

OOP Project

“Vita Nova”

Eric Aquilotti
Simone Mazzacano
Francesco Meloni
Elia Mami

11 giugno 2025

Indice

1 Analisi	3
1.1 Introduzione	3
1.2 Descrizione del gioco	3
1.3 Requisiti funzionali	3
1.4 Requisiti non funzionali	4
1.5 Modello di dominio	4
1.5.1 Mappa	4
1.5.2 Human (NPC)	5
1.5.3 Powerups	5
1.5.4 Malattie	5
1.5.5 Capitolo	5
1.5.6 Timer	6
1.5.7 Menu	6
2 Design	8
2.1 Architettura	8
2.2 Design dettagliato	9
2.2.1 Eric Aquilotti	9
2.2.2 Francesco Meloni	14
2.2.3 Simone Mazzacano	19
2.2.4 Elia Mami	26
3 Sviluppo	30
3.1 Testing automatizzato	30
3.1.1 Eric Aquilotti	30
3.1.2 Francesco meloni	30
3.1.3 Simone Mazzacano	31
3.1.4 Elia Mami	31
3.2 Note di sviluppo	32
3.2.1 Eric Aquilotti	32
3.2.2 Francesco Meloni	33

3.2.3	Simone Mazzacano	34
3.2.4	Elia Mami	34
4	Commenti finali	36
4.1	Autovalutazione e lavori futuri	36
4.1.1	Eric Aquilotti	36
4.1.2	Francesco Meloni	37
4.1.3	Simone Mazzacano	37
4.1.4	Elia Mami	38
4.2	Difficoltà incontrate e commenti per i docenti	38
A	Guida utente	39
B	Esercitazioni di laboratorio	40
B.0.1	francesco.meloni6@studio.unibo.it	40

Capitolo 1

Analisi

1.1 Introduzione

Il progetto consiste nello sviluppo di un gioco di simulazione 2D, in cui il giocatore gestisce una popolazione dinamica su una mappa. Il gioco è strutturato in capitoli, che diventano progressivamente più difficili.

1.2 Descrizione del gioco

Durante ogni capitolo, il giocatore controlla un umano che può muoversi e interagire con altri umani. Gli umani possono essere maschi o femmine, e quando due di genere opposto si scontrano, generano nuovi umani (figli), il giocatore è sempre maschio. I figli possono nascere come maschi o femmine e contribuiscono alla crescita della popolazione. Durante il capitolo, possono verificarsi malattie, che colpiscono alcuni umani e riducono temporaneamente la loro capacità di movimento o interazione. Inoltre, sulla mappa appariranno potenziamenti che migliorano le statistiche del giocatore, come la velocità di movimento o la resistenza alle malattie. L'obiettivo principale è raggiungere una popolazione target entro un tempo limite. Se il giocatore non riesce a raggiungere l'obiettivo, il capitolo viene considerato perso. Dopo ogni capitolo completato con successo, il giocatore può migliorare le proprie statistiche permanenti, rendendo più facile affrontare i livelli successivi.

1.3 Requisiti funzionali

- **Movimento:** il giocatore può muoversi liberamente nella mappa e interagire con altri umani. La mappa avrà ostacoli come acqua o sassi;

- **Collisioni:** quando il giocatore collide con altri umani, possono verificarsi eventi come la nascita di figli e la contrazione di una malattia;
- **Malattie:** il player può ammalarsi e diffondere la malattia, riducendo temporaneamente le statistiche sue e degli ammalati;
- **Powerups:** elementi sulla mappa che migliorano le statistiche del giocatore per un tempo limitato;
- **Obiettivi:** il giocatore deve raggiungere una popolazione target entro un tempo limite.
- **Menu:** il giocatore deve poter interagire con un menu principale e menu secondari per avviare il gioco, visualizzare le statistiche, mettere in pausa e uscire;
- **Timer:** il giocatore perde quando, senza aver raggiunto l'obiettivo, il timer raggiunge il tempo limite.

1.4 Requisiti non funzionali

- Il gioco deve essere molto efficiente per riuscire a gestire migliaia di umani simultaneamente sulla mappa;
- La finestra del gioco si può ridimensionare;
- Il gioco ha una frequenza di aggiornamento di 60 fps.

1.5 Modello di dominio

1.5.1 Mappa

La mappa è autogenerativa, il giocatore avrà un terreno differente per ogni capitolo ed è composta da:

- **Erba**, dove è possibile camminare;
- **Acqua, Costa e Roccia**, dove non è possibile camminare;

1.5.2 Human (NPC)

L'umano è colui che si muove in modo randomico nella mappa e ha interazione con gli altri umani. A seconda della skin (immagine rappresentativa di un umano) si vedrà se un NPC è maschio o femmina. Se è una femmina avrà un tempo di cooldown per la riproduzione dopo aver prodotto un figlio e quando nasce. Dalla skin si vedrà anche se un NPC è malato o no.

Player

Il player è sempre maschio e si muove nelle 8 direzioni in base all'input dato dall'utente. Il player ha diverse statistiche permanenti:

- **Velocità.**
- **Resistenza alle malattie** (la probabilità di ammalarsi è minore).
- **Raggio di collisione maggiorato.**
- **Fertilità** (Più probabilità di avere figlie femmine).

1.5.3 Powerups

I powerups appariranno in modo casuale sulla mappa e solo il player sarà in grado di raccoglierli, forniranno un incremento temporaneo delle statistiche viste sopra 1.5.2.

1.5.4 Malattie

Il player potrà contrarre malattie in modo casuale a seguito di una riproduzione durante il capitolo, con una conseguente riduzione temporanea delle statistiche elencate in 1.5.2. Inoltre, se uno dei genitori coinvolti in una riproduzione è malato, la malattia verrà trasmessa al partner e al figlio. Si noti che ogni umano potrà contrarre la malattia una sola volta.

1.5.5 Capitolo

Il capitolo è il livello da superare per progredire nel gioco. Inizialmente ci sarà un certo numero di femmine in base al numero del capitolo. Anche la popolazione target aumenterà a seconda del numero del capitolo.

1.5.6 Timer

Il giocatore deve poter visualizzare il tempo rimanente e perdere se l'obiettivo del capitolo non viene raggiunto entro il tempo limite. Inoltre, il timer deve essere resettato all'inizio di ogni capitolo e messo in pausa quando viene attivato il menu di pausa.

1.5.7 Menu

- **Menu iniziale**, permette di iniziare una nuova partita, continuare l'ultima salvata e uscire dal gioco;
- **Menu di pausa**, consente di mettere in pausa, riniziare il capitolo, generare una nuova mappa e uscire dal capitolo corrente;
- **Menu di statistiche**, mostra le statistiche del giocatore;
- **Menu di vincità e upgrade**, permette di potenziare una statistica a scelta e di progredire al capitolo successivo;

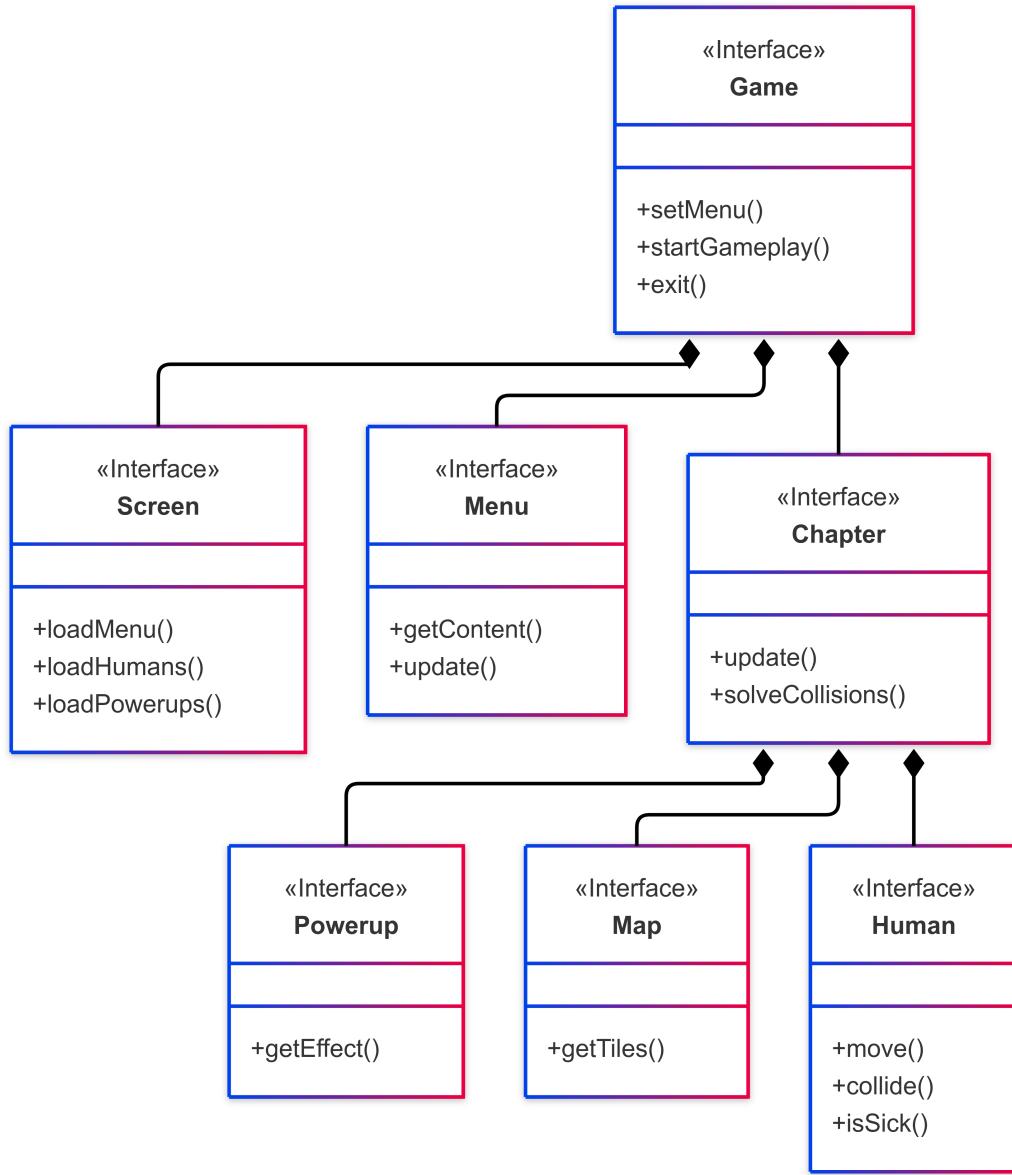


Figura 1.1: Schema UML dell’analisi.

Capitolo 2

Design

2.1 Architettura

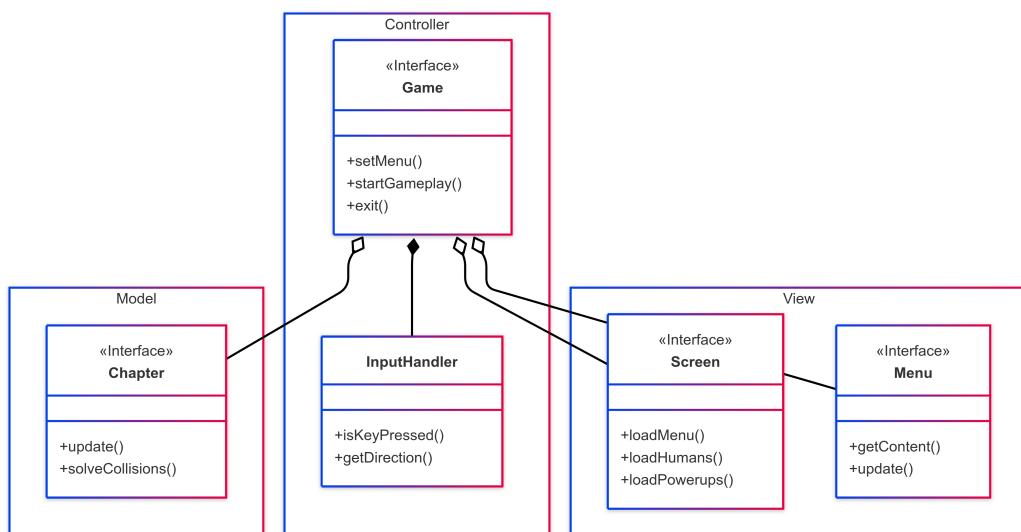


Figura 2.1: Schema UML dell’architettura MVC.

L’architettura del gioco segue il pattern architetturale MVC puro. Game è il controller che si occupa di ricevere gli input e interagire col model e la view di conseguenza per presentare i diversi menu o far muovere il player durante un capitolo. I 3 elementi sono opportunamente separati e non sarebbe un problema cambiare completamente la view.

2.2 Design dettagliato

2.2.1 Eric Aquilotti

Humans

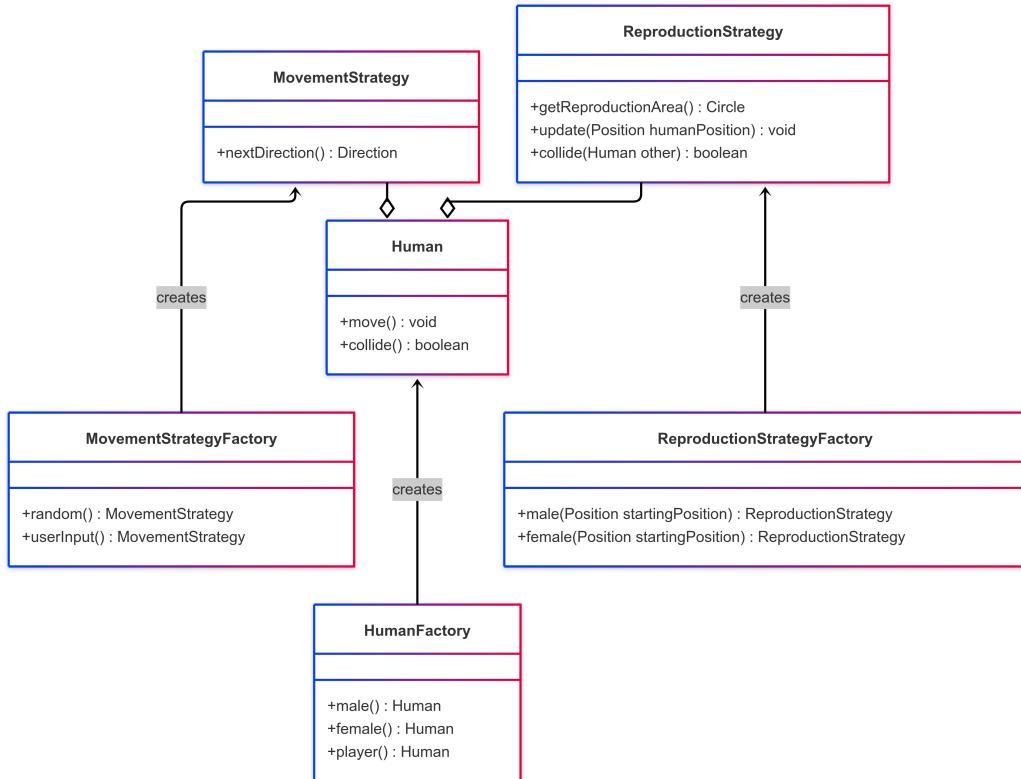


Figura 2.2: Schema UML gestione humans.

Per la gestione dei personaggi nel gioco è stata adottata una *factory*, in modo da facilitare l'estensione futura con nuovi tipi di personaggi, senza modificare il codice esistente ma semplicemente aggiungendo nuovi metodi. Attualmente, i personaggi generabili sono: **male** (NPC), **female** (NPC) e **player**, che corrisponde a un human di tipo male controllato dall'utente.

Per modellare il comportamento dei personaggi è stato impiegato lo *Strategy Pattern*, combinato con una *factory* dedicata alla gestione sia del movimento sia della riproduzione.

Per quanto riguarda il **movimento**, sono state definite due strategie:

- una per il movimento casuale degli NPC;

- una per il movimento controllato da input da tastiera per il player.

Anche la **riproduzione** è gestita tramite strategie distinte:

- i personaggi di tipo *male* non si riproducono, poiché non sono responsabili della generazione dei figli;
- i personaggi di tipo *female* si riproducono quando entrano in collisione con un *human* di tipo diverso (non *female*), rispettando un intervallo di *cooldown* prima di poter attivare nuovamente la riproduzione.

Collisioni

Per la gestione delle collisioni tra personaggi, ogni *Human* è dotato di un'area di riproduzione rappresentata da un cerchio, posizionato nella zona tra il busto e le gambe. In questo modo, per verificare l'avvenuta collisione tra due individui, è sufficiente controllare l'intersezione tra i rispettivi cerchi.

L'approccio iniziale prevedeva, per ogni personaggio di tipo *female*, un controllo diretto su tutti gli altri personaggi presenti nella mappa, con una complessità temporale pari a $\mathcal{O}(n^2)$. Tuttavia, poiché il sistema è progettato per gestire un elevato numero di umani contemporaneamente, tale soluzione risultava inefficiente.

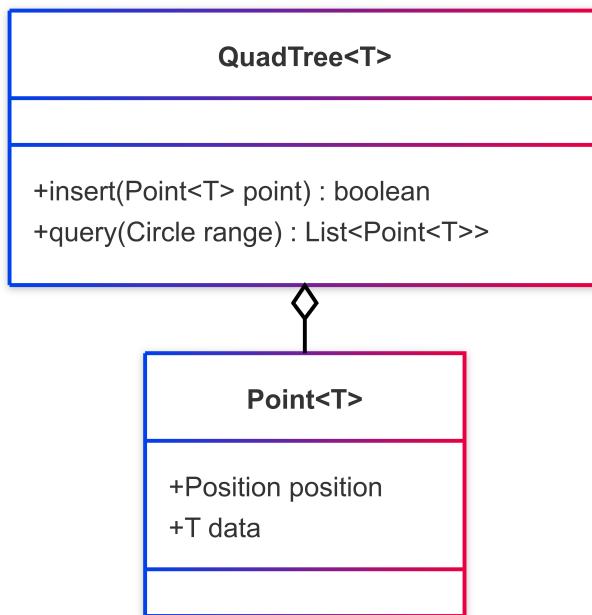


Figura 2.3: Schema UML quad tree.

Per migliorare le prestazioni, è stato introdotto l'utilizzo di una struttura dati di tipo *QuadTree*. Questa struttura consente di ottenere in modo efficiente tutti i punti (cioè i personaggi) all'interno di un determinato range spaziale, con una complessità di ricerca pari a $\mathcal{O}(\log n)$ nella media dei casi.

Di conseguenza, la complessità complessiva della verifica delle collisioni è stata ridotta a $\mathcal{O}(n \log n)$, poiché per ogni female viene eseguita una singola query sul *QuadTree*. Un ulteriore ottimizzazione introdotta è quella di eseguire la query solo con le female che possono riprodursi, ovvero che la riproduzione non è in cooldown.

Temporizzazione

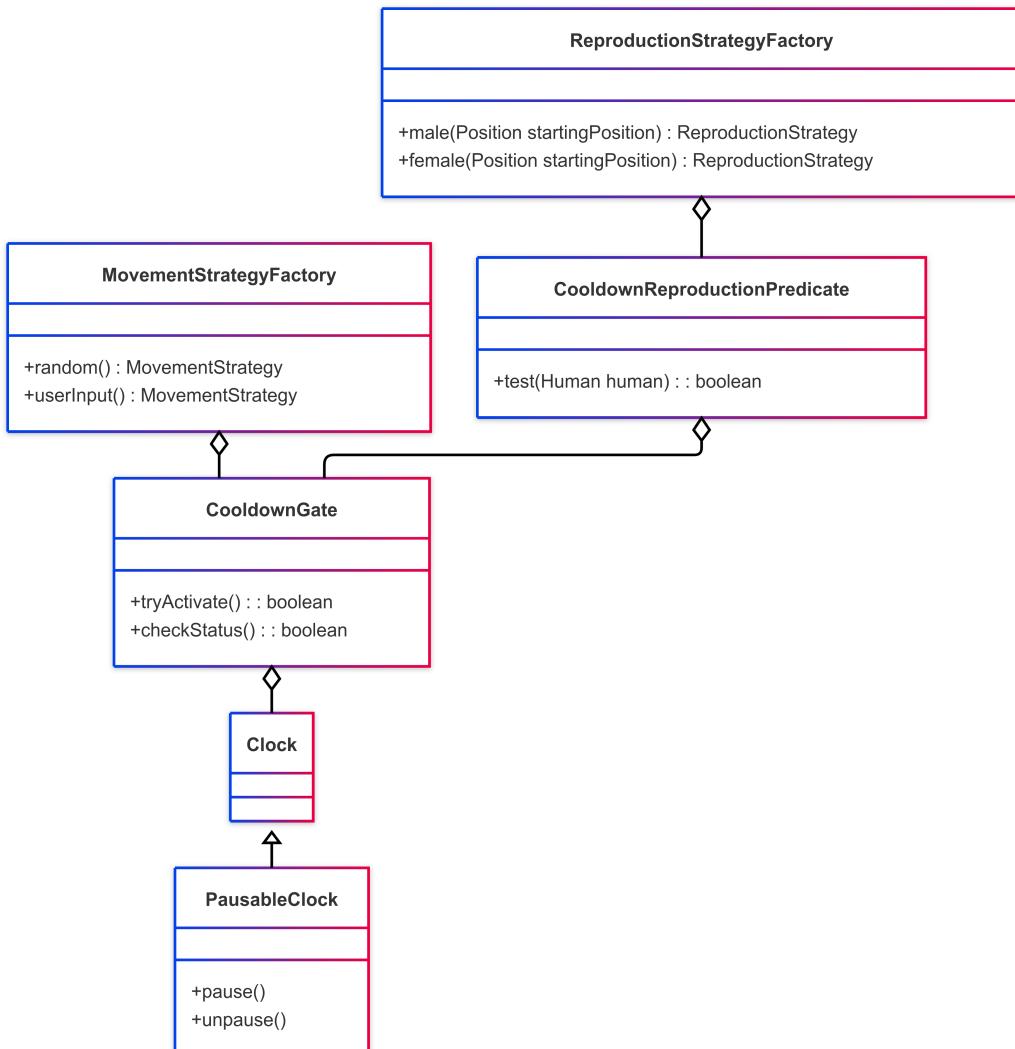


Figura 2.4: Schema UML gestione cooldown.

Per gestire gli elementi del gioco che richiedono una temporizzazione (come la riproduzione, il timer ...) ho implementato la classe **CooldownGate**, che consente di controllare i vari intervalli temporali utilizzando un oggetto **Clock**.

Dal momento che è necessario tenere conto anche delle pause di gioco, ho esteso la classe Java **Clock** creando una nuova classe **PausableClock**, la quale permette di sospendere e riprendere il tempo. In questo modo è

possibile mantenere il corretto funzionamento di tutte le temporizzazioni anche durante le fasi di pausa.

View

La parte della *view* si occupa della gestione dell’interfaccia grafica tramite l’astrazione di uno *screen* e la relativa implementazione, che ha il compito di mostrare a schermo i dati caricati: gli *humans*, i testi dei menu e la mappa.

Nel gestire un numero elevato di personaggi, è emerso un problema legato all’efficienza del rendering. In particolare, invocare la funzione `drawImage` separatamente per ogni *human* si è rivelato estremamente inefficiente, causando rallentamenti significativi, in quanto si tratta di un’operazione computazionalmente costosa.

Per risolvere il problema, è stata adottata una soluzione basata su una `BufferedImage`: tutti gli elementi della scena vengono disegnati prima su questa immagine in memoria. Successivamente, viene effettuata una singola chiamata a `drawImage` per renderizzare il contenuto della `BufferedImage` sullo schermo.

Questo approccio consente un rendering molto più efficiente e fluido, permettendo di mantenere un elevato frame rate anche in presenza di numerosi oggetti da visualizzare.

2.2.2 Francesco Meloni

Mappa

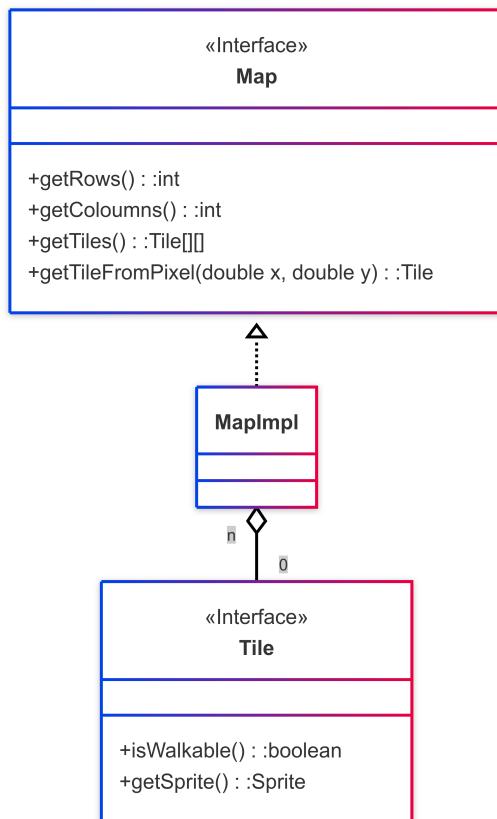


Figura 2.5: Schema UML della Mappa.

La mappa di gioco rappresenta il terreno dove gli umani interagiscono, ho scelto di rappresentarla con una matrice di Tile, avrei potuto usare una Map avente come key la posizione e value la Tile, ma non l'ho ritenuto necessario.

Generazione della Mappa

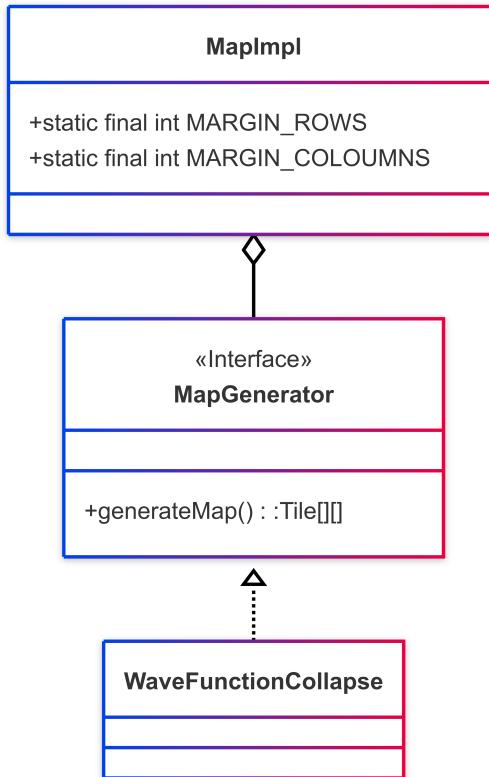


Figura 2.6: Schema UML della Generazione della Mappa.

La mappa di gioco 2.5 viene autogenerata per ogni capitolo, ho adottato il pattern strutturale **Strategy** per far fronte al problema dell'autogenerazione della mappa. L'interfaccia Strategy è **MapGenerator**, la quale ha solo una implementazione, ovvero **WaveFunctionCollapse**, che però possono aumentare nel caso in cui si decidesse di implementare un nuovo algoritmo di generazione di mappe 2d, questa soluzione è facilmente espandibile e risusabile grazie al pattern Strategy.

Wave Function Collapse

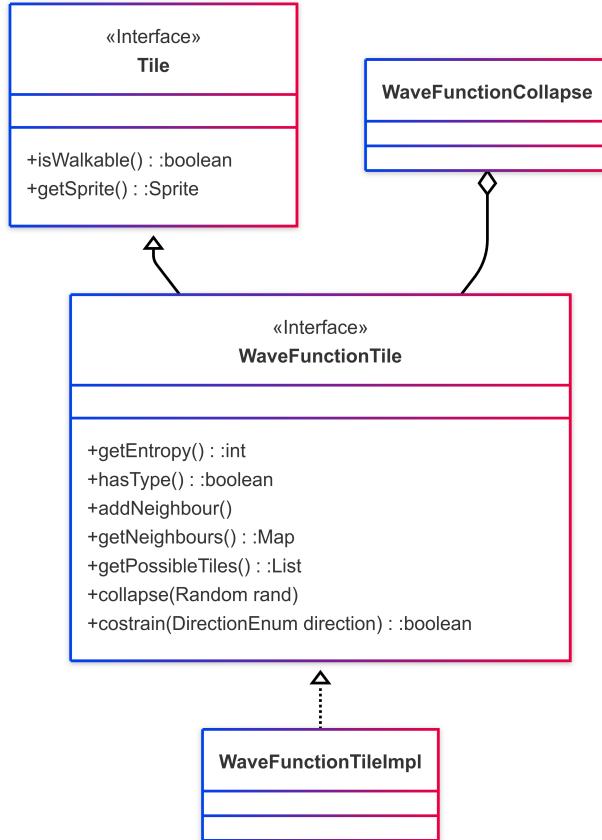


Figura 2.7: Schema UML della Wave Function Collapse.

Wave Function Collapse è un algoritmo di **Generazione Procedurale** per mappe 2d e 3d, questo algoritmo si basa su un Set di regole predefinite riguardanti la vicinanza tra i vari tipi di Tile (es. erba può stare solo vicino ad erba) e le varie possibilità di scelta di una Tile al, quindi i pesi del random per le singole Tile.

L'algoritmo consiste nel scegliere randomicamente una cella, dalla matrice della mappa, tra quelle con un'entropia minore, ovvero con il Set di possibili Tile adottabili, dalla cella stessa, di minor grandezza, e assegnarle una Tile ij modo casuale proprio da quest'ultimo Set, questa procedura si chiama **Collassamento** o **Collapse**. Una volta collassata una cella, si devono aggiornare le possibili Tile adottabili dai vicini semplicemente rimuovendo le Tile che non possono essere collegate con la Tile appena scelta.

Per questo motivo **Tile** ha una specializzazione **WaveFunctionTile** con

propria implementazione **WaveFunctionTileImpl**, usata da **WaveFunctionCollapse** per generare la mappa. Questo approccio rende invisibile a **MapImpl 2.6** che Tile in realtà possa essere una **WaveFunctionTile**, inoltre se in futuro si volesse implementare un'altro metodo di generazione per la mappa, si andrebbe direttamente a creare la specializzazione della tile per il nuovo metodo, quindi è facilmente scalabile.

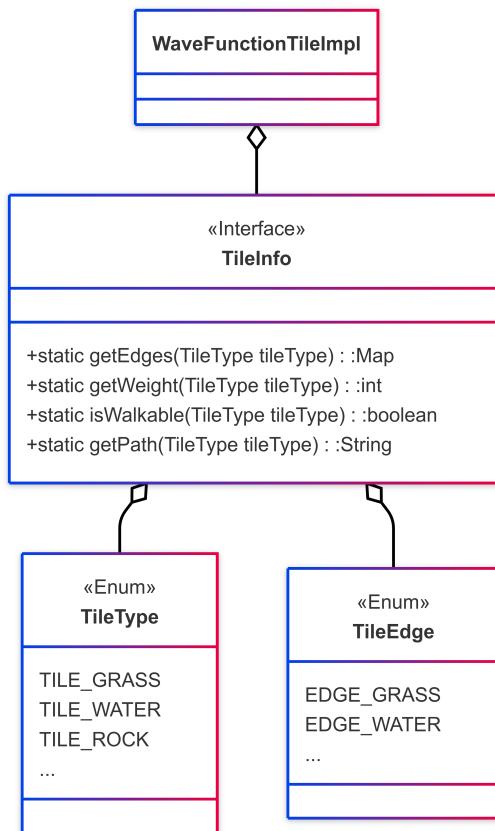


Figura 2.8: Schema UML delle regole delle Tile per Wave Function Collapse.

Le regole per ogni singola Tile sono definite in un'interfaccia **TileInfo**, inizialmente **TileInfo** non esisteva e quindi tutte le informazioni erano situate dentro **TileType**, ma ho voluto scorporare **TileType** per rispettare SRP e anche per incentivare il riuso, dato che **TileType** potrebbe essere utilizzato in futuro per altri metodi di generazione di mappe.

Collisioni Solide

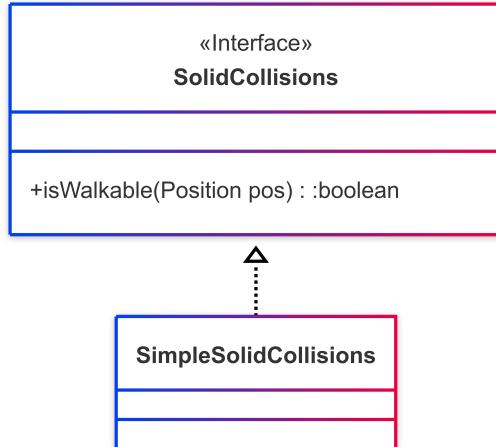


Figura 2.9: Schema UML delle Collisioni Solide.

Le collisioni solide sono una parte fondante del gioco e permettono agli umani di non poter camminare su certe porzioni di mappa (riguardano solo le collisioni umano-mappa e non umano-umano).

Ho scelto di adottare il pattern strutturale **Strategy** per l'implementazione delle collisioni solide. L'interfaccia Strategy è **SolidCollisions** e l'unica sua implementazione è **SimpleSolidCollision**, ovviamente avendo adottato il pattern Strategy è molto facile creare altre implementazioni per le collisioni solide (es. collisioni cerchi-rettangoli, rettangoli-rettangoli, etc...) e quindi è facilmente estendibile, ma anche riusabile.

L'implementazione che ho scelto è molto semplice, ma efficace, infatti per capire se un umano può camminare su una Tile o meno si guarda solo se quella Tile è camminabile o meno. Questa soluzione è poco dispendiosa, dato che non necessita di calcoli di distanze o aree, ed è stata scelta per il poco impatto nelle prestazioni, data la necessità di estrarre più prestazioni per migliori scenari nelle fasi avanzate del gioco. Altre possibili soluzioni più raffinate erano sviluppare le collisioni con cerchi (area di un umano) e rettangoli (area di una Tile) oppure solo rettangoli, ma sono state scartate per favorire le prestazioni.

2.2.3 Simone Mazzacano

Menu

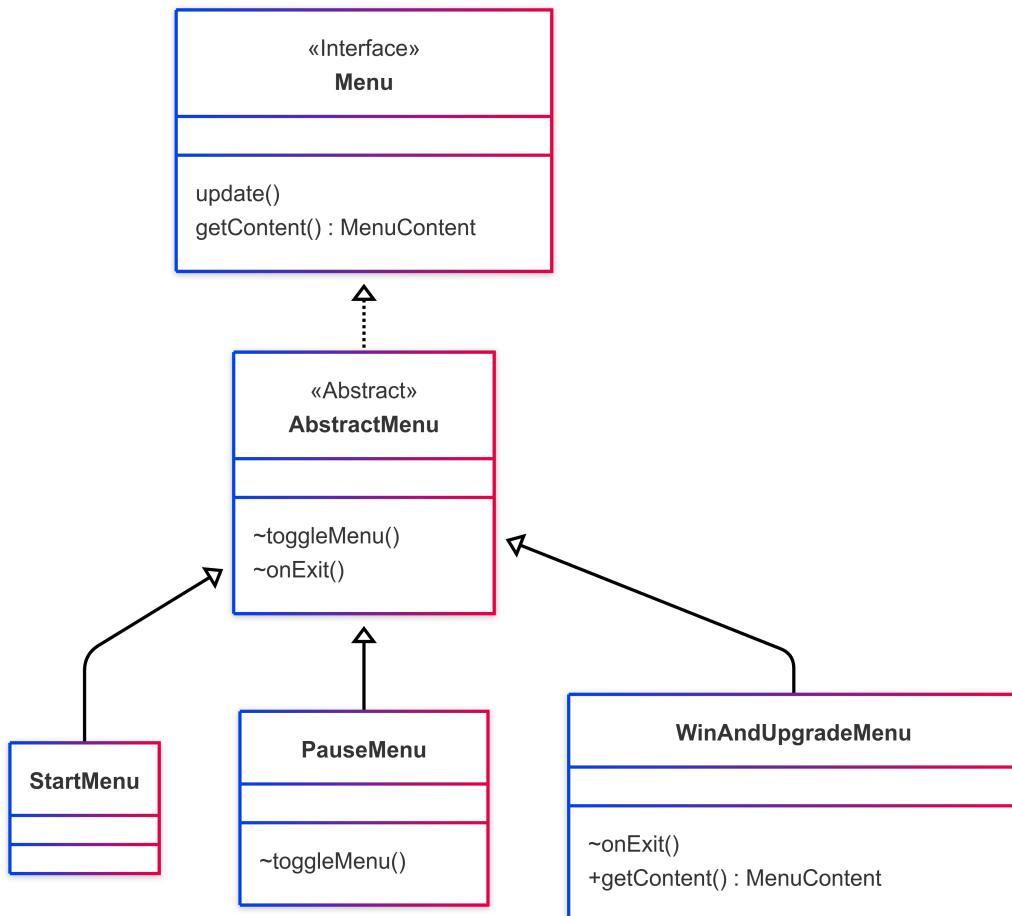


Figura 2.10: Schema UML gestione menu. I menu mostrati non sono gli unici presenti nel gioco, ma sono quelli più significativi per mostrare l'implementazione di **Template Method Pattern**. Si noti come nelle classi concrete vengano implementati alcuni metodi di **AbstractMenu**.

Durante la fase di sviluppo mi sono accorto che i menu condividevano molto codice, ma avevano anche delle differenze significative. Ho adottato perciò una interfaccia **Menu** implementata dalla classe abstract **AbstractMenu**, che rappresenta il comportamento di un menu generico. Per modellare i vari menu è stato usato il **Template Method Pattern**, usando come metodo template `update()` che si occupa di gestire l'input dell'utente e di aggiornare lo

stato del menu e del gioco di conseguenza. I vari menu sono implementati come classi concrete che estendono `AbstractMenu` e possono implementare gli altri metodi in modo specifico per il loro comportamento, oppure lasciare che la classe astratta gestisca il comportamento predefinito. Ho valutato l'uso di pattern creazionali come il Factory Method, ma ho optato per questa soluzione in quanto i menu hanno spesso bisogno di funzioni di utilità customizzate, potenzialmente complesse, che non sarebbero state facilmente implementabili con un Factory Method. Per modellare il comportamento delle opzioni selezionabili da parte dell'utente, ho adottato una classe `MenuItemOption`.

MenuItemOption

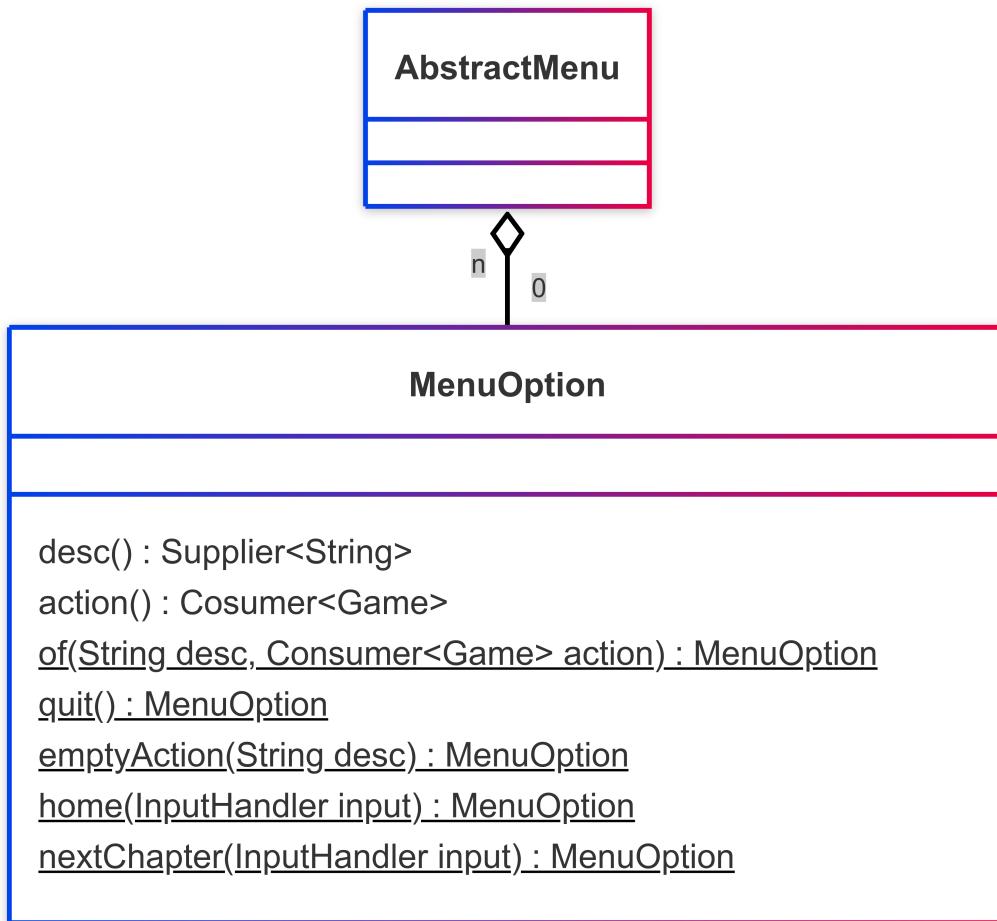


Figura 2.11: Schema UML gestione delle opzioni dei menu.

La classe `MenuOption` contiene una descrizione e un'azione da eseguire quando l'utente seleziona l'opzione. In questo modo, ognuna può avere un comportamento specifico senza dover modificare il codice del menu stesso. Questa classe è stata implementata come una **Static Factory**, in modo da semplificare la creazione di nuove opzioni di menu e non ripetere codice tra menu che necessitano della medesima opzione. Inoltre ciò permette di avere dei "named constructors" che rendono più chiaro il codice di creazione delle opzioni.

MenuContent

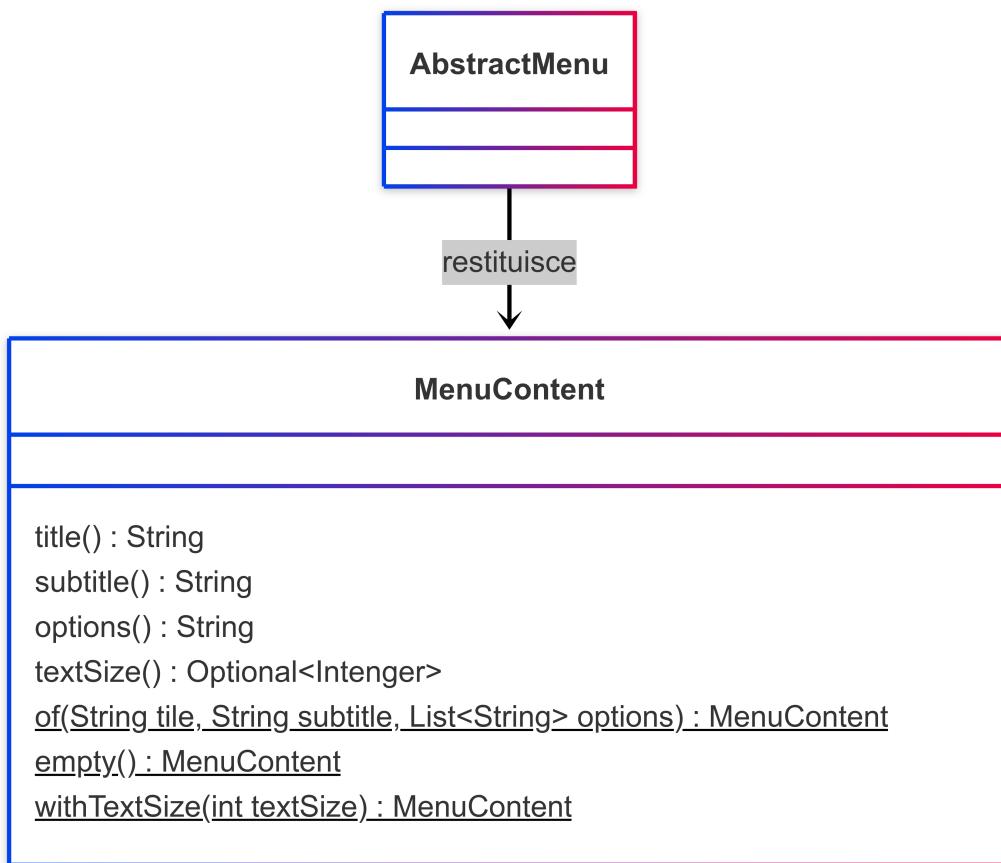


Figura 2.12: Schema UML gestione del contenuto dei menu.

Per fornire allo `screen` le informazioni necessarie per visualizzare i menu, ho implementato una classe `MenuContent`, anch'essa una Static Factory.

SicknessManager

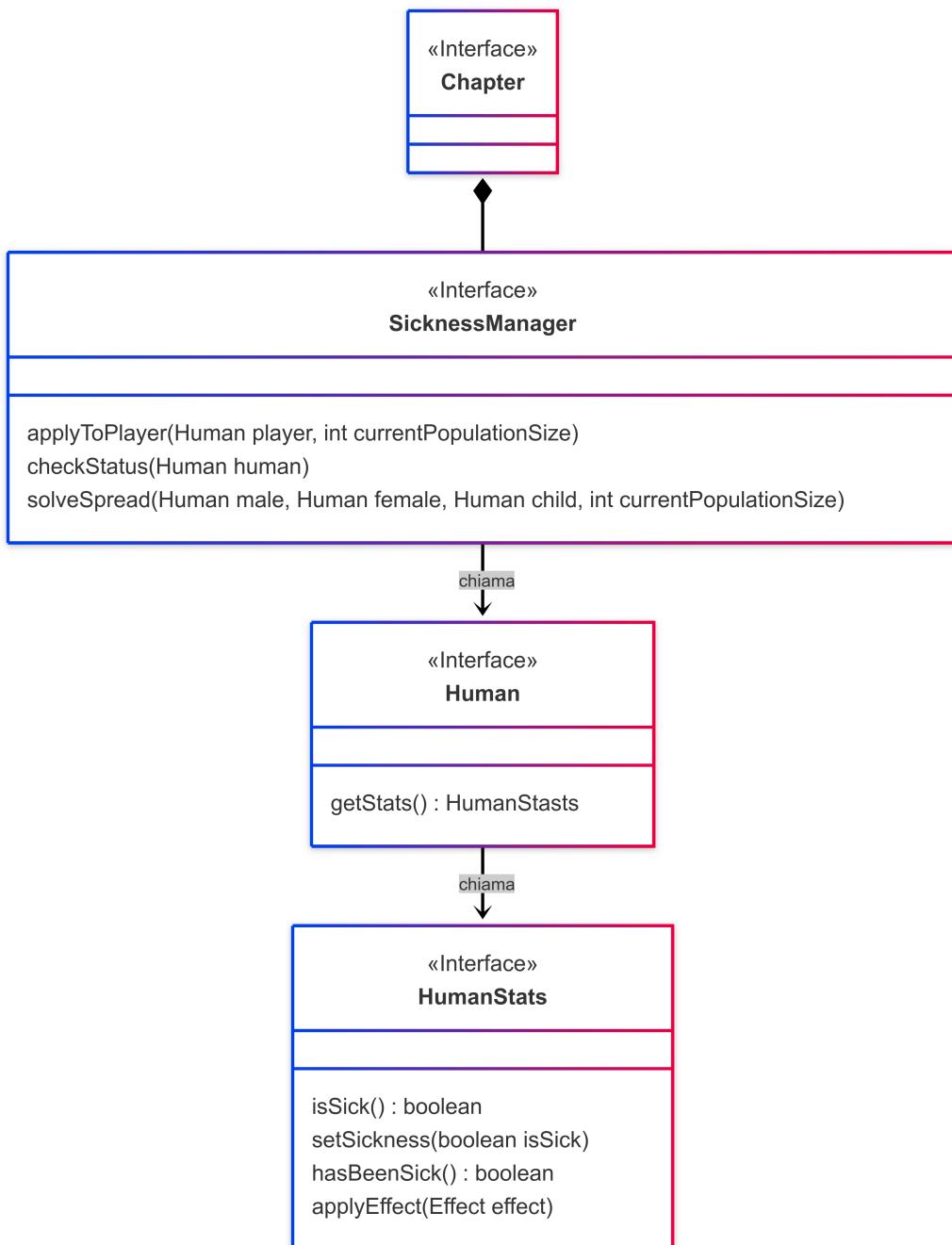


Figura 2.13: Schema UML gestione della malattia.

Per gestire la malattia degli umani ho implementato una classe `SicknessManager` che si occupa di:

- Controllare che la percentuale di umani minima per contrarre malattie sia raggiunta;
- Fare ammalare il player in modo casuale;
- Gestire la propagazione della malattia tra gli umani dopo una riproduzione, controllando che ognuno si ammali una sola volta;
- Applicare gli effetti della malattia sugli umani;
- Rimuovere la malattia dopo che gli effetti sono scaduti.

Timer

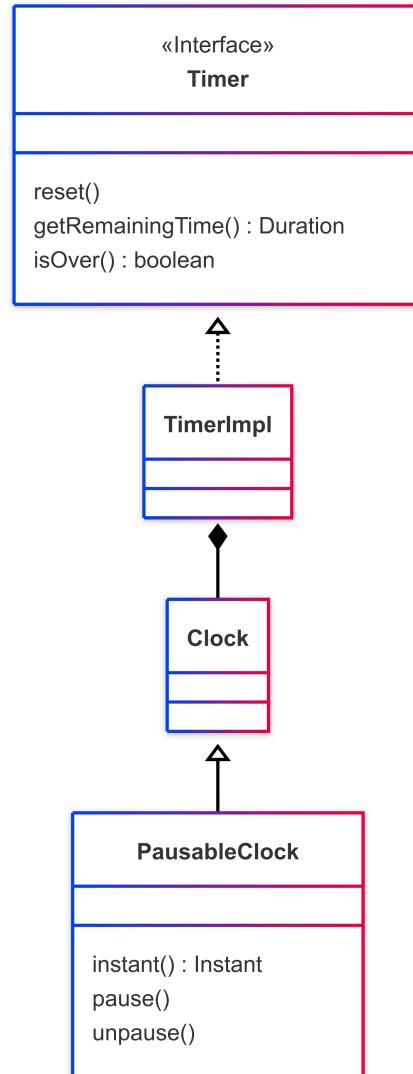


Figura 2.14: Schema UML gestione del timer.

Per gestire il timer del gioco ho implementato una classe **Timer**, utilizzata dal capitolo per verificare se il giocatore ha perso a causa dello scadere del tempo. Il valore corrente del timer viene mostrato a schermo. La classe **Clock** nello schema UML 2.14 corrisponde a `java.time.Clock`. Essa fornisce al timer le informazioni sull'orario attuale, delegando la gestione dello scorrere del tempo e semplificando le responsabilità di **Timer**. Tramite **PausableClock**, il tempo può essere messo in pausa e ripreso, permettendo così di sospen-

dere il timer di gioco. Ho scelto di non utilizzare soluzioni esistenti come `java.util.Timer` poiché non si adattavano bene al nostro caso d'uso, che richiede un timer facilmente suspendibile e riprendibile, mantenendo però un'interfaccia semplice e chiara. Inoltre, in questo modo più classi possono condividere lo stesso `PausableClock`, mantenendo la coerenza del tempo di gioco tra i vari componenti.

View

Oltre che alla visualizzazione dei menu, mi sono occupato di quella del timer e del contatore della popolazione.

La soluzione adottata ha permesso di risolvere i seguenti requisiti:

- Il timer deve rimanere in alto allineato al centro dello schermo
- Il contatore della popolazione deve rimanere in alto allineato a destra dello schermo
- Il testo dei menu deve essere centrato e rimanere al centro dello schermo e poter supportare i newline nei sottotitoli
- Il tutto deve essere ridimensionabile in modo da adattarsi a finestre di dimensioni diverse

La soluzione adottata è stata quella di creare un `JPanel` implementato da una classe `TopPanel` che contiene il timer e il contatore della popolazione, che usa `SpringLayout` per gestire il posizionamento. Anche per i menu ho usato un `JPanel` con `SpringLayout` che mantiene il `JTextPane` centrato e quest'ultimo gestisce i newline e il centramento del testo al suo interno. `TopPanel` e il `JPanel` del menu sono aggiunti allo `screen` tramite un `GridLayout`, in modo da mantere le proporzioni corrette e permettere il ridimensionamento della finestra.

2.2.4 Elia Mami

Statistiche del player

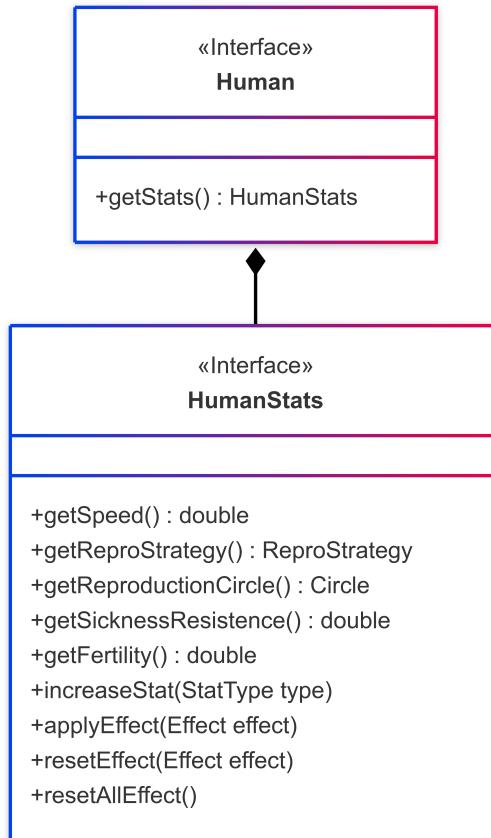


Figura 2.15: Schema UML della HumanStats.

Per descrivere le **statistiche** del player quali: **speed**, **sickness resistance**, **reproduction range** e **fertility**; ho creato *HumanStats*. *HumanStats* implementa tutti i metodi necessari per accedere alle **statistiche**, per incrementarle e per applicare e rimuovere effetti durante il corso della partita.

PowerUp

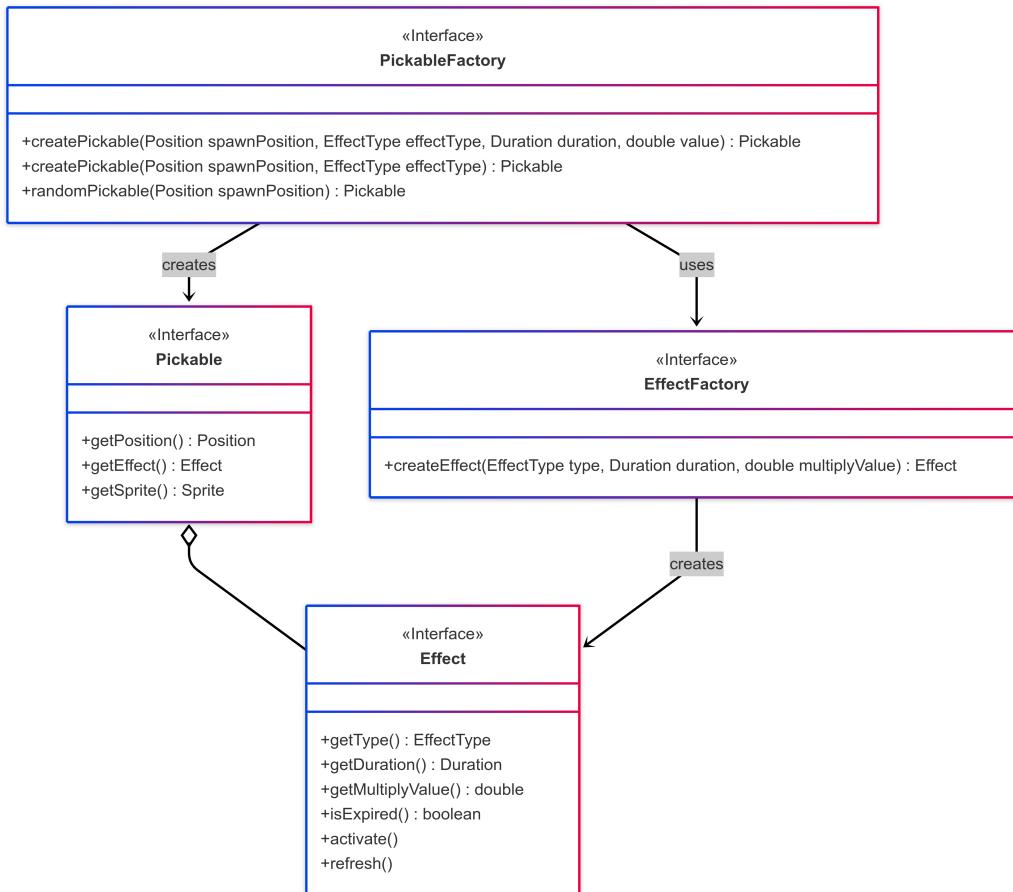


Figura 2.16: Schema UML della PickableFactory.

Per i **PowerUp** ho deciso di utilizzare il *SimpleFactory pattern*, questo mi permette in futuro di aggiungere metodi per creare dei *Pickable* personalizzati in aggiunta a quelli già presenti. I **PowerUp** sono dei *Pickable* ovvero degli oggetti raccoglibili dal player con associato un *Effect*, quest'ultimo ha il compito di descrivere qual è la **statistica** da modificare, di quanto sarà il moltiplicatore da applicare e la durata dell'*effetto*. Anche per gli *effetti* ho usato il *SimpleFactory pattern* sempre per permettere in futuro di aggiungere *effetti* personalizzati.

Gestione dei PowerUp

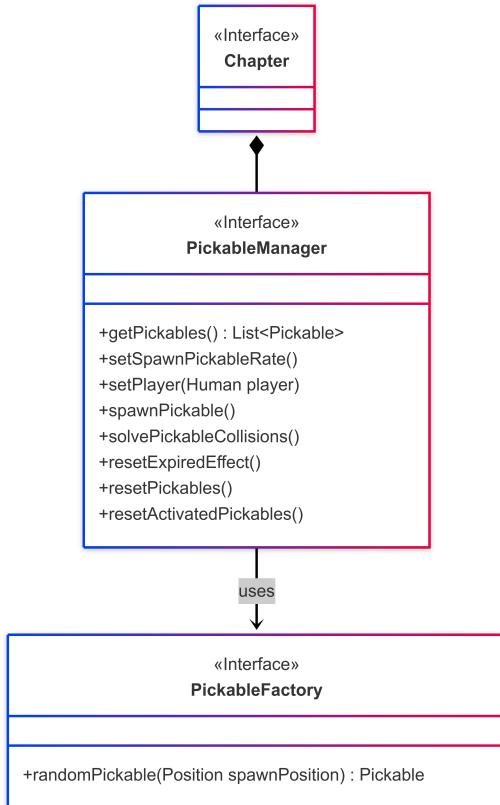


Figura 2.17: Schema UML del PickableManager.

Per la gestione dei **PowerUp** ho creato *PickableManager* che ha il compito di generare quando è il momento il *Pickable* all'interno della mappa, rivelare e risolvere le collisioni con il player e gestire la durata dei **PowerUp** attivati dal player. Include inoltre i metodi per resettare i *Pickable* generati nella mappa, quelli attivati in precedenza e lo spawn timer dei *Pickable*.

Salvataggio statistiche e chapter

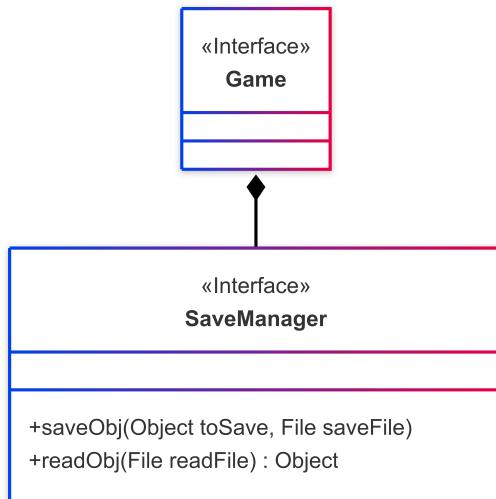


Figura 2.18: Schema UML del SaveManager.

Per il salvataggio delle **statistiche** e del **chapter** ho implementato *SaveManager* che si occupa di leggere da file e scrivere su file tramite la serializzazione.

Incremento delle statistiche

Per l'incremento delle **statistiche** tramite la schermata *WinAndUpgradeMenu* chiamo in Game *CheckAndIncreaseStats* passandogli il tipo di **statistica** che voglio incrementare, lui internamente controlla se ha gli *SkillPoint* necessari per farlo e in caso li avesse incrementa la **statistica** interessata e diminuisce gli *SkillPoint*.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

3.1.1 Eric Aquilotti

Humans

Per quanto riguarda la componente *Humans*, ho deciso di testare le *Strategies*, in particolare quelle che richiedono un periodo di *cooldown*, al fine di verificare il corretto funzionamento dei clock.

Collisioni

Per quanto concerne le collisioni, ho testato il *QuadTree* per verificarne la correttezza nelle operazioni di ricerca (*query*). Inoltre, ho condotto test sulle classi utilizzate per la gestione delle collisioni, in particolare *Circle* e *Rectangle*.

Temporizzazione

Per le temporizzazioni ho deciso di testare tutte le classi coinvolte nella logica di cooldown, con particolare attenzione al loro comportamento in presenza di pause. In questo modo è possibile verificare che le funzionalità temporali siano coerenti anche quando il tempo viene sospeso e successivamente ripreso.

3.1.2 Francesco meloni

- **SoldiCollisionsTest:** test per il corretto funzionamento delle collisioni solide 2.9

- **TileTest:** vengono testate le funzionalità dell'unica implementazione di Tile, ovvero WaveFunctionTile 2.7.

3.1.3 Simone Mazzacano

Malattia

Ho deciso di testare la classe SicknessManager (vedi 2.13), verificando che:

- la malattia si diffonda correttamente dopo la riproduzione;
- la malattia venga rimossa al termine del suo effetto;
- l'effetto della malattia riduca le statistiche del giocatore.

Timer

Ho verificato che il Timer (vedi 2.14) restituisca correttamente il tempo residuo e segnali se il tempo è scaduto in varie situazioni:

- al momento dell'istanziazione,
- immediatamente prima e dopo il raggiungimento del tempo obiettivo,
- durante la pausa quando il Clock è di tipo PausableClock,
- al richiamo di `reset()`.

HumanMockup

Ho aggiunto una classe *HumanMockup* per facilitare la crezione di oggetti *Human* durante i test. Questa classe consente di creare facilmente oggetti *Human* con parametri predefiniti, semplificando la scrittura dei test e migliorando la leggibilità del codice.

3.1.4 Elia Mami

- **HumanStats:** test per il corretto funzionamento dell'applyEffect, resetEffect, resetAllEffect, increase di ogni statistica e la corretta creazione dell'oggetto tramite la lista degli upgrade.
- **Effect:** test per il corretto funzionamento della creazione dell'Effect tramite EffectFactory, di activate, di isExpired e refresh.

- **PickableManager**: test per il corretto funzionamento di spawnPickable.
- **SaveManager**: test per il corretto funzionamento di saveObj, readObj e throw IOException quando necessario.

3.2 Note di sviluppo

3.2.1 Eric Aquilotti

Per ogni feature avanzata del linguaggio è presente solo un permalink anche se talvolta è presente in più parti del codice.

- uso di librerie della JDK moderna, come Clock, Duration e Instant:
<https://github.com/Eric-Aquilotti/OOP24-vitanova/blob/6cfe0a3a03e43bc33c31862f9f4292814e754973/src/main/java/it/unibo/common/PausableClock.java#L12>
- largo uso di lambda expressions: <https://github.com/Eric-Aquilotti/OOP24-vitanova/blob/6cfe0a3a03e43bc33c31862f9f4292814e754973/src/main/java/it/unibo/model/human/strategies/movement/MovStrategyFactoryImpl.java#L46C16-L51C10>
- largo uso di streams: <https://github.com/Eric-Aquilotti/OOP24-vitanova/blob/da4a2d0d1defc761e9c63ebde994e1fa7f971255/src/main/java/it/unibo/model/chapter/collisions/CollisionSolver.java#L43-L57>
- Uso di Optional <https://github.com/Eric-Aquilotti/OOP24-vitanova/blob/6cfe0a3a03e43bc33c31862f9f4292814e754973/src/main/java/it/unibo/view/screen/ScreenImpl.java#L213-L224>
- Uso di generici <https://github.com/Eric-Aquilotti/OOP24-vitanova/blob/a827886c868191de51b1926420c89b7816f6523a/src/main/java/it/unibo/model/chapter/quadtree/QuadTreeImpl.java#L15>
- Uso di Predicate <https://github.com/Eric-Aquilotti/OOP24-vitanova/blob/a827886c868191de51b1926420c89b7816f6523a/src/main/java/it/unibo/model/human/strategies/reproduction/CooldownReproductionPredicate.java#L14>
- Uso di Supplier <https://github.com/Eric-Aquilotti/OOP24-vitanova/blob/6ca1feb907fe8420926cc0bde4601ea6fe2d0473/src/main/java/it/unibo/model/human/strategies/reproduction/ReproStrategyFactoryImpl.java#L48>

La base di implementazione del QuadTree è stata presa da un mio altro progetto, è stata riadattata da JavaScript a java. Link alla repo: <https://github.com/Eric169/presentazioneStruttureDati/blob/master/QuadTree/QuadTree.js>

3.2.2 Francesco Meloni

Utilizzo di Generici

Permalink: <https://github.com/Eric-Aquilotti/OOP24-vitanova/blob/fd3e0978ebe9aa10deeb40f3606b89e9ec27f186/src/main/java/it/unibo/view/sprite/Sprite.java#L250-L265>

Utilizzo di Optional

Usati in altri punti. PermaLink di un esempio di utilizzo: <https://github.com/Eric-Aquilotti/OOP24-vitanova/blob/fd3e0978ebe9aa10deeb40f3606b89e9ec27f186/src/main/java/it/unibo/model/tile/wavefunction/WaveFunctionTileImpl.java#L59-L63>

Utilizzo di Lambda expression e Stream

Usati in molte occasioni. PermaLink di un esempio di utilizzo: <https://github.com/Eric-Aquilotti/OOP24-vitanova/blob/fd3e0978ebe9aa10deeb40f3606b89e9ec27f186/src/main/java/it/unibo/model/tile/wavefunction/WaveFunctionTileImpl.java#L116-L119>

Utilizzo di BiFunction

Permalink: <https://github.com/Eric-Aquilotti/OOP24-vitanova/blob/7e60673955bf28e47b915e0b88f05f191d00baff/src/test/java/it/unibo/model/human/SolidCollisionsTest.java#L42-L49>

Utilizzo di risorse esterne

Link repository github da cui sono state prese le immagini delle Tile e parte della logica dell'algoritmo Wave Function Collapse: <https://github.com/CodingQuest2023/Algorithms>

3.2.3 Simone Mazzacano

- Largo uso di lambda expressions: <https://github.com/Eric-Aquilotti/OOP24-vitanova/blob/31026369e56b2195590e548fad3bff1a8183068e/src/main/java/it/unibo/view/menu/MenuOption.java#L76C45-L79C12>
- Largo uso di Consumer <https://github.com/Eric-Aquilotti/OOP24-vitanova/blob/31026369e56b2195590e548fad3bff1a8183068e/src/main/java/it/unibo/view/menu/GameOverMenu.java#L20C35-L23C15>
- Uso di Stream: <https://github.com/Eric-Aquilotti/OOP24-vitanova/blob/31026369e56b2195590e548fad3bff1a8183068e/src/main/java/it/unibo/view/menudisplay/MenuDisplay.java#L101C1-L104C45>
- Uso di Optional <https://github.com/Eric-Aquilotti/OOP24-vitanova/blob/31026369e56b2195590e548fad3bff1a8183068e/src/main/java/it/unibo/view/screen/ScreenImpl.java#L180C1-L180C75>
- Uso di BiConsumer <https://github.com/Eric-Aquilotti/OOP24-vitanova/blob/31026369e56b2195590e548fad3bff1a8183068e/src/main/java/it/unibo/model/human/sickness/SicknessManagerImpl.java#L109>

3.2.4 Elia Mami

- Uso di lambda expressions: Usate in altri punti. Permalink di esempio: <https://github.com/Eric-Aquilotti/OOP24-vitanova/blob/f23f4189a388716e7fe9dc39855b2d6473424b5f/src/main/java/it/unibo/view/menu/WinAndUpgradeMenu.java#L23-L25>
- Uso di Predicate: Permalink: <https://github.com/Eric-Aquilotti/OOP24-vitanova/blob/f23f4189a388716e7fe9dc39855b2d6473424b5f/src/main/java/it/unibo/model/pickable/manager/PickableManagerImpl.java#L62-L64>
- Uso di Function Permalink: <https://github.com/Eric-Aquilotti/OOP24-vitanova/blob/f23f4189a388716e7fe9dc39855b2d6473424b5f/src/main/java/it/unibo/view/menu/WinAndUpgradeMenu.java#L47>
- Uso di Stream: Usati in altri punti. Permalink di esempio: <https://github.com/Eric-Aquilotti/OOP24-vitanova/blob/f23f4189a388716e7fe9dc39855b2d6473424b5f/src/main/java/it/unibo/model/pickable/manager/PickableManagerImpl.java#L65-L69>

- Uso di `Optional` Usati in altri punti. Permalink di esempio: <https://github.com/Eric-Aquilotti/OOP24-vitanova/blob/012f9d4b09ac4ed16bc7495a04csrc/main/java/it/unibo/model/effect/EffectFactoryImpl.java#L29>
- Uso di `Serializable` Permalink: <https://github.com/Eric-Aquilotti/OOP24-vitanova/blob/f23f4189a388716e7fe9dc39855b2d6473424b5f/src/main/java/it/unibo/model/saveobject/SaveObjectImpl.java#L9>

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

4.1.1 Eric Aquilotti

Per la parte relativa agli *humans* 2.2, sono soddisfatto della scalabilità ottenuta grazie all'utilizzo delle *strategies*. Questo approccio rende semplice l'aggiunta di nuovi personaggi con modalità di movimento differenti. Mi immagino un villain che nasce in momenti casuali e insegue le persone uccidendole al contatto.

Sono anche molto contento delle ottimizzazioni implementate, in particolare dell'uso del *QuadTree* 2.3, che avevo sviluppato in passato per una presentazione alle scuole superiori e che si è rivelato utile anche in questo contesto.

La creazione della classe `PausableClock` e in generale della gestione dei cooldown 2.4 è stata un'attività particolarmente interessante e si è rivelata estremamente utile in diverse funzionalità implementate. Inizialmente pensavo esistesse già una soluzione simile in qualche libreria (e probabilmente è così), ma svilupparla autonomamente è stato comunque interessante.

Non credo che questo progetto avrà sviluppi futuri, ma un'aggiunta di effetti sonori e un miglioramento della grafica rappresenterebbero senz'altro dei miglioramenti.

Per quanto riguarda il lavoro di gruppo, abbiamo incontrato alcune difficoltà nella comunicazione, soprattutto a causa delle diverse esigenze personali e della disponibilità di tempo non sempre coincidente. Questo è sicuramente un aspetto su cui sarà importante migliorare nelle prossime collaborazioni. Nonostante ciò, con il tempo siamo riusciti a suddividere la mole di lavoro in modo abbastanza equo. In generale sono molto soddisfatto del risultato.

4.1.2 Francesco Meloni

Ho sviluppato la mia parte di lavoro in modo decente, se dovessi darmi un voto, in una scala da 1 a 10, sarebbe 8.5.

Per la parte delle Collisioni Solide 2.9 e della Generazione della Mappa 2.6, non ho punti a sfavore, sono contento della scalabilità offerta.

Riguardo alla parte della Mappa 2.5 l'unica cosa che avrei potuto cambiare è la gestione della matrice, che avrei potuto affidarla ad una Map per semplificare l'utilizzo di Stream su di essa.

Passando alla Wave Function Collapse 2.7, nel caso dovessi fare un'altra mappa autogenerata di questo tipo, probabilmente non mi affiderei a questo metodo, ma bensì ad un metodo con meno regole strutturali, inoltre la gestione di TileInfo 2.8, nello specifico le configurazioni nei metodi getEdges, getWeight e isWalkable, potevano essere scritte in un file di configurazione, per poi essere lette, e non direttamente nell'interfaccia.

I ruoli nel gruppo sono stati abbastanza equi, forse all'inizio non avevamo stimato correttamente il tempo necessario per implementare ogni feature, quindi alcune persone avevano troppo lavoro in più rispetto ad altre, ma con il tempo ci siamo distribuiti il lavoro in modo più equo possibile.

Nel caso in cui questo progetto dovesse continuare di sicuro applicherei le migliorie dette in precedenza, quindi Map per le tiles e file di configurazione per TileInfo, inoltre espanderei le Tile presenti in modo da avere una Mappa con più diversità, si potrebbero aggiungere degli effetti sonori ed anche una storia, infine migliorerei la grafica, l'avanzamento dei capitoli e il sistema di ricompense per stimolare il giocatore nel continuare a giocare.

4.1.3 Simone Mazzacano

Sono soddisfatto dell'implementazione dei menu (2.10), in quanto ritengo offrano una buona flessibilità per future estensioni, mantenendo al contempo un'ottima leggibilità del codice. Ho apprezzato il fatto che altri membri del gruppo abbiano potuto aggiungere nuovi menu o modificare quelli esistenti con facilità. Sono inoltre soddisfatto della gestione del timer (2.14), che presenta un'interfaccia semplice ma consente di integrare la funzionalità, non banale, della pausa durante il gioco. Questa soluzione ha beneficiato di una buona collaborazione, riutilizzando il `PausableClock` già impiegato in altre classi. Per quanto riguarda la gestione della malattia (2.13), forse sarebbe stato opportuno scorporare parte della logica di `SicknessManager` in `HumanStats` e valutare l'implementazione dell'Observer pattern tra il manager e gli oggetti `Human`. Tuttavia, ho ritenuto che tale modifica non avrebbe apportato un miglioramento significativo. Infine, nella gestione della view,

ho incontrato alcune difficoltà nell'aggiungere il timer e il contatore della popolazione mantenendoli allineati e ridimensionabili. Anche la gestione del testo nei menu richiede che rimanga centrato e che il newline sia gestito correttamente. Ritengo che la soluzione adottata sia un buon compromesso, ma sarei curioso di approfondire se esistano approcci migliori in Swing. Per quanto riguarda il lavoro di gruppo, inizialmente avremmo potuto distribuirci meglio i compiti più urgenti di modo che il progetto prendesse piede in maniera più graduale, tuttavia successivamente ci siamo allineati meglio e sono soddisfatto del risultato ottenuto. Non penso che il progetto avrà sviluppi futuri, ma potrebbe essere un buon esempio delle nostre capacità di teamwork e progettazione da presentare in futuro.

4.1.4 Elia Mami

Sono abbastanza soddisfatto del lavoro fatto, in quanto sono contento di aver integrato la *SimpleFactory* per i *Pickable* e per gli *Effetti* 2.16, permettendo in futuro, se voglio, di aggiungere **PowerUp** e **Effetti** personalizzati. Non sono invece molto soddisfatto delle *HumanStats* 2.15 perché penso che avrei potuto svilupparle in maniera migliore cercando di renderle più scalabili. Seppur semplice sono felice di essere riuscito a salvare su file le informazioni necessarie tramite la **serializzazione** 2.18. Perciò ritengo di aver fatto un buon lavoro in generale.

All'interno del gruppo penso di aver avuto un ruolo rilevante, cercavo sempre il confronto per cercare migliori implementazioni e se possibile chiamate di gruppo per aggiornarci e avere un dialogo. Ciò nonostante abbiamo comunque incontrato alcune difficoltà di comunicazione, ma penso siano state superate con facilità. Ho comunque intenzione di prendere nota di questa esperienza per evitarle in futuro se possibile, cercando di essere più chiaro nelle mie spiegazioni o posizioni. Sono però contento della suddivisione della mole di lavoro tra di noi, perché l'ho trovata molto equa ed adatta.

Non credo che questo progetto avrà sviluppi futuri, ma mi piacerebbe tornarci ogni tanto per vedere quanto mi posso spingere oltre nella creazione di nuovi pickable o effetti, e con magari l'aggiunta di altre piccole feature come suoni.

4.2 Difficoltà incontrate e commenti per i docenti

Non abbiamo commenti rilevanti per i docenti.

Appendice A

Guida utente

I *comandi* sono molto basilari, per muoversi all'interno dei menù basterà usare i tasti **W** ed **S** o le frecce direzionali **SU** e **GIÙ**, per cliccare l'opzione selezionata nei menù basterà premere il tasto **INVIO** o il tasto **BARRA SPAZIATRICE**. Per muoversi all'interno della mappa basterà sempre usare i tasti **W** (**avanti**), **S** (**indietro**), **D** (**destra**) ed **A** (**sinistra**) oppure le frecce direzionali **SU**, **GIÙ**, **DESTRA**, **SINISTRA**. All'interno della mappa spawneranno dei **PowerUp**:

- **Rosso**: aumenta la resistenza alle malattie.
- **Blù**: aumenta la velocità.
- **Viola**: aumenta il range di riproduzione.
- **Giallo**: aumenta la probabilità di far nascere femmine.

Appendice B

Esercitazioni di laboratorio

B.0.1 francesco.meloni6@studio.unibo.it

- Laboratorio 07: <https://virtuale.unibo.it/mod/forum/discuss.php?d=177162#p246037>
- Laboratorio 08: <https://virtuale.unibo.it/mod/forum/discuss.php?d=178723#p247229>
- Laboratorio 09: <https://virtuale.unibo.it/mod/forum/discuss.php?d=179154#p247877>
- Laboratorio 10: <https://virtuale.unibo.it/mod/forum/discuss.php?d=180101#p249326>