Natural Selection Simulator
Relazione Progetto
"Programmazione ad Oggetti"

Ceredi Simone, Conti Alessio, Giulianelli Andrea

25/04/2020

Indice

1 Analisi **2**
1.1 Requisiti 2
1.2 Analisi e modello del dominio 3

2 Design **5**
2.1 Architettura 5
2.2 Design dettagliato 6

3 Sviluppo **24**
3.1 Testing automatizzato 24
3.2 Metodologia di lavoro 25
3.3 Note di sviluppo 26

4 Commenti finali **29**
4.1 Autovalutazione e lavori futuri 29
4.2 Difficoltà incontrate e commenti per i docenti 31

A Guida utente **33**

Capitolo 1

Analisi

1.1 Requisiti

Il gruppo si pone come obiettivo quello di realizzare un simulatore che visualizzi l'evoluzione di una popolazione di individui nel tempo a seguito di adattamenti generazionali che possono colpire ogni individuo nel momento della nascita in maniera casuale. Il progetto si ritiene utile per visualizzare come una data popolazione, che possiede determinate caratteristiche e proprietà iniziali si evolva nel tempo per meglio adattarsi all'ambiente in cui vive.

Requisiti Funzionali

- Durante il giorno gli individui dovranno muoversi alla ricerca di cibo il quale si troverà disposto in posizioni casuali all' interno dell'ambiente. Essi potranno muoversi e perderanno energia in base alle loro caratteristiche.
- Cibandosi essi acquistano una quantità di energia pari al valore energetico offerto dal cibo.
- Durante la notte le entità possono procreare. Ogni entità figlia avrà le stesse caratteristiche del padre più una eventuale mutazione.
- La simulazione procede fino a quando tutti gli organismi si estinguono oppure infinitamente se le condizioni della popolazione e dell'ambiente lo permettono.
- Analisi quantitativa dell'evoluzione degli organismi in vita tramite grafici in real-time.

Requisiti non funzionali

- Progettare il simulatore in modo tale da renderlo facilmente espandibile con altre mutazioni o con altri elementi dell'ambiente.
- L'applicativo dovrà essere efficiente nell'uso delle risorse.

1.2 Analisi e modello del dominio

La simulazione ha luogo in un Environment il quale possiede definite caratteristiche quali la temperatura e la dimensione. Esso viene popolato da Organismi e Cibo. Gli organismi sono contraddistinti da Trait, caratteri specifici:

- *Dimensione*: consente di stabilire quanto cibo un organismo può mangiare. Un organismo di dimensione maggiore sarà in grado di mangiare di più però consumerà più energie nel movimento. La dimensione influisce anche sulla resistenza alla temperatura, un individuo più grande sarà in grado di resistere meglio al freddo.
- *Velocità*: maggiore velocità aumenta il consumo di energie dell'organismo nel movimento.
- *Numero di figli generabili*
- *Food Radar*: capacità di individuare e mangiare cibo in un'area attorno a sè. Maggiore questa sensibilità più energie consumerà l'organismo.

Durante la procreazione i figli possono incorrere in mutazioni dei loro caratteri; questo porta all'evoluzione della specie. Solamente gli organismi con caratteristiche affini all'ambiente in cui si trovano a vivere potranno sopravvivere. Disseminati nell'environment gli organismi possono trovare del cibo il quale possiede una ben definita quantità di energia. Gli elementi costitutivi del problema sono sintetizzati in Figura 1.1. Il requisito non funzionale riguardante l'efficienza nell'uso delle risorse non potrà essere svolto all'interno del monte ore previsto in quanto prevede un'analisi dettagliata delle performance dell'applicativo: tale feature sarà oggetto di futuri lavori.

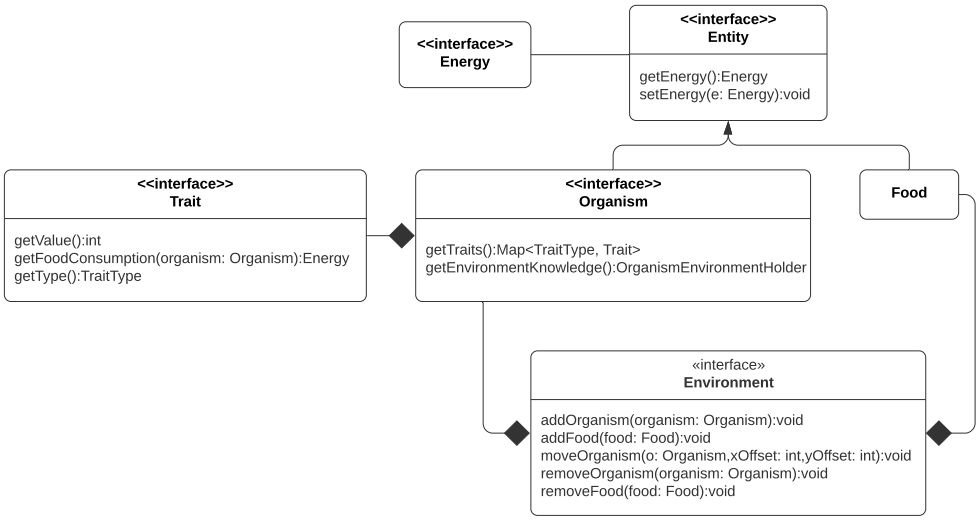


Figura 1.1: Schema UML dell'analisi del problema, con rappresentate le entità principali ed i rapporti fra loro

Capitolo 2

Design

2.1 Architettura

Per la realizzazione di Natural Selection Simulator abbiamo scelto di utilizzare il pattern architetturale Model-View-Controller. In questo modo è stato possibile isolare la logica funzionale che riguarda le tre componenti, le quali non interferiscono tra loro. Il progetto risulta essere così meglio organizzato e la modifica di una delle componenti non comporta cambiamenti nelle restanti. Nella nostra modellazione del pattern MVC il controller comanda le modifiche al modello e cambia di conseguenza la view. In particolare il controller gestisce il loop di simulazione aggiornando model (modello object-orented del dominio applicativo) e notificando alla view i cambiamenti da effettuare periodicamente. Il controller gestendo il coordinamento tra model e view si interpone tra questi, in questo modo le tre componenti rimangano estremamente separate, permettendo una sostituzione di view o model senza impattare il funzionamento del sistema. In Figura 2.1 é esemplificato il diagramma UML architetturale.

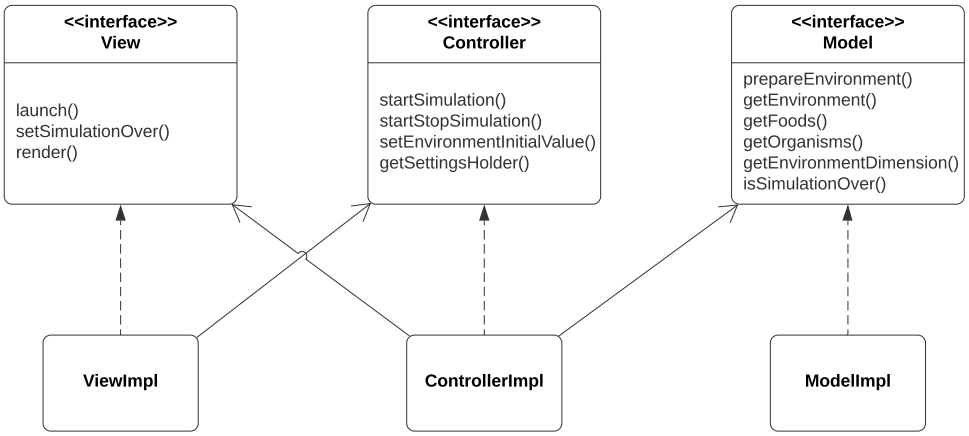


Figura 2.1: Schema UML architetturale di Natural Selection Simulator.

2.2 Design dettagliato

Alessio Conti

Il mio compito nel progetto è stato quello di modellare le *entità* (organismi e cibo) che popolano l'ambiente e in seguito gestire le *azioni* che questi svolgeranno durante il loro ciclo vitale.

Entity

Le entità che popolano l'ambiente: organismi e cibo, presentano alcuni tratti caratteristici comuni tra loro; uno di questi è il fatto che entrambi possiedono un livello di energia.
Per ottemperare a questo fatto ho deciso di creare un'entità padre, comune ad entrambi che raggruppasse quelle che sono le loro caratteristiche comuni. Ciò è stato fatto con un'interfaccia *Entity*, la quale assieme alla relativa classe astratta *AbstractEntity* rappresenta l'ossatura centrale delle entità in campo. Organismi e Cibo perciò non sono altro che estensioni della classe *AbstractEntity* raffinate con i relativi metodi propri che le caratterizzano. (Figura 2.2)

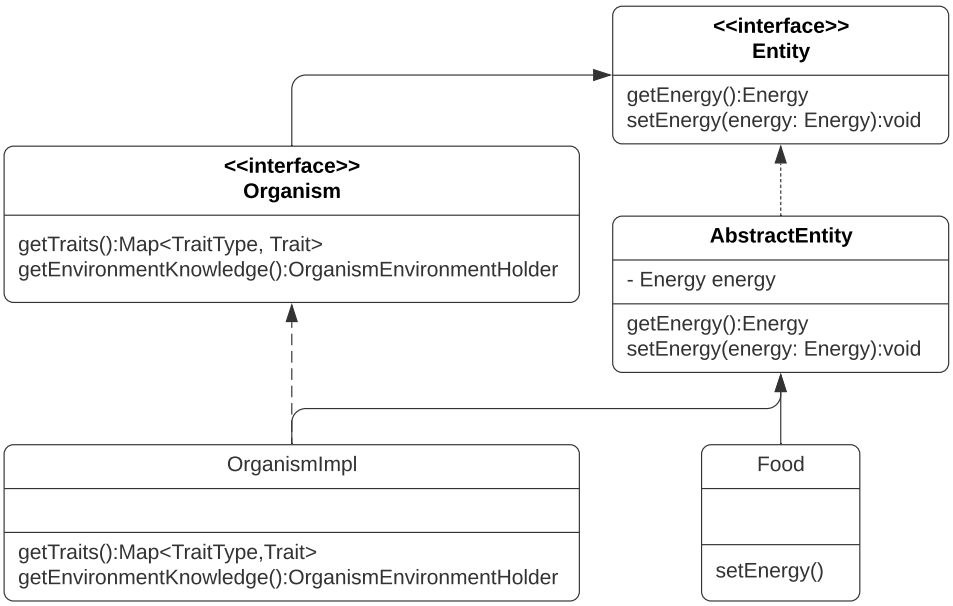


Figura 2.2: Schema UML gestione Entity.

Per istanziare tali elementi ho ritenuto utile fornire due Builder: uno per il cibo (*FoodBuilder*) e uno per gli organismi (*OrganismBuilder*).
Ritengo tale scelta opportuna in quanto: nel caso del cibo, in scenari futuri potrebbero voler essere inserite varie tipologie di cibo, le quali differiscono per esempio per quantità di energia conferita.
Per quanto riguarda il Builder degli organismi le considerazioni che mi hanno portato alla scelta di tale pattern costruttivo sono state dettate dal fatto che un organismo potrebbe teoricamente possedere un numero indefinito di caratteristiche, soprattutto tenendo presente come in futuro possano voler essere aggiunti alcuni caratteri piuttosto che altri. Grazie a questa scelta in futuro potranno essere istanziati anche diverse "specie" di organismi i quali possiedono caratteristiche diverse a seconda della specie di cui fanno parte. Descriverò solamente il Builder relativo all'organismo in quanto lo ritengo di maggior rilievo, per il cibo ho agito analogamente.

Con riferimento alla Figura 2.3: il Builder in se è l'interfaccia *OrganismBuilder* che definisce i metodi per settare le varie parti che compongono l'oggetto che deve essere costruito. Ad implementare tale interfaccia e quindi a costruire effettivamente gli oggetti è stata creata la classe *OrganismBuilde-*

rImpl. L'energia viene presa in ingresso dal costruttore in quanto elemento fondamentale per la costruzione di un organismo.

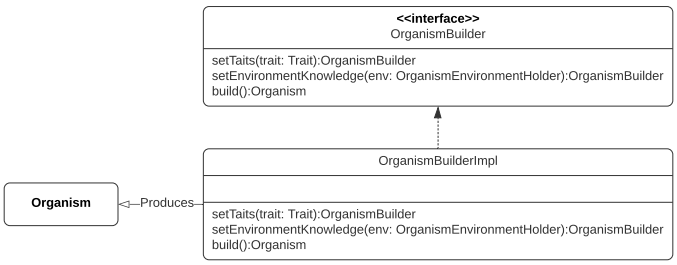


Figura 2.3: Schema UML Builder per un Organismo.

Actions

L'idea dietro alla simulazione è che vi sia un ciclo giornaliero. Un'alternanza tra giorno e notte definisce infatti quali azioni devono essere compiute dagli individui nei vari momenti della giornata.
L'ambiente (gestore dell'alternanza temporale) notifica di volta in volta ai singoli organismi quale ciclo di azioni intraprendere a seconda del momento della giornata in cui si trovano.
Sono state attualmente implementate due cicli di azioni, il ciclo delle azioni del giorno e quelle svolte durante la notte. Le azioni del giorno permettono ad un organismo di muoversi e di mangiare il cibo che trova nell'ambiente, mentre durante la notte potrà generare entità figlie.
Dal momento in cui i periodi si differenziano soltanto in base a quali compiti vengono portati a termine in una situazione piuttosto che nell'altra, ho agito in modo da lasciare piena liberà alle sottoclassi di implementare il comportamento che differisce e massimizzare il riuso.
Per fare ciò ho utilizzato il pattern *Command*.
Con riferimento alla Figura 2.4: l'interfaccia *Actions* svolge il compito di command, dichiarando l'operazione *perform()* da svolgere, mentre *DayAction* e *NightAction* rappresentano gli effettivi concreteCommand i quali implementano *perform()* invocando le specifiche operazioni da svolgere nelle differenti situazioni.

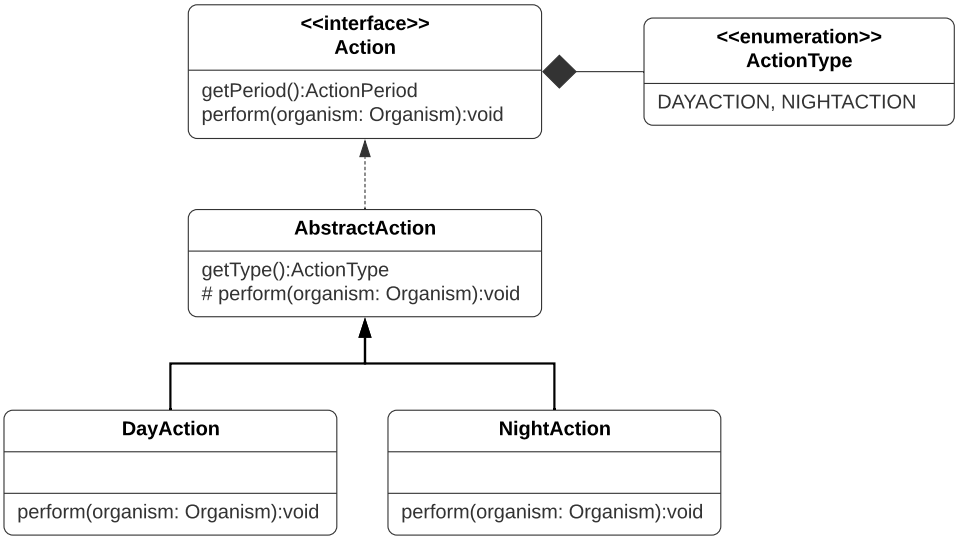


Figura 2.4: Schema UML pattern Command per gestire le Actions.

Per la gestione dei compiti assegnati ad ogni organismo si utilizza il pattern *Strategy*.
Come da Figura 2.5: le implementazioni dei singoli compiti possono essere così facilmente modificate, e la modifica impatta direttamente sul comportamento di ogni organismo.

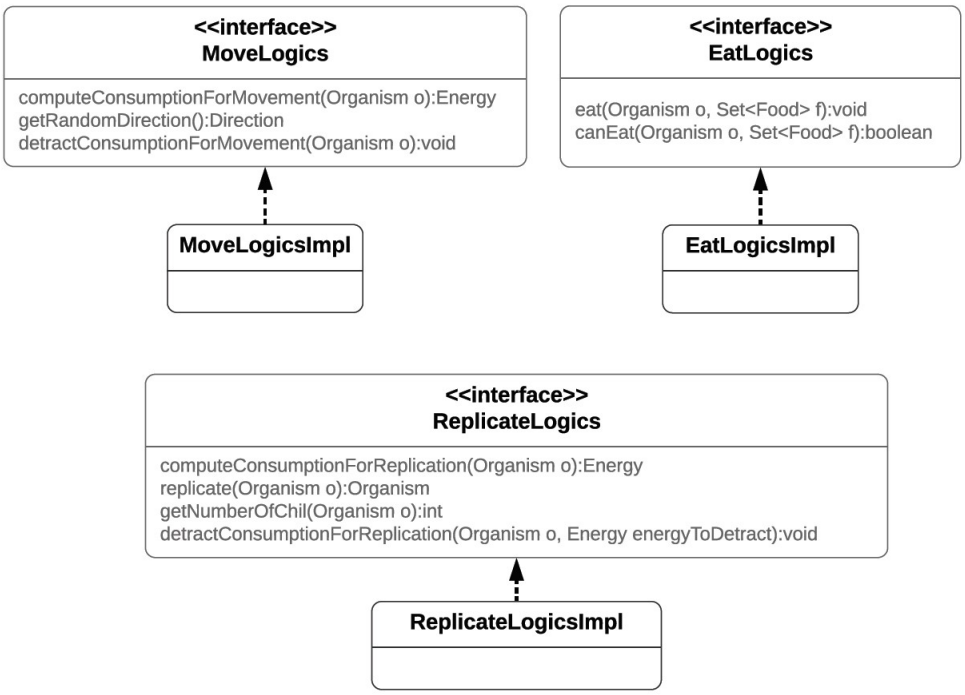


Figura 2.5: Schema UML pattern Strategy per la gestione dei compiti assegnati ad ogni organismo.

Andrea Giulianelli

All'interno del progetto mi sono occupato di gestire le caratteristiche e le mutazioni degli organismi, le impostazioni, il setup iniziale della simulazione e la generazione e navigazione nelle schermate del simulatore.

Caratteristiche

Le caratteristiche o trait degli organismi implementano l'interfaccia Trait in modo da permettere di applicare il principio DIP (Dependency inversion principle). Al fine di gestire il consumo energetico di ogni trait ho utilizzato il pattern Strategy. La strategia è identificata dall'interfaccia FoodConsumptionFunction (Figura 2.6).

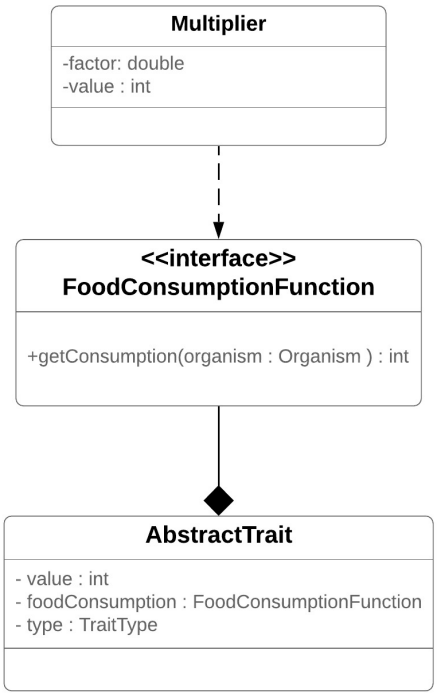


Figura 2.6: Rappresentazione UML del pattern Strategy per la strategia di consumo cibo dei Trait.

Ho implementato due tipi di consumo, un moltiplicatore (classe Multiplier) che restituisce il consumo calcolato come prodotto tra un fattore fornito in input e il valore assunto dal trait in quel particolare istante, e uno ad-hoc per la Sensibilità alla temperatura che utilizza i dati dell'organismo e del trait per modificare i parametri di una parabola al fine di ottenere il consumo. Il vantaggio di utilizzare questa struttura è che ogni qual volta si desidera aggiungere un nuovo trait agli organismi (quindi al simulatore) basterà creare una nuova classe che estenda la classe astratta AbstractTrait specificando la strategia da utilizzare per calcolare il consumo di cibo. Figura 2.7

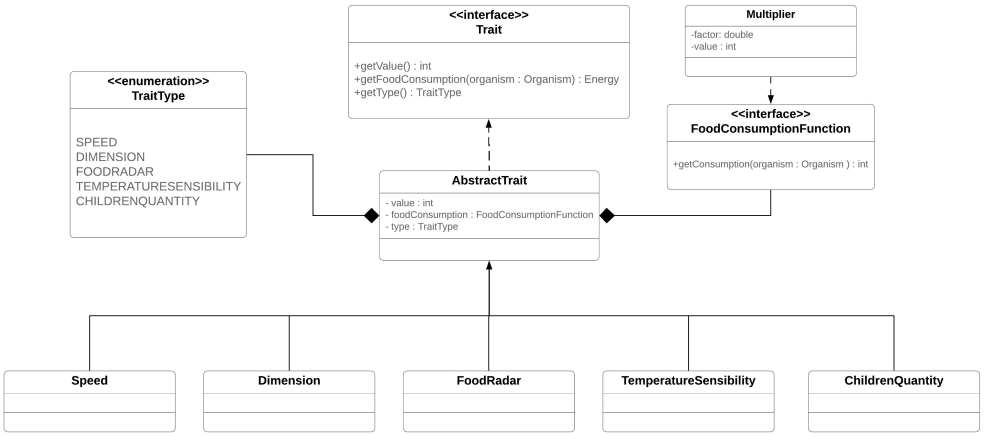
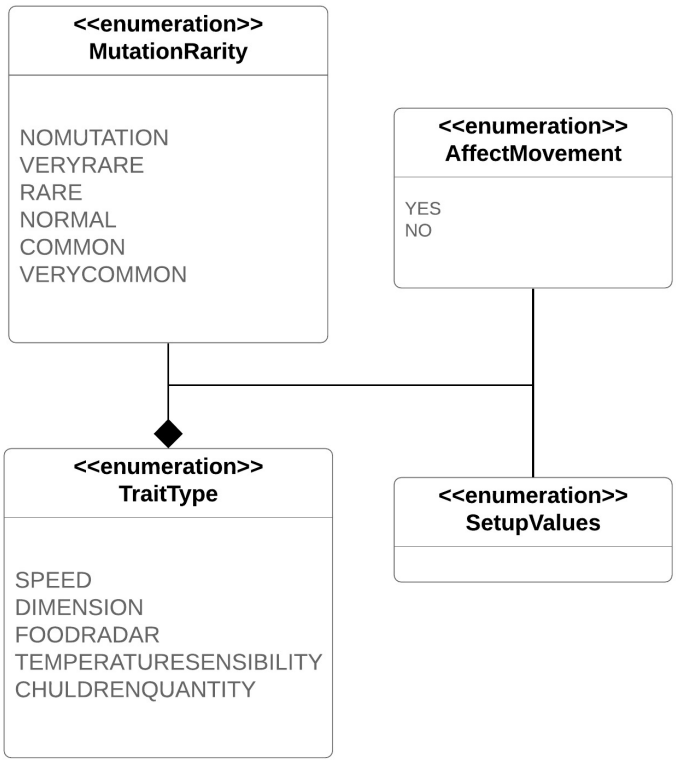


Figura 2.7: Schema UML gestione Trait

Al fine di descrivere le caratteristiche di ogni trait ho creato l'enum TraitType che al suo interno per ogni tipologia di trait memorizza:

- *Rarità di mutazione*: Mutation Rarity
- *AffectMovement*: indica il fatto che il trait consumi o meno cibo durante il movimento dell'organismo. Come suggerito da J. Block uso un'enum al posto di un booleano.
- *SetupValues* : Range di valori che assume



Mutation Rarity

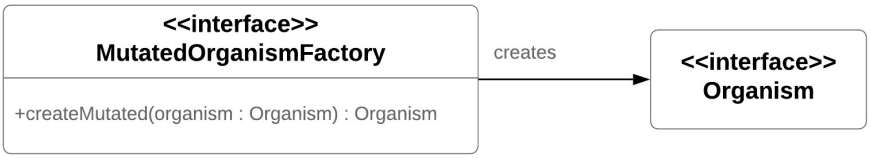
Ho ritenuto molto utile la presenza di una enum che descrivesse la rarità di mutazione, in quanto ogni caratteristica può mutare con una percentuale diversa. La percentuale di mutazione verrà utilizzata dall'algoritmo che gestisce la mutazione del trait.

Setup Values

Inoltre ho ritenuto utile la presenza anche di una enum che mi memorizzasse il range di valori possibili per ogni trait, in modo da rendere il controllo sulla validità dei valori veloce e unico, infatti è direttamente implementato all'interno della classe astratta AbstractTrait. In aggiunta, in questo modo ho tutti i valori assunti dai trait in un unico concetto, senza delegare questa responsabilità alle singole implementazioni.

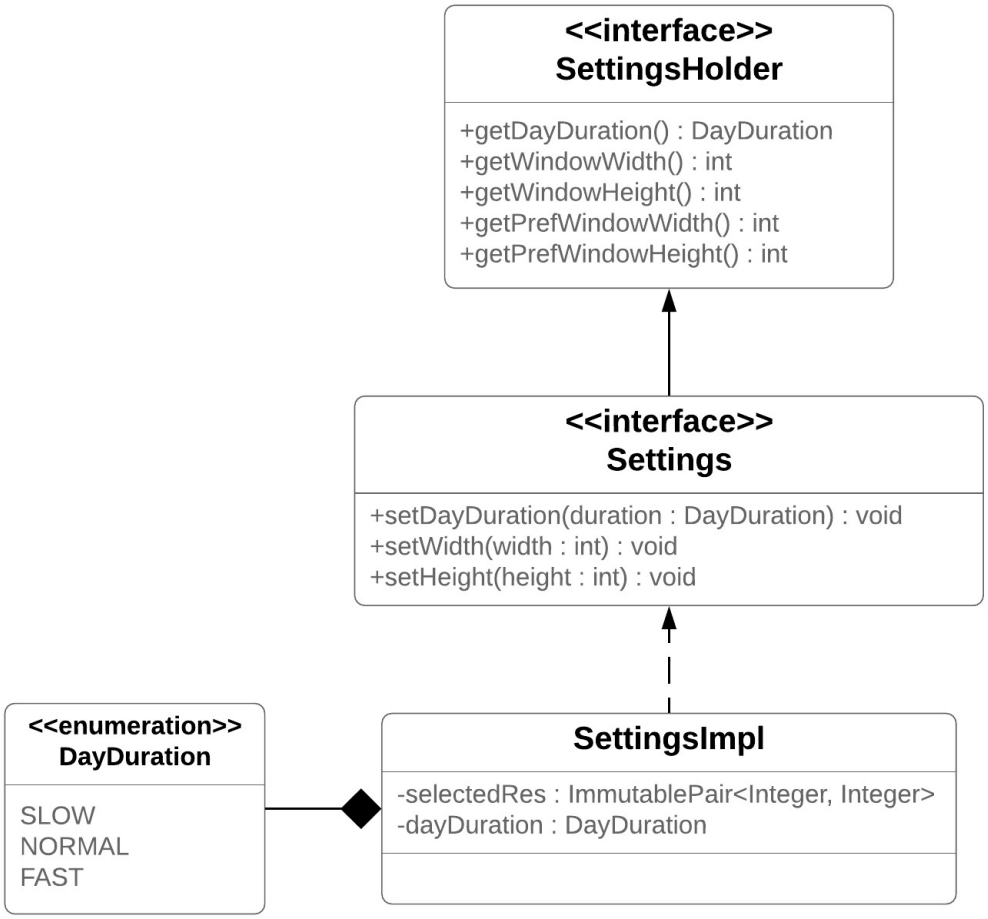
Mutazioni

Quando gli organismi creano degli organismi figli questi ultimi possono essere soggetti a mutazioni dei trait. Quindi per modellare la creazione di figli ho deciso di utilizzare il pattern Factory Method scorporando la creazione dei singoli individui, all'atto dell'inizio della simulazione (svolta da Alessio Conti), dalla creazione dei figli. Ho deciso di optare per il Factory Method perchè consente di creare oggetti lasciando alle sottoclassi il compito di decidere quale classe istanziare e soprattutto in questo modo se un domani occorrà modificare il modo con cui vengono creati i figli, basterà creare una nuova specializzazione. In particolare questa factory ha la peculiarità di prendere in ingresso un organismo di partenza, cioè il padre. Ho optato per il pattern Factory Method invece che per il Prototype soprattutto per il fatto che l'organismo è immutabile, quindi dopo la clonazione non avrei potuto mutare i trait.

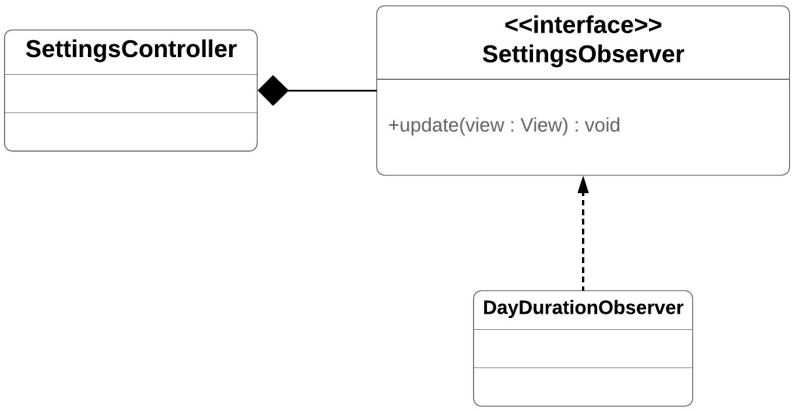


Impostazioni

Le impostazioni consentono all'utente di impostare la velocità di simulazione e servono per gestire le risoluzioni delle schermate. Al fine di rispettare il principio ISP (Interface segregation principle), ho costruito due interfacce specifiche per le richieste delle due tipologie di cliente delle impostazioni (View e Controller). L'interfaccia SettingsHolder consente di ottenere tutte le informazioni contenute nelle impostazioni senza la possibilità di modificare tali valori; mentre l'interfaccia Settings aggiunge i vari metodi di modifica. Ho adottato questa scelta perchè ad esempio all'interno dei controller delle view non è necessario avere il totale controllo delle impostazioni ed inoltre questo mi permette di evitare modifiche dirette dei settings da parte della view, obbligando così a passare per il controller (rispettando MVC).

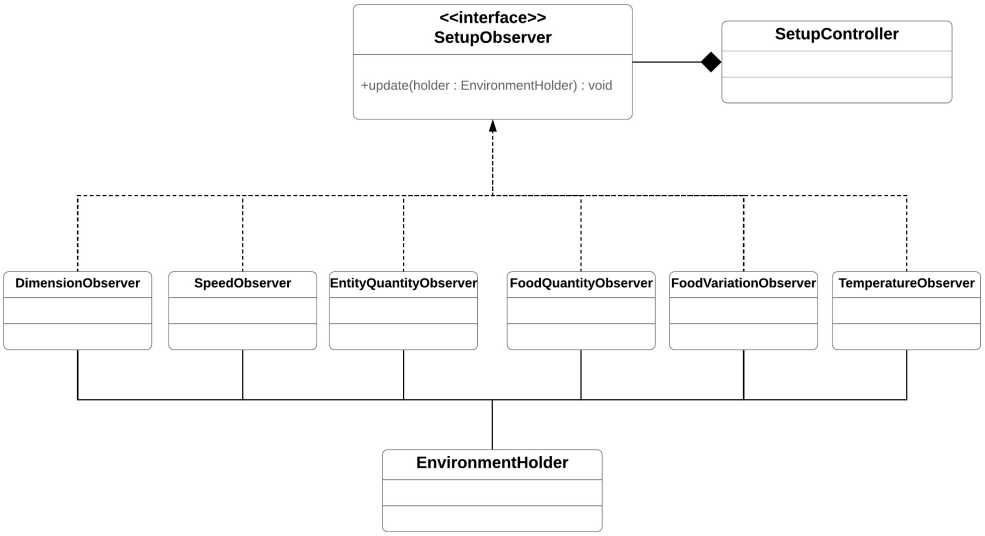


Inoltre per gestire le varie modifiche apportate alle impostazioni durante l'esecuzione ho utilizzato il pattern Observer in modo tale da creare diverse gestioni per ogni tipologia di impostazione. L'interfaccia Observer è SettingsObserver implementata attualmente dall'unica tipologia di impostazione presente: la durata del giorno. Ho collegato poi tutti gli observer al controller della scena delle impostazioni in modo tale da rendere l'aggiunta di nuove impostazioni molto semplice e poco onerosa.



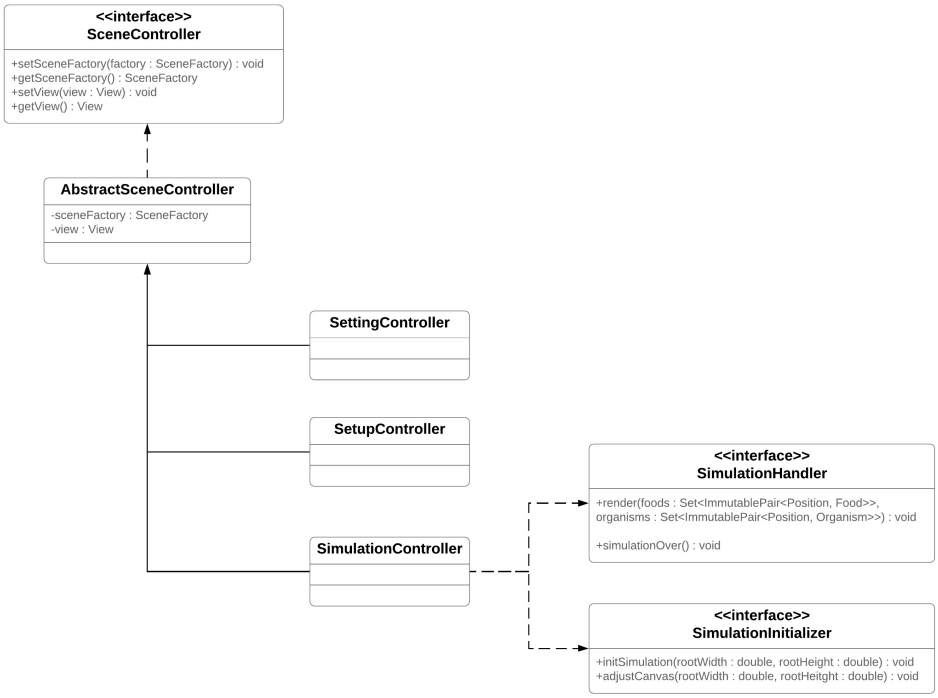
Setup Iniziale

Anche per la gestione del setup iniziale ho utilizzato il pattern Observer. Ho collegato alla scena di Setup Iniziale un observer per ogni configurazione (Quantità di cibo iniziale, velocità iniziale, temperatura dell'ambiente, ecc..) passando anche il componente grafico utilizzato per effettuare la scelta. Utilizzando il pattern observer si rende molto più semplice la possibilità di aggiungere nuove tipologie di configurazione: basterà creare un nuovo observer ed aggiungerlo all'observable (ruolo che in questo caso è ricoperto dal controller del Setup Iniziale).

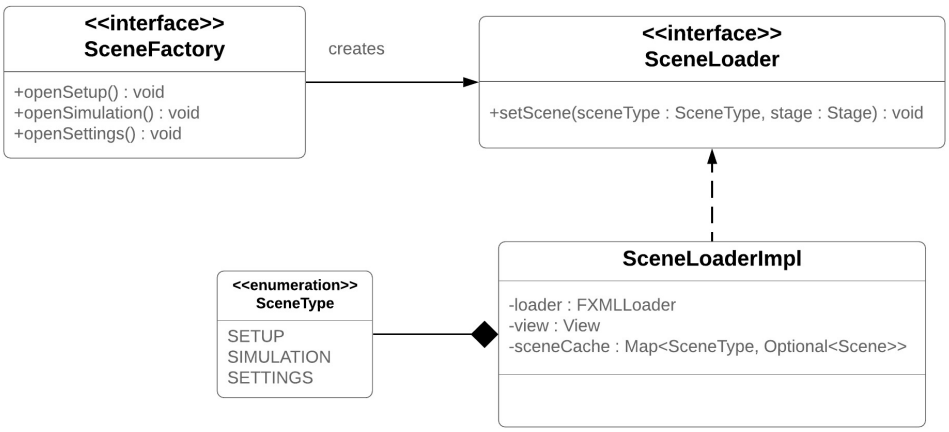


Gestione View

Questa parte di progettazione include la gestione delle varie scene del simulatore, cioè tutto ciò che riguarda la creazione, la chiusura, la transizione, quindi la navigazione tra le varie schermate in funzione dell'input dell'utente. Ho deciso di utilizzare JavaFx, con descrizione della GUI tramite il linguaggio FXML in modo da separare la parte grafica dalla logica che ne gestisce il comportamento. Al fine di controllare ogni scena indipendentemente le une dalle altre ho collegato ogni file FXML al rispettivo controller. Ogni controller in realtà ha delle parti in comune che sono state modellate nell'interfaccia SceneController ed implementate nella classe astratta AbstractSceneController da cui tutti i controller ereditano. Ovviamente nel caso in cui un controller necessiti di ulteriori operazioni, ho fatto in modo che implementasse sue specifiche interfacce oltre che alla classe astratta, come nel caso del Simulation Controller (nel rispetto del principio ISP).



Al fine di rendere univoco ed estendibile la gestione della navigazione tra le varie scene del simulatore ho creato due concetti separati: Scene Loader e Scene Factory. Innanzitutto per la creazione delle varie scene ho utilizzato il pattern Factory attraverso l'interfaccia SceneFactory rendendo veloce l'introduzione di nuove schermate. Lo SceneFactory demanda il caricamento della scena all'interno dello stage allo SceneLoader. Lo Scene Loader si occupa essenzialmente del caricamento della scena all'interno dello stage, quindi collega file FXML e relativo controller, imposta la risoluzione corrente, aggiunge i fogli di stile e lancia l'eventuale inizializzazione della schermata.



Simone Ceredi

Il mio compito all'interno del progetto è stato quello di modellare l'*ambiente*, popolato di organismi e di cibo per essi, di gestire il *ciclo della giornata* e insieme ad Alessio Conti dell'*aggiornamento della schermata di simulazione*.

Environment

Ho deciso di gestire l'*ambiente* all'interno del quale gli organismi si muovono alla ricerca di cibo utilizzando un'interfaccia *Environment* ed una sua implementazione tramite una classe astratta *AbstractEnvironment* contenente i metodi comuni a tutti gli ambienti.
Queste sono poi state raffinate per andare a creare *BasicEnvironment* permettendo così la gestione di un ciclo giorno/notte e la modifica giornaliera della quantità di cibo che gli organismi avrebbero trovato all'interno dell'ambiente.
Ho infine creato un *AdvancedEnvironment* all'interno del quale ho gestito il parametro *Temperatura* e il *FoodRadar*, ovvero la possibilità degli organismi di mangiare cibo anche a breve distanza dalla loro posizione.
Per la gestione delle collisioni ho utilizzato il pattern Template Method, realizzato grazie al metodo astratto *checkPosition*, in questo modo ogni sottoclasse di abstractEnvironment può creare una implementazione che sia in grado o meno di gestire le collisioni. (Figura 2.8)

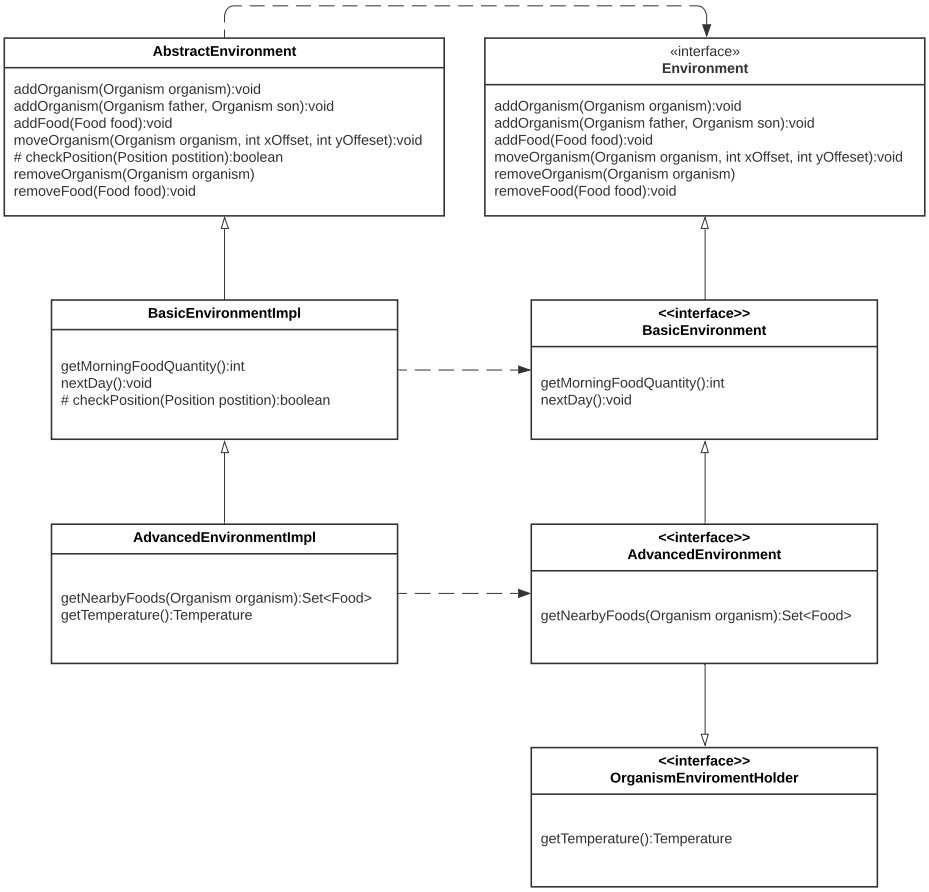


Figura 2.8: Schema UML gestione Environment.

Per permettere una semplice creazione dei suddetti elementi ho optato per la creazione di una Factory. Questa permette l'istanziazione sia di *BasicEnvironment* che di *AdvancedEnvironment*. Ho preferito l'utilizzo di una Factory al posto di due Builder in quanto tutti i parametri necessari per la creazione dei due tipi di ambiente sono obbligatori. (Figura 2.9)

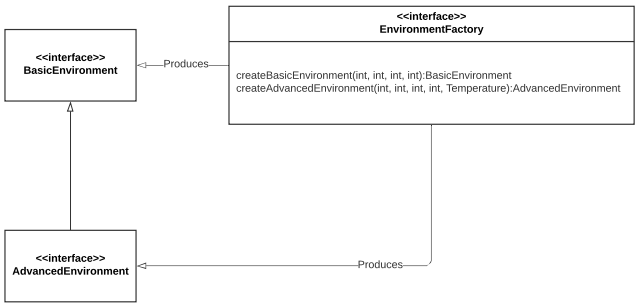


Figura 2.9: Schema UML Factory per l'Ambiente.

Un punto sul quale mi vorrei soffermare è la gestione delle *posizioni*. È stato deciso di non permettere ad un organismo di conoscere la sua posizione esatta (in coordinate) all'interno dell'ambiente. È invece l'ambiente che conosce le posizioni di tutti gli organismi e i cibi. I movimenti fatti da un organismo vengono passati all'ambiente sotto forma di offset rispetto alla posizione nella quale l'organismo si trovava all'istante precedente. Dato che l'organismo si muove in modo randomico era possibile che esso provasse ad uscire dai limiti dell'ambiente. Per evitare di lasciare l'ambiente in uno stato inconsistente (organismi fuori dall'environment) ho implementato una eccezione *OutOfEnvironmentException* che sarà poi gestita dal gestore dei movimenti. Questa scelta viene dal fatto che l'ambiente non è in grado di decidere se lasciare un organismo nel punto in cui si trovava prima del movimento oppure eliminarlo.

Gestione ciclo della giornata

La gestione del ciclo della giornata è stata implementata tramite una enum *DayPeriod*, un'interfaccia *DayCicle* e la sua implementazione.
All'interno di *DayPeriod* sono contenuti tutti gli stati in cui una giornata si può trovare ad un certo istante, ovvero in questo caso solamente *DAY* o *NIGHT*.
DayCicle permette di ottenere il *DayPeriod* corrente e inoltre gestisce la divisione tra i vari periodi della giornata. (Figura 2.10)

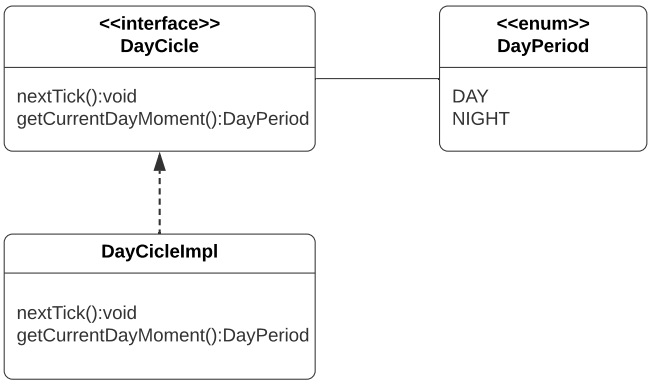


Figura 2.10: Schema UML gestione ciclo giorno/notte.

Simulation Loop

Per la temporizzazione e l'esecuzione di tutti i task all'interno della simulazione ho utilizzato il pattern *Game Loop* riadattato per la gestione delle simulazioni, andando a considerare la simulazione come un gioco dove non vi sono input da parte dell'utente, ma che si evolve autonomamente.
Il loop stabilisce il numero di aggiornamenti che vengono eseguiti durante una giornata. Durante la sua esecuzione scandisce il tempo e richiama l'update e il render della simulazione. Per fare questo si compone di un *DayCicle* al quale delega la gestione del ciclo della giornata.
Sfruttando una *mutex* sono andato a gestire la possibilità di mettere in pausa e riprendere la simulazione. (Figura 2.11)

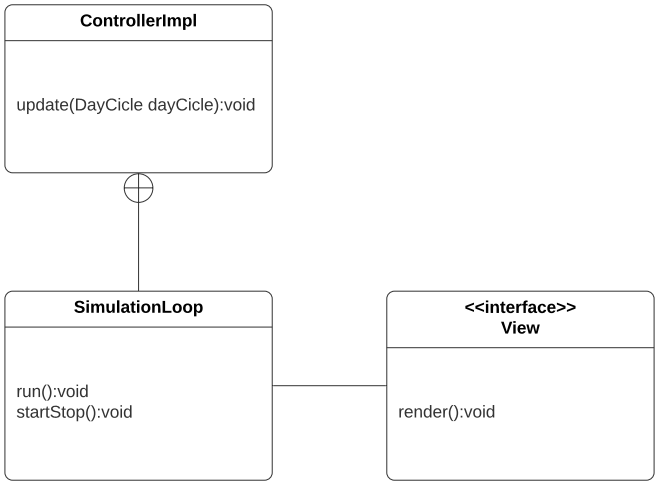


Figura 2.11: Schema UML loop di simulazione.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Durante lo sviluppo del nostro progetto abbiamo utilizzato la libreria JUnit per testare in modo automatizzato alcune funzionalità.
Funzionalità testate automaticamente:

- *Environment*: viene testata l'aggiunta e rimozione di organisms e foods, gestione della temperatura e passaggio da una giornata alla successiva.
- *Organism*: viene testata la creazione di un organismo tramite apposito Builder, sottoposti a test anche casi fallaci che determinano eccezioni. Verifica dei traits inseriti e della quantità di energia.
- *Food*: testata la creazione di un food tramite apposito Builder e verificato lo stato di energia.
- *Action*: vengono testate alcune casistiche delicate che possono risultare indice di errore durante il funzionamento dell'applicazione e passare inosservate; soprattutto nei casi in cui vi è una maggiorazione di organismi in vita.
- *Trait*: è stato testato la corretta memorizzazione dei valori e il calcolo del consumo di cibo.
- *MutatedFactory*: testato la creazione di un nuovo individuo, con particolare interesse nel controllare che i riferimenti agli oggetti padre e figlio fossero diversi.
- *Settings*: è stata testata la corretta gestione dei valori.

Funzionalità testate manualmente:

- *GUI*: effettuato test manuale in quanto il monte ore non risultava sufficiente per studiare anche il testing automatico delle GUI in JavaFX.

3.2 Metodologia di lavoro

A seguito della fase di sviluppo, possiamo asserire che la ripartizione dei compiti da noi concordata inizialmente e confermata nelle fasi precedenti è stata rispettata, risultando equa e piuttosto bilanciata. Non sono state necessarie sostanziali modifiche e le dipendenze tra le nostre parti sono limitate. Ciò è dovuto al fatto che parte di analisi architetturale è stata svolta assieme in un momento antecedente allo sviluppo, anche utilizzando strumenti di connessione virtuale quali "Skype" e "lucidchart.com", in tal modo abbiamo definito quali interfacce usare e solamente in un secondo momento ci siamo occupati in maniera indipendente di sviluppare le differenti parti del sistema.

Di seguito elenchiamo le funzionalità implementate dai singoli elementi del team:

- *Alessio Conti*: gestione delle entità della simulazione (organismi e cibo), gestione delle azioni svolte dagli organismi durante le varie fasi della giornata, gestione dei movimenti.
- *Andrea Giulianelli*: gestione dei trait degli organismi, gestione delle impostazioni e creazione, navigazione nelle schermate di simulazione, grafici.
- *Simone Ceredi*: gestione dell'ambiente di simulazione, gestione del loop di simulazione, gestione del ciclo della giornata.

Le parti comuni tra i componenti del team:

- Aggiornamento della simulazione Ceredi Simone, Conti Alessio per quanto riguarda l'aggiornamento di ambiente, organismo e cibo, Giulianelli Andrea per la parte che concerne l'aggiornamento in real time dei grafici. Le classi interessate sono SimulationController la quale gestisce la scena della simulazione e SimulationViewLogics corredata da relativa implementazione per gestire il render della canvas.

Nella fase di sviluppo abbiamo utilizzato il DVCS Git. Abbiamo creato una repository su BitBucket e ciascun membro del team ha effettuato la clone. Sul branch develop è stata sempre mantenuta una versione aggiornata del

codice che fosse priva di errori di compilazione. Ogni componente ha lavorato autonomamente sul proprio repository in locale, condividendo su develop tramite push al termine di ogni task a meno che non ci fossero errori di compilazione. Sul branch master è stata aggiunta solo la versione funzionante dell'applicazione.

3.3 Note di sviluppo

Alessio Conti

- *Stream*: uso degli stream per la gestione di collezioni.
- *Lambda expression*: ho specificato alcune lamda expressions con l'obiettivo di avere codice più compatto e leggibile, spesso in concomitanza di stream; in particolare nelle classi relative ai compiti assegnati ad ogni organismo.
- *Optional*: in ogni caso in cui una classe poteva avere delle informazioni assenti è stato scelto di usare Optional.
- *Libreria Apache Commons*: per non ricreare classi già implementate, limitatamente all'utilizzo di Pair.
- *JavaFX*: per gestire la parte relativa alla GUI del progetto.

Per quanto riguarda pattern progettuali ho consultato il materiale didattico delle lezioni in aula, approfondendo il tutto con il libro *"Design Patterns: Elements of Reusable Object-Oriented-Software"* - Gamma, Helm, Johnson, Vlissides. Grazie a tale testo sono entrato a conoscenza del pattern *Command*, utilizzato per gestire le azioni che devono essere svolte durante i vari periodi della giornata dagli organismi.

Andrea Giulianelli

- *Lambda expression*: impiegate al fine di rendere il codice più compatto, meno verboso e di maggiore qualità. Sono state utilizzate principalmente per definire la strategia all'interno degli Stream, per l'Handler degli eventi (GUI, oppure relativi al cambio di risoluzione).
- *Optional*: impiegati all'interno di SceneLoaderImpl, allo scopo di rappresentare oggetti che possono assumere valori nulli, consentendo di generare codice snello e meno incline agli errori di gestione dei null.

- *Stream*: utilizzati per le operazioni che coinvolgevano le collezioni lavorando in maniera efficiente e compatta. Utilizzati soprattutto nella mutazione degli organismi oppure per impostare i valori delle combobox del menu iniziale.
- *Libreria Apache Commons*: utilizzata per evitare di risviluppare le ImmutablePair.
- *JavaFX*: Tutta la GUI relativa al progetto è stata realizzata utilizzando JavaFX. Inoltre ho utilizzato questa libreria anche per la creazione e la gestione dei Grafici.
- *FXML*: utilizzato per descrivere il design delle varie scene.
- *CSS*: utilizzato per migliorare l'aspetto della GUI.

Per l'apprendimento di JavaFx ho utilizzato le slide del corso, documentazione online e corsi online. Come riferimento e studio dei pattern ho utilizzato il testo *"Design Patterns: Elements of Reusable Object-Oriented-Software"* - Gamma, Helm, Johnson, Vlissides. La creazione delle varie scene grafiche sono unicamente frutto di continui tentativi così come la realizzazione dei fogli di stile per decorare la GUI. Il design della mia parte è frutto di mie riflessioni ed osservazioni. Per la navigazione tra scene ho preso spunto da un vecchio progetto: Magnum Chaos, poichè mi sembrava un ottima divisione della parte di loading e creazione delle scene. Dopo aver studiato a fondo la gestione delle scene di JavaFx per mezzo di testi e siti web, ho reimplementato completamente entrambe le parti (SceneLoader e SceneFactory), aggiungendo inoltre alcune feature come ad esempio il caching delle varie scene create, quindi nessuna parte di codice è stata copiata. Ho inoltre studiato la gestione dei grafici offerta da JavaFx a questo link: `https://levelup.gitconnected.com/realtime-charts-with-javafx-ed33c46b9c8d` adattandolo alle mie richieste. Inizialmente ho anche seguito una serie di video su JavaFx che spiegavano i vari componenti: `https://www.youtube.com/watch?v=FLkOX4Eez6o&list=PL6gx4Cwl9DGBzfXLWLSYVy8EbTdpGbUIG`. Ho preso spunto su come rendere la Canvas ridimensionabile da alcuni post su StackOverflow e dalla documentazione di JavaFx.

Simone Ceredi

- *Stream*: utilizzati principalmente per le operazioni svolte sulle collezioni all'interno dell'Environment.

- *Lambda expression*: utilizzate solitamente all'interno degli stream così da rendere il codice più compatto e leggibile.
- *Optional*: ogni volta che una classe poteva avere delle informazioni assenti o che un metodo poteva ritornare un valore nullo.
- *Libreria Apache Commons*: utilizzata per sfruttare l'implementazione di ImmutablePair evitando di risvilupparla.
- *JavaFX*: per la gestione della parte di GUI della simulazione.

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 e infine al seminario del Professor Ricci e alcuni video di altri seminari svolti dallo stesso reperiti dalla piattaforma Youtube.com (`https://www.youtube.com/watch?v=xBoB1Q1yD_E`).

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

Alessio Conti

Sono personalmente soddisfatto del mio lavoro svolto all'interno del progetto. Fin dall'inizio è stato un progetto ambizioso e complesso ma proprio per questo interessante e stimolante.
Essendo la prima volta che affrontavo un progetto di tale portata e la prima volta ad affrontarlo in gruppo non nascondo il fatto che in un primo momento le incertezze sono state svariate e di differente natura. Tuttavia lavorare con un metodo ferreo e progettare l'ossatura del progetto prematuramente e in maniera dettagliata, per quanto concerne le parti che avrebbero dovuto in futuro collegarsi tra loro, ha giocato un ruolo decisivo nel riuscito sviluppo del progetto senza causare troppi rallentamenti nel momento di scrittura del codice; confermando quanto detto dai professori a lezione: una buona progettazione alla base è la chiave per un riuscito progetto.
Sono soddisfatto della mia parte del progetto. Ho cercato sempre di tenere alta la riusabilità ed estendibilità del codice anche grazie a diversi pattern sia costruttivi che di design. Sono consapevole del fatto che alcune implementazioni possano non essere le migliori, ma ritengo di aver messo in pratica tutte le conoscenze che al momento posseggo e di aver fatto del mio meglio in ogni sezione di cui mi sono occupato.
Ritengo di aver raggiunto un buon punto di estendibilità del codice, a riprova di ciò posso affermare che è risultato molto semplice e poco dispendioso implementare alcune funzionalità aggiuntive che non erano previste tra quelle ritenute obbligatorie. Per esempio aggiungendo il trait *"FoodRadar"*, mi è bastato modificare alcune parti relative alla logica del movimento per giunge-

re al risultato desiderato senza compromettere componenti terzi. Considero tale progetto una solida base di partenza per poterci lavorare in futuro. Le possibilità di arricchirlo sono svariate: dall'aggiunta di caratteristiche proprie degli Organismi a caratteristiche riguardanti l'ambiente in cui vivono.
Può essere interessante modellare il concetto di specie di individuo, ognuna con caratteristiche proprie, e farle convivere nello stesso ambiente così da studiare e mettere in campo una vera e propria "lotta alla sopravvivenza".
Allo stesso modo mettendo a disposizione varie tipologie di cibi si potrebbe modellare una sorta di AI propria degli organismi che gli permetta di scegliere il cibo "migliore" da ingerire in base alle caratteristiche e alle necessità della propria specie di appartenenza.
Inoltre sarebbe interessante estendere l'applicazione con opzioni aggiuntive e maggiormente dettagliate per lo studio dell'evoluzione degli organismi, in modo da poter interagire e meglio esaminare l'andamento di essa.

Andrea Giulianelli

Personalmente sono soddisfatto del lavoro svolto. Al fine di ottenere la maggiore efficienza e il miglior risultato possibile ho cercato di dedicare tanto tempo alla fase di analisi del problema e soprattutto alla modellazione della soluzione mediante schemi UML. Di questa scelta sono veramente tanto soddisfatto perchè mi ha concesso di implementare la mia parte senza troppe difficoltà. Sono molto contento anche perchè penso di essere arrivato ad un buon punto di estendibilità del codice, cosa che ho provato personalmente in quanto i trait "Food Radar" e "Temperature Sensibility" inizialmente non li avevo implementati, ma successivamente visto che mi avanzavano alcune ore ho deciso di implementarli ed è stato veramente un'operazione poco onerosa. Sono molto soddisfatto anche del caching dei grafici e delle scene che mi hanno risolto alcuni problemi nella navigazione tra schermate ed hanno reso il codice molto più leggero. Durante la progettazione ho cercato di aiutare anche gli altri componenti del team, quando vi era un problema ho sempre cercato di aiutarli a trovare la soluzione. Sicuramente c'è tanto margine di miglioramento nella mia progettazione e nel mio codice proposto, però mi ritengo soddisfatto considerando anche le tempistiche e il fatto di dover portare avanti il progetto durante esami e lezioni universitarie. Un problema che ho riscontrato è come gestire nel migliore dei modi lo scaling alle varie risoluzioni; inizialmente avevo utilizzato i metodi di scale offerti da JavaFX, ma sono metodi che vengono utilizzati solitamente per fare animazioni e quindi non erano adatti. Ho cercato varie soluzioni, ma nel monte delle 80 ore non sono riuscito a fare meglio di così. Sono molto contento di come

abbiamo lavorato in team: siamo partiti tutti e tre con le idee molto chiare e abbiamo fin da subito diviso i compiti in modo equo e senza creare troppe dipendenze. Inoltre c'è stata tanta collaborazione ma soprattutto rispetto delle varie deadline che ci imponevamo in modo da mantenere un ritmo di sviluppo elevato e costante. Mi piacerebbe portare avanti questo progetto, introducendo nuovi trait e soprattutto utilizzando l'intelligenza artificiale al fine di eseguire mutazioni che realmente mirano ad adattarsi all'ambiente circostante. E anche in caso non avesse un futuro, sono veramente felice perchè mi è stato utile per comprendere il lavoro in team e soprattutto l'importanza nel dedicare tanto tempo all'analisi e alla progettazione che è stata sicuramente un'ottima scelta.

Simone Ceredi

Mi ritengo soddisfatto del lavoro che ho svolto all'interno del progetto. Ritengo che un punto che ha giocato a favore dell'intero gruppo sia stata la progettazione svolta con precisione prima dell'effettiva implementazione, che mi ha permesso di implementare la mia parte senza difficoltà, nonostante inizialmente il progetto sembrasse complesso.
All'interno delle mie implementazioni ho cercato di mantenere alto il livello di riusabilità e estendibilità del codice utilizzando al meglio delle mie possibilità i diversi pattern di design. Quando è stata aggiunta la *temperatura* all'Environment e il tratto *FoodRadar* agli organismi, l'inserimento della mia sezione all'interno del codice mi è risultata molto agile. Ho realizzato le mie sezioni di progetto al meglio delle mie possibilità, sono tuttavia consapevole che alcune implementazioni non siano ottimali così come la progettazione.
Ritengo che il progetto svolto sia facilmente estendibile. Si potrebbe implementare un'intelligenza artificiale per gli organismi, oltre che aggiungere vari tratti agli organismi e all'ambiente, popolare l'ambiente con ostacoli che non permettano agli organismi di muoversi liberamente ed aggiungere valori di mutazioni plausibili in modo da creare vere e proprie simulazioni realistiche.

4.2 Difficoltà incontrate e commenti per i docenti

Andrea Giulianelli

Le difficoltà principali riguardano il ridimensionamento delle finestre alle varie risoluzioni. Ho provato in un'infinità di modi a risolvere questo problema

in una maniera efficiente, e alla fine ho cercato di risolvere il tutto al meglio delle mie possibilità. L'ultimo ostacolo presentatomi è stata la gestione del ridimensionamento della canvas della simulazione. Ho deciso di creare una mia classe ResizableCanvas che estendesse canvas rendendola ridimensionabile. Mi sento di dare due suggerimenti:

- Dedicare leggermente più tempo a JavaFX invece che a Swing in quanto la maggior parte dei progetti utilizza JavaFx e soprattuto dare delle linee guida su come gestire le varie risoluzioni.

- Inoltre dedicare del tempo anche a consigliare come gestire in modo appropriato le differenze in alcuni contesti tra Java e C# come ad esempio le enum. Nel progetto faccio largo uso dei metodi all'interno delle enum cosa che in C# non è concessa. Alla fine anche qui ho cercato di trovare la soluzione più efficiente ed estendibile, al meglio delle mie possibilità.

  Nonostante queste mie piccole osservazioni sono molto soddisfatto di come è stato svolto il corso, soprattuto per l'importanza che date al rispetto delle regole, all'analisi del problema e per la passione che trasmettete durante le vostre lezioni.

Appendice A

Guida utente

Avviando la simulazione si incorre nella schermata:

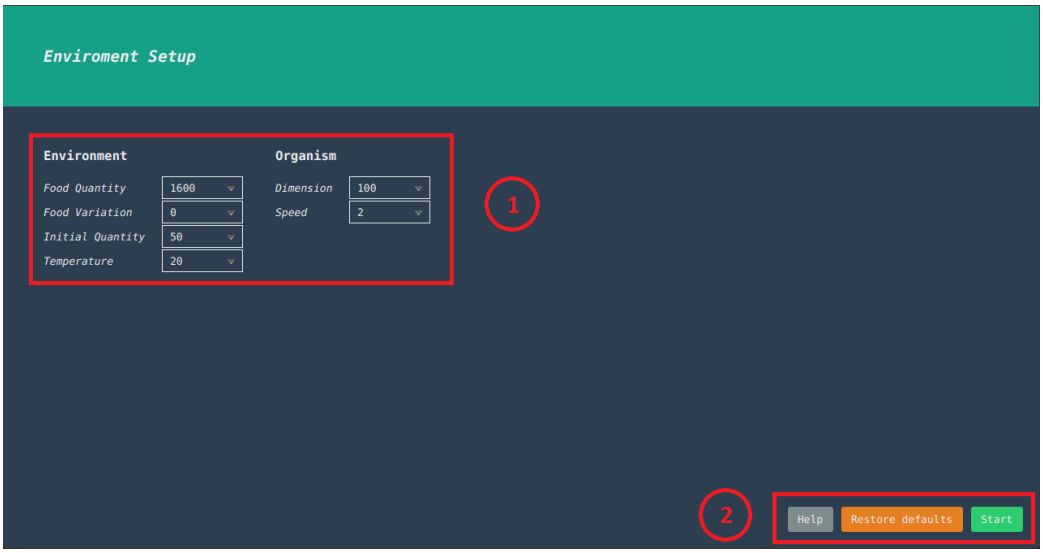


Figura A.1: Schermata Iniziale del simulatore

E' possibile scegliere alcuni valori iniziali dal menù **(1)**

- *Food Quantity*: quantità di cibo disponibile nell'ambiente
- *Food Variation*: variazione di cibo giornaliera
- *Initial Quantity*: quantità iniziale di individui
- *Temperature*: temperatura dell'ambiente

- *Dimension*: dimensione iniziale degli individui
- *Speed*: velocità inziale degli individui

Nella parte inferiore destra della schermata, **(2)** l'utente trova tre pulsanti:

- *Help*: descrive le informazioni riguardanti gli organismi durante la simulazione
- *Start*: inizia la simulazione
- *Restore default*: per reimpostare i valori iniziali

Alla pressione del tasto start la schermata visualizzata sarà similare all'Figura A.2

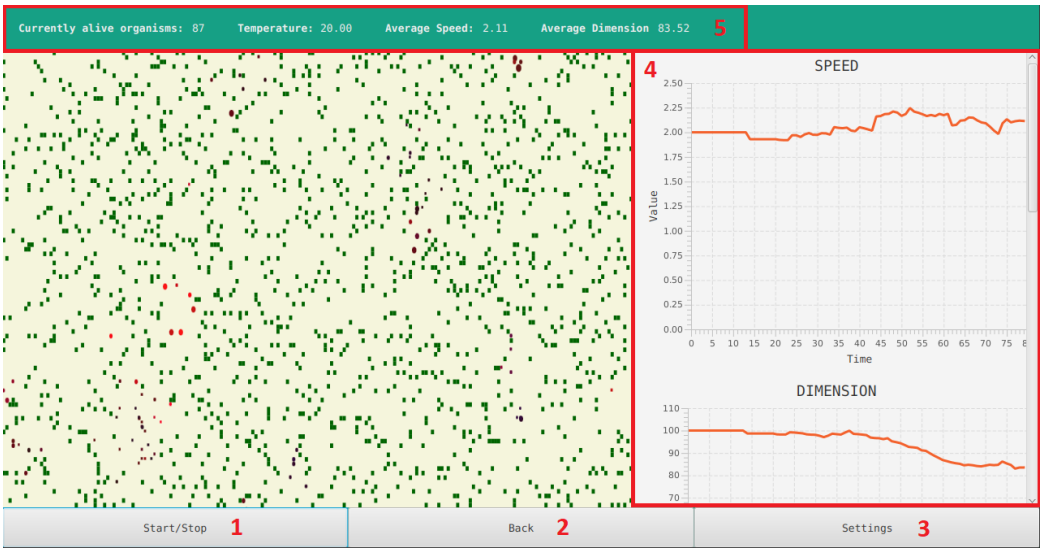


Figura A.2: Schermata Simulazione

Nella parte centrale verranno visualizzati in quadratini verdi le unità di cibo, mentre in varie gradazioni di rosso gli organismi. L'individuo a seconda dei valori assunti dalle varie caratteristiche cambia la sua rappresentazione:

- Il raggio dell'individuo è direttamente proporzionale alla sua dimensione.

- La gamma cromatica degli organismi spazia da colori scuri tendenti al nero fino al rosso acceso.
  Velocità (Rosso), Food Radar (Marrone) e Children Quantity (Viola) influiscono sul colore. Ciò che influenza fortemente il colore è la velocità dell'organismo. A velocità alte il colore tenderà al rosso acceso, mentre a velocità basse il colore sarà scuro. I restanti caratteri invece influenzano la gamma cromatica in minor rilievo determinando sfumature: più il valore è basso e più incidono con una tonalità chiara del colore associato, contrariamente più il valore aumenta e più la tonalità diventa scura.

I tasti nella parte inferiore **(1,2,3)** permettono di gestire la simulazione. **(1)** Permette di mettere in pausa e far ripartire la simulazione. **(2)** Permette di tornare al menù iniziale Figura A.1. **(3)** Consente di aprire la schermata delle impostazioni Figura A.3. Nella parte destra **(4)** vengono visualizzati i grafici sull'andamento medio dei caratteri dell'intera popolazione. Infine nella parte superiore **(5)** vengono visualizzate alcune informazioni riguardanti la popolazione in vita.

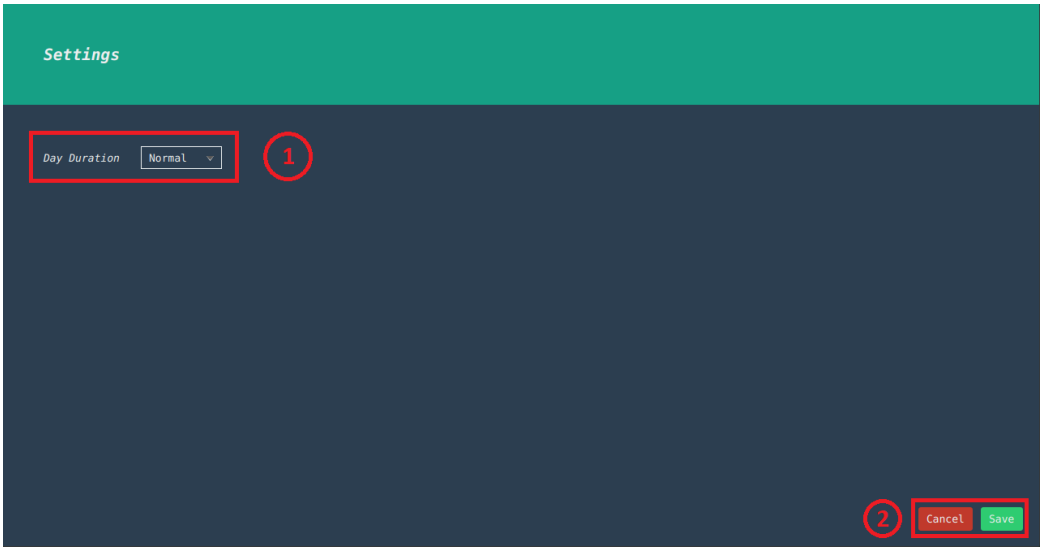


Figura A.3: Schermata Settings

La schermata delle impostazioni Figura A.3 permette di settare la velocità di aggiornamento della simulazione. **(1)**

Bibliografia

[1] Design Patterns Elements of Reusable Object-Oriented Software di Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides