relazione OOP
“ArtRat”

Manuel Benagli	manuel.benagli@studio.unibo.it
Cristian Di Donato	cristian.didonato@studio.unibo.it
Matteo Tonelli	matteo.tonelli8@studio.unibo.it
Samuele Trapani	samuele.trapani@studio.unibo.it

15 Febbraio 2025

Indice

1 Analisi **3**
- 1.1 Descrizione e Requisiti . . . 3
- 1.2 Modello Del Dominio . . . 5

2 Design **7**
- 2.1 Architettura . . . 7
- 2.2 Design dettagliato . . . 9
  - 2.2.1 Manuel Benagli . . . 9
  - 2.2.2 Cristian Di Donato . . . 13
  - 2.2.3 Gestione aggiornamento Model . . . 16
  - 2.2.4 Matteo Tonelli . . . 18
  - 2.2.5 Samuele Trapani . . . 23

3 Sviluppo **26**
- 3.1 Testing automatizzato . . . 26
- 3.2 Note di sviluppo . . . 27
  - 3.2.1 Manuel Benagli . . . 27
  - 3.2.2 Cristian Di Donato . . . 28
  - 3.2.3 Matteo Tonelli . . . 29
  - 3.2.4 Samuele Trapani . . . 30

4 Commenti finali **32**
- 4.1 Autovalutazione e lavori futuri . . . 32
  - 4.1.1 Matteo Tonelli . . . 32
  - 4.1.2 Samuele Trapani . . . 32
  - 4.1.3 Cristian Di Donato . . . 33
  - 4.1.4 Manuel Benagli . . . 33
- 4.2 Difficoltà incontrate e commenti per i docenti . . . 34

A Guida utente **35**
  - A.0.1 Menu . . . 35

A.0.2 Gameplay 36
A.0.3 Shop 37
A.0.4 Inventory 37
A.0.5 Missions 38

B Esercitazioni di laboratorio **39**
B.0.1 matteo.tonelli8@studio.unibo.it 39

Capitolo 1

Analisi

1.1 Descrizione e Requisiti

ArtRat vuole essere un gioco di azione stealth , ispirato al gioco già esistente "Robbery Bob", in cui il giocatore vestirà i panni di LuPino, un ladro d'arte con un debole per i quadri rari e costosi. Il tuo obiettivo è infiltrarti in una serie di appartamenti sorvegliati, raccogliere il bottino e fuggire prima che l'allarme scatti o che le guardie ti catturino. Una volta uscito dall'appartamento il giocatore potrà spendere il suo guadagno nello shop di gioco.

Requisiti funzionali

- La generazione delle stanze in ogni appartamento dovrà essere casuale
- I nemici dovranno essere capaci di individuare il giocatore e di rendergli difficile il compito di rubare.
- Il timer, rappresentante l'allarme, si attiverà una volta entrato in un appartamento e dovrà scorrere fino all'uscita del giocatore
- Il furto dovrà essere comunque possibile nonostante gli impedimenti
- Il ladro potrà muoversi in verticale e in orizzontale all'interno del piano
- Gli oggetti di interesse garantiranno un guadagno in punti una volta uscito dall'appartamento
- Il negozio in gioco dovrà essere accessibile tra un furto e l'altro e permettere l'acquisto di nuovi power-up e consumabili con i punti guadagnati

Requisiti non funzionali

- Il gioco dovrà essere ottimizzato per avere un gameplay fluido senza eccessivi cali di frame.
- ArtRat dovrà avere una grafica minimale in grado di rappresentare gli elementi principali e di far capire al giocatore le azioni disponibili
- La finestra di gioco dovrà garantire scalabilità, mantenendo comunque proporzioni adatte.

1.2 Modello Del Dominio

In ArtRat, il giocatore si dovrà muovere all'interno di una serie di appartamenti, ognuno dei quali costituito da un sistema di stanze ognuna con una disposizione precisa di oggetti e ostacoli.
La stanza dunque è il cuore del gameplay, ogni stanza potrà includere appunto Oggetti D'Interesse che saranno l'obbiettivo principale di LuPino, ma anche ostacoli d'ambiente che costituiranno il "labirinto" in cui il ladro si aggirerà.
Il ladro potrà però essere fermato dai sistemi di sicurezza dell'appartamento in questione:

- Allarme : ogni appartamento ha a disposizione un allarme che dopo un certo lasso di tempo suonerà rendendo vano il tentativo di furto
- Personale : vi sarà in ogni camera il personale di guardia, incaricato di sorvegliare i quadri, che una volta visto LuPino cercherà di catturarlo.

Più il giocatore sarà capace a collezionare quadri, più punti gli verranno assegnati una volta uscito dalla camera (simulando dunque la vendita del bottino).
Al ladro sarà anche disponibile un negozio per spendere i "punti" guadagnati dopo ogni furto comprando powerup e consumabili per rendere più facile il furto, o semplicemente per variare lo stile di gioco:

- PowerUp : acquistabili solo una volta.
- Consumabili : acquistabili infinite volte.

Tutti gli oggetti acquistati da LuPino saranno poi disponibili all'interno del suo inventario e potranno essere utilizzati a suo piacimento, ma solo una volta fuori da qualsiasi appartamento.

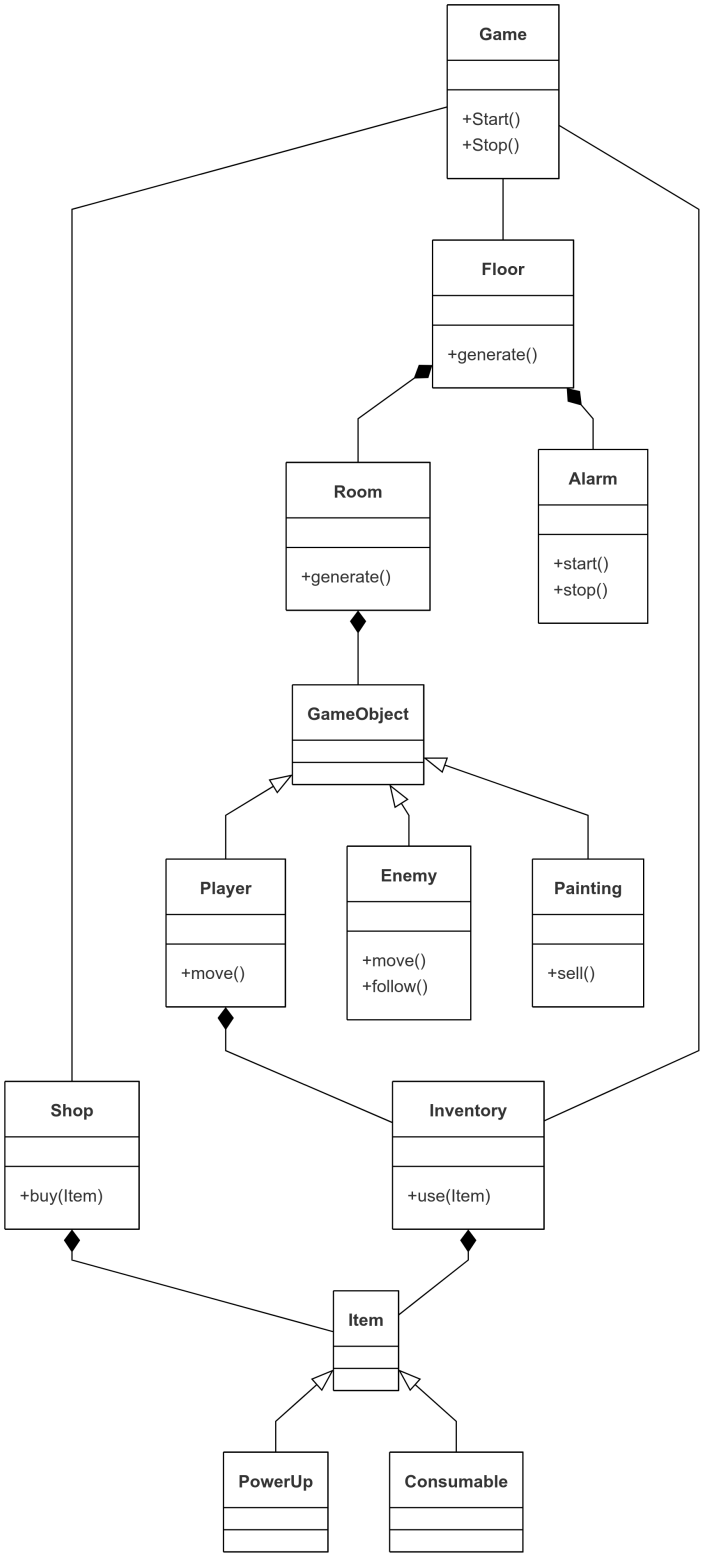

Figura 1.1: UML del modello di dominio

Capitolo 2

Design

2.1 Architettura

ArtRat seguirà il pattern architetturale MVC, in questo modo garantiamo una gestione modulare dei vari scenari, quali il menù principale, l'inventario, il negozio, le missioni e la porzione giocabile ovvero l'appartamento. Il mainController sarà l'entry point iniziale, incaricato di gestire il flusso del gioco attraverso il gameLoop principale. Il mainController dovrà anche avviare e coordinare i vari scenari, permettendo così anche un'alta scalabilità del codice e rendendo semplice una ipotetica aggiunta di nuovi stage, nonchè la sostituzione anche singola di quelli già esistenti. Per ogni stage vi sarà quindi un controller dedicato in grado di gestire gli input della propria view e di aggiornare il model se necessario, i sub-controller dedicati saranno utili a evitare una GodClass, dividendo così le responsabilità degli stage seguendo la logica Single Responsibility Principle (SRP). Il model sarà condiviso per tutti i sub-controller in modo tale da garantire la coerenza dei dati.

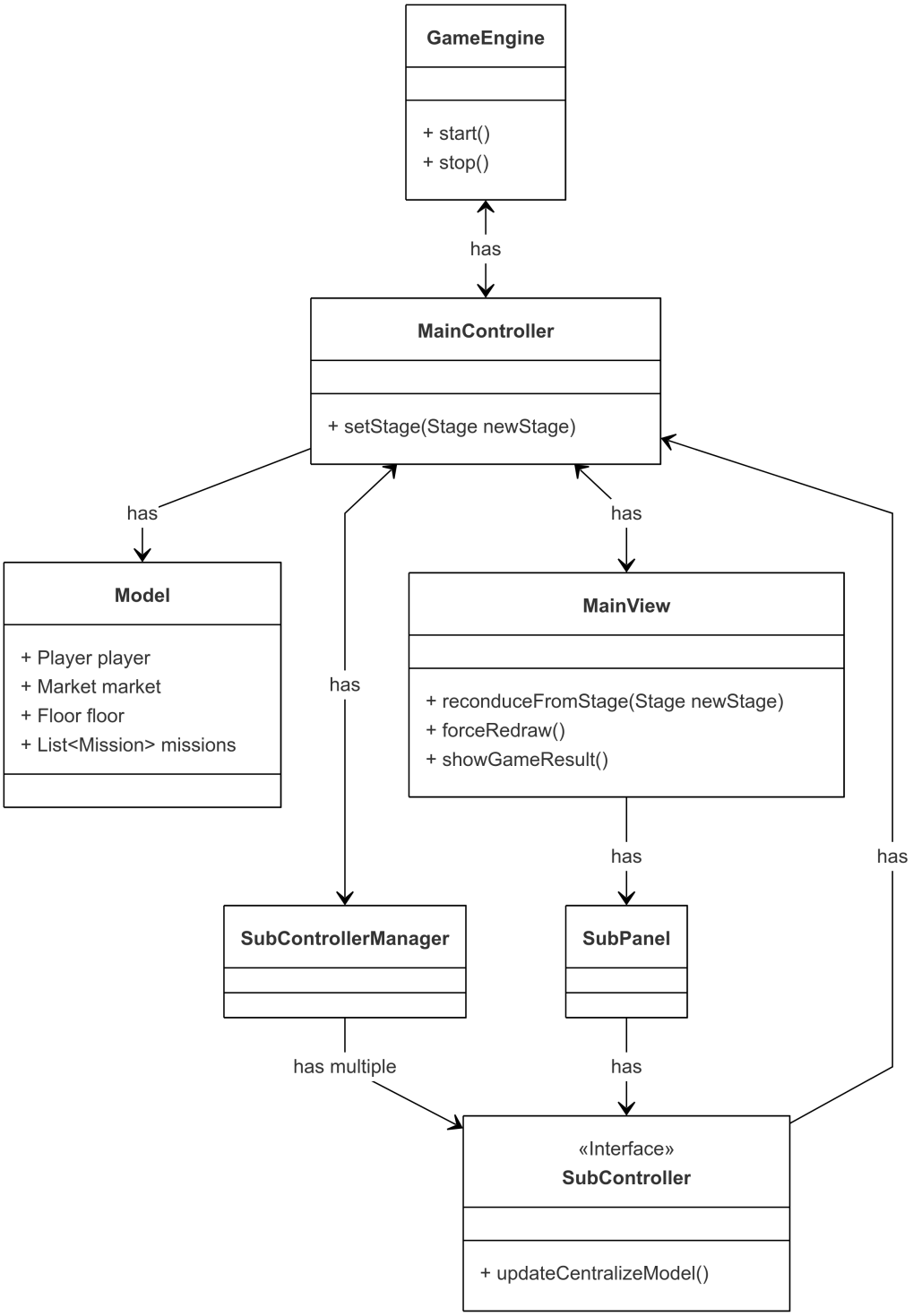

Figura 2.1: UML deL MVC

2.2 Design dettagliato

2.2.1 Manuel Benagli

Logica e gestione di ItemManager

Problema: Inizialmente il fatto che ricerca, filtraggio e ordinamento (funzionalità del market), agiscano in maniera dipendente l'uno dall'altro, ha posto due potenziali problemi, dove allocare queste funzionalità e come non rendere il codice troppo complesso.

Soluzione: L'ItemManager riflette la necessità di gestire diverse funzionalità all'interno del market (sort, filter, search), ho ritenuto opportuno creare questa classe per favorire pulizia di codice e chiarezza dei compiti all'interno della progettazione. La sua logica si appoggia all'utilizzo di tre differenti classi ed interfacce; quello che inizialmente si riscontra può essere un aumento di complessità a livello architetturale, e di mole di codice. Nonostante gli svantaggi sopra citati, ho intrapreso questa scelta in quanto mi avrebbe consentito di rendere il codice più estendibile, ma specialmente più pulito, perchè ci sarebbe stato il rischio che l'ItemManager diventasse poco leggibile, non modificabile e con elementi non propensi al riuso (andando ad intaccare il principio KISS). Le classi e le interfacce che vengono utilizzate dall'ItemManager si ispirano molto allo Strategy Pattern, ma sarebbe un po' forzato definirle tali (seppur non così concettualmente distanti). Per motivi di tempo e di mancato rilevamento del pattern in primo luogo, non è stato fatto uno Strategy puro che comprendesse tutte le operazioni.

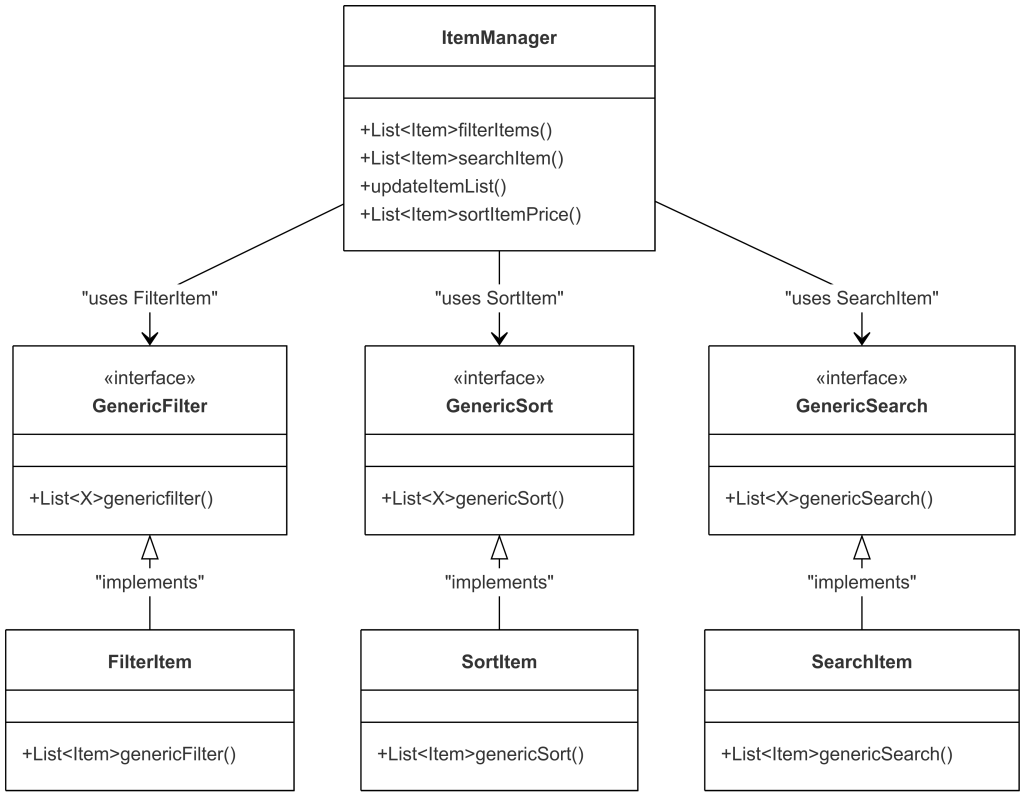

Figura 2.2: ItemManager UML Scheme

Inizializzazione e creazione delle missioni

Problema: Il principale problema riscontrato per le missioni è stato quello della loro effettiva creazione e logica di gestione.

Soluzione: La logica pensata ha come fulcro il MissionCenter, che si avvale dell'ausilio del MissionReader (leggendo dal fine yaml le missioni). Seppure ci siano alcune limitazioni,a livello di mole di codice, ho ritenuto essenziale l'utilizzo del `Factory Design Pattern`, per favorire maggior comprensione, compattazione e pulizia di codice, migliorandone la qualità e la manutenibilità nel lungo periodo (separandola dalla logica di lettura del MissionCenter). Il codice diventa più modulare e facilmente estendibile. A livello MVC, il MissionCenter è dislocato dagli altri stage, questa separazione permette di mantenere il codice chiaro, assegnando al MissionCenter un proprio controller (ha così una responsabilità definita).

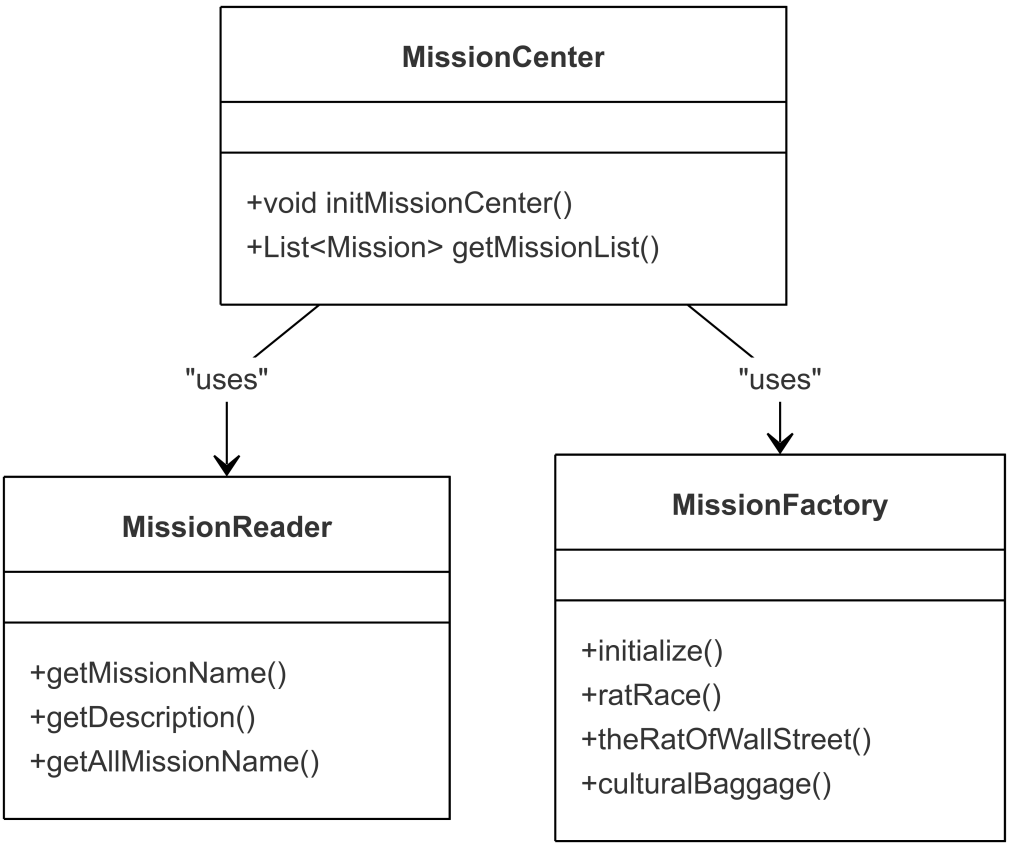

Figura 2.3: MissionCenter and MissionFactory UML

Collocazione del timer all'interno del progetto

Problema: L'allocazione del timer all'interno dell'architettura progettuale di gioco può essere importante, ed un potenziale problema per favorirne un suo riuso futuro.

Soluzione: Durante la progettazione, ho deciso di inizializzare il timer e gestirne il suo funzionamento mediante il MainController, al posto di collocarlo nel GameSubController, considerando che il suo effettivo utilizzo sarebbe stato rilegato solo all'effettivo avvio di una nuova partita. La mia scelta è stata presa per una ragione di estendibilità futura, poichè in questo modo il timer potrebbe essere avviato ed utilizzato non solo nel contesto di gioco, ma anche in tutti gli altri stage, ne sono un esempio lo Shop o il MissionCenter.

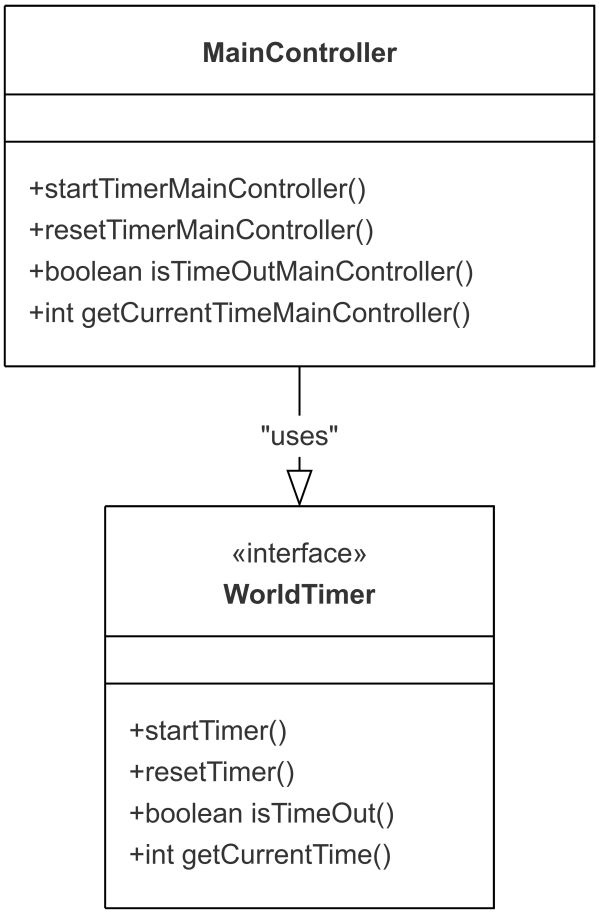

Figura 2.4: WorldTimer UML

2.2.2 Cristian Di Donato

Gestione Estendibile degli Item di Gioco in ArtRat

Problema : Nel contesto dello sviluppo di ArtRat, è stato ritenuto essenziale garantire flessibilità nell'aggiunta di nuovi elementi, senza compromettere, ogni volta, la struttura esistente. Un'implementazione basata su classi indipendenti avrebbe potuto rendere il codice rigido e difficile da estendere, complicando l'integrazione di nuovi item senza modificare le parti esisten e rendendo così tutto difficile da gestire e soggetto a errori.

Soluzione : Per affrontare questa problematica, ho scelto di adottare un approccio strutturato basato su interfacce e classi astratte, integrando il pattern Factory per centralizzare la creazione degli oggetti. L'interfaccia Item definisce i metodi comuni per tutti gli item, essenziali per ottenere informazioni come il nome, la descrizione, il prezzo e il tipo dell'oggetto, oltre a un metodo consume che ne determina l'effetto sul giocatore. Al fine di evitare la ripetizione di codice, è stata introdotta la classe astratta AbstractItem, che implementa i metodi comuni così in questo modo, le varie estensioni possono ereditare le funzionalità comuni e concentrarsi esclusivamente sull'implementazione del metodo consume, che varia in base all'effetto dell'oggetto nel gioco. Per favorire l'estendibilità del codice senza dover in futuro modificare classi esistenti si è utilizzato il pattern Factory attraverso la classe ItemFactory che centralizza la logica di istanzione fornendo all'utente un metodo per la creazione di ogni tipo di oggetto.

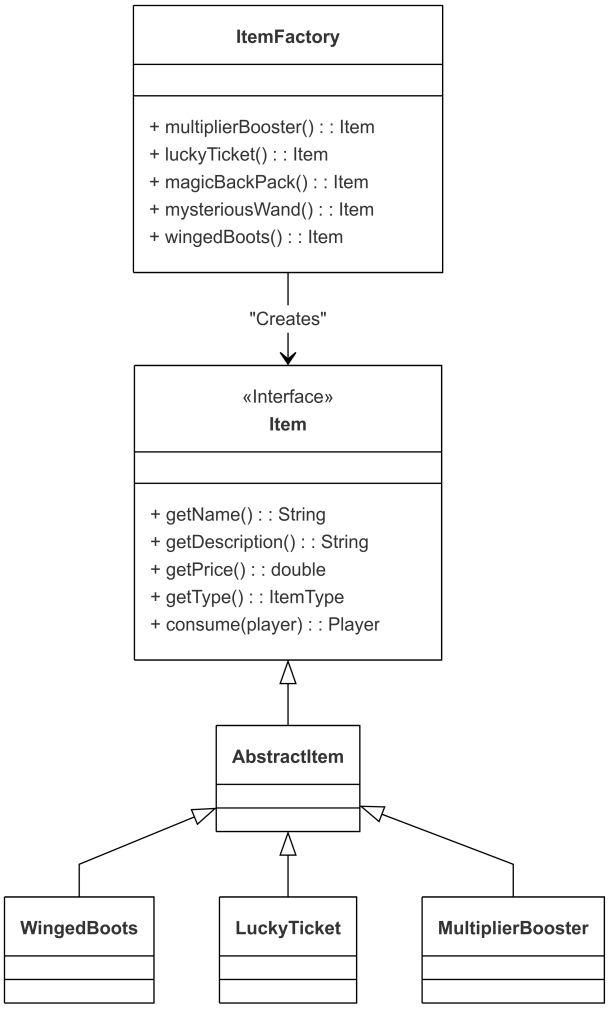

Figura 2.5: Items and Factory UML

Gestione dinamica dei campi degli Item

Problema : In pricipio di sviluppo di ArtRat, è emersa la necessità di gestire in modo dinamico i campi degli item, poiché questi potrebbero essere modificati, rimossi o aggiunti in futuro, o anche durante lo sviluppo. Questo richiede una struttura flessibile per consentire eventuali modifiche o espansioni senza compromettere l'architettura.

Soluzione : Per risolvere il problema, ho scelto di non utilizzare un enum, in quanto un tipo di struttura fissa ossia utile per valori fissi e immutabili (ad esempio, giorni della settimana), e perciò non adatti al nostro bisogno. Invece, ho è adottato un'architettura più flessibile basata su interfacce, classi astratte e implementazioni concrete. Questo approccio permette di aggiungere nuovi lettori (reader) o modificare quelli esistenti in modo semplice e senza compromettere la struttura complessiva. Interfaccia Reader: Definisce il metodo comune a tutti i reader, che è il metodo setPath, utilizzato per impostare il percorso del file da cui leggere i dati. Classe astratta AbstractReader: Fornisce un'implementazione di base che centralizza la logica comune per la gestione del percorso del file. In questo modo, se c'è necessità di una modifica o di un'estensione della logica di gestione del percorso, è sufficiente modificarla in un unico punto ed è quindi facilmente estendibile per l'aggiunta di nuovi reader senza dover riscrivere la logica comune. Classe concreta ItemReader: Implementa i metodi specifici per la lettura dei campi da file relativi agli item.

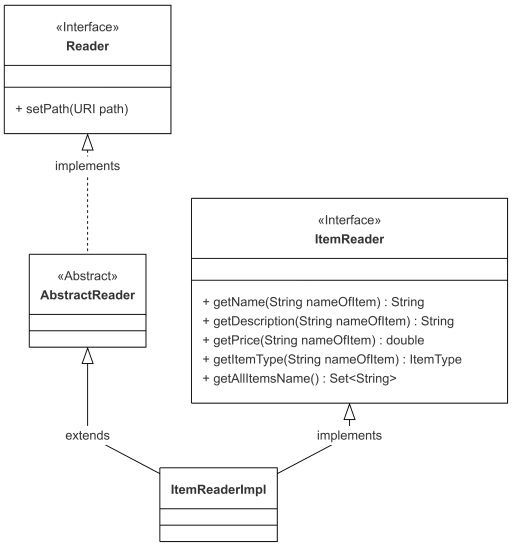

Figura 2.6: Reader UML

2.2.3 Gestione aggiornamento Model

Problema : Durante la progettazione di ArtRat, mi sono trovato ad affrontare la gestione della sincronizzazione tra il Model e le sue modifiche da parte di tutte le schermate. Dovevo perciò garantire che ogni modifica ai dati fosse correttamente sincronizzata tra le diverse viste del sistema.

Soluzione : La soluzione adottata è una versione modificata dell'Observer Pattern, adattata alle esigenze specifiche del sistema.

Model (Subject): Il Model contiene la logica dei dati e si occupa di gestire le informazioni.

SubController (Observer): I SubController contengono la logica di business e sono responsabili della gestione delle varie schermate o funzionalità del sistema. Ognuno di essi ottiene una copia del Model tramite il metodo getModel() dal loro rispettivo MainController ed ogni volta che vogliono apportare una modifica ai dati, inviano una notifica chiamando il metodo updateCentralizeModel() del MainController.
Lavorando su una copia isolata del Model non interferiscono direttamente tra loro, riducendo il rischio di conflitti tra le modifiche. Inoltre il metodo updateCentralizedModel() offre una buona base per estendere la logica di sincronizzazione, permettendo di aggiungere funzionalità più complesse come il merge delle modifiche.

MainController: Funziona come un "Observer Manager", gestendo la comunicazione tra i SubController e il Model. Quando riceve la notifica da un SubController, aggiorna il Model tramite il metodo setModel() e la copia passata aggiornata. In questo modo, il MainController funge da mediatore, centralizzando e controllando il flusso di aggiornamenti tra i vari componenti.

Differenze rispetto al pattern Observer tradizionale: Il pattern Observer tradizionale si basa su una notifica automatica, dove il Subject invia aggiornamenti ai suoi Observer senza che ci sia un controllo esplicito. La nostra soluzione, invece, sacrifica un po' dell'efficienza e dell'automazione tipica del pattern Observer per dare maggiore controllo e adattabilità alla sincronizzazione, rendendola più adatta al nostro caso d'uso dove le modifiche sono contenute in una singola schermata alla volta.

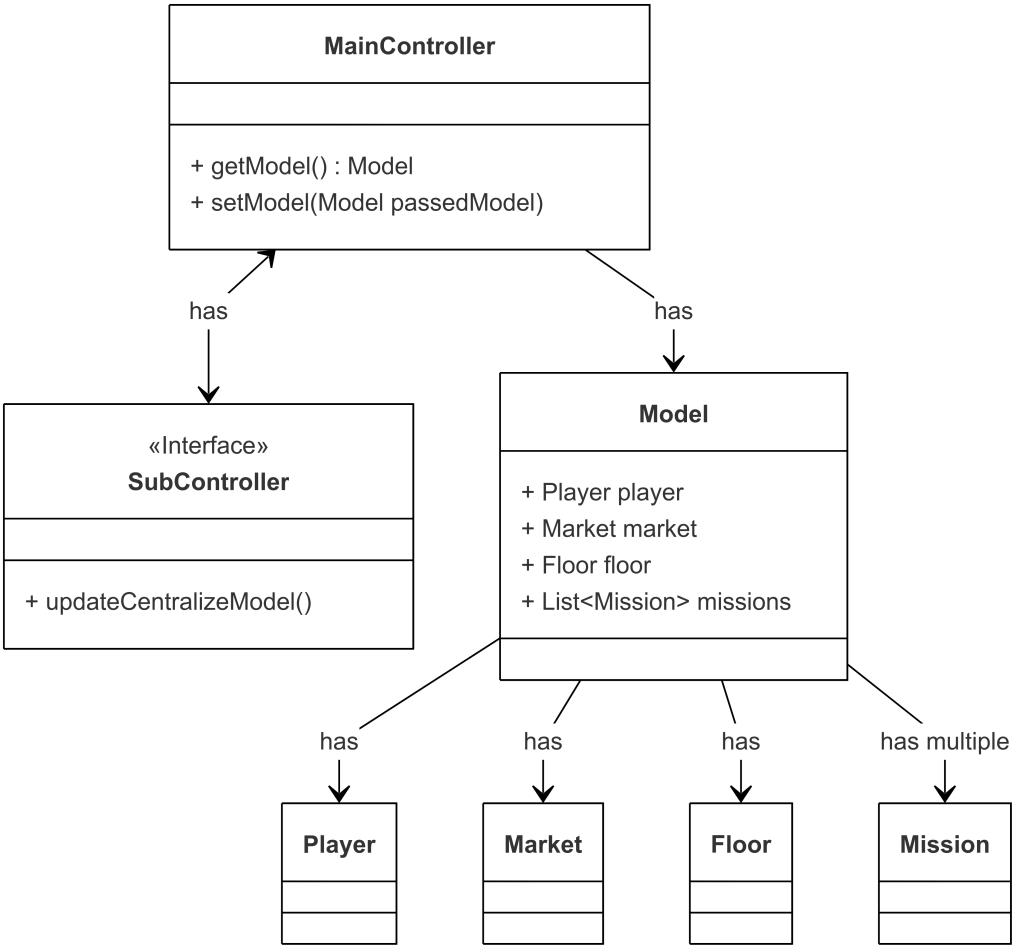

Figura 2.7: Model-Controller UML

2.2.4 Matteo Tonelli

Layout Contiguo della mappa

Problema: Le stanze devono essere disposte in modo contiguo sulla mappa, evitando che siano troppo sparse o isolate. Senza un metodo strutturato, la disposizione potrebbe risultare caotica e non garantire una connessione tra gli spazi. Utilizzando strutture prefabbricate sicuramente si eviterebbero i problemi di creazione, ma risultando così in una scarsa rigiocabilità.

Soluzione: Ho deciso di utilizzare un pattern Strategy per applicare diversi metodi di generazione della macro-struttura del piano, come ad esempio un Random Walk in cui un agente si muove casualmente sulla griglia creando stanze,ma comunque garantendo una disposizione connessa. Ovvimente le future implementazioni di generazione dovranno garantire contiguità. Inoltre, dividendo la struttura del piano da quella della singola stanza, si permette di sapere precedentemente la posizione delle camere, facilitando la creazione dei singoli passaggi.

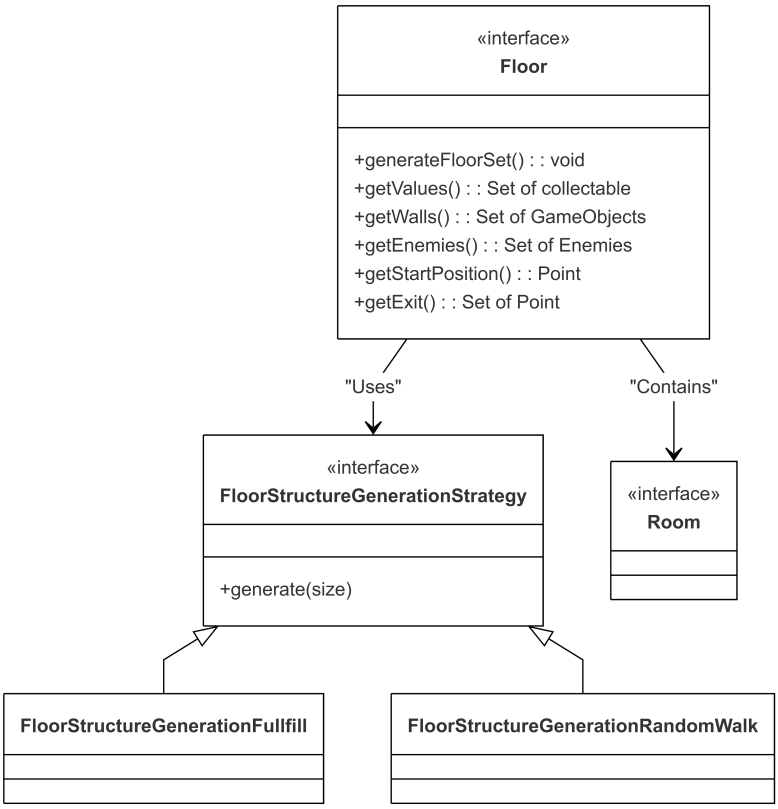

Figura 2.8: Floor Generation Strategy UML

Struttura delle stanze

Problema: Dopo aver definito la disposizione delle stanze, è necessario generare il contenuto del piano in modo modulare. L'uso esclusivo del pattern Strategy per la struttura non è sufficiente, poiché non considera elementi come nemici o oggetti di interesse. Integrarli direttamente complicherebbe il codice e ne ridurrebbe il riutilizzo.

Soluzione: Inizialmente si è considerato l'uso del Template Method per strutturare la generazione, ma questa soluzione avrebbe introdotto una forte dipendenza tra le classi, limitando il riutilizzo del codice. La scelta finale è ricaduta su una combinazione di Builder e Strategy. Così facendo il Builder può garantire una generazione standardizzata, mantenendo flessibilità e riducendo la duplicazione del codice. Inoltre, permette di integrare facilmente nuove strategie per la creazione di oggetti e ostacoli, sia dinamiche (es. DFS) che statiche (es. lettura da file).

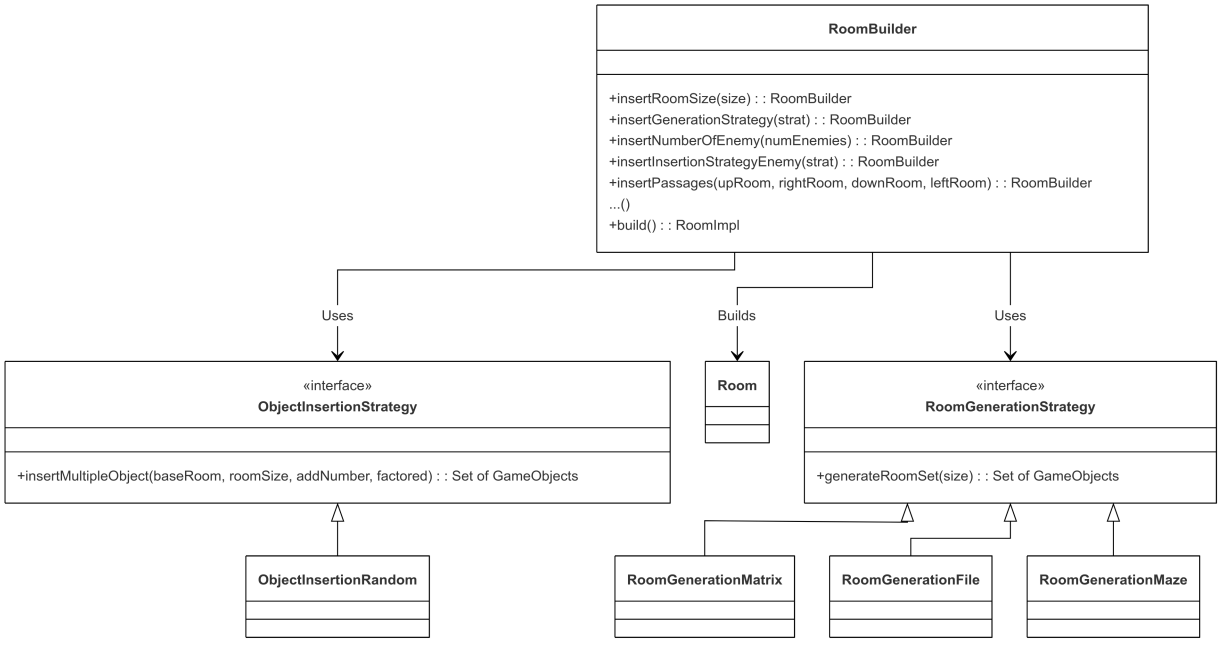

Figura 2.9: Room Builder UML

Orientamento nella mappa

Problema: In una mappa generata proceduralmente e quindi in maniera più o meno caotica sarebbe difficile orientarsi e quindi rendendo troppo difficile il gioco.

Soluzione: La prima idea è stata quella di creare una minimappa, ma è stata scartata in quanto sarebbe risultata difficile da leggere in labirinti troppo complessi e grandi. Ho scelto quindi di creare una semplice bussola per indicare l'uscita e, in ottica di uno sviluppo futuro, ho deciso di implementarla mediante un semplice template method. In questo modo avvantaggiando la creazione di bussole diverse come ad esempio che indicasse ai nemici o ad obbiettivi speciali.

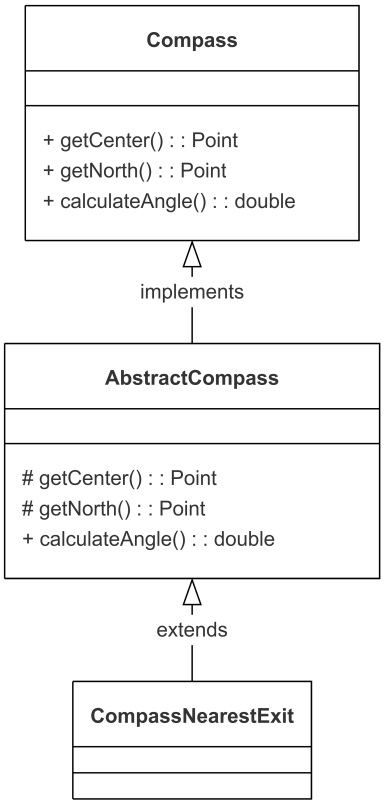

Figura 2.10: Compass UML

Lettura da File

Problema: È necessario un meccanismo per caricare e gestire risorse di configurazione in modo flessibile, indipendentemente dal tipo di input e output. Ad esempio servirebbero dei dati di configurazioni o un metodo di memorizzazione per le stanze prefabbricate.

Soluzione: Per gestire il caricamento delle risorse, ho deciso di implementare un'interfaccia generica, ResourceLoader, che garantisce modularità e riutilizzabilità, consentendo di scegliere il formato più adatto in base al contesto. Per le configurazioni, ho optato per YAML, apprezzato per la sua leggibilità e facilità di modifica da parte degli utenti. Per le stanze prefabbricate, invece, ho scelto JSON, grazie alla sua capacità di rappresentare strutture complesse con vari attributi, come le dimensioni, semplificando così la gestione e la personalizzazione.

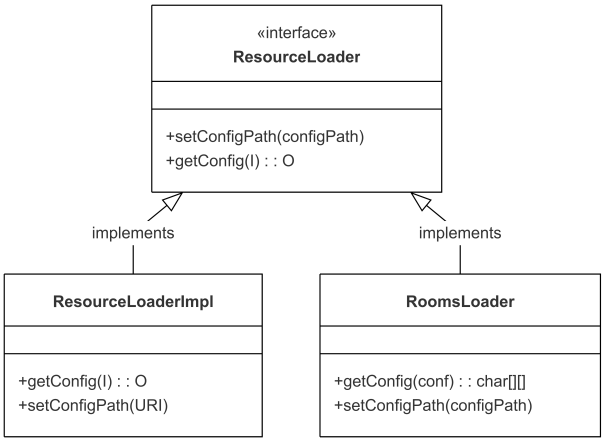

Figura 2.11: Resource Loader UML

2.2.5 Samuele Trapani

Gestione input da tastiera

Problema In fase di progettazione ci si era posto il problema di slegare il più possibile la logica (Model) dalla view e l'input se mal gestito può provocare un forte legame tra la vista dell'applicazione e le operazioni effettive eseguite.

Soluzione Per rendere tutto più indipendente si è optato per l'utilizzo del *Command Pattern*, che utilizza un sistema di notifica che aggiunge i comandi in input in un buffer nel `GameEngine` che ad ogni frame consuma i comandi notificati. Inoltre i comandi vengo etichettati da delle interfacce per capire meglio la loro funzionalità.
Dato che, per notificare i comandi dal `JPanel` al `GameEngine`, si è utilizzato una sorta di "passaggio di testimone" dei comandi, ogni `SubController` potrà implementare a piacimento nuovi comandi specifici per il proprio tipo di situazione, rendendo così il pattern estendibile ad altre funzionalità oltre al solo movimento.
Per la gesione del movimento, ogni comando aggiunge o rimuove in una collezione di `Vector2d` presente in `Entity` che viene utilizzata per il calcolo del movimento sommando i vettori di direzione presenti.

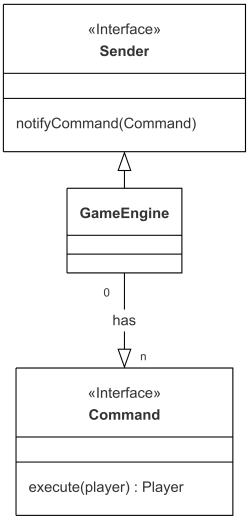

Figura 2.12: Command Pattern UML

Scalabilità delle entità

Problema In fase di progettazione si è puntato a rendere il più possibile le entità scalabili in modo da poter aggiungere in modo comodo e "pulito" nuove entità come:

- nuovi enemy con funzioni personalizzate
- nuovi tipi di entità con nuove meccaniche

Questo approccio di progettazione viene applicato in a medesimo modo anche con i `GameObject`, mettendo a disposizione a *factory* per possibili ampliamenti della struttura.

Soluzione In questo caso si va ad utilizzare un *Factory Method* che permette di rendere più facile una nuova implementazione di `Enemy`. Potendo così aggiungere nuovi tipi di nemici con nuove funzionalità o nuovi metodi di movimento e inseguimento del giocatore.

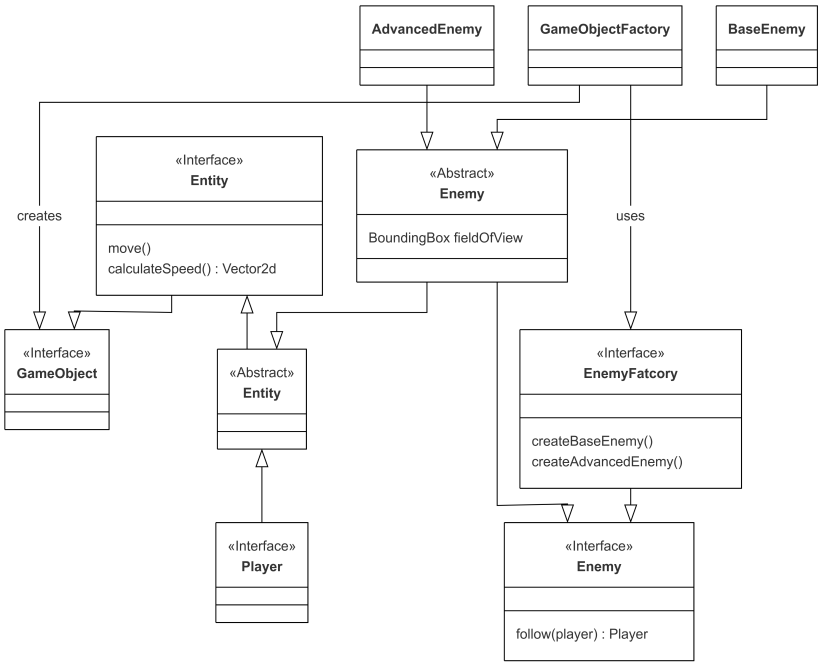

Figura 2.13: Entity structure UML

Gestione delle collisioni

Problema Nella fase di sviluppo, una volta implementata la gestione delle collisioni, il codice risultava poco elegante e confusionario. Tutta la logica risiedeva nel `GameEngine`, dove effettivamente ad ogni frame del gioco vengono effetuati i controlli opportuni per il movimento delle varie entità.

Soluzione È stato effettuato un refactoring spostando la logica della gestione delle collisioni in una classe a parte applicando il *Template Method* per rendere il codice più pulito e dando la possibilità di implementare logiche nuove e più complesse evitando ripetizioni di codice inutili, rispettanto così il principio di sviluppo *D.R.Y.*
Il `GameEngine` possiede un'istanza di `CollisionChecker` con la quale ad ogni loop di gioco viene chiamato `updateAndCheck()` nel quale risiede il funzionamento generale della gestione delle collisioni, la logica effettiva dipende dal tipo di implementazione delle classi che estendono il *Template*.

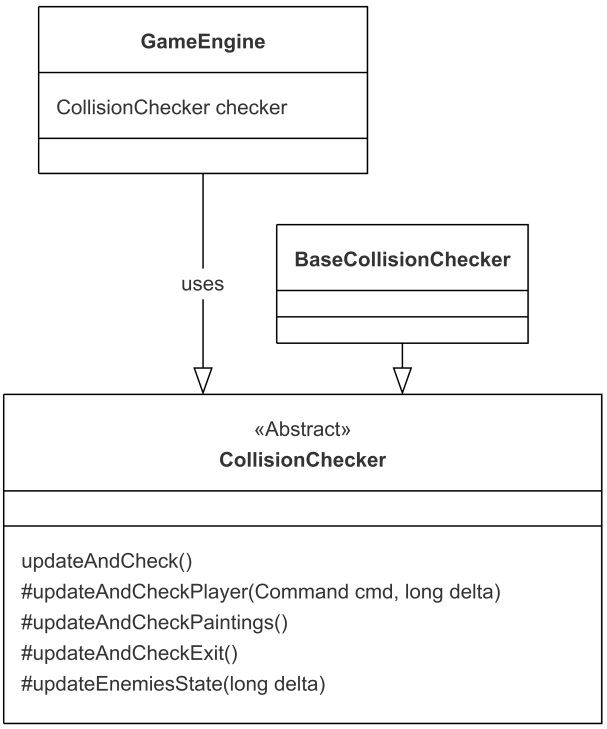

Figura 2.14: Collsion checker UML

Capitolo 3

Sviluppo

3.1 Testing automatizzato

CoinsTest: I test verificano il comportamento della classe Coin, assicurandosi che gestiscano correttamente valori invalidi (come numeri negativi o zero) e che aggiornino l'importo corrente delle monete come previsto. Inoltre, verificano che il saldo finale corrisponda a quello atteso dopo l'aggiunta e la spesa di monete.

ConverterTest: I test verificano la corretta conversione tra millisecondi e fotogrammi al secondo, gestendo input validi e non, controllando poi i risultati con una tolleranza minima (anche per la conversione inversa).

ItemReaderTest: I test verificano la corretta lettura di nome, descrizione, prezzo, tipo e elenco di oggetti da un file YAML, gestendo correttamente i percorsi errati e gli oggetti inesistenti, e assicurandosi che vengano lanciate eccezioni appropriate.

ItemTest: I test verificano l'effetto del consumo di vari oggetti nel gioco da parte di un giocatore, controllando che il moltiplicatore, la velocità, le monete, l'inventario e altri attributi vengano aggiornati correttamente, come nel caso del "Multiplier Booster", "Lucky Ticket", "Winged Boots", "Magic Backpack" e "Mysterious Wand".

MultiplierTest: I test verificano il corretto funzionamento della classe Multiplier, assicurandosi che vengano lanciate eccezioni per valori invalidi nel cambiamento del moltiplicatore e nel calcolo del moltiplicatore delle monete, e che il moltiplicatore influenzi correttamente il valore delle monete.

PlayerTest: I test verificano il corretto funzionamento della classe Player, assicurandosi che vengano gestiti correttamente i metodi di aumento del moltiplicatore, delle monete, della velocità e che questi lancino eccezioni per valori invalidi.

ResourceLoaderTest: Il test verifica il corretto funzionamento della classe ResourceLoader, assicurandosi che l'impostazione dei percorsi di configurazione venga gestita correttamente, che venga lanciata un'eccezione per percorsi di configurazione non validi e che i dati vengano letti correttamente da un file YAML.

TestBoundingBox: Il test verifica la corretta creazione e gestione di un BoundingBox, controllando la precisione nelle dimensioni, il calcolo del centro e la gestione dei bordi. Inoltre, testano vari scenari di collisione tra bounding box per garantire il corretto rilevamento delle sovrapposizioni.

CompassNearestTest: Il test verifica il comportamento del CompassNearestExit, calcolando l'angolo rispetto ai punti dati. I test sono eseguiti con punti null e con array di punti di cui prendere il più vicino al centro.

FloorTest: Il test verifica la creazione di Floor, assicurando che l'oggetto non sia nullo. Inoltre, controlla la corretta generazione di elementi come muri, nemici e collezionabili. Infine, verifica il comportamento in caso di configurazioni invalide o l'assenza di file di risorse.

RoomTest: Il test verifica la creazione e il comportamento della stanza, inclusi gli oggetti all'interno, come nemici e collezionabili. Controlla la struttura della stanza, i passaggi e la corretta gestione dei parametri di configurazione. Sono anche verificati casi con valori invalidi.

3.2 Note di sviluppo

3.2.1 Manuel Benagli

Utilizzo di stream Permalink d'esempio: `https://github.com/2AA-Team/OOP24-artrat/blob/1371e6a407e6229501c7a2b49fb3c6767f049613/src/main/java/it/unibo/artrat/model/impl/market/SearchItem.java#L23`

Utilizzo di lambda Permalink d'esempio: https://github.com/2AA-Team/OOP24-artrat/blob/1371e6a407e6229501c7a2b49fb3c6767f049613/src/main/java/it/unibo/artrat/view/impl/MarketSubPanel.java#L131

Utilizzo di method reference Permalink d'esempio: https://github.com/2AA-Team/OOP24-artrat/blob/1371e6a407e6229501c7a2b49fb3c6767f049613/src/main/java/it/unibo/artrat/model/impl/market/SortItem.java#L21

Utilizzo di generics Permalink d'esempio: https://github.com/2AA-Team/OOP24-artrat/blob/1371e6a407e6229501c7a2b49fb3c6767f049613/src/main/java/it/unibo/artrat/model/api/market/GenericSearch.java#L13

Utilizzo di libreria esterna Timer Permalink d'esempio: https://github.com/2AA-Team/OOP24-artrat/blob/1371e6a407e6229501c7a2b49fb3c6767f049613/src/main/java/it/unibo/artrat/model/impl/WorldTimerImpl.java#L27

Utilizzo di libreria esterna TimerTask Permalink d'esempio: https://github.com/2AA-Team/OOP24-artrat/blob/547256c144d821a825280d4f66fb406ca5c366c0/src/main/java/it/unibo/artrat/model/impl/WorldTimerImpl.java#L36

Fonti https://www.geeksforgeeks.org/java-util-timertask-class-java

3.2.2 Cristian Di Donato

Utilizzo di wildcard Permalink d'esempio: https://github.com/2AA-Team/OOP24-artrat/blob/1371e6a407e6229501c7a2b49fb3c6767f049613/src/main/java/it/unibo/artrat/model/api/world/roomgeneration/ObjectInsertionStrategy.java#L24

Utilizzo di method reference Permalink d'esempio: https://github.com/2AA-Team/OOP24-artrat/blame/1371e6a407e6229501c7a2b49fb3c6767f049613/src/main/java/it/unibo/artrat/model/impl/characte

rs/Lupino.java#L208

Utilizzo di lambda Permalink d'esempio: https://github.com/2AA-Team/OOP24-artrat/blob/1371e6a407e6229501c7a2b49fb3c6767f049613/src/main/java/it/unibo/artrat/model/impl/inventory/InventoryImpl.java#L38

Utilizzo di stream Permalink d'esempio: https://github.com/2AA-Team/OOP24-artrat/blob/1371e6a407e6229501c7a2b49fb3c6767f049613/src/main/java/it/unibo/artrat/controller/impl/subcontroller/InventorySubControllerImpl.java#L75

Utilizzo di libreria esterna per lettura file Permalink d'esempio: https://github.com/2AA-Team/OOP24-artrat/blob/1371e6a407e6229501c7a2b49fb3c6767f049613/src/main/java/it/unibo/artrat/utils/impl/AbstractReader.java#L33

3.2.3 Matteo Tonelli

Utilizzo di stream Permalink di esempio: https://github.com/2AA-Team/OOP24-artrat/blob/1371e6a407e6229501c7a2b49fb3c6767f049613/src/main/java/it/unibo/artrat/model/impl/world/roomgeneration/RoomGenerationEmpty.java#L23

Utilizzo di Lambda Permalink di esempio: https://github.com/2AA-Team/OOP24-artrat/blame/1371e6a407e6229501c7a2b49fb3c6767f049613/src/main/java/it/unibo/artrat/controller/impl/subcontroller/GameSubControllerImpl.java#L94

Utilizzo di Method Reference Permalink di esempio: https://github.com/2AA-Team/OOP24-artrat/blob/1371e6a407e6229501c7a2b49fb3c6767f049613/src/main/java/it/unibo/artrat/model/impl/world/roomgeneration/RoomGenerationMatrix.java#L42

Utilizzo di Generics Permalink di esempio:
https://github.com/2AA-Team/OOP24-artrat/blob/1371e6a407e6229501c7a2b49fb3c6767f049613/src/main/java/it/unibo/artrat/utils/impl/ResourceLoaderImpl.java#L19

Utilizzo di Libreria esterna snakeyaml Permalink di esempio:
https://github.com/2AA-Team/OOP24-artrat/blob/17f15c4da1e99bef08cfd923a268134956d92285/src/main/java/it/unibo/artrat/utils/impl/ResourceLoaderImpl.java#L42

Utilizzo di Libreria esterna jackson Permalink di esempio:
https://github.com/2AA-Team/OOP24-artrat/blob/17f15c4da1e99bef08cfd923a268134956d92285/src/main/java/it/unibo/artrat/utils/impl/RoomsLoader.java#L63

Utilizzo di fonti esterne

Algoritmo DFS Fonte:
https://medium.com/@nacerkroudir/randomized-depth-first-search-algorithm-for-maze-generation-fb2d83702742
Permalink:
https://github.com/2AA-Team/OOP24-artrat/blob/17f15c4da1e99bef08cfd923a268134956d92285/src/main/java/it/unibo/artrat/model/impl/world/roomgeneration/RoomGenerationMaze.java#L30

MainLoop Fonte:
https://github.com/aricci303/game-as-a-lab/blob/91336ae1dc0c58594af0b733330518c3d335b960/Game-As-A-Lab-Step-2-input-proc/src/rollball/core/GameEngine.java#L34
Permalink:
https://github.com/2AA-Team/OOP24-artrat/blob/17f15c4da1e99bef08cfd923a268134956d92285/src/main/java/it/unibo/artrat/app/GameEngineImpl.java#L88

3.2.4 Samuele Trapani

Utilizzo di stream Permalink: https://github.com/2AA-Team/OOP24-artrat/blob/60e36917f7d667edbd342d2a8ec69a0d78849a10/src/main/java/it/unibo/artrat/utils/impl/collisions/BaseCollisionChecker.java#L75

Utilizzo di method reference e lambda Permalink: https://github.com/2AA-Team/OOP24-artrat/blob/9c491126a203c146dcf1e9d789ec79748e5bf403/src/main/java/it/unibo/artrat/model/impl/characters/AdvancedEnemy.java#L55

Utilizzo di fonti esterne

Utilizzo dell'algoritmo AABB Permalink:https://github.com/2AA-Team/OOP24-artrat/blob/fe44ab2cc25ef04bf16b26f41e78bf656a1e7393/src/main/java/it/unibo/artrat/utils/impl/BoundingBoxImpl.java#L82
Fonte: https://kishimotostudios.com/articles/aabb_collision/

Utilizzo della distanza di Manhattan Permalink: https://github.com/2AA-Team/OOP24-artrat/blob/fe44ab2cc25ef04bf16b26f41e78bf656a1e7393/src/main/java/it/unibo/artrat/utils/impl/Point.java#L117
Fonte: https://www.datacamp.com/tutorial/manhattan-distance

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

4.1.1 Matteo Tonelli

Mi sono occupato principalmente della generazione della mappa di gioco e mi reputo soddisfatto della mappa finale, nonostante sia al corrente che si possa raggiungere una più alta pulizia del codice. Nonostante alcuni problemi organizzativi gli obiettivi prefissati sono stati raggiunti in modo soddisfacente. In futuro, relativamente alla mia parte, si potrebbe gestire la mappa utilizzando un grafo in modo da facilitare molte operazioni abbastanza macchinose e, ad esempio, creare metodi di risoluzione di labirinti per eseguire test più efficienti e non basare la percorribilità del piano completamente al programmatore.

4.1.2 Samuele Trapani

Mi sono occupato principalmente della creazione di tutte le strutture di base della fisica di gioco, delle varie entità e le relative meccaniche di movimento e collisione. La mia parte sulla struttura dell'entità è stata pensata cercando di rendere il tutto più estendibile possibile, per poter aggiungere in futuro nuovi nemici o nuovi personaggi da utilizzare con nuove funzionalità. La struttura della gestione degli input permette anche di aggiungere in futuro nuovi comandi per nuove meccaniche di gioco aggiungibili.
Ho trovato l'idea di fare un progetto molto stimolante a livello creativo e che dà possibilità anche di migliorare nella programmazione ad oggetti sia dal punto di vista di sviluppo che progettuale. Tuttavia sconsiglierei di fare un videogioco, perchè richiede alcune logiche non improntate sulla OOP che possono rallentare notevolmente lo sviluppo se dovessero essere commessi

errori, invece un gestionale risulterebbe più familiare agli esercizi e agli esempi fatti a lezione.
Purtroppo il gruppo non si è ben organizzato per riuscire a terminare il progetto entro la deadline causa un'intensa sessione di esami e una scarsa coesione tra i componenti del gruppo.

4.1.3 Cristian Di Donato

La creazione di ArtRat è stata una grande sfida personale, che mi ha portato a riflettere su me stesso e sulle mie capacità. Ritengo di aver affrontato il progetto in modo dignitoso, producendo un codice di qualità discreta.
La parte più impegnativa è stata senza dubbio quella relativa al Model e ai Controller, su cui mi sono concentrato soprattutto nelle fasi iniziali del lavoro. Ho incontrato diverse difficoltà nel trovare una soluzione ottimale per le esigenze del progetto e, sebbene il risultato finale non sia perfetto, posso ritenermi abbastanza soddisfatto di quanto realizzato.
Se avessi l'opportunità di proseguire il progetto, mi concentrerei sul miglioramento della gestione dinamica dei campi degli Item, sull'integrazione di oggetti interagibili nella mappa e sull'ottimizzazione della struttura del Model e dei vari Subcontroller. Nel complesso, questa esperienza mi ha permesso di crescere sia a livello tecnico che organizzativo, offrendomi spunti preziosi per affrontare le prossime sfide con maggiore consapevolezza e sicurezza.

4.1.4 Manuel Benagli

Mi sono occupato della creazione e gestione del market, oltre al timer e alla gestione e creazione delle missioni. Mi ritengo soddisfatto di quanto appreso in questo progetto, nonostante tutte le difficoltà incontrate. Il lavoro ha infatti ampliato la mia comprensione della progettazione orientata agli oggetti, ma sconsiglio la creazione di un videogioco, in quanto ha una mole di lavoro troppo pesante per la sessione universitaria. Lo sviluppo di Art Rat è stato disorganizzato a seguito di una sessione esami estenuante, che non ci ha concesso un attimo di respiro, rendendo difficoltosa l'organizzazione generale. Questo ha avuto un impatto sulla pianificazione delle tempistiche, portando a sfide nella definizione delle priorità e nell'ottimizzazione dei processi di sviluppo.

4.2 Difficoltà incontrate e commenti per i docenti

Ora che le lezioni e le spiegazioni sono complete, possiamo ritenerci soddisfatti di ciò che abbiamo appreso. Il percorso è stato impegnativo, ma ci ha permesso di acquisire conoscenze importanti e di approfondire gli argomenti trattati. Detto questo, riteniamo che la sessione d'esami sia stata particolarmente dura, e affrontare tutto insieme sia stato complesso, richiedendo un grande sforzo di organizzazione e concentrazione.

Appendice A

Guida utente

A.0.1 Menu

Il menu è diviso in:

- **Game** avvia direttamente il gioco.
- **Shop** apre la schermata in cui è possibile acquistare consumabili e powerup.
- **Inventory** apre la schermata dove sono presenti i consumabili e i powerup acquistati.
- **Missions** apre la schermata dove vengono elencate le missioni.
- **Exit** Chiude l'applicazione.

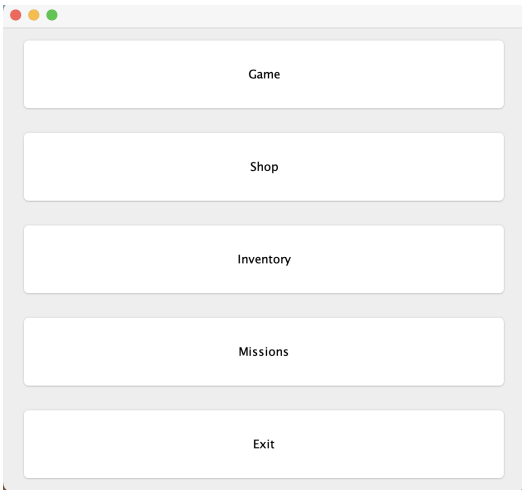

Figura A.1: Main Menu

A.0.2 Gameplay

Per completare la partita, il giocatore deve attraversare la mappa e raggiungere l'uscita, indicata dalla porta verde, sempre segnalata dalla bussola in alto a sinistra. Lungo il percorso, può raccogliere i quadri presenti per ottenere un punteggio più alto, ma non è obbligatorio prenderli tutti. L'obiettivo principale è uscire dalla mappa senza farsi catturare dalle guardie, altrimenti la partita terminerà immediatamente con un punteggio pari a zero.
Alla fine della partita, il punteggio viene assegnato in base al numero di quadri raccolti. Tuttavia, se il timer scade prima di raggiungere l'uscita, la partita si conclude con una sconfitta e nessun punto viene assegnato.

Movimento

Per raccogliere i quadri basta passare sopra di essi con il proprio personaggio e i tasti per il movimento sono:

- **W** vai su.
- **A** vai a sinistra.
- **S** vai giù.
- **D** vai a destra.

Figura A.2: Game Screenshot

A.0.3 Shop

In questa schermata è possibile acquistare powerup e consumabili applicando anche dei filtri di ricerca.

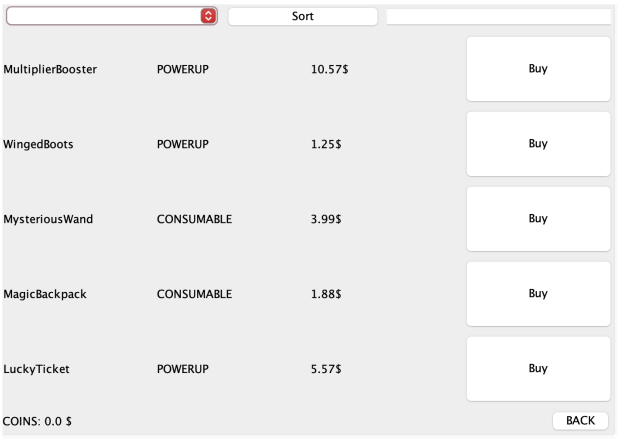

Figura A.3: Shop Screenshot

A.0.4 Inventory

In questa schermata ci sono tutti i consumabili e i powerup acquistati in precedenza.
Con un click sul nome dell'item viene visualizzata su dialog la descrizione dell'elemento selezionato, invene cliccando su *Use* l'item viene consumato e gli effetti vengono applicati.

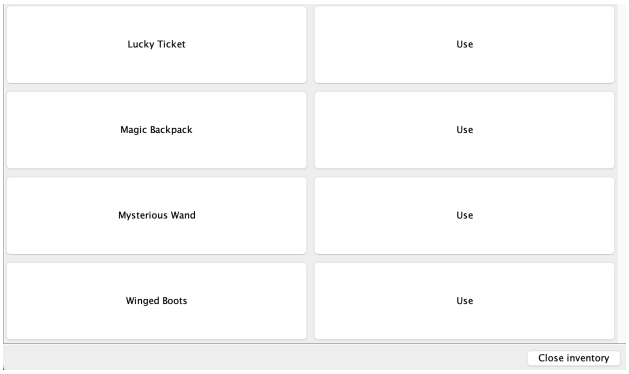

Figura A.4: Inventory Screenshot

A.0.5 Missions

In questa schermata è presente l'elenco delle missioni da compiere all'interno del gioco.

Attenzione! Le missioni vengono completate solamente all'apertura della specifica schermata destinata alle missioni.

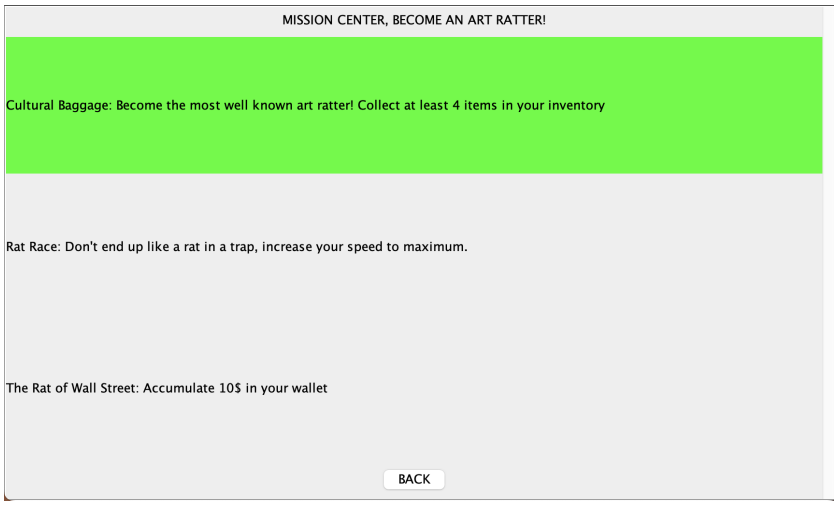

Figura A.5: Missions Screenshot

Appendice B

Esercitazioni di laboratorio

B.0.1 matteo.tonelli8@studio.unibo.it

- Laboratorio 09: `https://virtuale.unibo.it/mod/forum/discuss.php?d=179154#p247904`
- Laboratorio 11: `https://virtuale.unibo.it/mod/forum/discuss.php?d=181206#p250224`