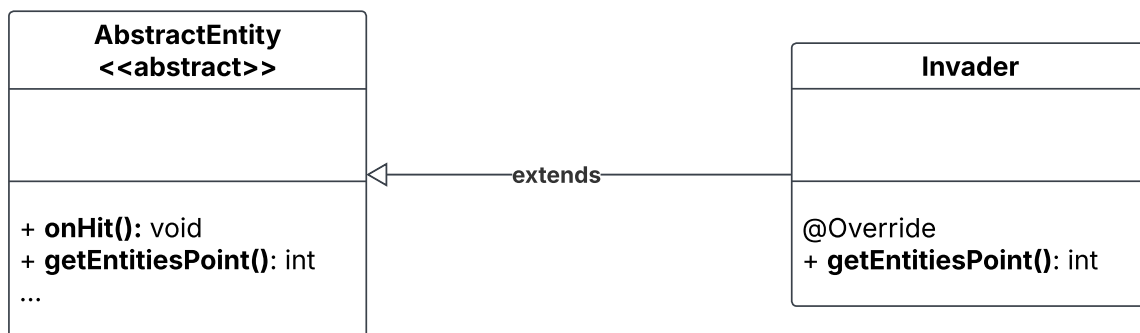


Figure 2.10: Schema UML della gestione del pattern.

2.2.3 Mario Lungu

1) Definizione centralizzata dei parametri

Problema: Durante lo sviluppo di un progetto di dimensioni abbastanza elevate, uno degli errori più frequenti è avere costanti sparse all'interno delle classi. Questo approccio rende il codice poco leggibile, poiché non è immediato capire cosa rappresenti un determinato numero incontrato nel programma. Inoltre, definire parametri di gioco (dimensioni degli invader, dimensioni del player) e costanti grafiche in modo disordinato crea confusione, viola la separazione delle responsabilità richiesta dal pattern MVC e rende il codice difficile da mantenere.

Soluzione: Per risolvere queste criticità, ho scelto di centralizzare tutte le costanti in due classi statiche dedicate, divise in base al loro utilizzo:

- **Constants:** Questa classe raccoglie le costanti che riguardano la parte logica del gioco. Contiene i valori matematici come le dimensioni della mappa, i punti vita delle entità, i punteggi e le velocità. Trovandosi nel pacchetto common, è utilizzata sia dalla logica del Model, che dalla View.
- **UIConstants:** Questa classe contiene le costanti legate alla grafica. Qui sono definiti i percorsi delle immagini (.png), i font utilizzati e le palette di colori. Questa classe si trova nel package della View, essendo accessibile solo ad essa.

Questa organizzazione porta tre vantaggi:

- **Manutenibilità:** Il codice diventa auto-esplicativo. Sostituire un numero grezzo con una costante dotata di un nome significativo rende più facile la comprensione del suo scopo.
- **Facilità di "Tuning":** Per bilanciare il gioco, è sufficiente modificare un singolo valore nella classe Constants per aggiornare l'intero sistema, senza dover cercare tra decine di classi.
- **Architettura pulita:** Il Model utilizza i dati di Constants ma non ha accesso a UIConstants. In questo modo, la logica di gioco rimane completamente indipendente dalla sua rappresentazione grafica.

Di seguito, in Figure 2.11, è riportato lo schema che visualizza la separazione delle responsabilità: il Model accede esclusivamente alle costanti logiche, mentre la View gestisce le costanti grafiche.

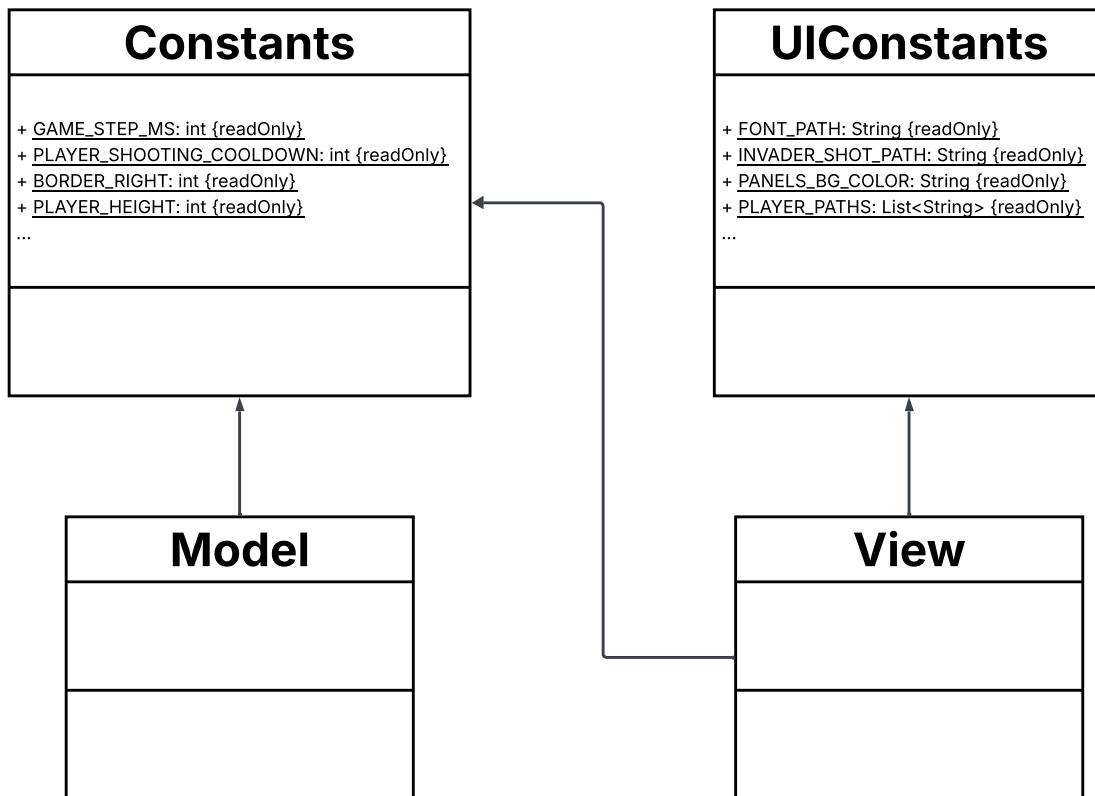


Figure 2.11: Diagramma delle dipendenze per le costanti di gioco e grafiche.

2) Rilevamento e gestione centralizzata delle collisioni

Problema: In un gioco con molti elementi che si muovono (proiettili, nemici, giocatore), è necessario capire quando due oggetti si toccano e decidere le conseguenze (es. togliere vita o eliminare un nemico). Se ogni oggetto dovesse controllare da solo chi sta toccando, si creerebbe una certa confusione: ogni classe dovrebbe conoscere i dettagli di tutte le altre, rendendo il codice "intrecciato" e molto difficile da modificare in futuro.

Soluzione: Per risolvere questo problema, la gestione delle collisioni è stata affidata interamente alla classe `GameLogic` (la classe che gestisce le regole del `Model`, come spiegato nella sezione 2.2.1 1)), aiutata da una classe di supporto chiamata `CollisionReport`.

Il meccanismo si basa sull'esecuzione sequenziale di due metodi principali ad ogni aggiornamento del gioco:

- Il metodo `checkCollisions()` scansiona tutti i proiettili presenti e controlla se colpiscono qualche altra entità. Durante la scansione applica anche le regole di validità: ignora il "fuoco amico" (es. un invader che colpisce un'altro invader) e le entità già distrutte. Genera un oggetto `CollisionReport` (anche se non c'è impatto, viene generato un `CollisionReport` vuoto), che racchiude al suo interno l'elenco completo delle entità coinvolte negli scontri (sia i proiettili andati a segno, sia i "bersagli" colpiti).
- Il metodo `handleCollisionReport()` viene invocato subito dopo e riceve in input il report appena creato. Esso scorre la lista delle entità coinvolte e applica gli effetti

concreti: diminuisce la vita, rimuove gli oggetti distrutti dal gioco e restituisce gli eventuali punti guadagnati (in caso di collisioni con degli invader).

Questa organizzazione rispetta due principi fondamentali:

- SRP: Le entità (Invader, Player) devono solo "esistere" (sapere dove sono), mentre è la logica di gioco che fa da arbitro e decide quali scontri sono validi.
- Ordine del codice: Separare il momento in cui si "rilevano" i contatti (creazione del report) dal momento in cui si "applicano" i danni rende il sistema molto più ordinato e facile da espandere.

La soluzione per la centralizzazione e risoluzione delle collisioni è esposta nel diagramma in Figure 2.12.



Figure 2.12: Schema UML dell'utilizzo della classe di supporto CollisionReport per centralizzare il rilevamento e la risoluzione delle collisioni all'interno del Model.

3) Gestione della difficoltà: Progressione dinamica e bilanciamento

Problema: Nelle prime fasi di sviluppo, i parametri fondamentali di gioco, come la velocità degli invader o la frequenza di sparo, erano definiti semplicemente come costanti fisse. Questo approccio rendeva il gioco statico (la difficoltà non aumentava mai) e rigido da bilanciare: per modificare l'esperienza di gioco era necessario intervenire direttamente sul codice sorgente cambiando i valori fissi, rendendo la manutenzione complessa e poco flessibile.

Soluzione: Per risolvere il problema è stata introdotta la classe **DifficultyManager** all'interno del **Model**. Questa classe viene utilizzata come un componente interno della **GameLogic** (che è la classe del **Model** che si occupa delle regole del gioco, come spiegato nella sezione 2.2.1 1)), rispettando il Single Responsibility Principle: tutti i calcoli matematici sulla difficoltà sono isolati in questa classe specifica. Questo permette di modificare o regolare la progressione del gioco agendo solo su questa classe, senza dover mai toccare il codice del motore di gioco principale (**GameLogic**), che rimane così più ordinato e sicuro.

Il meccanismo prevede che, ad ogni ciclo di aggiornamento del gioco, la **GameLogic** prelevi dal **DifficultyManager** i valori aggiornati per il livello corrente. Nello specifico, richiede:

- La velocità di movimento: quanto tempo deve passare tra un passo e l'altro degli invader.
- La velocità di generazione dei colpi: quanto devono aspettare i nemici prima di poter sparare di nuovo.
- Il numero di colpi da generare: quanti proiettili devono essere sparati in quel preciso momento.

In questo modo, il sistema adatta la sfida automaticamente, aumentando la difficoltà man mano che il giocatore avanza nei livelli.

La Figure 2.13 mostra la relazione di delega utilizzata per calcolare i parametri di bilanciamento del gioco.



Figure 2.13: Diagramma UML che evidenzia l'associazione per la gestione della difficoltà: il Model interroga DifficultyManager per ottenere i valori di configurazione correnti.

4) Struttura della View: suddivisione di responsabilità e modularità

Problema: La gestione dell'intera interfaccia grafica all'interno di un'unica classe avrebbe comportato la creazione di una "God Class", violando il Single Responsibility Principle e rendendo il codice più difficile da leggere e da mantenere. Inoltre, sorgerà un problema di navigazione: i pannelli secondari (come il Menu) devono poter chiedere alla finestra principale di cambiare schermata. Se i pannelli avessero un riferimento diretto alla classe concreta View, si creerebbe un accoppiamento ciclico e rigido, rendendo impossibile testare i pannelli isolatamente o riutilizzarli in contesti diversi.

Soluzione: È stata adottata un'architettura basata su pannelli specializzati, coordinati tramite un'interfaccia di navigazione dedicata (ViewActionsInterface):

- View: Agisce come coordinatore principale. Gestisce il JFrame e il layout, ma non contiene la logica specifica delle singole schermate.
- ViewActionsInterface: È un'interfaccia definita nel pacchetto API della View. Espone solo i metodi necessari alla navigazione. La classe View implementa questa interfaccia.
- Pannelli (MenuPanel, GamePanel): I pannelli non conoscono la classe View. Nel loro costruttore ricevono un riferimento di tipo ViewActionsInterface.

Questa suddivisione porta diversi vantaggi:

- SRP: Ogni pannello fa una cosa sola. Se devo modificare la grafica dei pulsanti, so che non romperò per sbaglio la grafica del gioco.
- Input separati: L'input da tastiera è abilitato esclusivamente durante la fase di gioco. Grazie alla separazione dei pannelli, si elimina il rischio che la pressione dei tasti venga erroneamente rilevata o processata mentre l'utente naviga all'interno del menu.
- Facilità di modifica: Aggiungere una nuova pagina al menu (come le "Opzioni") è facile e non richiede di toccare il codice che fa funzionare il gioco.
- Chiarezza: Il codice è più pulito: chi legge la classe GamePanel vede solo codice che disegna il gioco, senza trovarsi in mezzo righe che controllano i bottoni del menu.

L'organizzazione gerarchica dei pannelli, progettata per separare le responsabilità di interfaccia da quelle di rendering, è rappresentata in Figure 2.14.

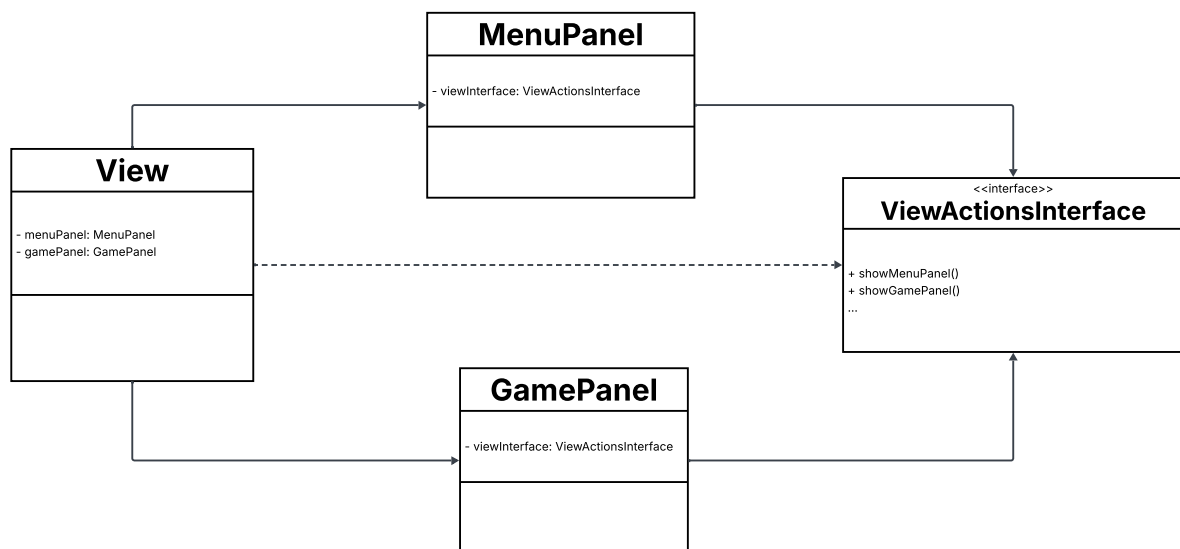


Figure 2.14: Architettura della View, dove i pannelli interagiscono con il contenitore principale esclusivamente tramite l'interfaccia ViewActionsInterface.

Chapter 3

Sviluppo

3.1 Testing automatizzato

Abbiamo realizzato una serie di test automatici utilizzando `JUnit`, con l'obiettivo di verificare il corretto funzionamento dei principali meccanismi della logica di gioco. I test sono completamente automatici, non richiedono interazione dell'utente.

Funzionalità testate

- **Gestione delle entità nel GameWorld**
Verifica che l'aggiunta di un'entità aggiorni correttamente le strutture interne e i riferimenti specifici (ad esempio il player).
- **Movimento dei proiettili**
Controllo della corretta variazione della coordinata verticale in base alla direzione del colpo.
- **Sistema di collisioni**
Verifica che una collisione tra proiettile e invader venga rilevata e che l'entità colpita perda punti vita e di conseguenza muoia.
- **Rimozione dei colpi non validi**
Controllo che i proiettili morti vengano correttamente rimossi dal GameWorld.
- **Vincoli di movimento del player**
Verifica che il player non possa superare i limiti del mondo di gioco.
- **Cambio direzione degli invader**
Controllo del corretto aggiornamento della direzione quando viene richiesto il cambio.
- **Condizioni di fine partita**
Verifica sia della vittoria degli invader (morte del player) sia della vittoria del player (tutti gli invader eliminati).
- **Gestione della difficoltà**
Verifica dell'incremento del livello tramite `DifficultyManager`.

Abbiamo scelto di concentrare i test esclusivamente sulla **logica di gioco**, in quanto riteniamo sia l'aspetto critico del software.

I test sono volutamente essenziali ma coprono i punti critici del sistema: gestione dello

stato, collisioni, movimento, vincoli di gioco e progressione della difficoltà. In questo modo si riduce il rischio di regressioni a seguito di modifiche future alla logica del gioco.

3.2 Note di sviluppo

3.2.1 Girolamo Ronzoni

Utilizzo di Stream e lambda expressions

Le ho utilizzate diffusamente, qui di seguito riporto solo alcuni esempi.

- [Permalink 1](#)
- [Permalink 2](#)
- [Permalink 3](#)

Utilizzo di classi anonime

Anche in questo caso riporto solo alcuni esempi.

- [Permalink 1](#)
- [Permalink 2](#)

Utilizzo di variable arguments

- [Permalink](#)

Utilizzo di `synchronized` per operazioni di lettura e scrittura su liste condivise, e utilizzo di `Collections.synchronizedList` per rendere thread-safe le strutture dati principali

- [Permalink 1](#)
- [Permalink 2](#)

Utilizzo di `AtomicInteger` per la gestione thread-safe di alcune variabili intere condivise, per prevenire corse critiche

- [Permalink 1](#)
- [Permalink 2](#)

Utilizzo di blocco di inizializzazione statica per il caricamento e la registrazione di risorse grafiche costanti

- [Permalink](#)

3.2.2 Leonardo Lioi

Utilizzo della libreria `javax.sound.sampled` per la gestione audio

Permalink: <https://github.com/girolamooo/OOP25-SCAT/blob/f160ce4dbff464019f340caaaa35d1src/main/java/it/unibo/scat/view/util/AudioManager.java#L16>

Utilizzo di Method Reference e Comparator Chaining

Permalink: <https://github.com/girolamooo/OOP25-SCAT/blob/f160ce4dbff464019f340caaaa35d1src/main/java/it/unibo/scat/model/leaderboard/Leaderboard.java#L102>

Utilizzo di Method Reference

Permalink: <https://github.com/girolamooo/OOP25-SCAT/blame/f160ce4dbff464019f340caaaa35d1src/main/java/it/unibo/scat/view/game/GameOverPanel.java#L56>

Implementazione del volume del sistema

Permalink: <https://github.com/girolamooo/OOP25-SCAT/blame/bbb3d0009ad0640fdf88c83845f62src/main/java/it/unibo/scat/view/util/AudioManager.java#L41>

Indipendenza dall'ambiente per l'archiviazione dei dati

Permalink: <https://github.com/girolamooo/OOP25-SCAT/blob/16302b6c353b1671fdff934f6a047b1src/main/java/it/unibo/scat/model/leaderboard/Leaderboard.java#L32>

3.2.3 Mario Lungu

Utilizzo di Stream API

Permalink:

- <https://github.com/girolamooo/OOP25-SCAT/blob/a0efd0162f176c1e699d21aeba14748fd7f1/src/main/java/it/unibo/scat/model/game/GameLogic.java#L302>

Utilizzo di Method Reference

Permalink:

- <https://github.com/girolamooo/OOP25-SCAT/blob/e7f683a29595654eda30342d8ca6f816d4e/src/main/java/it/unibo/scat/view/game/PausePanel.java#L54>
- <https://github.com/girolamooo/OOP25-SCAT/blob/e7f683a29595654eda30342d8ca6f816d4e/src/main/java/it/unibo/scat/view/game/PausePanel.java#L56>
- <https://github.com/girolamooo/OOP25-SCAT/blob/e7f683a29595654eda30342d8ca6f816d4e/src/main/java/it/unibo/scat/view/game/PausePanel.java#L58>

Rendering grafico avanzato (Graphics2D)

Permalink:

- <https://github.com/girolamooo/OOP25-SCAT/blob/e7f683a29595654eda30342d8ca6f816d4e/src/main/java/it/unibo/scat/view/game/PausePanel.java#L129>

Chapter 4

Commenti finali

4.1 Autovalutazione e lavori futuri

4.1.1 Girolamo Ronzoni

Il mio compito principale era quello di definire la struttura del progetto e la comunicazione tra le varie componenti; posso considerarmi, in fin dei conti, soddisfatto del risultato.

Ho progettato un'architettura solida utilizzando MVC, pattern Observer e separazione delle responsabilità, sulla quale i miei compagni hanno potuto implementare efficacemente le funzionalità del gioco.

I tempi ristretti per lo sviluppo, e l'uscita di un membro del gruppo dopo 2 settimane dall'inizio del progetto sicuramente hanno inciso negativamente sul risultato finale, ma nonostante questo siamo riusciti a portare a termine il progetto ottenendo a mio avviso un ottimo risultato. Alcuni aspetti aggiuntivi che mi piacerebbe implementare in futuro, sono l'aggiunta dei loot (droppati ad intervalli random che potenziano la navicella del player) e di uno o più boss finali. L'architettura del progetto è solida ed estendibile pertanto queste aggiunte saranno molto facili da implementare.

Nel complesso ritengo il progetto molto formativo. Mi ha permesso di prendere dimestichezza con git, di imparare a gestire una repository, di apprendere alcuni aspetti di progettazione del software, di implementare pattern che non avevo mai utilizzato (come observer e factory) e infine di imparare a lavorare in squadra.

4.1.2 Mario Lungu

Il mio ruolo si è focalizzato principalmente sull'implementazione della logica di interazione fisica tra le entità e sulla gestione dell'input utente, oltre alla realizzazione di componenti ausiliari dell'interfaccia grafica.

Considero la parte matematica del codice il mio principale punto di forza: implementare l'algoritmo di rilevamento delle intersezioni e gestirne le conseguenze è stata la sfida più stimolante e, a mio avviso, quella meglio riuscita. Ho trovato grande soddisfazione nel vedere la logica teorica tradursi in un gameplay reattivo. Inoltre, l'esperienza mi ha permesso di comprendere a fondo l'importanza di separare la logica di controllo (input) dalla rappresentazione grafica.

Tuttavia, riconosco come punto di debolezza un utilizzo non sempre costante delle funzionalità più moderne del linguaggio Java. Rileggendo il codice, mi rendo conto che alcune sezioni avrebbero potuto beneficiare di una maggiore adozione di Stream API e Lambda Expressions per rendere la sintassi più concisa, e avrei potuto esplorare ulteriori librerie esterne per arricchire le funzionalità.

In un'ottica di un eventuale sviluppo futuro, mi piacerebbe rifattorizzare il sistema di collisioni per introdurre una rilevazione più avanzata e ampliare il sistema di input per supportare controller esterni, rendendo il progetto un vero e proprio arcade moderno.

4.1.3 Leonardo Lioi

All'interno del progetto, il mio contributo principale è stato lo sviluppo del nucleo centrale del Model, dove ho progettato il sistema che gestisce tutte le entità di gioco e ho realizzato la parte per il salvataggio permanente dei punteggi. In particolare, ho implementato la gerarchia delle classi tramite la classe astratta `AbstractEntity`, garantendo che attributi comuni come posizione e salute fossero centralizzati per evitare duplicazioni di codice.

Mi sono inoltre occupato della Leaderboard, definendo l'interfaccia e la modellazione dei dati tramite `GameRecord`, e del comparto Audio, gestendo la riproduzione di musica ed effetti sonori. Tra i punti di forza del mio lavoro figura la scelta architetturale della gerarchia, che permette di aggiungere nuovi tipi di nemici semplicemente estendendo la classe base, e la netta separazione delle responsabilità tra logica di gioco e archiviazione dei dati. Di contro, ho riscontrato alcune difficoltà iniziali nella gestione dei flussi audio paralleli con la libreria `javax.sound.sampled`.

Per quanto riguarda gli sviluppi futuri, il sistema è già strutturalmente predisposto per l'introduzione di meccaniche di gioco più profonde, come il rilascio di loot o potenziamenti (power-up) da parte degli invasori alla loro distruzione, sfruttando la gerarchia delle entità già esistente. Inoltre, l'architettura attuale permetterebbe di evolvere la classifica locale verso una leaderboard globale online, sincronizzando i record su un server remoto senza dover stravolgere il codice esistente grazie al forte disaccoppiamento tra i moduli.

4.2 Difficoltà incontrate e commenti per i docenti

L'unico punto sul quale ci sentiamo di dare un suggerimento è il seguente:

il plugin di Q.A. che utilizza SpotBugs, PMD e CheckStyle è risultato a nostro avviso un pò troppo "aggressivo", rallentando in certi casi lo sviluppo. Soprattutto durante la fase iniziale del progetto, nella quale abbiamo definito tutti i campi (che temporaneamente erano inutilizzati) e i metodi delle interfacce (implementati inizialmente con metodi vuoti, e inutilizzati), abbiamo passato molto tempo a sopprimere warning, questo ha inciso leggermente sul processo di sviluppo.

Il nostro suggerimento per il futuro è quello di rendere lievemente meno sensibili i tool di analisi statica.

Appendix A

Esercitazioni di laboratorio