Global Outbreak
“Programmazione ad Oggetti”

Luca Carabini, Mattia Farabegoli, Giacomo Severi, Pietro Ventrucci

14 giugno 2023

Indice

Capitolo 1

Analisi

1.1 Requisiti

Il nostro obiettivo è quello di realizzare un videogioco ispirato a Plague Inc., un gioco strategico-simulativo in cui i giocatori controllano lo sviluppo di una malattia al fine di annientare l'intera popolazione umana. È importante sottolineare che il gioco non intende riflettere accuratamente la realtà degli sviluppi e la diffusione di malattie o pandemie. Non è una rappresentazione precisa o dettagliata delle dinamiche reali, ma è concepito come un semplice strumento di intrattenimento.

Requisiti funzionali

- La creazione di una malattia personalizzata per ogni partita, basata sulla scelta delle mutazioni che influenzano caratteri come letalità, gravità, resistenza e altri.
- Il contagio della popolazione deve poter avvenire attraverso mezzi di contagio realistici o meno.
- Il monitoraggio dello stato del gioco e le relative statistiche sul progresso e la diffusione della malattia, come numero di contagiati, morti, etc.
- La divisione del mondo in varie regioni che possano essere differenti e con caratteristiche peculiari, in grado quindi di rallentare o velocizzare la diffisione della malattia.
- L'azione attiva della popolazione, che potrebbe decidere di chiudere le frontiere, cominnciare lo sviluppo di una cura o altre azioni che possano rallentare la malattia.
- La possibilità di prendere decisioni strategiche, come il punto di partenza per la malattia e la relativa strategia di diffusione, la gestione delle risorse etc.

Requisiti non funzionali

- L'interfaccia utente del gioco dovrebbe essere abbastanza intuitiva da permettere al giocatore di prendere decisioni e interagire con il gioco in modo semplice e chiaro.
- Il gioco deve essere facilmente configurabile, senza la necessità di ricompilare o modificare il codice.
- L'applicativo dovrà essere efficiente nell'uso delle risorse.

1.2 Analisi e modello del dominio

Global Outbreak è un gioco nel quale il giocatore deve cercare di diffondere una malattia attraverso le varie regioni del mondo, tenendo conto delle possibili condizioni, come la differenza di clima o di sviluppo che queste potrebbero avere. Per fare ciò ha a disposizione una serie di mutazioni, che vanno ad influenzare il comportamento della malattia, come capacità infettiva, letalità, resistenza ed altre. L'obbiettivo della malattia è diffondersi e annientare totalmente la popolazione. Le mutazioni consentono al giocatore di adattarsi e migliorare le capacità della malattia nel corso del gioco. La diffusione della malattia avviene attraverso vari mezzi di trasmissione, come il contatto diretto, il trasporto aereo o marittimo e altri fattori. La scelta del modo di diffusione influenza la velocità e l'efficacia della propagazione della malattia. Le regioni reagiscono alla diffusione della malattia adottando misure di prevenzione e controllo. Possono chiudere i confini, investire nella ricerca di una cura, implementare politiche sanitarie o cercare di limitarne lo sviluppo usando iniziative interne. La cura infatti è il principale avversario del giocatore, e la priorità della ricerca potrebbe aumentare in relazione alla gravità della situazione, bisogna stare attenti a non farla sviluppare, altrimenti la sconfitta della malattia è inevitabile. In questo contesto già complicato l'unico aiuto al giocatore è dato dagli eventi naturali che possono uccidere la popolazione del mondo. Gli elementi che costituiscono il problema sono sintetizzati in Figura 1.1. Il requisito non funzionale riguardante l'efficienza nell'uso delle risorse non sarà svolto all'interno del monte ore previsto in quanto prevede un'analisi dettagliata delle performance dell'applicativo, ma sarà oggetto di miglioramenti futuri, bisognerà quindi che l'utente valuti autonomamete se sia in grado o meno di reggere l'applicativo.

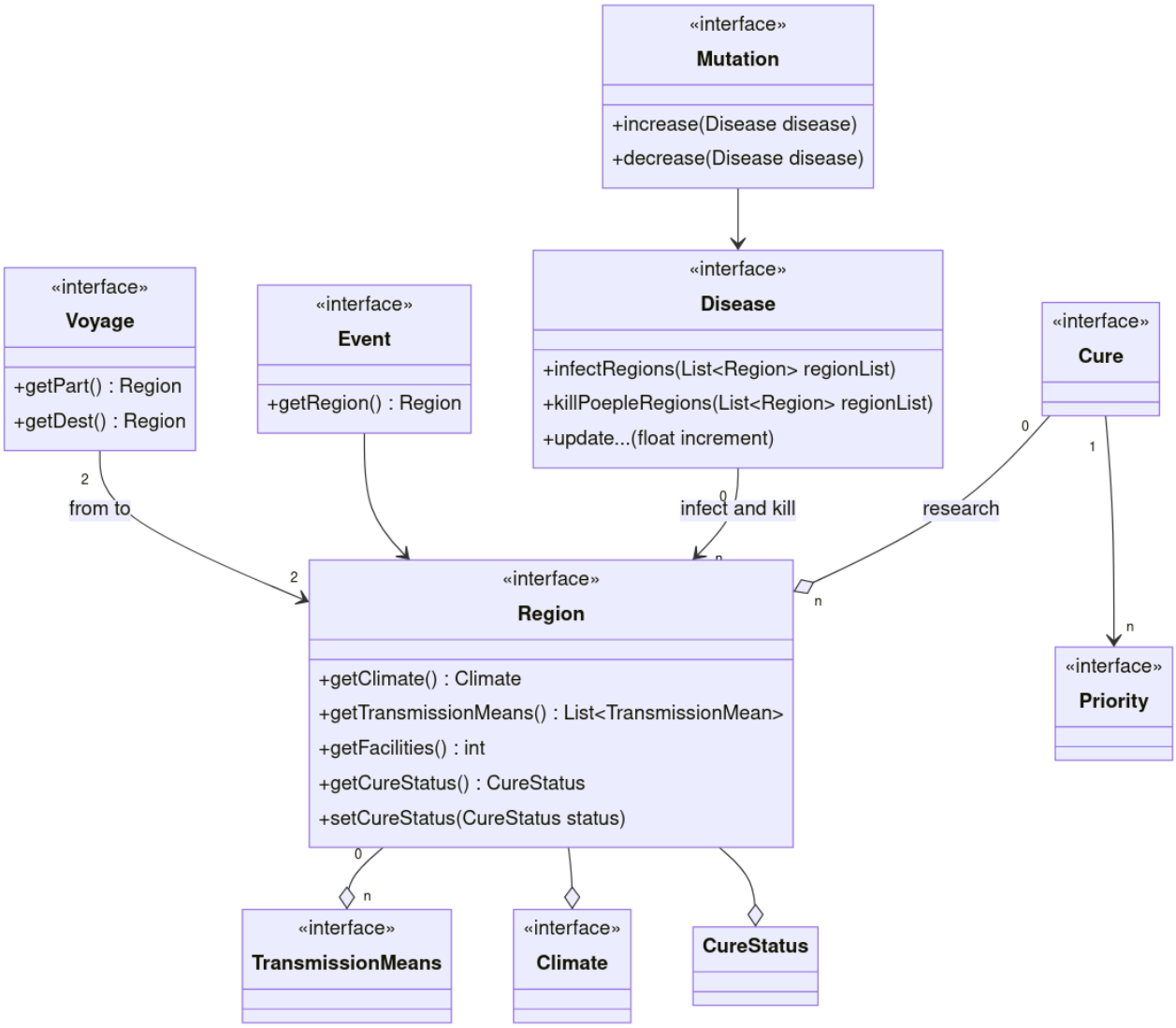

Figura 1.1: Schema UML dell'analisi del problema, con rappresentate le entità principali ed i rapporti fra loro

Capitolo 2

Design

2.1 Architettura

Per la realizzazione del gioco abbiamo utilizzato il pattern architetturale MVC(Model View Controller). La scomposizione nelle tre parti ci grantisce l'indipendenza necessaria a eventuali modifiche future La scelta di utilizzare questo pattern, combinata all'utilizzo di interfacce ci ha permesso di separare la logica del dominio applicativo dall'implementazione. Il Model definisce la logica di dominio dell'applicazione, questa parte è stata progettata per implementare i comportamenti delle principali entità in gioco. Il Controller si occupa di gestire l'interazione tra le varie entità. Il Controller manipola i dati del Model comununicando alla View le modifiche effettuate. Quest'ultima si occuperà di rappresentare graficamente tali modifiche. La View dialoga con il Controller notificando gli eventi scatenati dall'interazione con l'utente. Si è cercato di rendere la View indipendente da Model e Controller. Mentre il cambio della scena è gestito dallo Scene Manager.

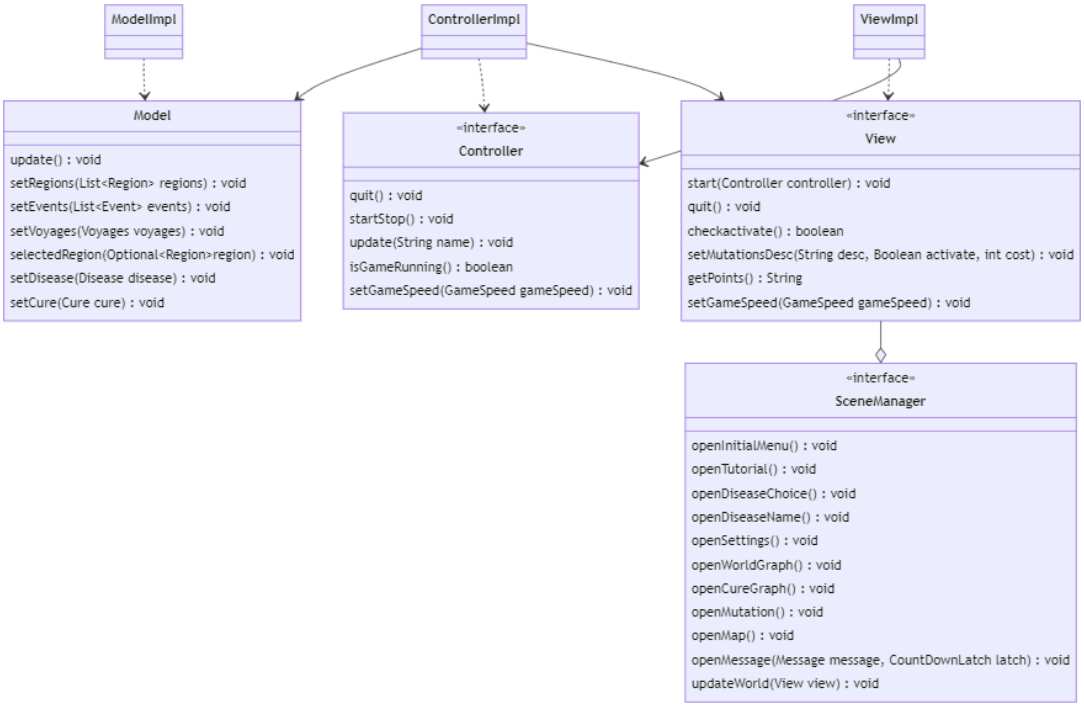

Figura 2.1: Schema UML architetturale di Global Outbreak

2.2 Design dettagliato

Luca Carabini

Nel prgetto, per quanto riguarda la parte model e controller i miei compiti principalisono stati: gestire la creazione delle regioni(o Stati), e i rispettivi viaggi tra di esse e la creazione di eventi catastrofici. Per quanto riguarda la parte grafica mi sono occupato della crazione della mappa di gioco e la fruizione di informazioni riguardanti la singola regione o il mondo intero.

Regione

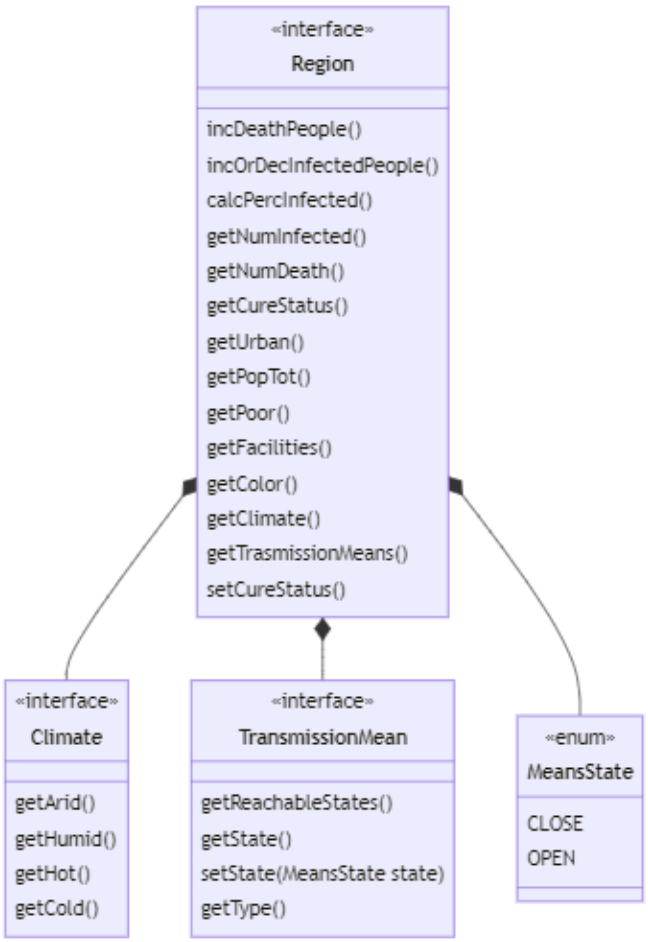

Figura 2.2: Schema UML della Regione

La Regione è una delle classi principali del gioco, infatti al suo interno contiene tutte le informazioni (es. umidità, temperatura ecc.), le quali verrano poi lavorate dalle altre classi. Il problema principale era come gestire le informazioni al suo interno, in particolar modo quelle del Clima e dei vari mezzi di trasporto. Inizialmente avevo messo tutto dentro la classe principale ma successivamente ho pensato di creare due ulteriori classi per dividere meglio le inormazioni, e rendere facilmente raggiungibili le informazioni dei trasporti. Un'altro problema è stato identificare la chiusura delle frontiere, che ho risolto crando un

enum. Un aspetto positivo della mia implementazione è la possibilità di aggiungere stati senza modificare il codice.

Voyages

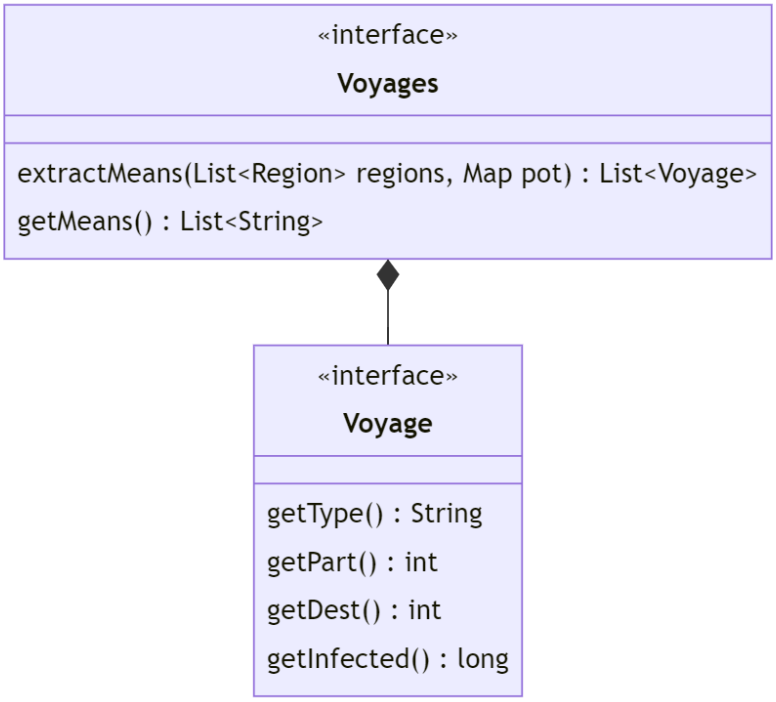

Figura 2.3: Schema UML dei Viaggi, e utilizzo del pattern Strategy

I Viaggi si occupano di infettare gli altri paesi, tramite i vari mezzi di trasporto. Ogni mezzo di trasporto ha un nome e un numero di posti. Il problema principale è stato capire con quale probabilità vengono estratti i mezzi infetti, l'ho risolto prendendo come probabilità di essere infetto il rapporto tra infetti e popolazione totale(alla quale dovrà sommare il rispettivo potenziamento). Inoltre ogni regione ha dei confini che vengono usati quanto il mezzo è "via terra", un un punto debole di questa soluzione è che cambiando mezzi di trasmissione la classe che estrae i viaggi andrà leggermente modificata.

Causa Evento

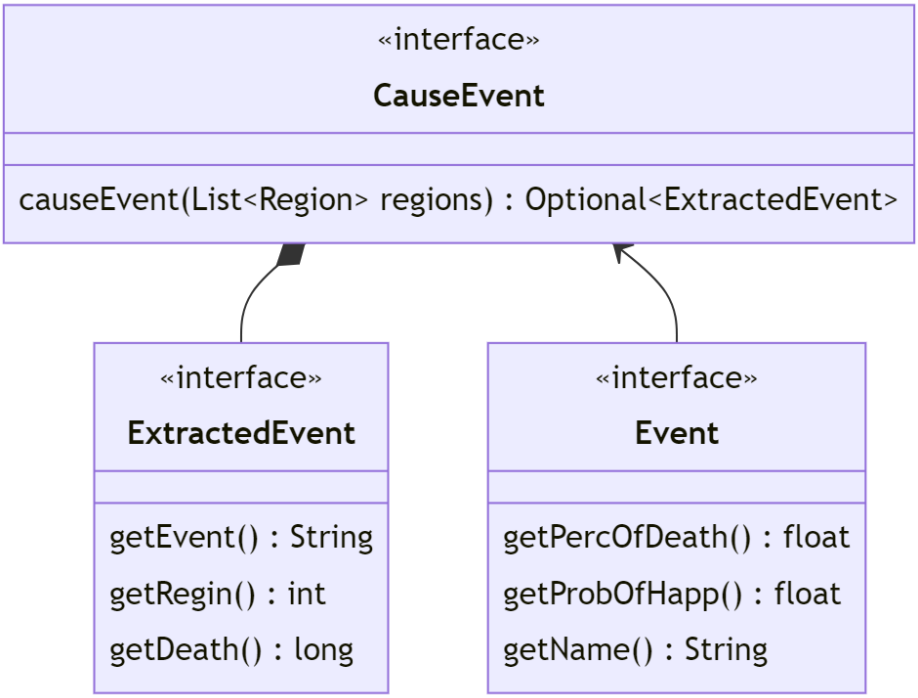

Figura 2.4: Schema UML degli Eventi

Causa evento si occupa di causare una catastrofe(es. Terremoto, Alluvione) quidi genera delle morti per causa naturale. Ogni evento ha una probabilità di avvenimento e una percentuale di morti che causa. È inoltre molto generale infatti si possono aggiungere quasi qualsiasi evento senza modificare il codice(ad eccezione degli eventi collegati con il mare es. Tsunami).

Mattia Farabegoli

Il mio compito è stato gestire la creazione di una malattia e implementare i suoi metodi di sviluppo e diffusione. Ho anche lavorato sulla creazione del menu iniziale, la gestione del cambio delle schermate, il sistema di punti per acquisire potenziamenti e la visualizzazione dei dati tramite grafici.

Malattia

Disease Factory

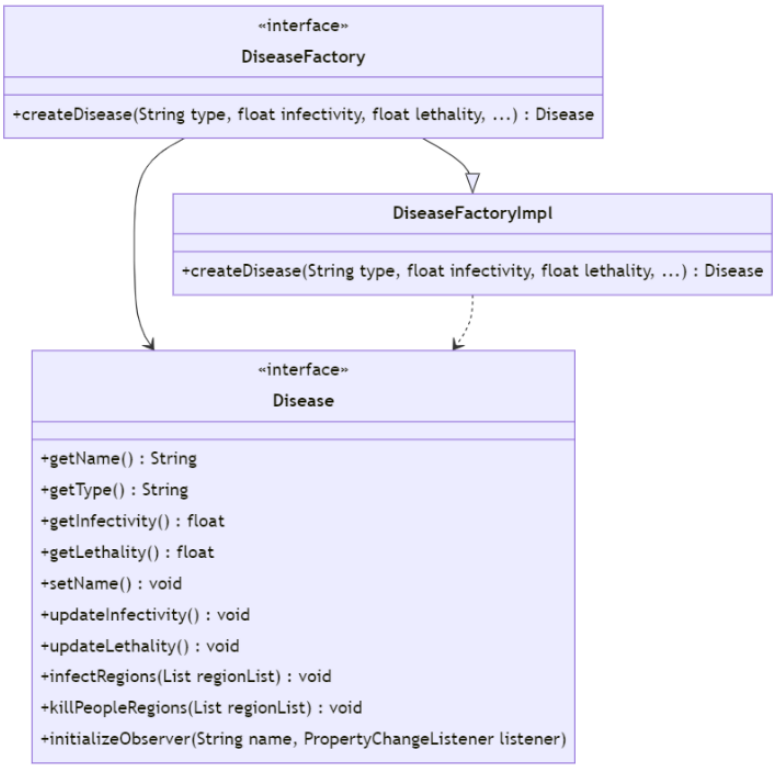

Figura 2.5: Schema UML di Disease, con pattern Factory Method

Per la creazione della malattia, ho scelto di utilizzare il *pattern Factory Method* 2.13. Il Factory Method consente di gestire contemporaneamente diverse varianti di virus all'interno della simulazione. Questo è particolarmente utile quando si desidera aggiungere nuovi tipi di malattie senza dover modificare direttamente le classi esistenti. Invece, è sufficiente creare una nuova sottoclasse del factory method che implementa la logica per creare l'oggetto rappresentante la nuova malattia. L'incapsulamento delle logiche di creazione delle malattie in un metodo factory dedicato promuove una migliore organizzazione del codice. Le logiche specifiche per la creazione di un particolare tipo di malattia possono essere implementate all'interno della corrispondente sottoclasse factory, separando efficacemente le responsabilità e garantendo una maggiore modularità.Inoltre, si riducono le dipendenze dirette tra le classi coinvolte nella creazione delle malattie. Le classi che richiedono un'istanza di un oggetto malattia non devono preoccuparsi di conoscere i dettagli specifici sulla sua creazione. Si basano semplicemente sull'interfaccia fornita dal metodo factory per ottenere l'oggetto desiderato.

Disease data

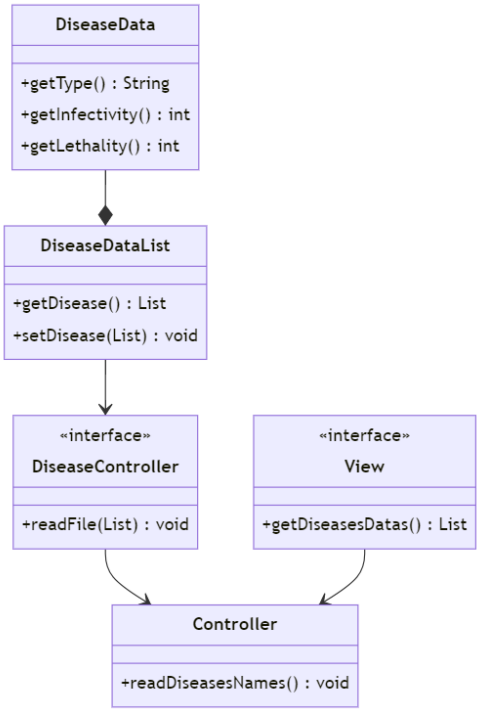

Figura 2.6: Schema UML di DiseaseReader

Ho creato DiseaseData e DiseaseDatalist per tenere traccia di tutte le informazioni delle malattie lette dal file Json, di modo da tenere un disaccoppiamento tra Disease e DiseaseData. Inoltre queste classi possono essere utilizzate, interagendo con DiseaseController, per visualizzare tutte le statistiche delle malattie durante la scelta del tipo nel menu iniziale (2.6).

Menu iniziale

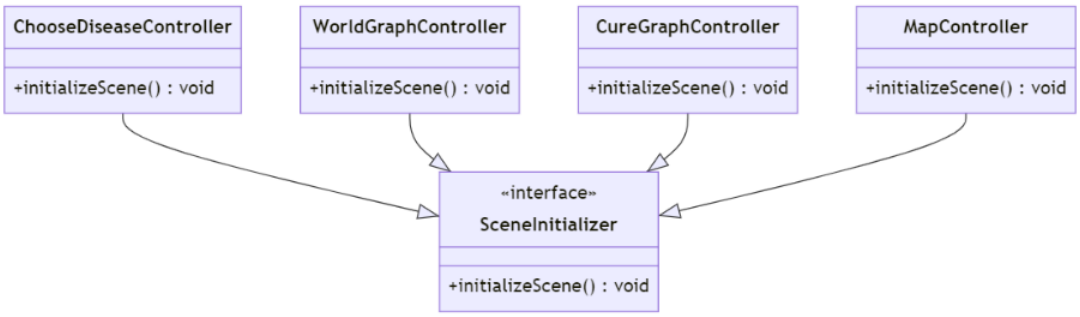

Figura 2.7: Schema UML di gestione delle scene

Per quanto riguarda il menu iniziale ho creato le due interfacce SceneLoader e SceneManager 2.7 di modo da favorire una netta separazione delle responsabilità, infatti la prima si occupa di gestire il caricamento dei FXML e di definire tutte le impostazioni dello Stage e della finestra, mentre la seconda si occupa di settare la scena sullo stage. Per i controller dei file FXML, invece, ho utilizzato il pattern Strategy, creando l'interfaccia SceneInitializer che poi viene implementata da alcuni controller che necessitano di inizializzare la loro scena tramite metodi diversi. In questo modo permetto di gestire in modo flessibile l'inizializzazione delle scene, utilizzando algoritmi diversi a seconda delle necessità.

InfoData, interfaccia informazioni e gestione dei punti

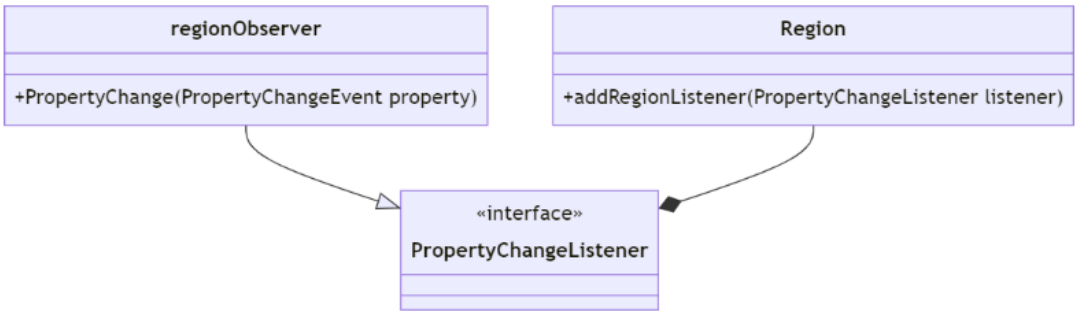

Figura 2.8: Schema UML per regionObserver

Per i dati generali, come per la gestione dei punti ho creato l'interfaccia InfoData. Per l'ottenimento ho implementato un PropertyChangeListener 2.8, essendo Observer deprecato. Il listener rimane in ascolto della classe Region, che invierà i dati sui nuovi infetti, indicando se è la prima volta che si infetta o no.

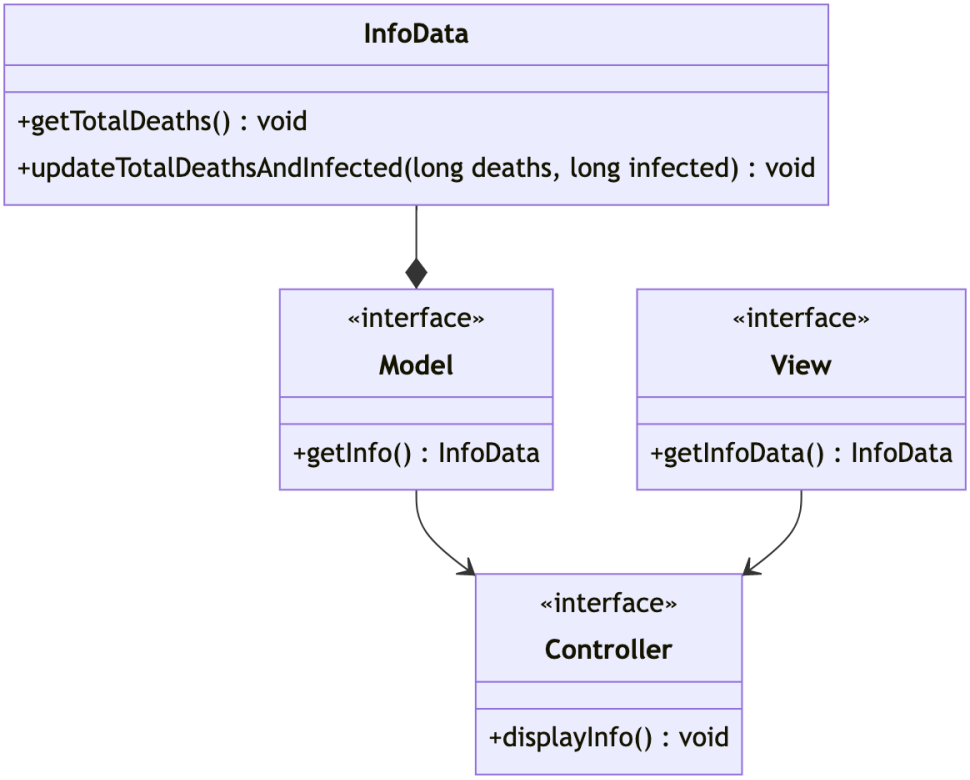

Figura 2.9: Schema UML per infodata display

Mentre per creare i grafici Ho suddiviso la parte del Model(InfoData) dalla parte dei grafici (View). In questo ho separato i concetti di dominio applicativo e dahsboard visiva delle informazioni, che comunicano attraverso il controller (2.9).

Giacomo Severi

Il mio ruolo all'interno del progetto si è concentrato sulla creazione dell'entità "Potenziamento". Questo ha coinvolto sia la gestione del potenziamento che la gestione delle impostazioni per il caricamento.

Potenziamenti

Mutation Factory

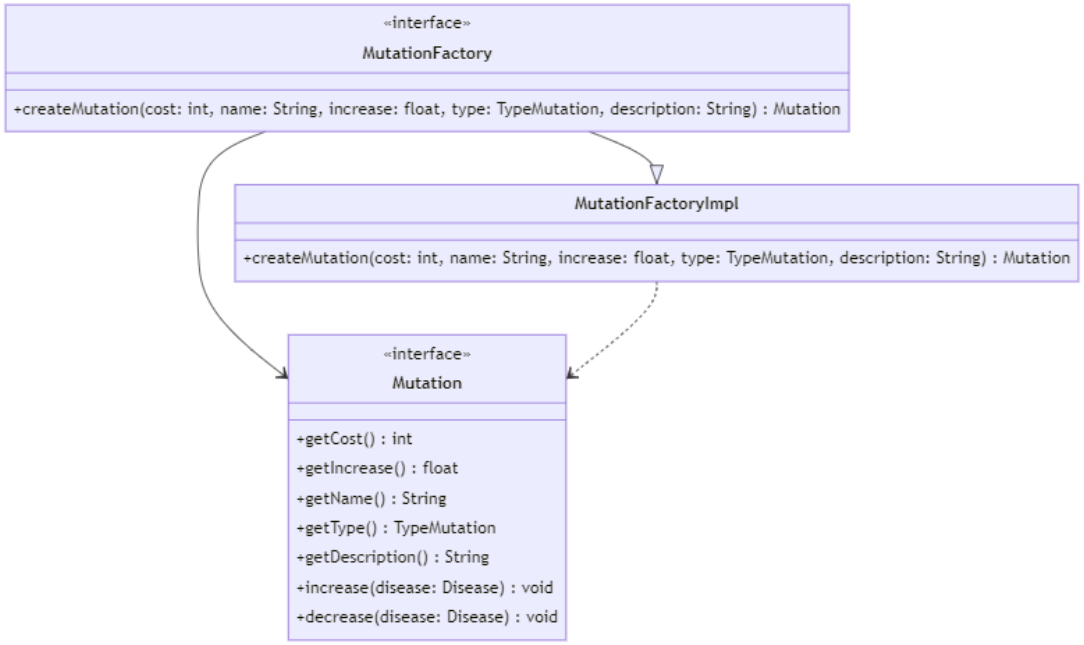

Figura 2.10: Schema UML di Mutation Factory

L'interfaccia "Mutation" definisce i metodi e le proprietà che devono essere implementati dalle classi di mutazione. Include metodi per ottenere il costo, l'incremento, il nome, il tipo e la descrizione della mutazione, nonché metodi per aumentare e diminuire i parametri associati alla mutazione in una malattia. Ho deciso di utilizzare una factory perché il Factory Method è un pattern flessibile e potente che fornisce astrazione e separazione delle responsabilità nella creazione degli oggetti. Questo approccio può migliorare l'estensibilità, la manutenibilità e la modularità del codice per la gestione della creazione di oggetti complessi all'interno di un'applicazione.

Caricamento potenziamenti

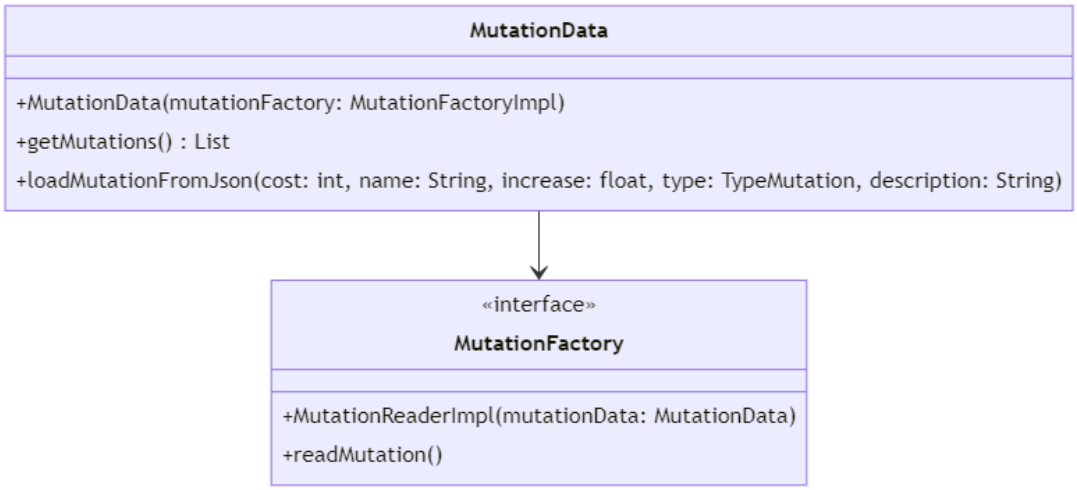

Figura 2.11: Schema UML di MutationData

Per il caricamento delle mutazioni, ho utilizzato la classe "MutationData". Questa classe rappresenta un insieme di mutazioni e fornisce metodi per accedere e manipolare questa collezione. Per convenzione ho supposto che i nomi delle mutazioni fossero univoci. La classe include un metodo "loadMutationFromJson" per istanziare le mutazioni da una fonte esterna, come un file JSON, Per istanziare tutte le mutazioni, la classe "MutationData" utilizza una factory chiamata "MutationFactory".

Lettura file

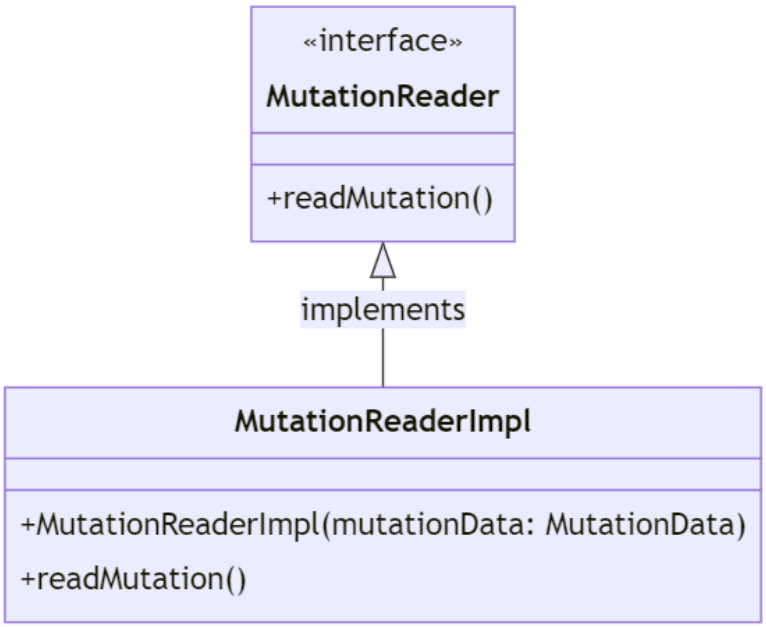

Figura 2.12: Schema UML di MutationReader

Per recuperare le informazione delle mutazioni contenute nei file ho creato la classe mutationreader che si occupa di leggere il file, recuperare i dati e utilizzando la classe mutationData istanziarli. Ho deciso di utilizzare un file json per salvare i dati principalmenti per la facilità di modifica: È relativamente semplice apportare modifiche al file JSON utilizzando un editor di testo o uno strumento apposito, consentendo di aggiornare rapidamente i dati senza dover ricompilare l'applicazione.

Gestione potenziamenti attivi

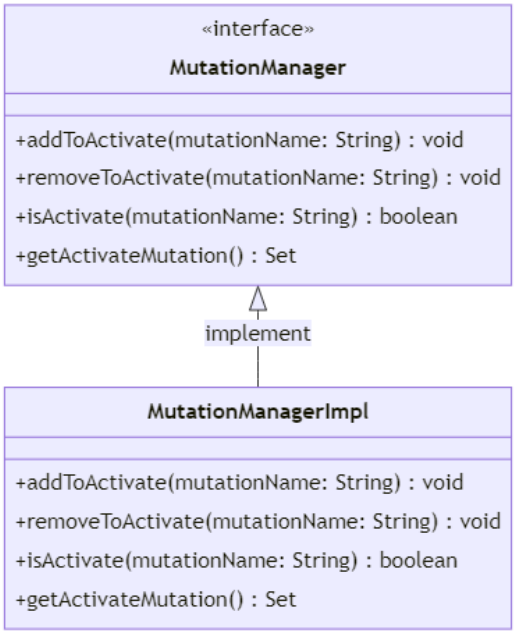

Figura 2.13: Schema UML di Mutation Manager

Le classi "MutationManager" e "MutationManagerImpl" gestiscono l'attivazione e la rimozione delle mutazioni attivate. L'interfaccia "MutationManager" definisce i metodi per aggiungere, rimuovere e controllare lo stato di attivazione di una mutazione. L'implementazione "MutationManagerImpl" gestisce effettivamente l'insieme delle mutazioni attive utilizzando una struttura dati di tipo Set.

Pietro Ventrucci

Nel progetto, il compito è stato quello di gestire la creazione e lo sviluppo di una cura per una malattia. Inoltre, ero responsabile di tenere aggiornato il mondo di gioco, monitorando gli eventi giornalieri e garantendo il corretto funzionamento delle notifiche di gioco.

Cura

Cura intercambiabile

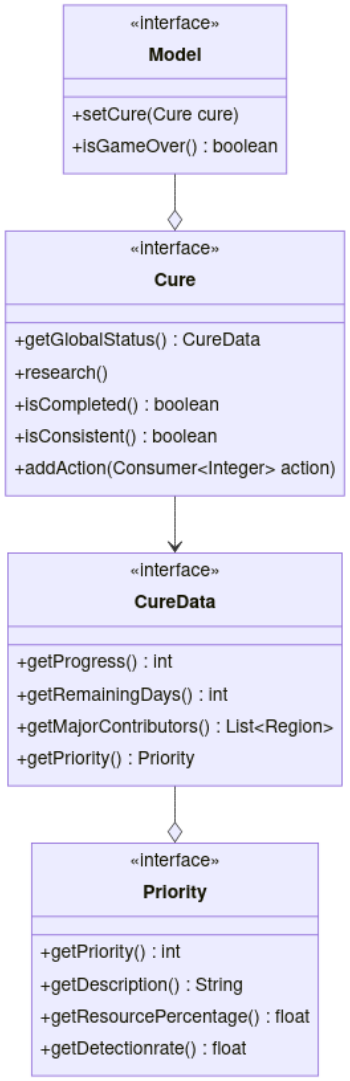

Figura 2.14: Schema UML della cura, con pattern Strategy e dependency injection

Una delle condizioni che possono determinare la conclusione del gioco è il completo sviluppo della cura per la malattia. Tuttavia, ciò può avvenire attraverso diverse condizioni o metodi. Ho quindi deciso di utilizzare il *patter Dependency Injection* 2.14 per fare si che il Model dipendesse da una istanza di Cure. Puo essere che la cura voglia far conoscere lo stato del prorprio sviluppo. Ho quindi utilizzato il *pattern Strategy* 2.14 per potere passare a Cure un Consumer che definisca la strategia da utilizzare in questi casi. Per

quanto riguarda CureData non esiste una vera implementazione, ma viene creata una sua istanza su richiesta creando una classe anonima.

Aggiornamenti sulle mutazioni della malattia

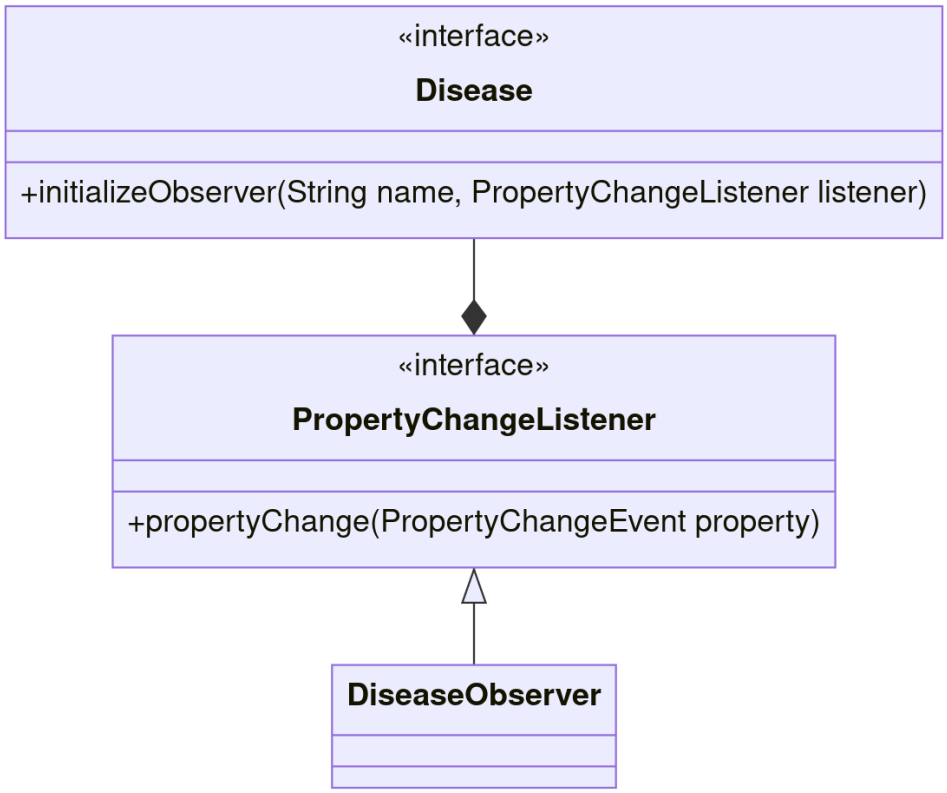

Figura 2.15: Schema UML della cura, con pattern Observer per lo stato di Disease

Per gestire le varie mutazioni della malattia avevo pensato che sarebbe stato sensato che ci fosse un controllo ogni volta che veniva effettuata la ricerca sulla malattia, ma sarebbero stati un numero sicuramente eccessivo di controlli rispetto alle volte che la malattia mutava effettivamente, ho quindi convenuto che utilizzare il *pattern Observer* 2.15 sarebbe stato migliore. Ho utilizzato l'interfaccia PropertyChangeListener fornita da java.beans piuttosto che quella Observer (in quanto deprecata) per creare il mio DiseaseObserver che ho poi collegato all'observable, Disease.

Costruzione della cura e delle priorità

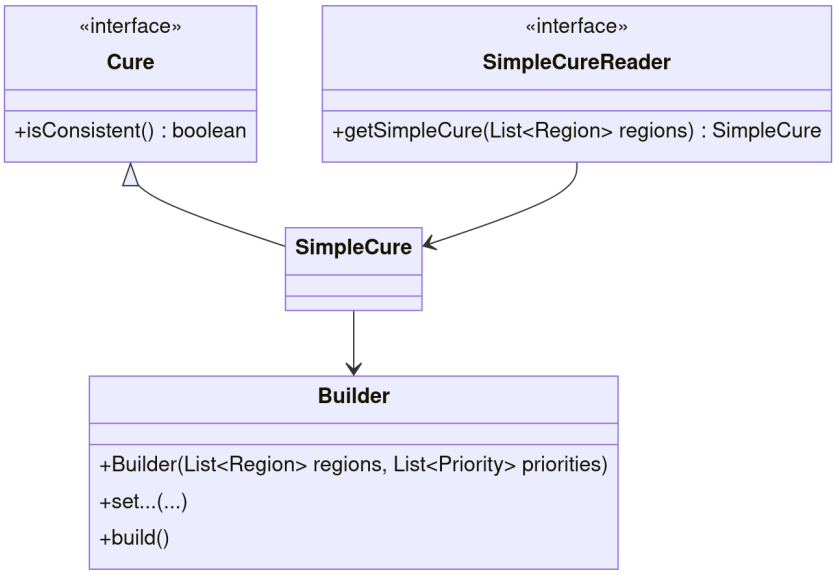

Figura 2.16: Schema UML della cura, con pattern Builder per la creazione di SimpleCure

Per l'istanziazione di SimpleCure l'unica implementazione di Cure nel gioco, avendo la maggiore flessibilità possibile, ho scelto di utilizzare il *patter Builder* 2.16 (non tutti i metodi di Builder sono presenti per una maggiore facilità di lettura e importanza puramente implementativa), creando una una classe innestata Builder. Questa scelta mi ha permesso di combinare il pattern con la lettura dei parametri da file, utilizzando una possibile implementazione di SimpleCureReader. Inizialmente pensavo di creare un interfaccia generica per CureReader, ma considerando che una cura potrebbe avere bisogno o meno di parametri esterni (in questo caso l'elenco delle Region che contribuiscono alla cura) ho optato per un CureReader più specifico, consentendomi la possibilità di creare diverse istanze di SimpleCure, ognuna leggermente diversa, semplicemente modificando il file di configurazione anziché ricompilare il codice ogni volta. Ho scelto il pattern Builder perché mi permette di creare un'istanza di SimpleCure anche se il file di configurazione è vuoto, grazie ai valori di default dei parametri. Allo stesso modo, non ho bisogno di configurare ogni singolo parametro da file. Ho preferito mantenere la verifica della coerenza dei parametri passati al Builder all'interno dell'interfaccia Cure stessa, in modo da avere il controllo il più vicino possibile all'oggetto che sto creando. Il builder evita la necessità di avere un numero eccessivo di costruttori personalizzati in base ai diversi set

di parametri, evitando potenziali confusioni tra parametri di tipo simile che potrebbero causare errori. Anche per la creazione delle Priorità utilizzo un approccio con Builder e lettura da file strutturato similmente a SimpleCure.

Aggiornamento gioco

Game loop

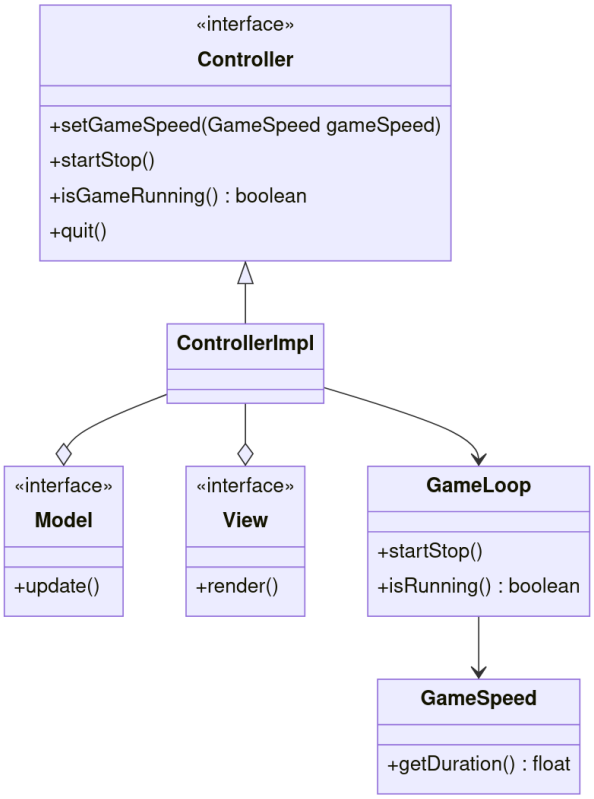

Figura 2.17: Schema UML del Game Loop, con patter Game Loop

Per la gestione dell'aggiornamento degli eventi, ho utilizzato il *pattern Game Loop* 2.17, che è stato adattato per soddisfare le esigenze specifiche del nostro caso. Ho implementato il GameLoop come una classe innestata all'interno di ControllerImpl. Questa classe è responsabile della regolazione della velocità di aggiornamento utilizzando un oggetto GameSpeed, che può essere modificato durante il gioco per regolare la velocità di aggiornamento, durata di una giornata. Il controller gestisce anche la possibilità di mettere in pausa il loop utilizzando una Lock e una Condition. Questo consente di controllare il flusso di esecuzione del GameLoop e metterlo in attesa fino a quando non viene dato il

segnale per riprendere l'esecuzione. In questo modo, è possibile gestire l'aggiornamento degli eventi in base a un intervallo di tempo, considerando l'intera giornata come unità di tempo. Il GameLoop si occupa di eseguire le operazioni di aggiornamento e rendering in modo sincronizzato e controllato. L'utilizzo del pattern Game Loop in particolare permette di mantenere un controllo preciso sulla frequenza di aggiornamento degli eventi.

Creazione di GameSpeed

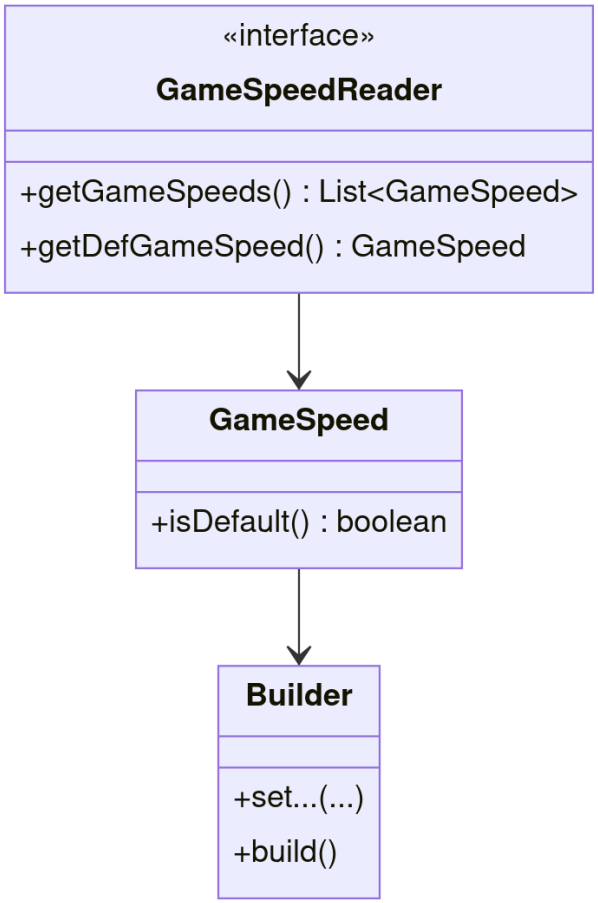

Figura 2.18: Schema UML per GameSpeed, con patter Builder

Per permettere una facile configurazione delle velocità di aggiornamento del GameLoop 2.17 ho deciso di utilizzare un file di configurazione dedicato per specificare le diverse opzioni di GameSpeed, avendo così la possibilità di avere un numero arbitrario di velocità. Per facilitare la gestione e l'istanziazione di queste opzioni, ho introdotto un GameSpeedReader che si occupa di leggere il file di configurazione e restituire le diverse istanze di

GameSpeed. Per garantire una maggiore flessibilità e facilità di creazione delle istanze di GameSpeed, ho utilizzato il *pattern Builder* 2.18. Questo mi consente di avere un'istanza predefinita di GameSpeed nel caso in cui il file di configurazione sia vuoto o non correttamente impostato. Inoltre, l'utilizzo del pattern Builder mi consente di avere un codice più leggibile e di gestire facilmente l'aggiunta di nuove opzioni di configurazione in futuro.

Aggiornamento GameSpeed

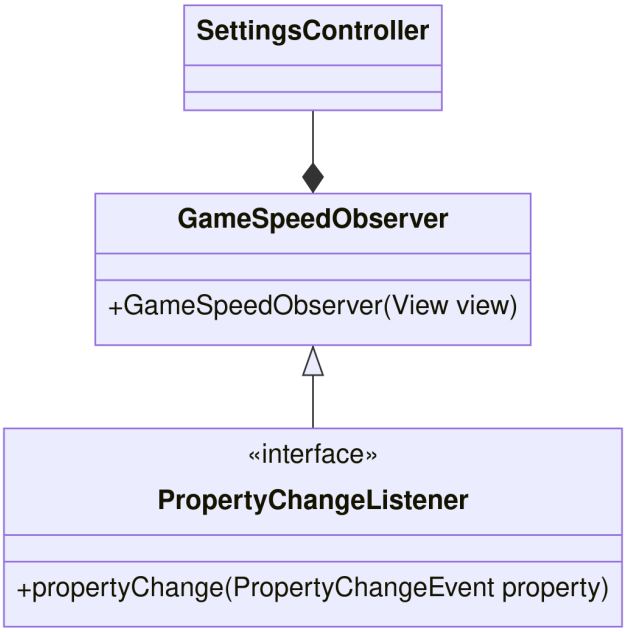

Figura 2.19: Schema UML per GameSpeedObserver, con patter Observer

Per consentire la modifica della velocità di aggiornamento del GameLoop, che è determinata dalla GameSpeed di riferimento, ho scelto di utilizzare il *patter Observer* 2.19, similmente a quando fatto 2.15, implementando PropertyChangeListener e avendo SettingsController come observable.

Gestione notifiche

Visualizzazione notifiche

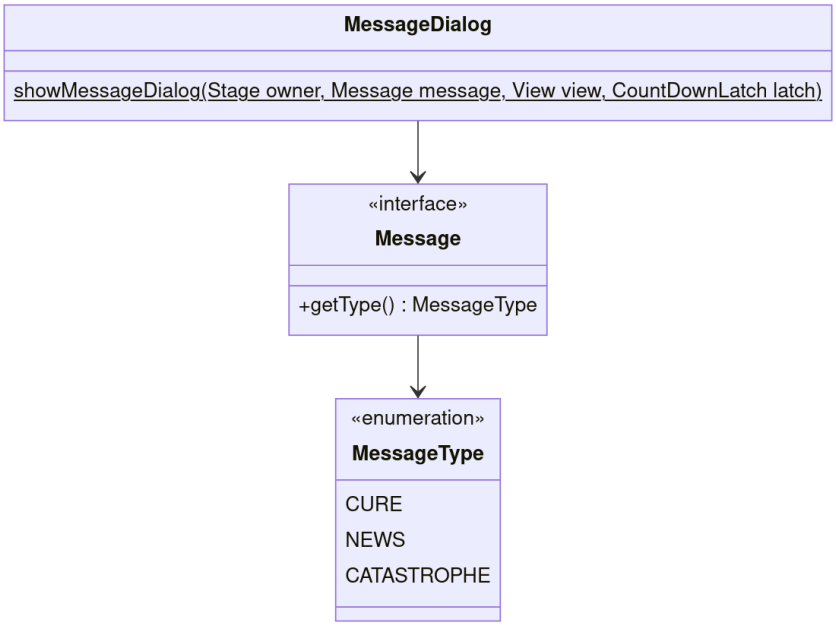

Figura 2.20: Schema UML per la gestione dei messaggi

La gestione delle notifiche in gioco, viene gestita da un MessageDialog 2.20, il quale per non particolari esigenze di grafica gestisce tute le notifiche in maniera analoga, ma grazie al MessageType che ogni Message possiede sarebbe possibile andare a modificare il MessageDialog in maniera personalizzata, viene utilizzato un CountDownLatch per aspettare che venga chiuso prima di eseguire altre azioni. Ho preferito utilizzare una Enumeration piuttosto che una configurazoine da file, come nei casi precedenti, in quanto le notifiche non sono un elemento fondamentale per il comportamento dell'applicazione, quindi la loro modifica è principalmente di carattere estetico, per esempio aggiungere un tipo di notifica vuole dire che si potrebbe avere un messaggio personalizzato nella View. Andando a creare un nuovo tipo di notifica sarebbe necessario poi una modifica in ogni caso al codice anche se la configurazione fosse avvenuta da file perchè vuole dire gestire un evento che fino a prima non era gestito.

Messaggi di News

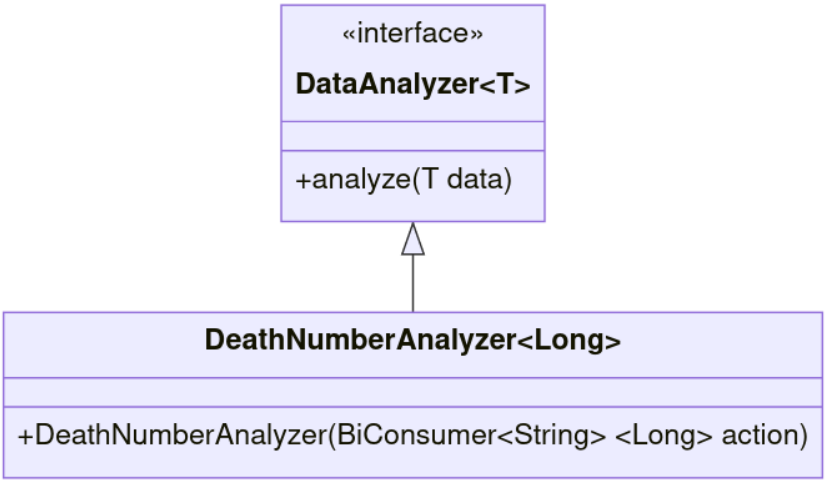

Figura 2.21: Schema UML per l'analisi dei dati, con pattern Strategy

Per avere un confronto fra le statistiche della malattia nel gioco e altri dati provenienti dal mondo reale, o meno, ho creato un interfaccia DataAnalyzer generica che permette la specializzazione sul tipo di dato che si vuole analizzare, che siano interi, stringhe, long o altro. In particolare ho implementato DeathNumberAnalyzer che si basa su un file per il riperimento delle informazioni sulle cause di morte. Ho utilizzato il *patter Strategy* 2.21 passando un Biconsimer al costruttore di DeathNumberAnalyzer, così facendo definisco la strategia da utilizzare. Avevo inizialmente cercato di reperire le informazioni utilizzando le API di WHO, permettendo così un aggiornamento nel tempo, ma mi è risultato troppo complicato e dispendioso, ho quindi preferito un salvataggio di dati in locale.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Durante lo sviluppo del nostro progetto abbiamo utilizzato le funzionalità offerte dalla libreria JUnit per testare in maniera automatizzata.
Funzionalità testate automaticamente:

- *SimpeCure:* siccome abbiamo adottato l'approccio del Builder per l'instanziazione della classe SimpleCure. Ciò ci ha permesso di testare che, se vengono passati parametri non validi, la configurazione risulti incoerente. Abbiamo anche eseguito test per verificare i risultati attesi ottenuti tramite la classe CureData.
- *Priority:* viene testato il corretto funzionamento del Builder per l'instanziazione di Priority. Viene anche testato che le priorità vengano create in ordine crescente.
- *GameLoop:* viene testato che se non viene impostata nessuna cura nel gioco il gioco si conclude.
- *Disease:* viene testato il corretto funzionamento dei metodi che calcolano gli infetti e i morti di ogni regione.
- *InfoData:* viene testato il corretto funzionamento dell'observer che deve notificare quando avvengono dei nuovi infetti per verificare se il totale degli infetti di uno stato passa da 0 a n, per poter aggiungere punti Dna. Viene anche verificato il corretto incremento dei punti dna ogni volta che il totale mondiale di morti e infetti supera una certa soglia.
- *Mutation:* viene testato il corretto funzionamento del pattern factory, l'aggiunta o rimozione delle mutazioni attive e la lista delle mutazioni.

- *Region:* viene testato l'incremento e il decremento degli infetti e morti nei casi limite.
- *Voyage:* Viene testata l'estrazione dei viaggi. (La correttezza viene controllata manualmente essendo l'estrazione casuale)
- *Events:* Viene testata, l'estrazione degli eventi(poi controllata manualmente essendo casuale)

Funzionalità testate manualmente:

- *GUI:* effettuato test manuale in quanto monte ore non sufficiente per studiare testing automatico di GUI.

3.2 Metodologia di lavoro

A seguito della fase di sviluppo, possiamo confermare che la ripartizione del lavoro è stata rispettata ed equa. Avremmo potuto migliorare il tempo e il lavoro utilizzato durante l'analisi per evitare di dovere riformulare alcune classi/interfaccie, ma siamo riusciti a tenerci aggiornati e coerenti con quanto fatto anche grazie a strumetni di connessione come "Meet" quando non riuscivamo a vederci per discuterne, così da poter essere coerenti con le interfacce che venivano utilizzate. Particolare difficoltà è dovuta alla scelta di utilizzare javaFx per la parte di Gui.

Implementazioni dei singoli elementi del team:

- *Mattia Farabegoli:* Gestione del menu iniziale e utilizzo di uno sceneManager e uno sceneLoader. Creazione dell'entità malattia e gestione del calcolo degli infetti e delle morti. Implementazione di Infodata per la gestione del calcolo dei punti di potenziamento della malattia e per salvataggio informazioni generali su morti infetti e stato della cura.
- *Luca Carabini:* Creazione della Mappa di gioco, gestione delle regioni, causare catastrofi, e gestione dei viaggi.
- *Giacomo Severi* gestione dei potenziamenti.
- *Pietro Ventrucci* gestione della Cura, gestione delle notifiche, game loop e velocità di gioco.

Parti in comune tra i componenti:

- Ci siamo confrontati di gruppo per capire come meglio far funzionare l'aggiornamento della View.

Nella fase di sviluppo abbiamo utilizzato il DVCS Git. Creato una repository su GitHub che abbiamo utilizzato per sincronizzare le versioni del progetto in locale. Abbiamo tenuto un branch master dove tenere le versioni più aggiornate dei nostri task, mentre lavoravamo sullo sviluppo su branch secondari.

3.3 Note di sviluppo

Luca Carabini

- `Lambda` Utilizzate in combinazione con lo Stream e per foreach `https://github.com/0rteip/OOP22-global-outbreak/blob/786c3488e368d0b1e9dbc7635ce69ee48e7bada3/src/main/java/globaloutbreak/model/voyage/VoyagesImpl.java#L42`
- `Stream` utilizzato principalmente filter e toList, in VoyageImpl e CauseEvent per generare nuove liste.`https://github.com/0rteip/OOP22-global-outbreak/blob/786c3488e368d0b1e9dbc7635ce69ee48e7bada3/src/main/java/globaloutbreak/model/voyage/VoyagesImpl.java#L34`
- `Utilizzo della libreria SLF4J` utilizzata per i logger `https://github.com/0rteip/OOP22-global-outbreak/blob/786c3488e368d0b1e9dbc7635ce69ee48e7bada3/src/main/java/globaloutbreak/model/region/RegionImpl.java#LL106C1-L106C1`
- `Libreria FasterXML Jackson`: per la lettura di file json. Permalink `https://github.com/0rteip/OOP22-global-outbreak/blob/786c3488e368d0b1e9dbc7635ce69ee48e7b src/main/java/globaloutbreak/controller/region/RegionControllerImpl.java#L32`
- `Libreria JavaFX`: utilizzata per gestire le schermate dell'applicazione. `https://github.com/0rteip/OOP22-global-outbreak/blob/786c3488e368d0b1e9dbc7635ce69ee48e7b src/main/java/globaloutbreak/view/scenecontroller/MapController.java#L33`
- `Optional`: Utilizzati per la selezione della Regione, per gestire le estrazioni degli eventi e per la lista degli stati raggiungibili. `https://github.com/0rteip/OOP22-global-outbreak/blob/786c3488e368d0b1e9dbc7635ce69ee48e7bada3/src/main/java/globaloutbreak/model/events/CauseEventsImpl.java#L32`

Mattia Farabegoli

- `Lambda`: utilizzate in combinazione agli stream per avere un codice compatto e intuitivo. `https://github.com/0rteip/OOP22-global-outbreak/blob/0e1c7da7c4e6d6aec4ed75c src/main/java/globaloutbreak/model/disease/DiseaseFactoryImpl.java#LL213C25-L213C`

- `Stream`: utilizzati principalmente per le iterazioni sulle collezioni di DiseaseData, per leggere i tipi di tutte le malattie e con le collezioni di regioni nella classe DiseaseFactoryimpl per calcolare gli infetti e le morti di ogni regione. `https://github.com/0rteip/OOP22-global-outbreak/blob/0e1c7da7c4e6d6aec4ed75c6fb92763d0d41f44f/src/main/java/globaloutbreak/model/disease/DiseaseFactoryImpl.java#LL213C25-L213C` Ho utilizzato le Stream in quanto permettono di manipolarle e filtrarle con semplicità garantendo una scrittura del codice compatta.

- `Optional`: utilizzati nella scena DiseaseChoice per gestire il primo ed il secondo click dei pulsanti per scegliere la malattia. `https://github.com/0rteip/OOP22-global-outbreak/blob/0e1c7da7c4e6d6aec4ed75c6fb92763d0d41f44f/src/main/java/globaloutbreak/view/scenecontroller/ChooseDiseaseController.java#LL23C3-L23C4`

- `Libreria slf4j`: utilizzata principalmente per l'utilizzo di un logger per gestire messaggi di informazione, avvertimenti ed errori. `https://github.com/0rteip/OOP22-global-outbreak/blob/0e1c7da7c4e6d6aec4ed75c6fb92763d0d41f44f/src/main/java/globaloutbreak/model/disease/DiseaseFactoryImpl.java#LL252C22-L252C22`

- `Libreria fasterxml jackson`: utilizzata per la lettura di file json. `https://github.com/0rteip/OOP22-global-outbreak/blob/0e1c7da7c4e6d6aec4ed75c6fb92763d0d41` `src/main/java/globaloutbreak/diseasereader/DiseaseReaderImpl.java#LL39C1-L39C92`

- `Libreria Apache Commons`: utilizzata principalmente per l'utilizzo di Pair. `https://github.com/0rteip/OOP22-global-outbreak/blob/956221f7bc92a85ef4b1bb0d3375ffab74` `src/main/java/globaloutbreak/settings/windowsettings/WindowSettingsImpl.java#L15`

- `Libreria JavaFX`: utilizzata per gestire le schermate dell'applicazione. `https://github.com/0rteip/OOP22-global-outbreak/blob/d56b68be40e62fe0ac3eb0230b1ffdbf6648` `src/main/java/globaloutbreak/view/sceneloader/SceneLoaderImpl.java#L50`

Per quanto riguarda JavaFX ho utilizzato la documentazione ufficiale presente nel seguente link `https://www.oracle.com/java/technologies/javase/javafx-docs.html`, insieme alle slide del corso e qualche nota sui forum, soprattutto riguardo all'utlizzo dello Stage per il settaggio le scene, in quanto SpotBugs rivelava l'errore del tipo "may expose internal representation by storing an externally mutable object" nel passaggio dello Stage al SceneManager, poi trattato autonomamente con un @SuppressWarning (`https://github.com/0rteip/OOP22-global-outbreak/blob/0e1c7da7c4e6d6aec4ed75c6fb92763d0d41f44f/src/main/java/globaloutbreak/view/scenemanager/SceneManagerImpl.java#L27`), in quanto era necessaria la corretta istanza dello Stage per cambiare le scene. Mentre, per quanto riguarda i pattern progettuali, ho fatto riferimento alle slide del corso ed in generale al materiale fornito a lezione

Giacomo Severi

- Libreria slf4j: utilizzata per l'utilizzo di un logger per gestire messaggi di informazione, avvertimenti ed errori. https://github.com/0rteip/OOP22-global-outbreak/blob/786c3488e368d0b1e9dbc7635ce69ee48e7bada3/src/main/java/globaloutbreak/mutationreader/MutationReaderImpl.java#L91
- Lambda e Stream: utilizzate per le iterazioni sulle collezioni di dati. https://github.com/0rteip/OOP22-global-outbreak/blob/786c3488e368d0b1e9dbc7635ce69ee48e7b
src/main/java/globaloutbreak/view/scenecontroller/MutationViewController.java#L57
- Libreria fasterxml jackson: utilizzata per la lettura da file. https://github.com/0rteip/OOP22-global-outbreak/blob/786c3488e368d0b1e9dbc7635ce69ee48e7bada3/src/main/java/globaloutbreak/mutationreader/MutationReaderImpl.java#L58
- Libreria JavaFX: utilizzata per gestire le schermate dell'applicazione. https://github.com/0rteip/OOP22-global-outbreak/blob/786c3488e368d0b1e9dbc7635ce69ee48e7b
src/main/java/globaloutbreak/view/scenecontroller/MutationViewController.java

Pietro Ventrucci

- Libreria logging SLF4J :utilizzata per logging in vari punti. Permalink: https://github.com/0rteip/OOP22-global-outbreak/blob/786c3488e368d0b1e9dbc7635ce69ee48e
src/main/java/globaloutbreak/gamespeed/GameSpeedReaderImpl.java#L24
- Libreria FasterXML Jackson: per la lettura di file json. Permalink https://github.com/0rteip/OOP22-global-outbreak/blob/786c3488e368d0b1e9dbc7635ce69ee48e7b
src/main/java/globaloutbreak/model/cure/SimpleCureReaderImpl.java#L38
- Stream e Lambda: utilizzati per facilitare la operazioni sul calcolo, in particolare della cura, avendo un codice più compatto e efficiente. Permalink: https://github.com/0rteip/OOP22-global-outbreak/blob/786c3488e368d0b1e9dbc7635ce69ee48e7b
src/main/java/globaloutbreak/model/cure/SimpleCure.java#L110
- Interfaccia generica: usata una interfaccia generica per permettere una maggiore specializzazione. Permalink: https://github.com/0rteip/OOP22-global-outbreak/blob/786c3488e368d0b1e9dbc7635ce69ee48e7bada3/src/main/java/globaloutbreak/model/dataanalyzer/DataAnalyzer.java#L8
- Optional: usati per verificare o meno la presenza di certi valori. Permalink: https://github.com/0rteip/OOP22-global-outbreak/blob/786c3488e368d0b1e9dbc7635ce69ee48e
src/main/java/globaloutbreak/model/cure/CureData.java#L28

- `Utilizzo di Thread, Lock e Condition`: per permettere la sincronizzazione. Permalink:`https://github.com/0rteip/OOP22-global-outbreak/blob/786c3488e368d0b1e9dbc`
`src/main/java/globaloutbreak/controller/ControllerImpl.java#L235`

- `Utilizzo di CountDownLatch`: usato per sincronizzare messaggi e view. Permalink
:`https://github.com/0rteip/OOP22-global-outbreak/blob/786c3488e368d0b1e9dbc7635ce`
`src/main/java/globaloutbreak/view/messagedialog/MessageDialog.java#L48`

- `JavaFx`: utilizzata per la GUI. Permalink: `https://github.com/0rteip/OOP22-global-outbrea`
`blob/786c3488e368d0b1e9dbc7635ce69ee48e7bada3/src/main/java/globaloutbreak/`
`view/messagedialog/MessageDialog.java#L38`

Per quanto riguarda i pattern progettuali ho fatto riferimento al materiale didattico delle lezioni in aula, al libro *"Design Patterns: Elements of Reusable Object-Oriented-Software"* - Gamma, Helm, Johnson, Vlissides, al seminario del Professor Ricci. Per quanto riguarda JavaFX ho seguito un tutorial `https://www.youtube.com/watch?v=9XJicRt_FaI`

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

Luca Carabini

Sviluppare questa applicazione ‘e stato molto interessante perchè non avevo mai realizzato un programma così complesso e lungo. Sinceramente non mi ritengo totalmente soddisfatto della parte grafica, volevo fare lampeggiare le regioni ma ho havuto grossi problemi con la gestione dei Thread, essendo che java fx non lo avevo mai utilizzato, e avrei voluto organizzare meglio la classe della view, per quanto riguarda la parte logica invece mi ritengo sufficentemente soddisfatto anche se riconsco che se avessi gestito meglio il tempo avrei realizzato un codice più ordinato, nonostatnte questo ho sempre cercato di fare le classi in modo da garantire l'estendibilità nel tempo. Ritengo inoltre che avrei potuto gestire in modo migliore il tempo, magari concentrandomi di più sulla parte logica del gioco.

Mattia Farabegoli

La realizzazione di questo progetto è stata molto interessante, soprattutto perchè permette di entrare in un'ottica più ampia di realizzazione di progetti che si avvicina sicuramente di più ad un contesto aziendale. Lavorare in team ha aspetti positivi e negativi, ma il bilancio di questo progetto penso sia sicuramente positivo. E' stato allo stesso modo molto impegnativo, soprattutto nella gestione del tempo e nell'unione dei singoli lavori svolti. Anche l'utilizzo della libreria JavaFX non è stato facile nell'immediato. Sono comunque contento della partecipazione da parte di tutti e della sintonia che abbiamo creato in poco tempo. Il progetto poteva essere realizzato meglio, ma sicuramente è un punto di partenza.

Giacomo Severi

Sono soddisfatto del risultato ottenuto con questo progetto. È stata un'esperienza stimolante e gratificante lavorare su questo progetto. Lavorare in team è stato un aspetto molto positivo, in quanto abbiamo potuto combinare le nostre diverse prospettive e competenze per raggiungere gli obiettivi comuni. Nonostante le sfide incontrate lungo il percorso, siamo riusciti a superarle con successo grazie alla nostra comunicazione aperta e alla collaborazione. Una delle parti che ho apprezzato di più è stata l'implementazione rivolta alla riusabilità. Tuttavia, riconosco che ci sono ancora aree in cui il progetto potrebbe essere ulteriormente migliorato. Ad esempio, la gestione degli errori potrebbe essere resa più robusta e l'interfaccia utente potrebbe essere resa più intuitiva ed esteticamente gradevole. Questi sono aspetti su cui lavorare per futuri sviluppi e iterazioni. In generale, sono grato per l'opportunità di partecipare a questo progetto e per il supporto e l'entusiasmo del team. È stato un percorso di apprendimento significativo e mi sento motivato a continuare a sviluppare le mie abilità e conoscenze nel campo.

Pietro ventrucci

Sono abbastanza soddisfatto del mio lavoro nel progetto. Ho dedicato tempo ed energie per svolgere la mia parte utilizzando le mie competenze. Tuttavia, riconosco che avrei dovuto impegnarmi seriamente fin dall'inizio del progetto anziché procrastinare. Questo ha avuto un impatto negativo sullo sviluppo, poiché ho trascurato alcuni problemi e soluzioni migliori che avrei potuto individuare se avessi dedicato più tempo. Ho cercato di aiutare gli altri membri del gruppo durante lo sviluppo e di trovare le soluzioni migliori, anche se non sempre sono stato pienamente soddisfatto dei risultati. Nel complesso, il gruppo ha lavorato bene insieme, ma ritengo che ci sia mancata un po' di costanza e impegno generale, e sono consapevole che anche io ne sono stato parte. Tuttavia, il mio obiettivo principale durante tutte le fasi di sviluppo era quello di ottenere un alto livello di riusabilità, e credo di esserci riuscito grazie alle scelte di design pattern come il Builder e la configurazione da file. Sicuramente ci sono molte parti che avrei potuto svolgere meglio, in particolare la parte GUI, su cui non mi sono concentrato particolarmente a causa delle difficoltà che ho incontrato nell'apprendere JavaFX. Alla fine del progetto, ho compreso l'importanza di condurre un'analisi e definire un'architettura solida per facilitare lo sviluppo. In futuro, presterò maggiormente attenzione a questi aspetti in qualsiasi altro progetto. Non penso di volere sviluppare ulteriormente questo progetto nel suo stato attuale, ma probabilmente mi piacerebbe riuscire a capire come si sarebbe potuto strutturare meglio, su questo potrei pensarci.

Appendice A

Guida utente

Nella prima schermata l'utente dovrà decidere se leggere le regole del gioco, iniziare una nuova partita oppure uscire e quindi chiudere l'appicazione. Una volta scelto nuova partita dovrà decidere con quale malattia iniziare il gioco(al momento una sola), la schermata successiva riepiloga i valori iniziali della malattia, poi una volta cliccato sulla malattia ancora una volta, l'utente dovrà scegliere il nome della malattia. Infine potrà iniziare iniziare il gioco cliccando una regione, che sarà quella da cui partirà ad espandersi il virus (piccolo bug nella schermata della mappa per vedere la schermata completa bisegnerà ingradire la schermata).

Schermata Principale

La prima azione da effettuare è scegliere la regione dalla quale iniziare. Premendo su una regione la partita ha inizio ed il primo infetto appare nella regione scelta. Da questo momento l'infezione e la morte delle persone è automatica, ma il giocatore ha la possibilità di controllare l'andamento dell'epidemia potenziando i vari parametri della Malattia. Di seguito tutti gli elementi importanti nella schermata principale:

- `Malattia` Serve per andare alla schermata dei potenziamenti
- `Mondo` Serve per vedere lo stato della cura e il grafico dei morti, infetti e sani
- `Play` Serve per stoppare e riprendere il gioco, bisognera premere questo pulsante ogni volta che si premerà su qualisiasi altro bottone all'interno della schermata. Attenzione: per iniziare la partita non bisogna premere su questo bottone ma premere sulla regione da cui si vuole fare partire il primo infetto della partita.
- `Settings` Al momento serve soltanto per settare la velocità di gioco
- `Info regioni` (Schermata sotto la mappa) Qui vengono mostrate tutte le informazioni sulla regione selezionata (Occorre cliccare sulla regione che si vuole selezionare,

mentre per visualizzare le info sull'intero mondo senza cambiare schermata basta premere sul mare.

Schermata Potenziamenti

Qui l'utente può scegliere quali potenziamenti fare. In alto a sinistra sono indicati i punti. Premendo su un potenziamento, apparirà una breve descrizione, il costo, l'incremento e il pulsante che può indicare la scritta 'Evolvi', se il potenziamento non è ancor stato effettuato, o 'Involvi' se il potenziamento è già stato effettuato. In quest'ultimo caso premendo sul pulsante 'Involvi' il potenziamento sarà rimosso dalla Malattia e verranno restituiti tutti i punti spesi in precedenza per il potenziamento.