Gioele Bucci
gioele.bucci@studio.unibo.it
Edoardo Ceresi
edoardo.ceresi@studio.unibo.it
Nicolò Morini
nicolo.morini2@studio.unibo.it
Simone Mosconi
simone.mosconi2@studio.unibo.it

18 febbraio 2024

Indice

Capitolo 1

Analisi

1.1 Requisiti

Il software mira allo sviluppo del videogioco "TBOOOP", il quale sarà un demake di "The Binding Of Isaac". Per demake si intende che TBOOOP prenderà forte ispirazione dal gioco originale, disponendo tuttavia di meno funzionalità. Seguirà il modello dei videogiochi stile "roguelike", il cui gameplay è caratterizzato dall'esplorazione di una mappa di gioco generata casualmente.

Requisiti funzionali

- Il giocatore si muoverà all'interno di piani, ognuno dei quali corrisponderà ad un livello. Ogni piano sarà costituito da un insieme di più stanze. La generazione di ogni piano (e dei suoi contenuti) dovrà essere casuale.
- Ogni stanza sarà distinta in base al suo contenuto. In particolare dovranno essere presenti:
  - Stanze dei nemici, i quali attaccheranno il giocatore;
  - Stanze del negozio, contenenti una serie di oggetti acquistabili con delle monete (ottenibili durante l'esplorazione);
  - Stanze dell'oggetto, contenenti un singolo oggetto raccoglibile senza costi;
  - Stanze della botola, che porteranno il giocatore al piano successivo.
- Per poter accedere ad una stanza dell'oggetto o ad una stanza del negozio il giocatore dovrà disporre di una chiave. Le chiavi sono consumabili (monouso) ed ottenibili tramite l'esplorazione.
- Il giocatore dovrà essere in grado di muoversi liberamente all'interno del piano di gioco, sparare ai nemici per eliminarli e raccogliere i vari oggetti.
- Lo scopo del giocatore è di avanzare nei vari piani ed esplorarli: la partita si concluderà con l'eliminazione del giocatore da parte dei nemici.

Requisiti non funzionali

- Il gioco dovrà avere una frequenza di aggiornamento tale da risultare fluido e dovrà essere responsivo agli input dall'utente.
- La finestra di gioco dovrà essere ridimensionabile mediante dei tasti predefiniti, mantenendo un aspect ratio (rapporto tra larghezza e altezza) costante.

1.2 Analisi e modello del dominio

Per ogni livello verrà generato un piano e le relative stanze in esso contenute, che verranno collegate tra loro tramite delle porte. Il numero di stanze per ogni piano e il numero di nemici presenti nelle varie stanze aumenteranno con l'aumentare del livello. Al termine della generazione di un piano il giocatore verrà collocato nella relativa stanza iniziale. All'ingresso da parte del giocatore in una stanza contenente dei nemici le porte di tale stanza si chiuderanno, impedendo al giocatore di uscire. Le porte si riapriranno solamente al completamento della stanza, che consiste nell'eliminazione di tutti i nemici in essa presenti. In ogni piano, oltre alle stanze dei nemici, sono presenti anche due stanze speciali: una stanza dell'oggetto e una stanza del negozio. Per potervi accedere il giocatore avrà bisogno di una chiave.

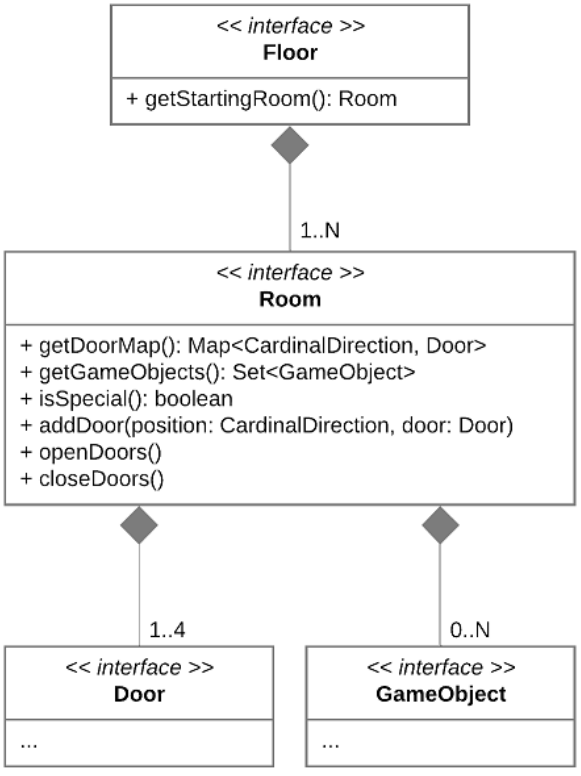


Figura 1.1: La struttura di un piano.

Le stanze contengono al loro interno un insieme eterogeneo di entità di gioco, ognuna caratterizzata da una posizione, uno stato aggiornabile e la capacità di rispondere ad eventi di collisione.
I nemici sono suddivisi per tipologia, ciascuno con un proprio pattern di movimento e di attacco (ad esempio alcuni infliggeranno danni a contatto, altri a distanza).

Sono presenti anche delle entità di gioco statiche (inamovibili), suddivise in due sottocategorie:

- Oggetti (item), che appaiono nei negozi o nelle stanze dell'oggetto e incrementano le statistiche del giocatore.
- Consumabili (pickup), come chiavi e monete. Un singolo consumabile apparirà al completamento di ogni stanza contenente dei nemici.

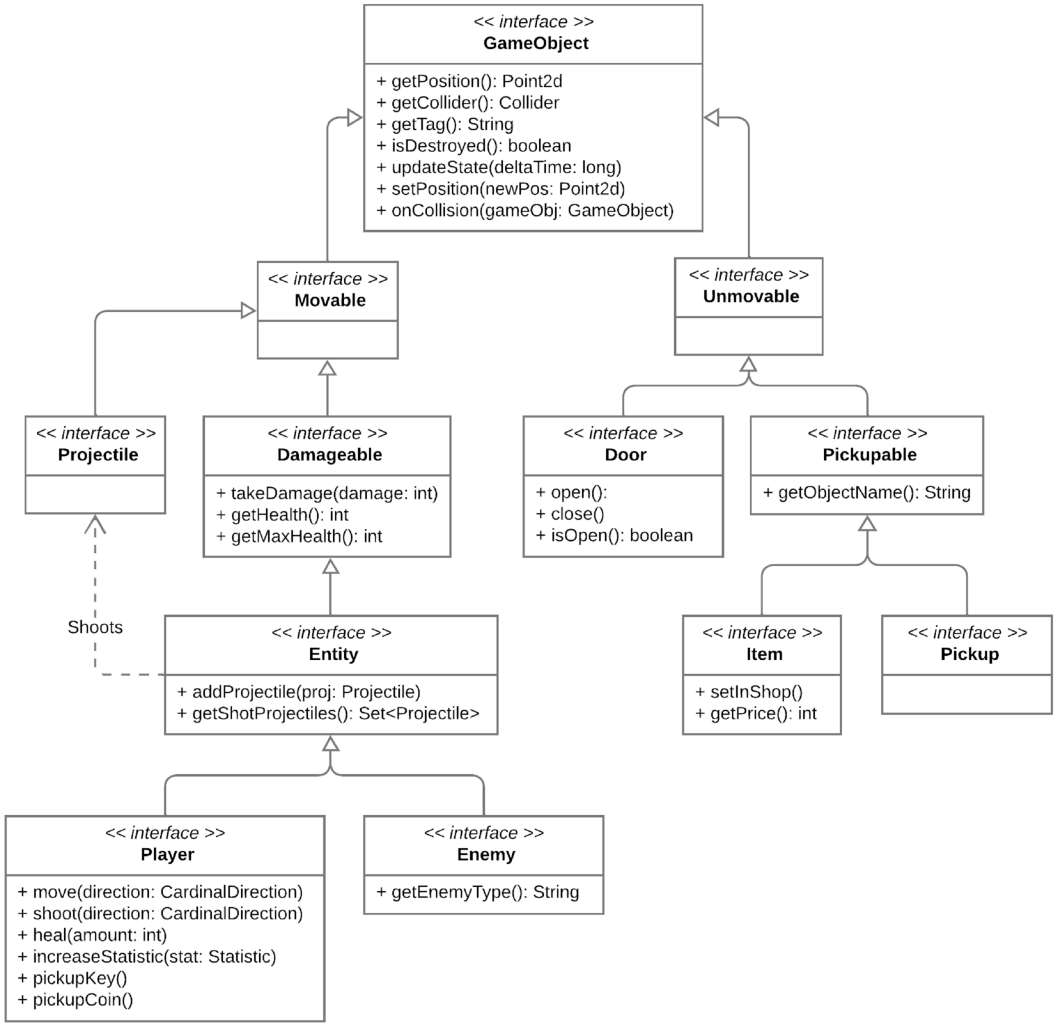


Figura 1.2: Gerarchia delle varie entità di gioco.

Capitolo 2

Design

2.1 Architettura

L'architettura dell'applicazione segue il pattern architetturale MVC. L'entry point dell'applicazione è la View, la quale si occupa poi di istanziare il Controller. Il Controller a sua volta inizializza il Model, salvandone gli elementi all'interno di un "World", ovvero un'unità di supporto al Controller contenente le varie entità di gioco. Le informazioni sul piano corrente e le relative stanze sono invece contenute all'interno di un "FloorManager", anch'esso accessibile dal Controller.

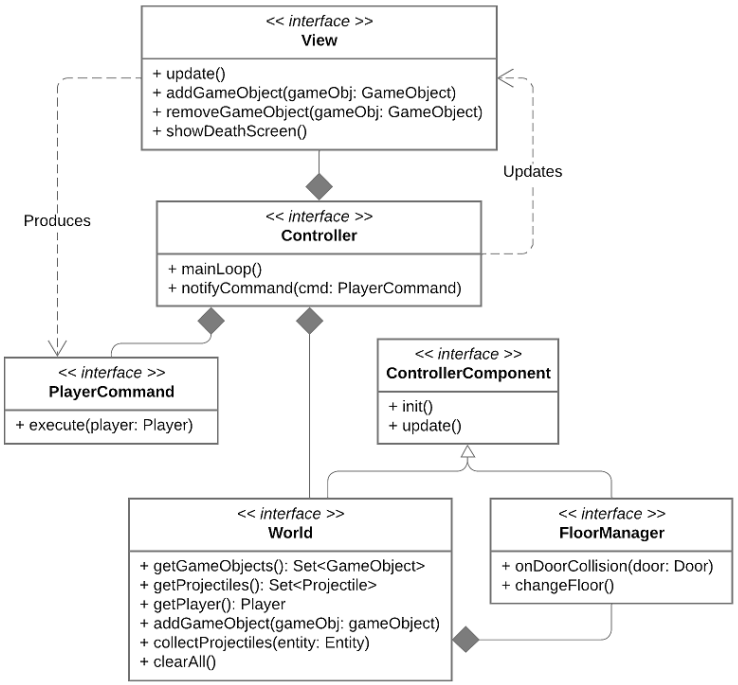


Figura 2.1: Architettura dell'applicazione (MVC)

La View delega la gestione degli input ad un "InputManager", che notifica al controller il comando richiesto. Il Controller ad ogni frame di gioco, oltre ad aggiornare il Model, si occupa anche dell'elaborazione di un comando ricevuto, se presente. Al termine di questa fase (indipendentemente dall'elaborazione o meno di un comando) la View viene aggiornata in base al nuovo stato del Model.
L'architettura di TBOOOP è stata progettata in una maniera tale da rendere possibile la sostituzione in blocco della View, senza bisogno di dover modificare

né il Controller né il Model. Ciò è reso possibile dalla capacità della View di aggiornarsi in un contesto isolato rispetto al resto dell'applicazione, non esponendo i propri dettagli implementativi.

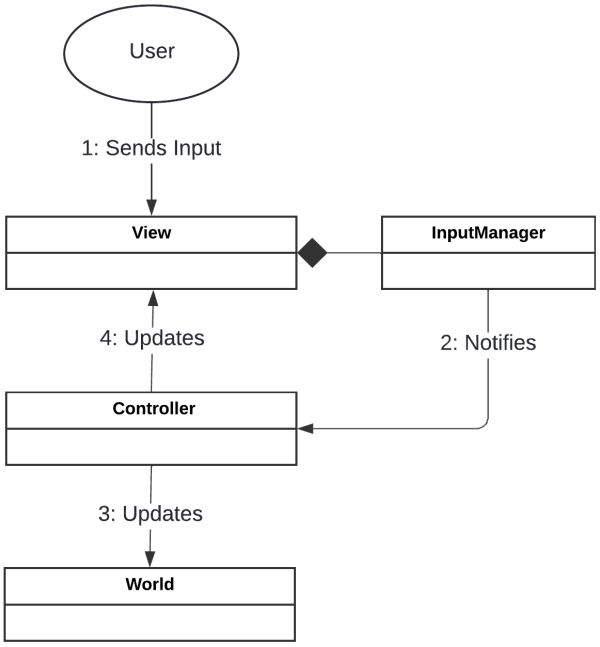


Figura 2.2: Scambio di informazioni tra i componenti dell'applicazione

2.2 Design dettagliato

2.2.1 Gioele Bucci

Il mio ruolo all'interno del gruppo riguardava principalmente la creazione della mappa di gioco. Ogni livello (o piano) è composto da un insieme di stanze connesse tramite delle porte. La generazione dei piani è un elemento distintivo in "The Binding of Isaac": per questo motivo è stato deciso che il generatore di "TBOOOP" avrebbe dovuto replicare fedelmente l'originale. Questa decisione implica che ogni piano debba rispettare una serie di restrizioni, principalmente legate alla disposizione delle stanze. Ogni piano deve includere esattamente due stanze speciali: una stanza dell'oggetto e una stanza del negozio, che vanno posizionate in modo tale da essere adiacenti ad una sola stanza.

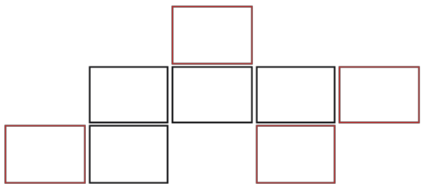


Figura 2.3: Eempio di layout di un piano. Una stanza speciale può essere generata solamente in corrispondenza di una delle stanze evidenziate in rosso.

Ogni piano deve anche possedere una stanza iniziale e una stanza finale, che consente al giocatore di accedere al livello successivo. La grandezza minima di un piano è perciò di 4 stanze.

Generazione del piano

Problema Creare un generatore di piani che rispetti i requisiti sopra elencati e che dia la possibilità di creare livelli dalla difficoltà variabile.

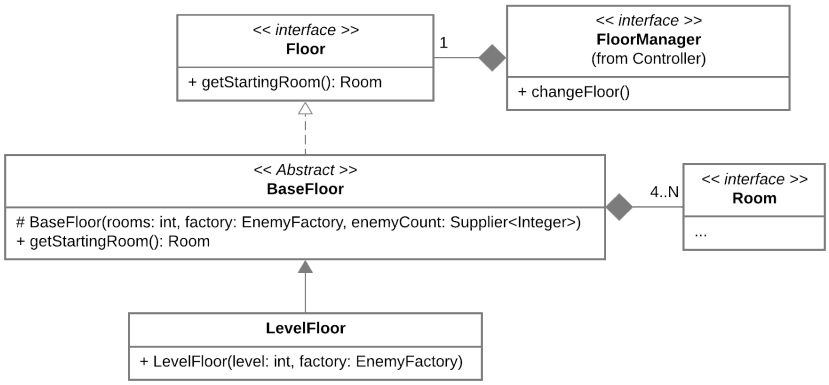


Figura 2.4: La classe astratta BaseFloor contiene la logica per la creazione di un piano generico. La sottoclasse LevelFloor ne specifica alcuni criteri di generazione.

Soluzione Ho racchiuso l'implementazione legata alla generazione del piano (effettuata tramite un ibrido di Random Walk), all'interno di una classe astratta BaseFloor. Il costruttore di questa classe consente di modificare, per ogni piano,

il numero di stanze presenti e il numero di nemici per stanza. L'utilizzatore FloorManager accede al layout del piano attraverso l'interfaccia funzionale Floor. Essa fornisce un'astrazione del piano ad una sorta di grafo, esplorabile a partire da un nodo di partenza che corrisponde alla stanza iniziale (i dettagli sulle stanze e la navigazione tra esse verranno descritti in seguito).
Nonostante l'unica sottoclasse di BaseFloor presente sia LevelFloor, che crea un piano dalla difficoltà proporzionale al livello scelto, grazie all'approccio utilizzato è possibile ottenere facilmente ulteriori varianti di un piano creando nuove sottoclassi di BaseFloor.

Generazione delle stanze

Problema Per poter creare un piano il generatore deve poter istanziare i vari tipi di stanze. Il sistema legato alla loro creazione deve essere facilmente utilizzabile ed estendibile, in previsione dell'aggiunta di ulteriori varianti.

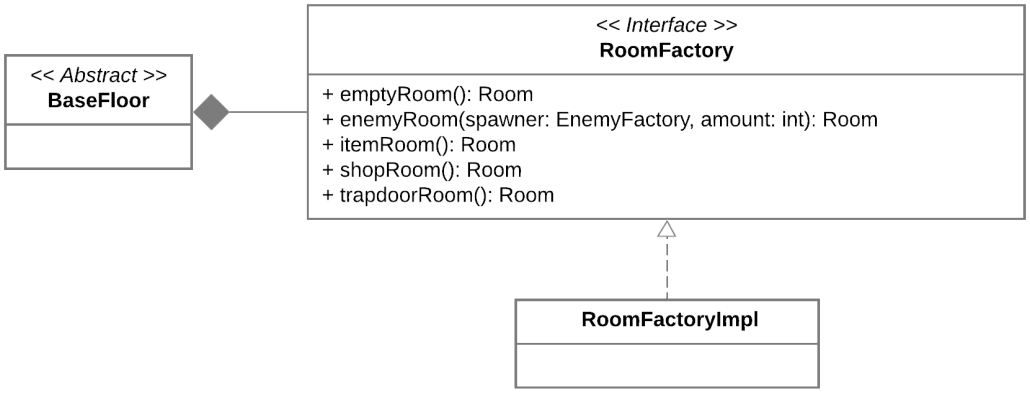
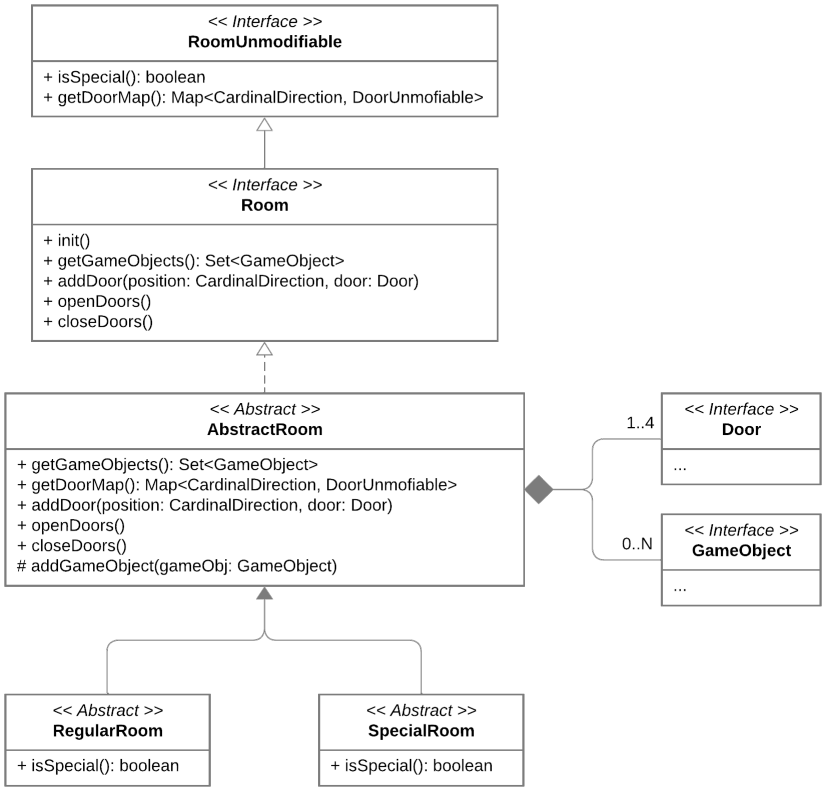


Figura 2.5: Rappresentazione UML del pattern Abstract Factory per la creazione dei vari tipi di stanza.

Soluzione Le stanze vengono create utilizzando una Abstract Factory. Tutti i metodi ritornano un'istanza di Room e non richiedono parametri, rendendo la factory facile da utilizzare, con l'eccezione del metodo enemyRoom(), che prende in ingresso il numero di nemici da posizionare all'interno della stanza e un relativo oggetto responsabile della loro creazione.

Gerarchia delle stanze

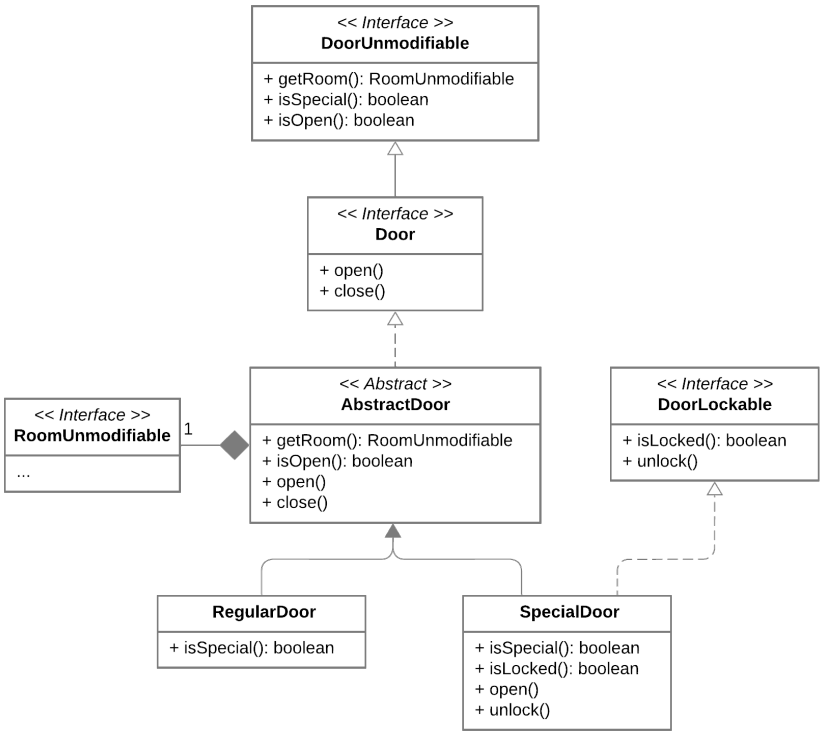
Problema Strutturare la gerarchia delle stanze in un'ottica di estendibilità, minimizzando la quantità di modifiche necessarie per poter creare una specifica tipologia di stanza.



Soluzione Ho modellato il concetto di stanza facendo uso di interfacce e classi astratte, rispettando il principio DRY. I metodi comuni ad ogni tipologia di stanza sono implementati all'interno della classe astratta AbstractRoom. Le sue due rispettive sottoclassi, anch'esse astratte, si occupano invece della distinzione tra stanza normale e speciale. Quest'ultima suddivisione è necessaria, sia al fine di posizionare correttamente la stanza all'interno di un piano (già discusso in precedenza), che per determinare il tipo di porta da associare a tale stanza (i dettagli sulle porte verranno trattati in seguito). Grazie a questo approccio, una RoomFactory può creare le varie implementazioni di ogni stanza a partire da RegularRoom o da AbstractRoom, facendo override solamente del metodo init(), che viene utilizzato per determinare il contenuto iniziale della stanza.

Generazione delle porte

Problema Collegare tutte le stanze presenti in un piano tramite delle porte. Ciascuna porta deve poter essere aperta o chiusa. In aggiunta, se una porta conduce ad una stanza speciale deve essere bloccata da un "lucchetto", che ne impedisce l'apertura fino a che non viene rimosso utilizzando una chiave.



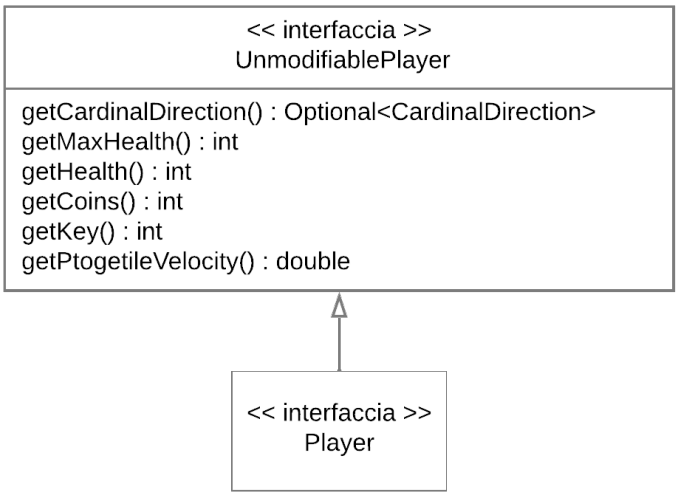
Soluzione Similmente a quanto fatto per le stanze, ho posto la logica comune ad entrambe le tipologie di porte in una classe astratta, rispettando DRY. Ad ogni porta viene anche associata la rispettiva stanza alla quale essa conduce. Questo consente di "visitare" un piano mantenendo un'unica istanza di Room (a partire dalla stanza iniziale) e ottenendo accesso alla stanza successiva tramite il riferimento fornito da una sua porta.

2.2.2 Edoardo Ceresi

Organizzazione del Player

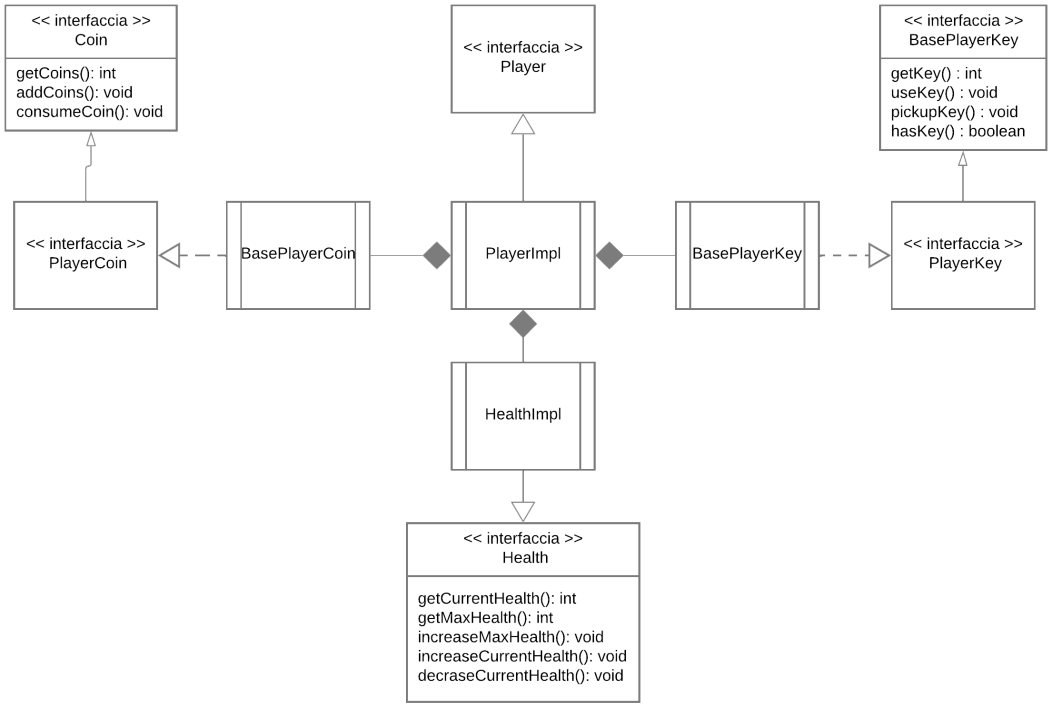
Problema Bisogno di far sapere le informazioni del Player alle altre classi senza però dare la possibilità di modificarle.

Soluzione Creazione di una classe per salvaguardare la classe Player, facendo vedere solamente i getter. È stata creata quest'interfaccia per aiutare l'implementazione della view.



Problema Gestione dei cuori, delle monete e chiavi del player in modo più semplice ed estendibile in futuro.

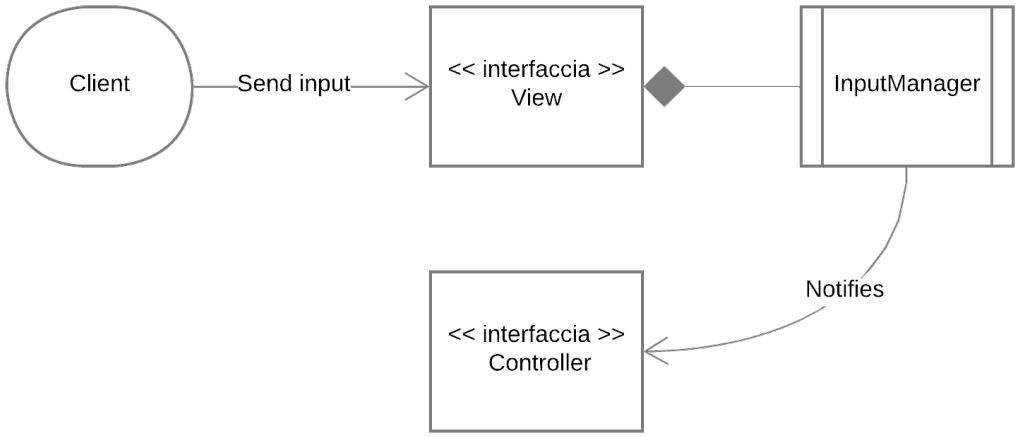
Soluzione la gestione dei cuori, delle monete e delle chiavi del giocatore l'ho affrontata utilizzando interfacce e implementazioni separate,rendendo più facile l'estensione in futuro, ho creato tre classi che poi verranno incapsulate nella classe Player per facilitare la gestione.



Gestione dell'input da tastiera

Problema Gestione dei comandi senza creare dipendenze tra model e view.

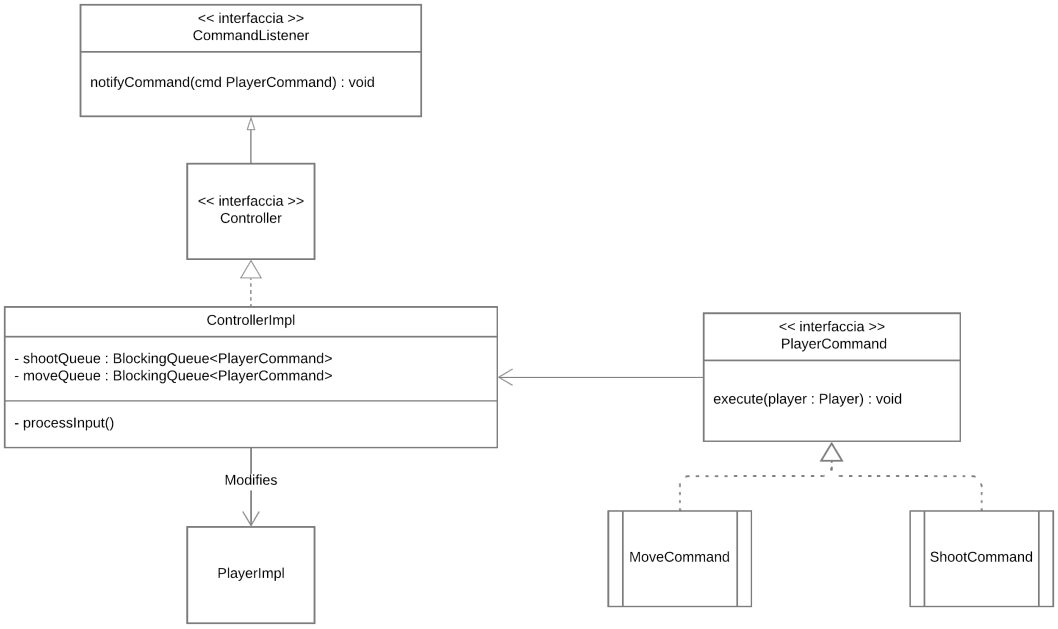
Soluzione Utilizzo del pattern di progettazione Observer.



La view riceve in input diversi Keybinds e in base al tasto premuto possono verificarsi vari eventi. L'InputManager è responsabile di notificare il controller del tasto premuto. In seguito spiegherò come il controller gestisce questi eventi. La view è la classe observable mentre l'InputManager è la classeObserver. Ho scelto di utilizzare questo pattern per non infrangere DRY e per migliorare l'estensibilità, consentendo di aggiungere in futuro molte più funzionalità.

Problema Gestione dei comandi presi dall'InputManager.

Soluzione Uso del pattern Command.



Quando l'InputManager comunica al controller tramite il metodo notifyCommand, il controller aggiunge alla relativa coda (moveQueue o shootQueue). Ad ogni frame, viene chiamato il metodo processInput che, semplicemente, chiama il metodo execute dei PlayerCommand, successivamente, tali metodi verranno eseguiti. L'invoker di questo pattern è il Controller, mentre il receiver è il Player. Ho scelto di utilizzare Command per non infrangere DRY, inoltre , utilizzandolo, possiamo aggiungere facilmente molti più comandi in futuro, rendendo il sistema estensibile. Un altro vantaggio è che viene creata una grande sinergia con il pattern Observer.

2.2.3 Nicolò Morini

Item e Pickup Il mio ruolo all'interno del progetto è stato quello di implementare gli "item" e i "pickup". Item e pickup sono entità di gioco modellate come GameObject di tipo unmovable. Entrambi estendono in maniera indiretta una classe astratta "GameObjectAbs" attraverso la classe "UnmovableAbs" ereditando l'implementazione comune a tutti i "GameObject".

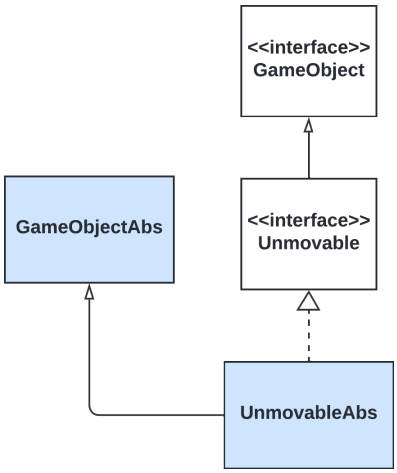


Figura 1: Relazione tra oggetti Unmovable e GameObject

Problema Il primo problema che è sorto riguardo alla gestione di questi due elementi di gioco è stato cercare di modellare item e pickup in maniera separata, andando però ad effettuare un'astrazione generale che ne contenesse le caratteristiche in comune.

Soluzione Questa parte è stata gestita tramite un'interfaccia madre "Pickupable", contenente un solo metodo necessario a riconoscere nel dettaglio il nome di un item/pickup. Successivamente ho creato una classe di astrazione che contiene una serie di metodi la cui implementazione è la stessa per tutti gli oggetti di tipo "pickupable" all'interno del gioco.

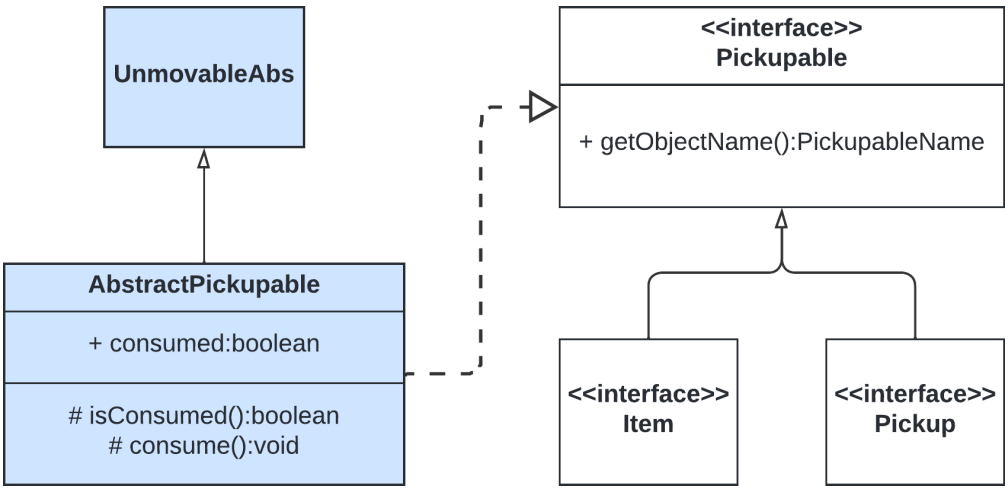


Figura 2: Suddivisione tra Item e Pickup

Problema considerando la "sottocategoria" degli oggetti del tipo "pickup", istanziare classe per classe i vari tipi di oggetti risultava macchinoso e poco propenso alla facile manutenzione del codice.

Soluzione ogni tipo di "pickup" viene istanziato grazie all'utilizzo del pattern di progettazione Abstract Factory, con il quale un'interfaccia "PickupFactory" contiene un metodo per ogni pickup istanziabile, implementati da una classe apposita che ne ritorna l'istanza con i relativi parametri passati al costruttore. Non è presente nessuna particolare astrazione o decorator in quanto nel caso dei pickup non ho pensato ricorresse nessuna implementazione condivisa al di fuori di quella generale per i GameObject di tipo "unmovable", ereditata da tutti i pickup utilizzando un'apposita classe astratta.

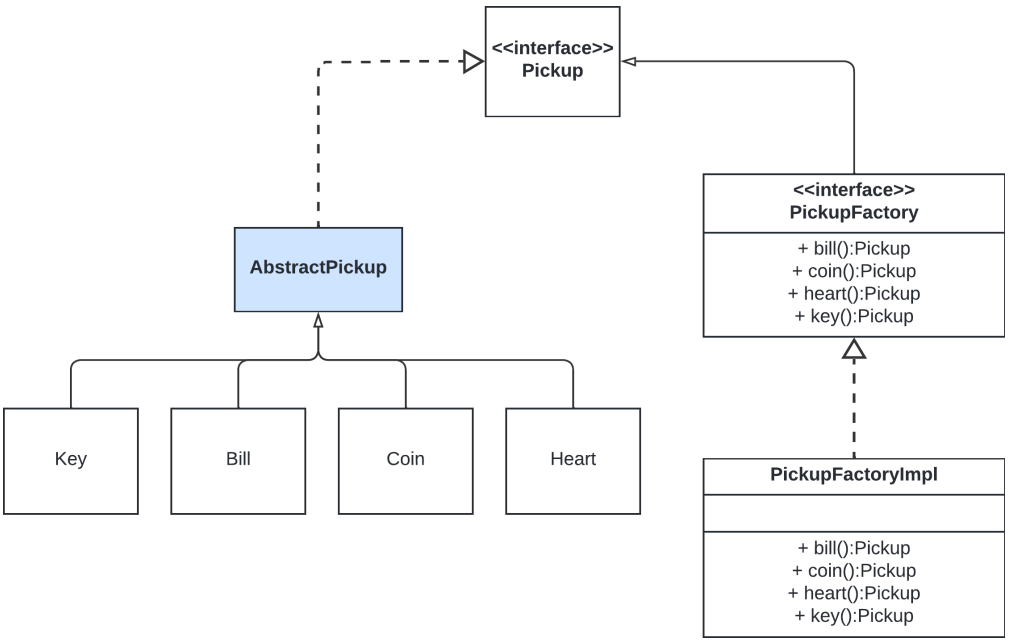


Figura 3: Scelta progettuale per la gestione dei "pickup"

Problema oltre alla ricorsione di una struttura simile a quella dei "pickup", in questo caso il problema è sorto anche nella presenza di una grande quantità di codice duplicato e di una scarsa propensione all'estendibilità e al riuso, nel caso si fossero implementati metodi comuni in ogni sottoclasse.

Soluzione come nel caso dei "pickup", ho optato per la scelta di un pattern di progettazione di tipo Abstract Factory, tale per cui è presente una interfaccia "ItemFactory" che contiene l'intestazione per ogni metodo che andrà a istanziare un nuovo oggetto per ogni tipo di "item" implementato. Riguardo alla duplicazione del codice, ho pensato ad una soluzione che utilizzasse un astrazione (classe astratta Item Abstract) che fungesse da decoratore per ogni Item, dal quale ne estendono il contenuto. In questo modo la soluzione progettuale risulta il più possibile estendibile e riutilizzabile, si possono infatti creare numerose sottoclassi di oggetti diversi, estendo quella astrazione ed implementando solo il comportamento specifico dell'item.

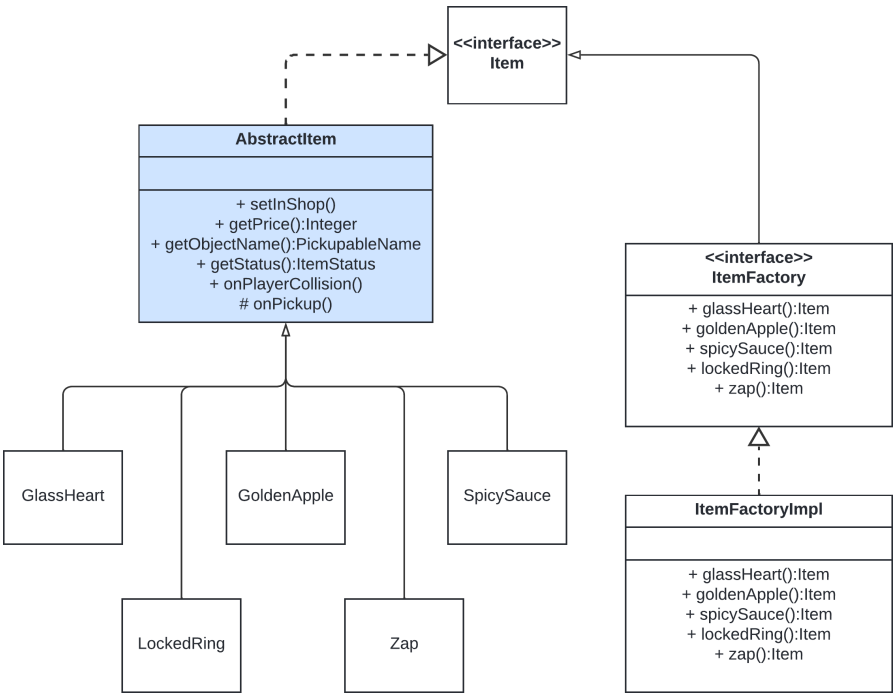


Figura 4: Scelta progettuale per la gestione degli “item”

2.2.4 Simone Mosconi

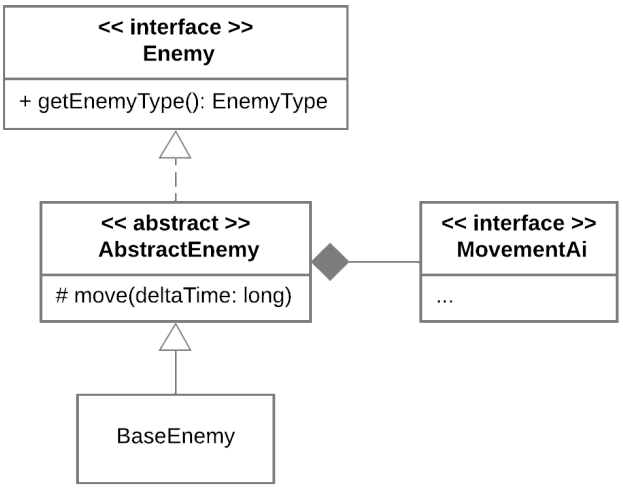
Struttura base nemici

Problema I nemici presenti in gioco possono apparire molto diversi tra loro, tuttavia ogni nemico presenta la stessa struttura di base: qualsiasi nemico infatti possiede, ad esempio:

- statistiche indispensabili come salute e velocità;
- una posizione nella stanza;
- un algoritmo di movimento;
- una hitbox;
- un tag che ne identifica la tipologia.

Soluzione E' stato deciso di fornire un'unica implementazione concreta di nemico (BaseEnemy), la quale estendendo AbstractEnemy contiene già tutti gli elementi essenziali che compongono la struttura base di un nemico, il quale potrà poi essere arricchito con comportamenti aggiuntivi.
La scelta di questa soluzione è stata fatta per permettere:

- estendibilità: a partire da questa struttura base è possibile creare nemici diversi;
- riuso: ogni nemico ha la stessa struttura base ma senza duplicare codice;



Movimento nemici

Problema Ogni nemico si muove nella mappa grazie ad un algoritmo di movimento, il quale potrebbe essere:

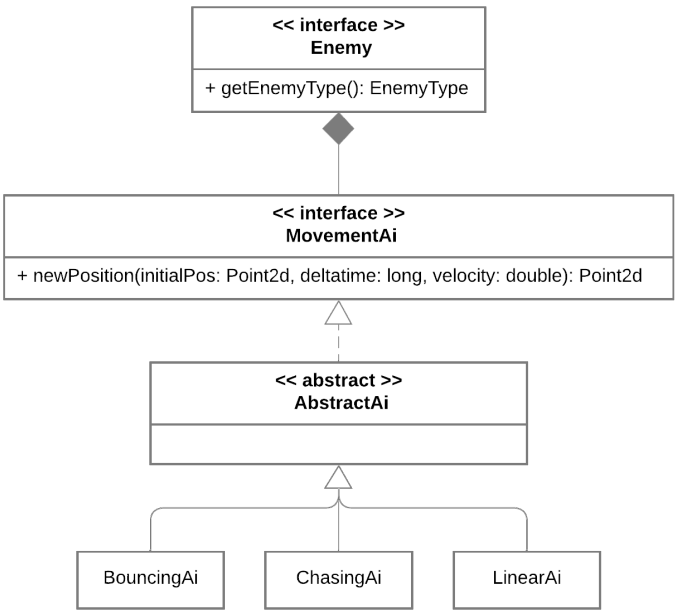
- uno stesso algoritmo comune a più tipi di nemici;
- intercambiabile nello stesso tipo di nemico con algoritmi diversi;

Soluzione E' stato deciso di delegare la gestione del movimento usando il pattern Strategy, dove delle classi esterne accomunate dall'interfaccia "MovementAi" (lo strategy in questione), si occupano di calcolare la posizione in cui il nemico dovrebbe spostarsi durante il gioco. Ogni nemico si comporrà quindi di una classe che implementa MovementAi.
Tra le implementazioni di MovementAi realizzate vi sono:

- BouncingAi: spostamento lineare con rimbalzamento sui muri della stanza;
- LinearAi: spostamento su un asse orizzontale o verticale;
- ChasingAi: inseguimento del player.

La scelta di questa soluzione è stata fatta per permettere:

- di evitare dipendenze con classi concrete;
- riuso: grazie all'interfaccia un algoritmo è assegnabile a qualsiasi nemico;
- flessibilità: per cambiare il modo in cui un nemico si muove è sufficiente assegnargli una diversa implementazione di MovementAi;
- divisione delle responsabilità: viene delegato a classi specializzate un compito che non dovrebbe essere svolto necessariamente dal nemico;
- estendibilità: è possibile creare e applicare nuovi algoritmi di movimento senza modificare i nemici.



Comportamento nemici

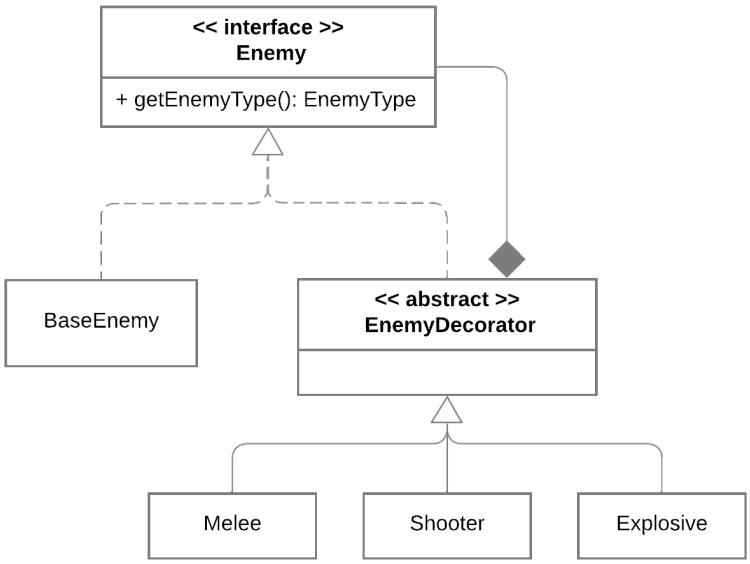
Problema Ogni nemico può avere una quantità variabile di comportamenti che lo caratterizzano, ad esempio, nel modo in cui attacca il giocatore. Inoltre alcuni nemici potrebbero condividere alcuni tipi di comportamento. Questo comporta che con l'aumentare della quantità di comportamenti assegnabili si possano creare svariate configurazioni di nemici.

Soluzione E' stato deciso di usare il pattern Decorator per gestire l'assegnamento di comportamenti ai nemici, facendo in modo che ogni comportamento aggiuntivo corrisponda ad una decorazione. In questo caso la classe astratta che funge da Decorator è "EnemyDecorator", il quale implementa l'interfaccia Enemy e al tempo stesso si compone di un Enemy, questo permette di aggiungere una quantità a piacere di decorazioni e con un ordine variabile. La maggior parte dei metodi vengono delegati all'istanza interna di Enemy, tuttavia ogni EnemyDecorator può arricchire i metodi esistenti e aggiungerne altri, per fornire quello che sarà il comportamento aggiuntivo del nemico in gioco.
Tra gli esempi di implementazione di EnemyDecorator realizzati vi sono:

- Melee: il nemico può danneggiare il giocatore mediante un contatto fisico;
- Shooter: il nemico può sparare proiettili a distanza per colpire il giocatore;
- Explosive: il nemico alla morte "esplode" sparando proiettili in tutte le direzioni.

La scelta di questa soluzione è stata fatta per permettere:

- estendibilità: creare nuovi decoratori permette di aumentare la quantità di configurazioni possibili di nemici;
- facilità d'uso: il pattern permette di creare agevolmente nuove configurazioni;
- riuso: ogni decorazione può essere usata in qualsiasi configurazione;
- prevenzione della proliferazione di classi: usare esclusivamente l'ereditarietà per fornire comportamenti multipli ai nemici sarebbe stato poco efficiente.

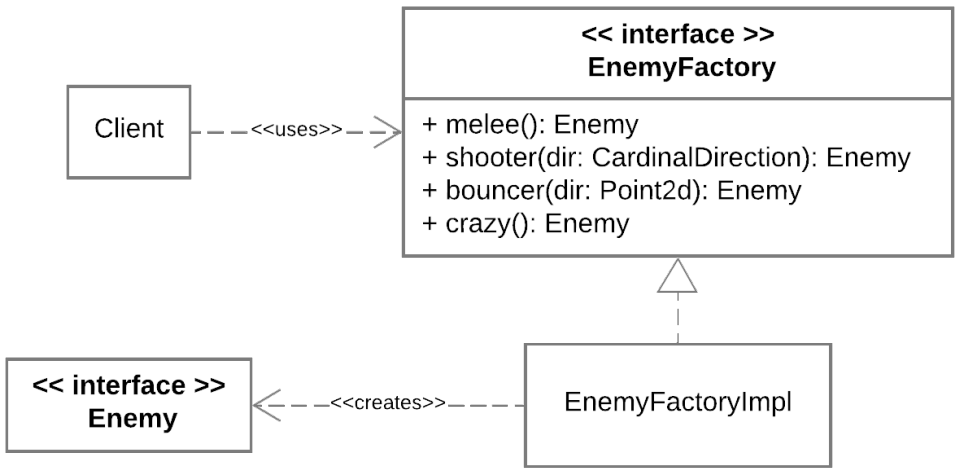


Creazione nemici

Problema Ogni tipo di nemico presente nel gioco è frutto di una particolare configurazione, non solo per quanto riguarda le statistiche (attacco, salute, velocità...) ma anche per il modo in cui si muove, come attacca il giocatore ed altri comportamenti variabili. Per questo motivo creare ed istanziare una configurazione di nemico non è un compito semplice e immediato.

Soluzione E' stato deciso di usare il pattern Abstract Factory, per fare in modo che sia una factory specializzata nella creazione dei nemici a doversi occupare di fornire istanze già configurate correttamente, in questo modo qualsiasi componente dell'applicazione potrà istanziare dei nemici già pronti per essere inseriti nel gioco. La scelta di questa soluzione è stata fatta per permettere:

- divisione delle responsabilità: viene delegato a una componente specializzata un compito di cui non si dovrebbero occupare altre classi;
- estentendibilità: è possibile ampliare la factory per fornire più configurazioni;
- facilità d'uso: fornisce un'interfaccia semplice per creare istanze di nemici.



Capitolo 3

Sviluppo

3.1 Testing automatizzato

I principali elementi del Model sono stati testati utilizzando JUnit. Di seguito le classi in esame:

- Player, di cui viene testato il movimento, la vita e i relativi metodi di cura e danno, vengono testate anche le chiavi e le monete del player, principalmente la loro creazione e alcuni metodi.
- Stanze e porte, con particolare attenzione alla meccanica di "lock/unlock" delle porte speciali.
- Collider, testando la hitbox circolare utilizzata dai vari GameObject.
- Alcune implementazioni di nemici, applicandovi un numero variabile di decoratori, in modo da verificarne la correttezza nel comportamento.
- Alcune implementazioni di MovementAi, per verificare il funzionamento dei relativi algoritmi di movimento.
- Item e Pickup, dei quali vengono testate le modifiche alle statistiche del giocatore dopo la raccolta. Per gli oggetti presenti nel negozio viene inoltre verificato il corretto controllo (e la corrispondente modifica) alle monete possedute dal giocatore.

3.2 Note di sviluppo

3.2.1 Gioele Bucci

- **Utilizzo di `Stream` e lambda expressions:**
  Permalink: `https://github.com/GioeleBucci/TBOOOP/blob/e0307fecf8601b05486f8a27e`
  `src/main/java/tbooop/model/dungeon/doors/impl/SpecialDoor.java#`
  `L45-L48`

- **Utilizzo di `Optional`:**
  Permalink: `https://github.com/GioeleBucci/TBOOOP/blob/e0307fecf8601b05486f8a27e`
  `src/main/java/tbooop/view/api/Keybinds.java#L130-L137`

- **Utilizzo di Wildcard:**
  Permalink: `https://github.com/GioeleBucci/TBOOOP/blob/e0307fecf8601b05486f8a27e`
  `src/main/java/tbooop/model/dungeon/rooms/api/AbstractRoom.java#`
  `L66-L68`

- **Utilizzo dei binding di JavaFX:**
  Permalink: `https://github.com/GioeleBucci/TBOOOP/blob/e0307fecf8601b05486f8a27e`
  `src/main/java/tbooop/view/ViewImpl.java#L131-L136`

- **Utilizzo di Vector2D della libreria Apache Commons Math:**
  Permalink: `https://github.com/GioeleBucci/TBOOOP/blob/e0307fecf8601b05486f8a27e`
  `src/main/java/tbooop/commons/impl/Point2dImpl.java#L18`

Link a risorse esterne utilizzate:

- Video di Florian Himsl sulla generazione dei piani: `https://www.youtube.com/watch?v=1-HIA6-LBJc`

- Articolo con un'ulteriore analisi sulla generazione dei piani: `https://www.boristhebrave.com/2020/09/12/dungeon-generation-in-binding-of-isaac/`

3.2.2 Edoardo Ceresi

- **Utilizzo di `Stream` e lambda expressions:**
  Permalink: `https://github.com/GioeleBucci/TBOOOP/blob/3c87628dedfb57af1d10150c6`
  `src/main/java/tbooop/view/InputManagerImpl.java#L63`

- **Utilizzo di `Optional`:**
  Permalink: `https://github.com/GioeleBucci/TBOOOP/blob/3c87628dedfb57af1d10150c6`
  `src/main/java/tbooop/model/player/api/AbstractPlayer.java#L44`

- **Utilizzo di JavaFX:**
  Permalink: `https://github.com/GioeleBucci/TBOOOP/blob/3c87628dedfb57af1d10150c6`
  `src/main/java/tbooop/view/player/HealthViewImpl.java#L19`
  Permalink: `https://github.com/GioeleBucci/TBOOOP/blob/3c87628dedfb57af1d10150c6`
  `src/main/java/tbooop/view/player/HealthRender.java#L11`

3.2.3 Nicolò Morini

- **Utilizzo di lambda expressions:**
  Permalink: `https://github.com/GioeleBucci/TBOOOP/blob/16f83aacb7fc5618421461e70`
  `src/main/java/tbooop/model/pickupables/items/impl/ItemShopLogic.`
  `java#L41`

- **Utilizzo di JavaFX**:
  Permalink: `https://github.com/GioeleBucci/TBOOOP/blob/16f83aacb7fc5618421461e70`
  `src/main/java/tbooop/view/pickupables/PlayerPickupsRenderAbs.java#`
  `L14`

3.2.4 Simone Mosconi

- **Utilizzo di `Stream`, lambda expressions e di JavaFX:**
  Permalink ad un frammento di codice che contiene un esempio di utilizzo di tutti e tre i casi: `https://github.com/GioeleBucci/TBOOOP/blob/`
  `412e7daaaa97149c5a487b241abe57253cbf77e3/src/main/java/tbooop/view/`
  `enemy/EnemyAnimatorImpl.java#L56-L60`

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

4.1.1 Gioele Bucci

Il mio ruolo principale è stato quello di implementare la generazione dei piani in maniera robusta, occupandomi secondariamente anche dell'implementazione dei commons (punti, vettori e collider), utilizzati pervasivamente da tutto il gruppo. Mi ritengo estremamente soddisfatto del lavoro svolto da me e dai miei colleghi: siamo riusciti ad implementare tutte le funzionalità ritenute obbligatorie e anche alcune feature aggiuntive e tra noi c'è sempre stato un clima di collaborazione. Penso che un fattore importante del nostro successo sia dovuto ad una solida analisi del dominio fatta a monte, che ci ha permesso di iniziare lo sviluppo partendo da un'idea molto chiara di cosa fare. In futuro mi piacerebbe tornare su questo progetto ed aggiungervi ulteriori feature, in particolare un boss: penso che grazie alla premura avuta nel creare del codice facilmente estendibile questa ed altre modifiche si possano fare facilmente.

4.1.2 Edoardo Ceresi

Il mio ruolo principale nel gruppo è stato quello di implementare il Player con la relativa gestione dei comandi, mi sento soddisfatto del mio lavoro svolto e anche di quello dei miei compagni, abbiamo completato tutti la nostra parte obbligatoria e ci siamo spinti a fare le funzionalità opzionali, riuscendo a creare un codice che segue i alcuni principi della programmazione ad oggetti, cercando di non infrangere DRY e soprattutto cercando di rendere il codice più estendibile. Se dovessi continuare il progetto sicuramente implementerei un menù di gioco, la possibilità di cambiare Keybinds, meccaniche diverse di shooting, la possibilità di scegliere le statistiche iniziali. È stata un'esperienza molto proficua per me, poiché questo progetto mi ha dato la forza di continuare gli studi, il gruppo è stato coeso per tutta la durata del progetto, ci siamo sempre messi a disposizione per qualsiasi problema e ci siamo spinti a metterci in gioco per implementare alcune funzionalità facoltative, abbiamo trovato molte difficoltà che ci hanno portato sempre di più ad approfondire la nostra conoscenza che all'inizio era molto superficiale.

4.1.3 Nicolò Morini

Giunto alla conclusione del progetto mi ritengo soddisfatto del risultato generale e della mia parte personale, nonostante l'esperienza totalmente nuova del lavorare in team per il compimento di un progetto di questa rilevanza. Fin dall'iniziale stadio di progettazione il team è risultato compatto, ognuno con idee che potessero aiutare il collettivo. Ogni membro del team ha sempre cercato di dare il massimo e di essere presente nei momenti del bisogno. Mi sento di avere sicuramente imparato molto da questo lavoro non solo per le dinamiche dello sviluppo software, ma dato la mia inesperienza riconosco qualche elemento che poteva essere migliorato. Principalmente ritengo che avrei potuto dedicare una parte maggiore di tempo sulla progettazione nel particolare della mia parte, in quanto sono incappato in modifiche durante il percorso che mi hanno occupato tempo evitabile, che magari non ho utilizzato per concentrarmi maggiormente sull'utilizzo di tecniche avanzate di programmazione in Java. Il risultato finale mi ha comunque soddisfatto molto, tale per cui ritengo aperta la possibilità che in futuro possa ritornare ad apportare modifiche e migliorie al codice di "TBOOOP!", in quelle parti in cui si possono realizzare.

4.1.4 Simone Mosconi

Personalmente mi ritengo soddisfatto del lavoro prodotto da me e i miei colleghi nella realizzazione del progetto. Progettare e sviluppare un applicazione di tali dimensioni per la prima volta non è stato facile, tantomeno farlo lavorando in gruppo, dove è stato necessario coordinarsi efficacemente per riuscire a portare risultati. Nonostante ciò ritengo che questo progetto sia stato estremamente formativo in ambito di programmazione e progettazione. Oltre ad essermi occupato della mia parte di progetto riguardante i nemici, ho cercato di essere flessibile e di dare il mio supporto, per quanto possibile, in ogni angolo del progetto, dalla progettazione all'implementazione. Nonostante le porzioni di struttura da me sviluppate possano essere indubbiamente soggette a miglioramento, credo di essere riuscito a strutturare quantomeno la sezione dei nemici in maniera vantaggiosa in ottica di eventuali cambiamenti futuri, grazie anche all'utilizzo dei pattern di progettazione visti durante il corso. In futuro è probabile che riprenderò in mano il progetto, principalmente per aggiungere nuove funzionalità.

Appendice A

Guida utente

I comandi di gioco sono i seguenti:

- **Movimento**: WASD
- **Sparo**: Freccie direzionali
- **Resize**: P per ingrandire, M per rimpicciolire
- **Fullscreen**: F11

Figura A.1: Gli oggetti presenti nel gioco, con i relativi effetti e prezzi

Appendice B

Esercitazioni di laboratorio

B.0.1 gioele.bucci@studio.unibo.it

- Laboratorio 06: `https://virtuale.unibo.it/mod/forum/discuss.php?d=146511#p208287`
- Laboratorio 07: `https://virtuale.unibo.it/mod/forum/discuss.php?d=147598#p209243`
- Laboratorio 08: `https://virtuale.unibo.it/mod/forum/discuss.php?d=148025#p209738`
- Laboratorio 09: `https://virtuale.unibo.it/mod/forum/discuss.php?d=149231#p211505`
- Laboratorio 10: `https://virtuale.unibo.it/mod/forum/discuss.php?d=150252#p212520`
- Laboratorio 11: `https://virtuale.unibo.it/mod/forum/discuss.php?d=151542#p213928`