ARES
"AgentBased Realistic Environment Simulation"

Leonardo Naddei, Domeniconi Lorenzo
Abou El Kheir Uhalid, Di Varano Lorenzo

26 luglio 2024

Indice

Capitolo 1

Analisi

1.1 Requisiti

L'obiettivo del progetto è lo sviluppo di un motore di calcolo per simulazioni di modelli ABM. Un ABM (Agent Based Model) è un tipo di modello computazionale utilizzato per simulare sistemi complessi in cui gli agenti individuali seguono regole e interagiscono tra loro e con l'ambiente circostante. Gli "agenti" in un ABM possono rappresentare entità eterogenee come individui, gruppi o qualsiasi altra unità che agisce autonomamente all'interno del sistema.

1.1.1 Requisiti funzionali

- L'applicazione dovrà presentare un'interfaccia grafica intuitiva per consentire agli utenti di configurare e avviare le simulazioni in maniera più semplice possibile nonostante l'elevata complessità dell'operazione.
- Deve essere possibile visualizzare i risultati delle simulazioni in modo chiaro e comprensibile tramite la GUI.
- L'interfaccia utente deve permettere agli utenti di avviare, mettere in pausa, fermare e riprendere le simulazioni in corso.
- Il sistema deve supportare l'avvio simultaneo di più simulazioni, anche da parte di utenti differenti in modo concorrente, garantendo la corretta isolazione dei dati e dei risultati.

Oltre al sistema in se ogni modello che verrà implementato dovrà rispettare una serie di requisiti, a seguito una breve descrizione

Fire spread model (Di Varano Lorenzo)

Simula il comportamento della propagazione del fuoco in un ambiente forestale.
Il modello è costituito da tre tipi di agenti:

- gli agenti Fire che rappresentano il fuoco.
- gli agenti Tree che rappresentano gli alberi presenti nell'ambiente.
- gli Agenti Extinguished che rappresentano agenti Fire estinti.

Gli agenti Fire si propagano nell'ambiente, ogni agente Fire ha:

- una posizione all'interno della griglia
- un certo quantitativo di combustibile (fuel)
- un tasso di consumo (consumption)

I suddetti agenti seguono una direzione, essa è definita in maniera casuale.
Gli agenti Tree rappresentano gli alberi nell'ambiente.
Ogni agente Tree ha:

- una posizione, una capacità di combustione (flammability)
- n quantitativo di combustibile (fuel)

Se colpito dal fuoco, si trasformerà in un agente Fire, con la già presente capacità di combustibile e un tasso di consumo calcolato utilizzando quella del agente fuoco che l'ha trasformato e la sua capacità di combustione.

Schelling's segregation model (Naddei Leonardo)

Modello basato ad agenti atto a simulare dinamiche di segregazione tra due gruppi in funzione di una soglia di soddisfazione.
Ogni agente ha:

- una posizione
- un tipo che potrà essere 'A' oppure 'B' θ
- un raggio di visione ρ ovvero la distanza (in celle) entro cui l'agente valuta la sua soddisfazione
- una soglia di tolleranza α

A ogni iterazione del modello, per ogni agente, viene verificata se è soddisfatto, in caso contrario l'agente si muove in una nuova posizione casuale.

La soddisfazione di una agente x è determinata dalla formula

$$\frac{\{\#agent \in \rho | \theta(agent) = \theta(x)\}}{\{\#agent \in \rho | \theta(agent)! = \theta(x)\}} > \alpha \tag{1.1}$$

Il modello termina quando lo stato corrente e il precedente sono uguali ovvero quando la soglia di tolleranze è soddisfatta per ogni agente

Virus model (Domeniconi Lorenzo)

Modello basato su agenti atto a simulare la diffusione di un virus all'interno di una popolazione, composta da due agenti: infetti e sani (persone). Gli agenti rappresentanti individui sani hanno:

- Una posizione
- Una stepSize ovvero la dimensione dello spostamento (in celle) che l'agente può compiere per iterazione
- Una direzione di movimento (inizializzata in maniera casuale)
- InfectionRate, ovvero una probabilità d'infezione ogni volta che avviene lo scontro con un agente di tipo I

Gli agenti rappresentanti individui infetti hanno:

- Una posizione
- Una stepSize ovvero la dimensione dello spostamento (in celle) che l'agente può compiere per iterazione
- Una direzione di movimento (inizializzata in maniera casuale)
- recoveryRate, una probabilità di guarire, a ogni iterazione, in maniera autonoma.

A ogni iterazione ogni agente sano può muoversi o essere infettato, mentre un agente infetto può muoversi o essere guarire.

Quando si spostano gli agenti seguono il seguente algoritmo:

1. Viene calcolata una nuova posizione in base a direzione e stepsize

2. Se la nuova posizione è fuori dai limiti della board allora la direzione viene invertita e la posizione ricalcolata (punto 1)

3. Se la nuova posizione è libera l'agente viene spostato altrimenti

    - Se l'agente è del tipo sano e nella cella è presente un paziente del tipo infetto si avvia il processo d'infezione dell'agente
    - Altrimenti si calcola una posizione casuale, se libera l'agente viene spostato altrimenti rimane fermo

Processo d'infezione:
La probabilità d'infezione è espressa in centesimi, quando si verifica un contatto viene generato un numero n, l'esito dell'infezione è il seguente

- Infetto se $n < InfectionRate$
- Non infetto altrimenti

Se si verifica che l'agente venga infettato allora l'agente sano viene eliminato e ne viene creato uno infetto al suo posto

Boid's model (Naddei Leonardo)

Modello basato ad agenti atto a simulare le dinamiche di volo di un gruppo di uccelli. In questo modello la direzione del singolo agente è risultante di cinque elementi:

- La direzione media 'a' degli agenti presenti nel suo cono di visione
- Il baricentro 'b' (posizione media) degli agenti presenti nel suo cono di visione
- La direzione che mi permette di evitare collisione 'c' (media delle direzioni opposte a quelle che mi porterebbero alla collisione)
- La direzione originale dell'agente 'd'
- La direzione che mi allontana dal perimetro dell'ambiente di simulazione 'e'

Ogni agente ha i seguenti parametri

- Una direzione (inizializzata in maniera casuale)
- Un raggio di visione ρ ovvero la distanza (in celle) entro cui l'agente percepisce la presenza di altri agenti

- Un angolo di visione α ovvero l'angolo che assieme al raggio determina il cono all'interno di cui l'agente percepisce la presenza di altri agenti, assume valori da 1 a 180 dove 180 significa il sia davanti che dietro
- alignemnt weight w_a, varia da 0.0 a 1.0 e rappresenta il peso della direzione data da a
- cohesion weight w_b, varia da 0.0 a 1.0 e rappresenta il peso della direzione data da b
- alignemnt weight w_a, varia da 0.0 a 1.0 e e rappresenta il peso della direzione data da c

A ogni tick la direzione di ogni agente è ricalcolata secondo la seguente formula:

$$direction = 0.4 * (w_a * a + w_b * b + w_c * c) + 0.6 * d + 0.2 * e \tag{1.2}$$

Il modello non ha condizioni di uscita

Predator Prey Model (Abou El Kheir Uhalid)

E' un modello basato su agenti utilizzato per simulare le dinamiche di interazione tra predatori e prede in un ambiente chiuso. Gli agenti in questo modello rappresentano due tipi di entità biologiche: i *predatori* e le *prede*.
Caratteristiche degli agenti predatori:

- **Posizione:** Ogni predatore è posizionato in uno spazio bidimensionale.
- **Raggio di visione:** Distanza entro la quale un predatore può identificare la presenza di prede.
- **Comportamento:** I predatori cercano le prede nel loro raggio visivo e si spostano verso di esse per catturarle.

Caratteristiche degli agenti prede:

- **Posizione:** Anche le prede sono posizionate nello spazio bidimensionale.
- **Raggio di visione:** Distanza entro la quale una preda può percepire la presenza di predatori.

- **Comportamento di fuga:** Quando una preda percepisce un predatore nel suo raggio di visione, tenta di muoversi nella direzione opposta per evitare di essere catturata.

Dinamica del modello: A ogni iterazione del modello:

1. **Rilevamento:** Ogni predatore verifica la presenza di prede nel proprio raggio di visione. Se una preda è presente, il predatore si muove verso di essa.

2. **Cattura:** Se un predatore raggiunge una preda, questa viene rimossa dall'ambiente.

3. **Movimento di fuga:** Le prede rilevano i predatori nel loro raggio di visione e si muovono nella direzione opposta per sfuggire.

Il comportamento degli agenti predatori e delle prede è definito tramite strategie specifiche che considerano lo stato corrente dell'ambiente e la posizione degli altri agenti. Queste strategie sono implementate attraverso lambda espressioni che modificano lo stato del modello a ogni iterazione.

Condizioni di terminazione: Il modello termina quando:

- Tutte le prede sono state catturate, oppure

- Non ci sono cambiamenti nello stato tra due iterazioni consecutive, indicando che un equilibrio è stato raggiunto.

Il Predator-Prey Model fornisce un interessante esempio di simulazione delle interazioni predatorie in un ambiente controllato, mostrando come comportamenti semplici a livello individuale possano portare a dinamiche complesse a livello di sistema.

Sugarscape Model (Abou El Kheir Uhalid)

Il modello Sugarscape è un modello basato su agenti che simula un ambiente in cui agenti consumatori interagiscono con risorse di zucchero. Il modello è costituito da due tipi di agenti:

- Agenti Consumer: rappresentano i consumatori di zucchero

- Agenti Sugar: rappresentano le risorse di zucchero nell'ambiente

Caratteristiche degli agenti Consumer:

- Posizione: ogni agente ha una posizione nell'ambiente

- Raggio di visione: distanza entro cui l'agente può percepire le risorse di zucchero
- Tasso di metabolismo: quantità di zucchero consumata a ogni iterazione
- Quantità di zucchero: zucchero posseduto dall'agente
- Capacità massima di zucchero: quantità massima di zucchero che l'agente può immagazzinare

Caratteristiche degli agenti Sugar:

- Posizione: ogni agente ha una posizione fissa nell'ambiente
- Quantità di zucchero: zucchero disponibile nella cella
- Capacità massima di zucchero: quantità massima di zucchero che può essere presente nella cella
- Tasso di crescita: velocità con cui lo zucchero si rigenera

Dinamica del modello: a ogni iterazione:

1. Gli agenti Consumer consumano zucchero in base al loro tasso di metabolismo
2. Se un Consumer non ha abbastanza zucchero per il suo metabolismo, viene rimosso dal modello
3. I Consumer cercano la risorsa di zucchero più conveniente nel loro raggio di visione, considerando la quantità di zucchero e la competizione con altri Consumer
4. I Consumer si muovono verso la risorsa scelta o la consumano se adiacente
5. Gli agenti Sugar rigenerano il loro zucchero in base al tasso di crescita, fino alla capacità massima

Il comportamento degli agenti è definito tramite strategie implementate con lambda espressioni che modificano lo stato del modello a ogni iterazione.

Condizioni di terminazione: Il modello termina quando tutti i Consumer sono stati rimossi.

Il modello Sugarscape fornisce una simulazione interessante delle dinamiche di consumo e rigenerazione delle risorse in un ambiente chiuso, mostrando come comportamenti individuali semplici possano portare a dinamiche complesse a livello di sistema.

1.2 Analisi e modello del dominio

ARES deve poter effettuare simulazioni e modelli più o meno complessi. Questi modelli sono costituiti da agenti ognuno del quale ha un comportamento. Anche i modelli stessi possono avere un comportamento o regole proprie. Un elemento fondamentale di tutto il sistema sono i parametri, essi vanno a definire, dinamicamente, il comportamento degli algoritmi di agenti e modelli, possono essere soglie minime o massime, tipi, temperature.... . Un problema di tutto ciò è la necessità di gestire e garantire la correttezza di una moltitudine di dati eterogenei. Un altra criticità è la complessità che deriva dall'avere una così puntuale parametrizzazione di ogni singolo elemento di calcolo. Il programma dovrà restituire, all'utilizzatore, a ogni chiamata un risultato rappresentate l'attuale stato della simulazione. Di seguito uno schema esemplificativo degli elementi costitutivi del problema

1.2.1 Requisiti non funzionali

- Il sistema deve essere in grado di gestire un elevato numero di simulazioni contemporanee senza compromettere le prestazioni.
- Il software deve essere robusto e resistente a malfunzionamenti, garantendo la continuità delle simulazioni anche in presenza di errori.
- Il motore di calcolo deve essere progettato per consentire l'aggiunta di nuove funzionalità e algoritmi di simulazione in modo semplice e modulare.
- Deve essere fornita una API ben documentata per consentire l'integrazione del motore di calcolo in progetti esterni.

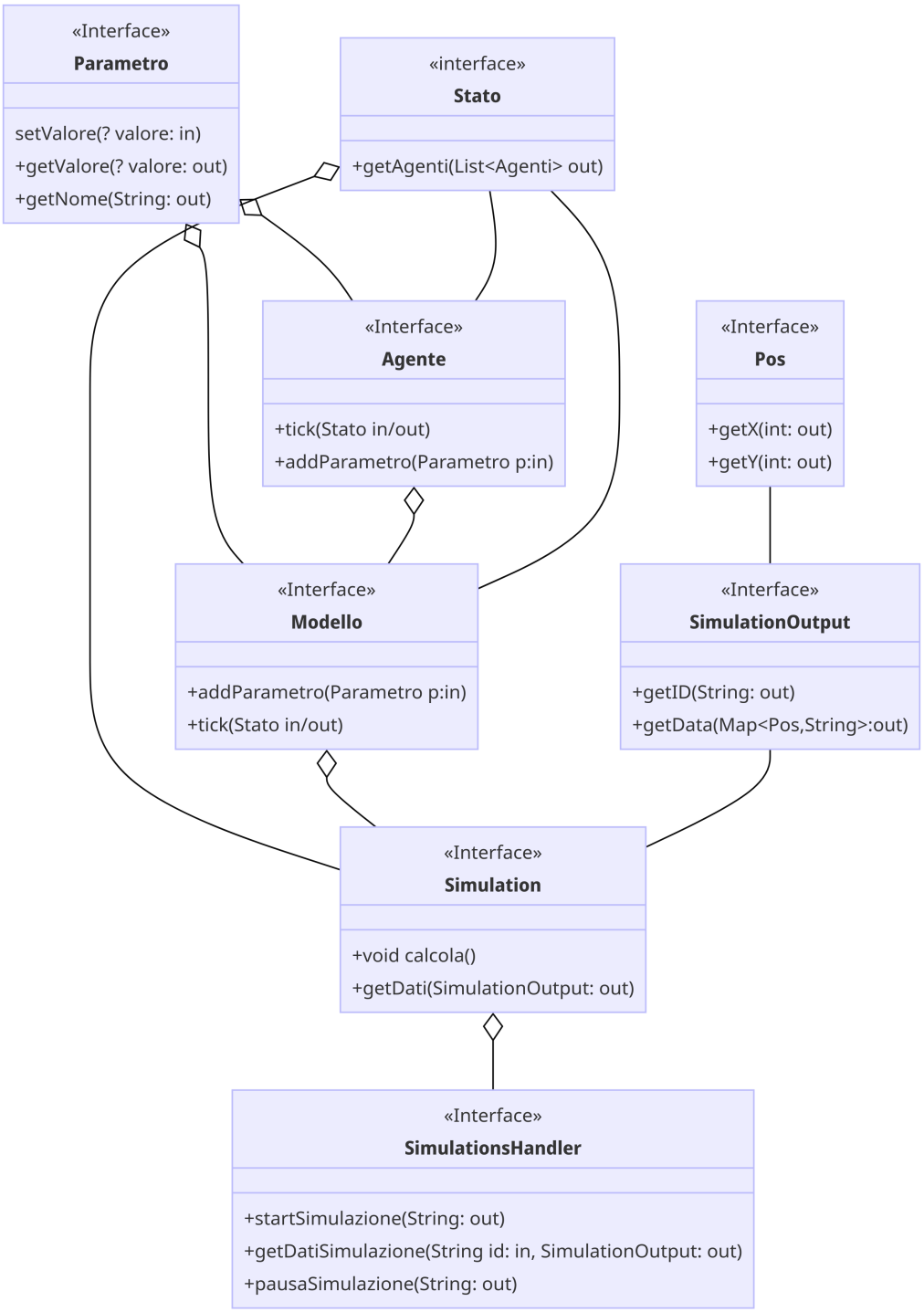

Figura 1.1: Schema UML dell'analisi del problema, con rappresentate le entità principali ed i rapporti fra loro

Capitolo 2

Design

2.1 Architettura

Ares è diviso in più parti, in particolare è presente una di calcolo e di gestione delle simulazioni (da qui in avanti core), il cui compito è l'esecuzione dei calcoli e la gestione dello stato delle simulazioni, l'architettura si potrebbe definire MC ovvero model controller in quanto il motore di calcolo non ha logica di visualizzazione (view) vera propria ma espone solo delle API per permettere l'utilizzo prendendo in input e restituendo dati. Il core utilizza le interfacce reattive di java per poter smistare e far ricevere alle applicazioni che lo utilizzino le informazioni necessarie, questo permette di avere, molteplici utilizzatori (anche applicazioni diverse da quella grafica e cli al momento sviluppata) a patto che implementino le interfacce necessarie e si interfacciano al core con le relative API. Affianco alla parte core sono presenti due applicativi che hanno lo scopo di permettere l'utilizzo, da parte di un utente della parte core.Seguono il pattern MVC, in particolare il model è rappresentato dalla parte core, mentre ogni applicativo (uno GUI e uno CLI Figura 2.2) ha la propria parte di view e di controller. In Figura 2.1 è esemplificato il diagramma UML architetturale. Del core

2.1.1 Architettura Core

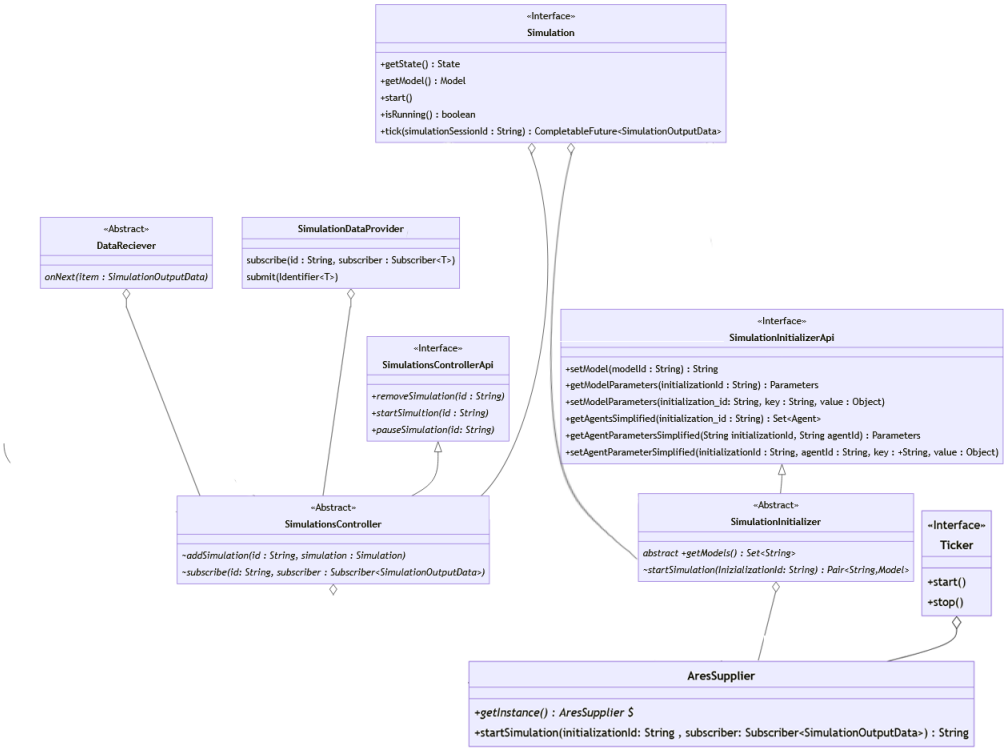

Figura 2.1: Schema UML della parte core di ARES. `CalculatorSupplier` fa da tramite per l'accesso alle funzionalità del core. Le classi astratte `SimulationsInitializer` e `SimulationController` permettono agli utilizzatori del servizio di creare una simulazione e di gestirla poi, attraverso queste due interfacce arriveranno tutti gli input verso il sistema. Per poter ricevere i dati della simulazione in corso, l'eventuale classe dovrà estendere la classe astratta `DataReciever` implementandone il relativo metodo

2.1.2 Architettura Cli

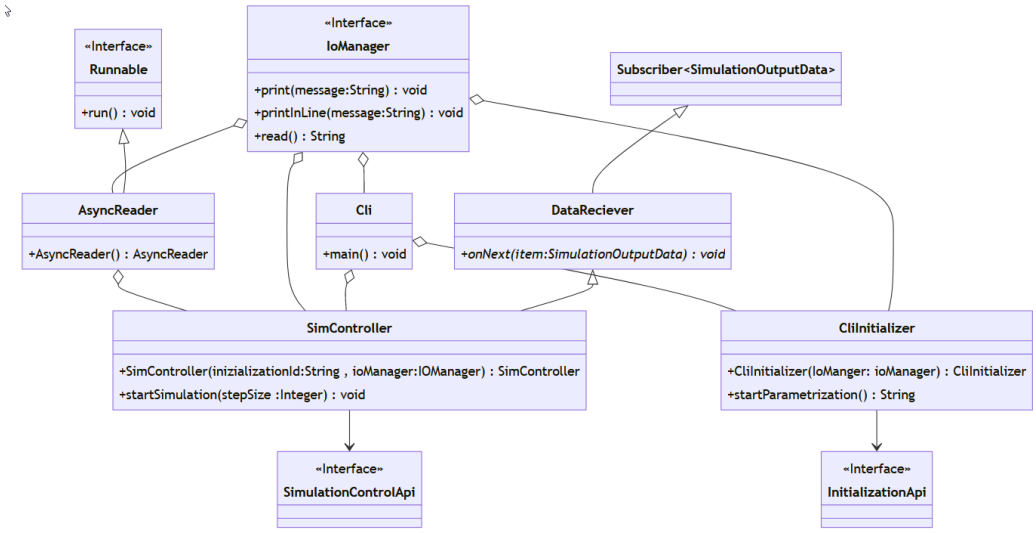

Figura 2.2: Schema UML dell'applicativo CLI, composto da due parti principali: CliInitializer e SimController che comunicano con Core tramite le interfacce SimulationInitializer e SimulationController e implementando la classe astratta DataReciever. CliInitializer e SimController sono i due elementi "Controller" del pattern MVC, il model è rappresentato dall'applicativo core con le sue interfacce esposte. La parte di View è astratta dall'interfaccia IOManager che fornisca api per leggere e scrivere da una interfaccia CLI

2.1.3 Architettura GUI

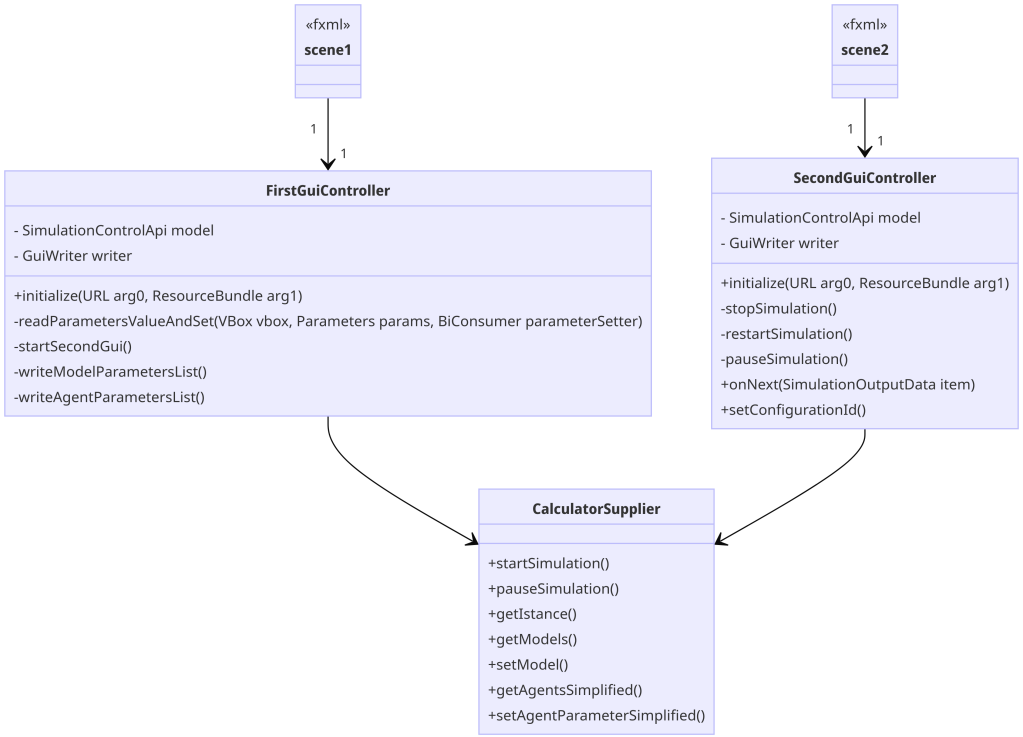

Figura 2.3: Architettura dell'applicativo GUI, integration con modulo core analogo a CLI

2.2 Design dettagliato

2.2.1 Naddei Leonardo

Creazione degli agenti

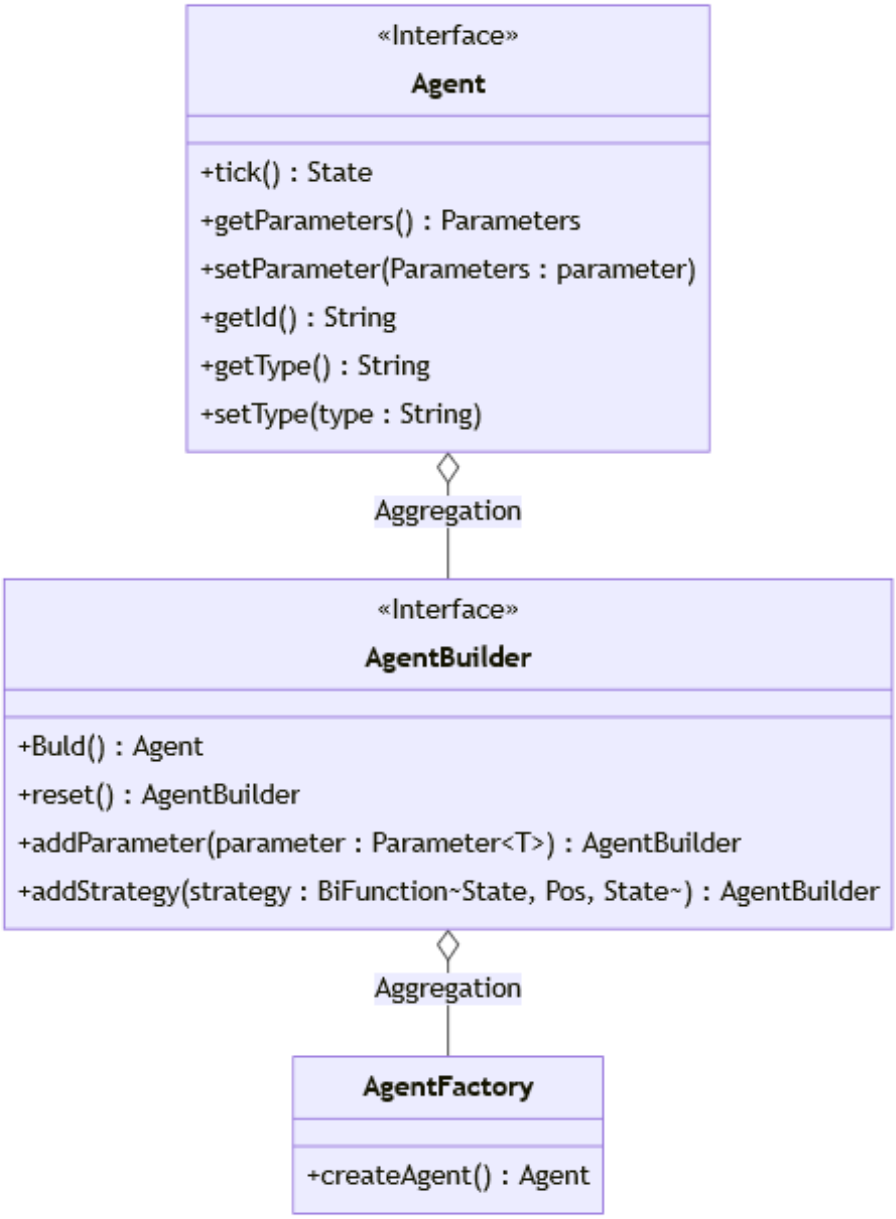

Figura 2.4: Rappresentazione UML dei pattern per la creazione degli agenti

Problema Ares ha vari possibili modelli, ognuno dei quali ha uno o più agent. E' necessario poterli creare in maniera rapida con il maggior riuso di codice possibile.

Soluzione Si sono utilizzati, insieme i pattern creazionali: *Builder*, *Factory* e *Strategy*, come da Figura 2.4. Nell'implementazione ogni agente è generato dalla relativa concrete factory, per creare l'Agente ed evitare il proliferare di classi anonime che implementano l'interfaccia *Agent* le factory utilizzano l'*AgentBuilder* che permette di creare agenti in maniera concisa. Infine per parametrizzare il comportamento dell'agente in base alla tipologia si è fatto uso dello strategy pattern, in particolare la strategy è rappresentata da una bifunzione che prende in input lo stato corrente e la posizione dell'agente nello stato e restituisce il nuovo stato aggiornato.

Gestione dei parametri

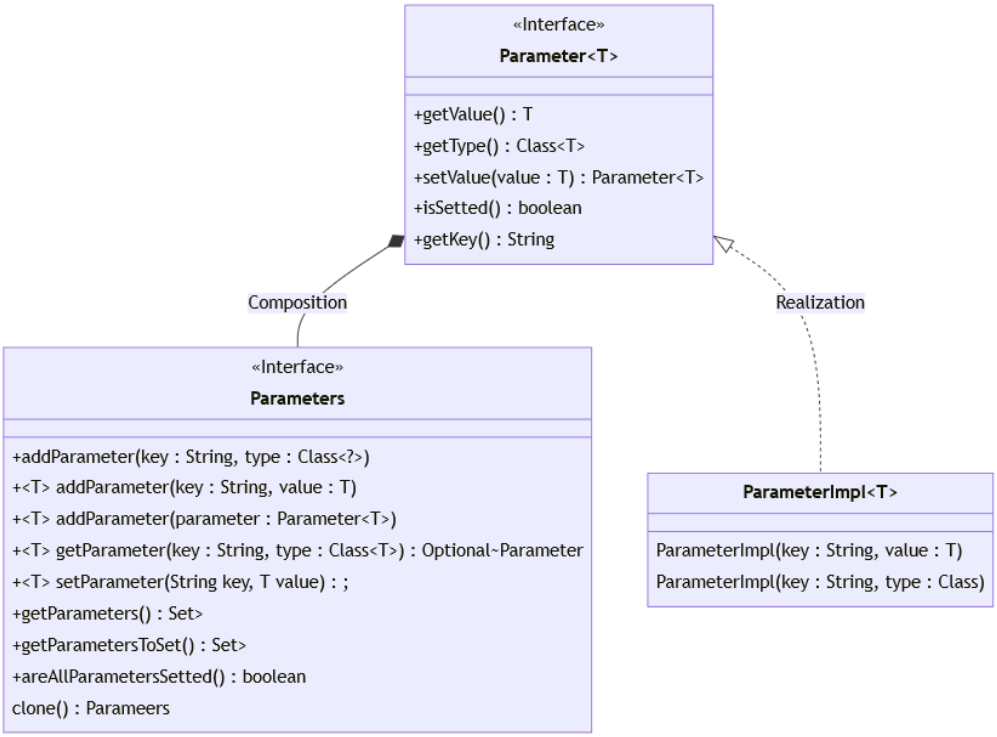

Figura 2.5: Rappresentazione UML delle classi per la gestione dei parametri

Problema Sia i modelli che gli agenti hanno bisogno di svariati parametri per funzionare, questi modelli sono difficilmente attribuibili a un unico set

di parametri comuni, sono anzi molto specifici e di svariati tipi. E' inoltre necessario sapere quali sono i parametri di un agente/modello e il poterli settare in maniera dinamica senza una conoscenza pregressa di quali sono (si pensi ad esempio a un interfaccia grafica generica che debba permettere la parametrizzazione dei modelli presenti e futuri)

Soluzione E' stato necessario un gestore di parametri, esso è composto di più classi, in particola un interfaccia generica *Parameter* che rappresenta il singolo parametro. Questi parametri sono poi conservati dentro a *Parameters*. Lo scopo di parameters è di avere una collezione generica di tipi eterogenei il cui accesso sia typesafe. Gli oggetti *Parameter* sono immutabili, è quindi delegato *Parameters* il compito di gestire la valorizzazione, questo allo scopo di facilitare la condivisione di questi parametri tra diversi oggetti del calcolatore senza rischio che qualcuno ne alteri lo stato provocando side effects indesiderati

Possibili soluzioni alternative Sono state identificate due possibili soluzioni alternative, la prima era qualla più banale ovvero dichiarare esplicitamente come campi assieme ai realtivi getter e setter i parametri, questo avrebbe più difficile astrarre gli agenti e avrebbe portato le applicazioni utilizzatrici di ARES-core a dovre implementare per ogni modello/agente tutti i metodi per gestire le chiamate ai getter e setter. Un altra possibile via sarebbe potuta essere quella di utilizzare la reflection per accedere dinamicamente ai parametri, crearli all'interno delle factory ecc. L'idea è stata scartata per vari motivi tra cui la lentezza, la poca robustezza, i problemi con la compilazione nativa...

Gestione del dominio di un parametro

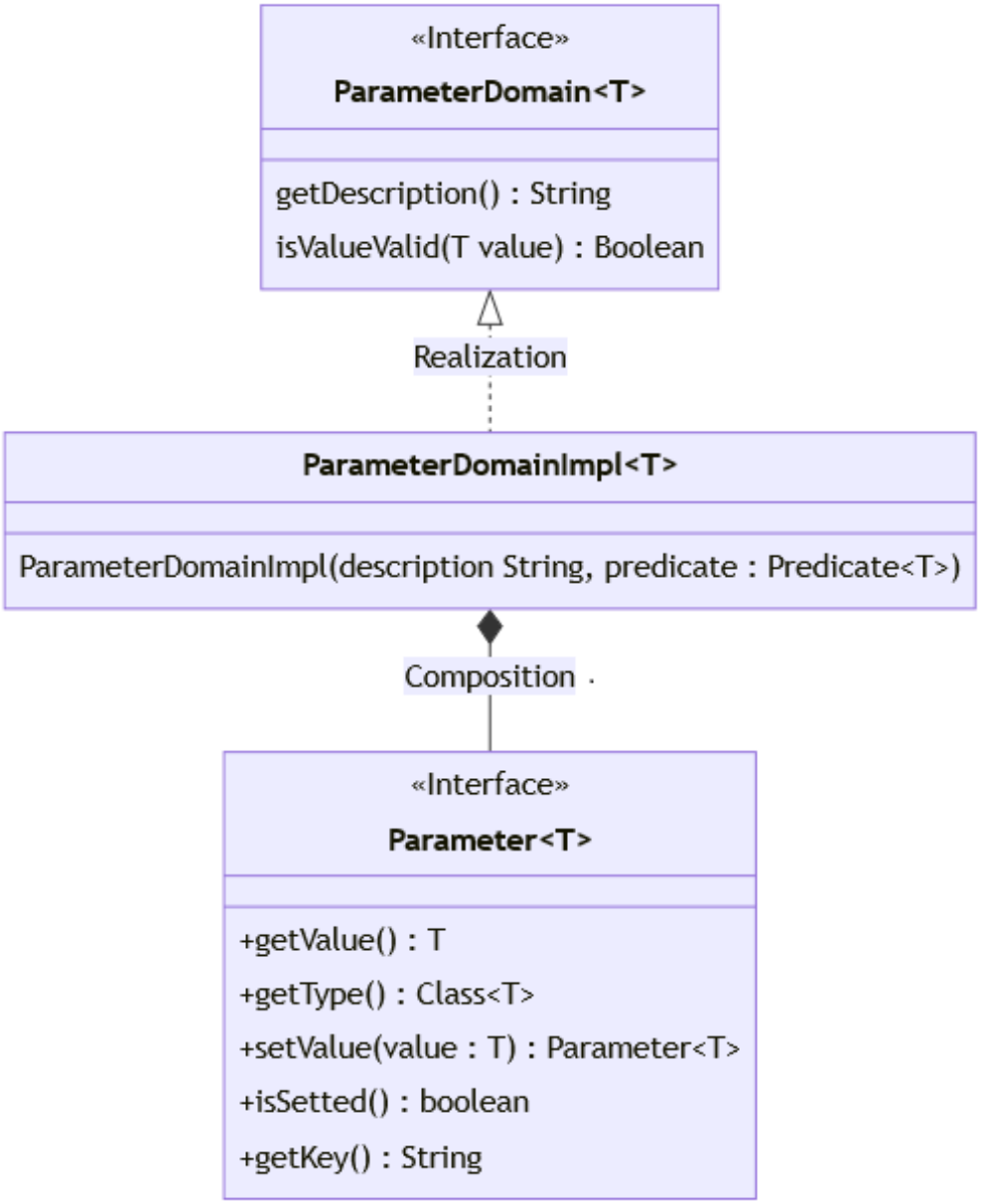

Figura 2.6: Rappresentazione UML delle classi per la gestione del dominio dei parametri

Problema A un utilizzatore esterno (una gui, un altro applicativo...) del sistema Ares core è necessario sapere oltre il tipo anche l'eventuale dominio

di un determinato parametro

Soluzione E' stata sviluppata l'interfaccia generica *Domain* che permetta di definire l'appartenenza o no di un valore all'interno del dominio e che fornisca descrizione in linguaggio naturale del dominio stesso.

Conversione della coppia stringa e tipo in un oggetto del tipo richiesto

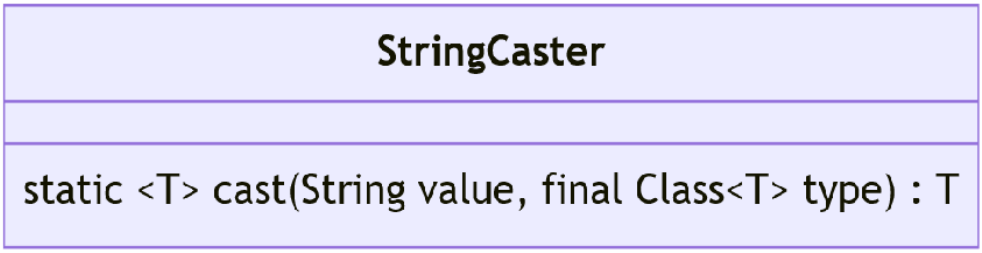

Figura 2.7: Rappresentazione UML della classe per il casting a oggetto

Problema All'interno della cli e della gui è necessario trasformare un input (di tipo stringa) in un oggetto tipizzato ciò che è richiesto dal parametro associato

Soluzione E' stata sviluppata una classe statica di utility che permettesse ti trasformare un valore di tipo stringa nell'oggetto richiesto

Gestione delle direzioni

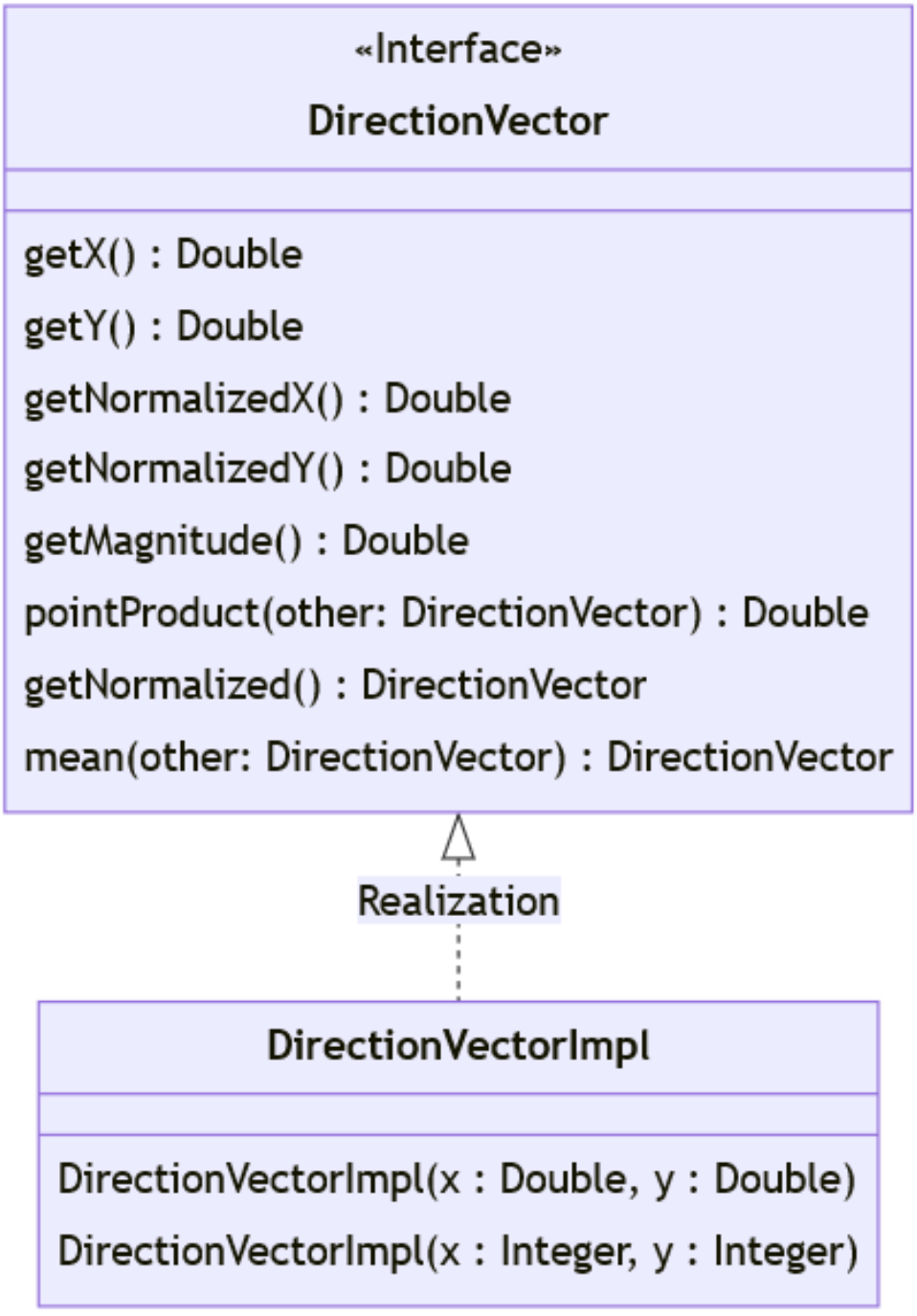

Figura 2.8: Rappresentazione UML delle classi per la gestione della direzione

Problema In molti modelli è necessario tracciare una direzione di movimento degli agenti

Soluzione E' stata sviluppata una classe che permettesse ti trasformare un valore di tipo stringa nell'oggetto richiesto

Astrarre la gestione della posizione di entità generiche

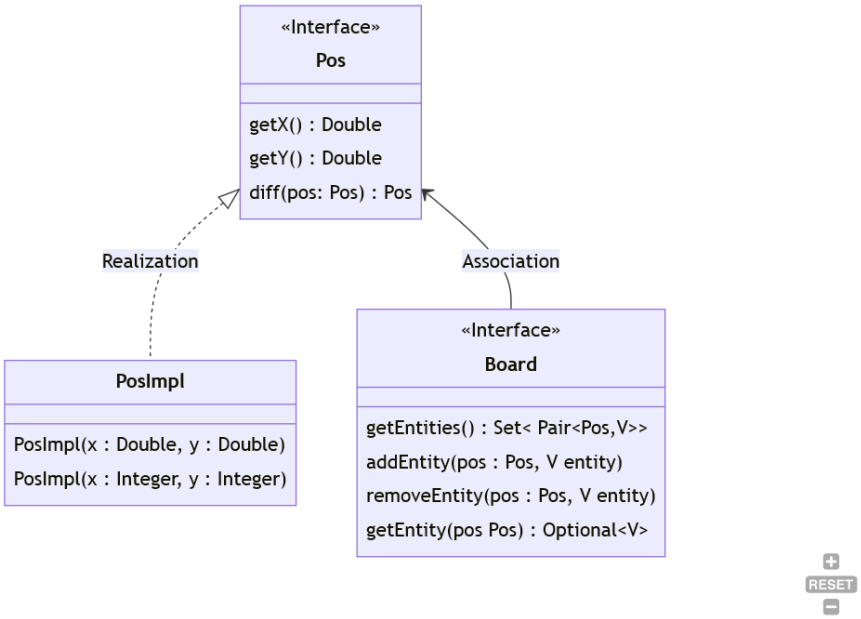

Figura 2.9: Rappresentazione UML delle classi per astrarre la gestione della posizione di entità generiche

Problema E' necessario poter tenere traccia della posizione di generici oggetti

Soluzione E' stata sviluppata una classe generice che si occupa di tracciare le posizioni di generici oggetti e di fornire un accesso diretto agli stessi, allo scopo è inoltre stata creata l'interfaccia posizione che astre una posizione (2D)

Astrarre lo stato di una simulazione

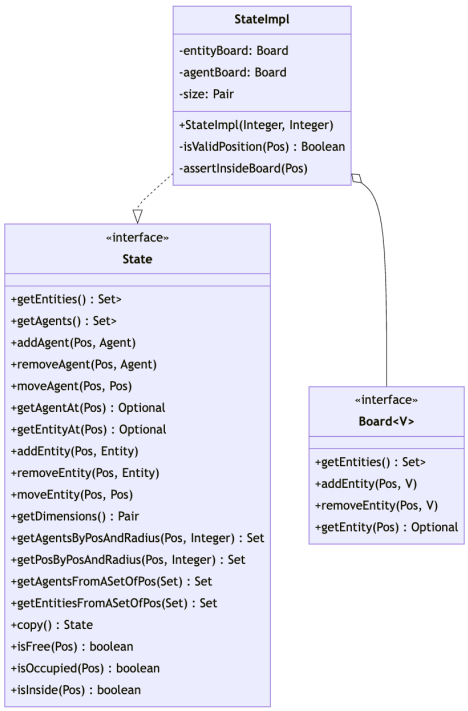

Figura 2.10: Rappresentazione UML delle classi per la gestione dello stato di una simulazione

Problema Ogni simulazione ha uno stato ovvero un insieme di dati che la definiscono, è necessario poter incapsulare tutto ciò in un unico oggetto mantenendo inoltre sempre validi certi vincoli (esempio dimensione NxN prefissata)

Soluzione E' stata sviluppata una classe Stato che utilizza la classe board precedentemente creata, allo scopo di gestire posizione di agenti e entità ma anche di gestire la dimensione della simulazione

Gestione del flusso di dati delle simulazioni

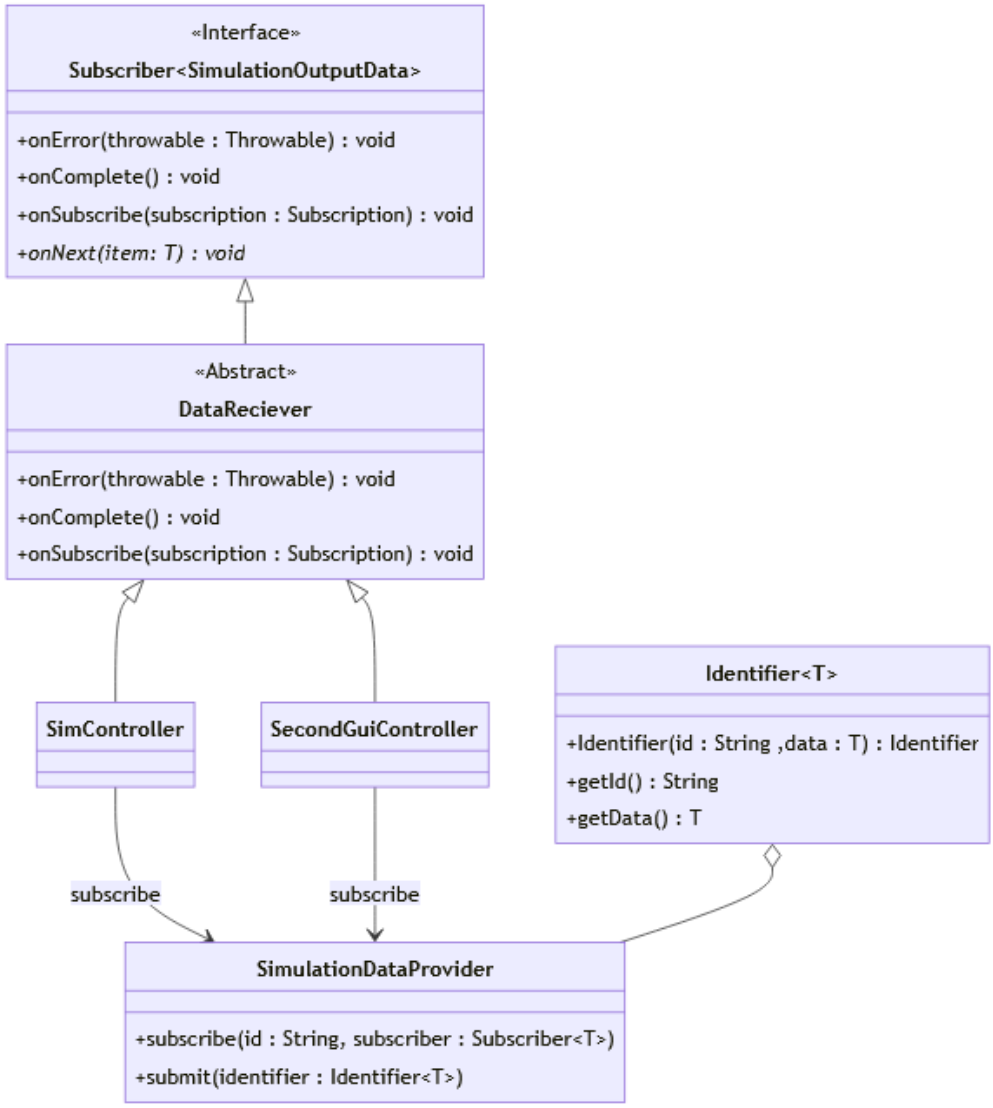

Figura 2.11: Rappresentazione UML delle classi flusso di dati delle simulazioni

Problema Il sistema deve permettere di mostrare i dati della simulazione creata, in particolare è richiesta un'interfaccia grafica che mostri una rappresentazione grafica del sistema ogni volta che la simulazione viene aggiornata (trasmissione di output).

Soluzione Attraverso le interfacce flow di java viene implementato un sistema per gestire l'invio dei corretti dati. In particolare ogni ricevitore di dati implementa la generica interfaccia subscriber (ricevendo oggetti SimlationOutputData) attraverso la classe atratta DataReciever che implementa metodi comuni o non usati. Il ricevitore quindi è un observable, osservato dalla classe SimulationDataProvider che riceve (tramite il metodo submit) i dati della simulazione appena calcolati e li reinvia ai corretti destinatari.

2.2.2 Domeniconi Lorenzo

Implementazione del pattern mvc all'interno della gui

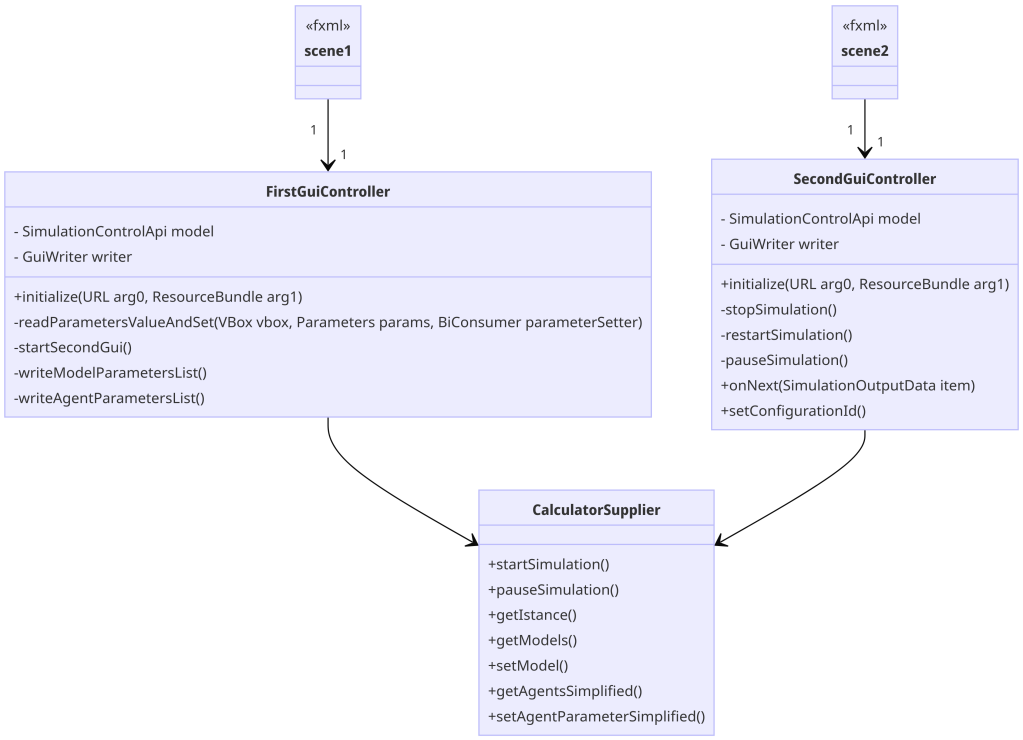

Figura 2.12: Rappresentazione UML delle classi per l'implementazione del pattern mvc all'interno della gui

Problema E' necessario gestire gli eventi generati dall'interfaccia utente e associare a ognuno di essi azioni specifiche.

Soluzione Il sistema per la gestione degli eventi delle interfacce è realizzato attraverso il pattern MVC, andando a creare un pattern MVC annidato

all'interno del pattern MVC generale dell'applicazione. Il pattern MVC annidato avrà la sua parte di view (file fxml), il suo controller associato e il model sarà composto dal package core (controller del pattern MVC generale). Disponendo di due interfacce grafiche, sono stati implementati due pattern MVC annidati, ogni view dispone del suo controller associato, condividendo lo stesso model. Avremmo potuto incorporare i due controller in uno solo in modo da gestire tutte le varie view della GUI attraverso un unico controller ma andando a utilizzare il pattern MVC per ogni singola view creata abbiamo un impatto positivo sulla chiarezza dell'applicazione, e sul riuso in quanto potremmo andare a prendere soltanto la view (associata al suo controller) che permette l'inserimento dei parametri e utilizzarla in un altro applicativo per esempio. Inoltre l'associazione di un proprio controller a ogni view permette di andare a modificare una parte del sistema (visualizzazione simulazione o scelta del modello e creazione) senza andare a modificare necessariamente altre parti.

Utilizzo di pattern Adapter per gestione degli eventi della gui

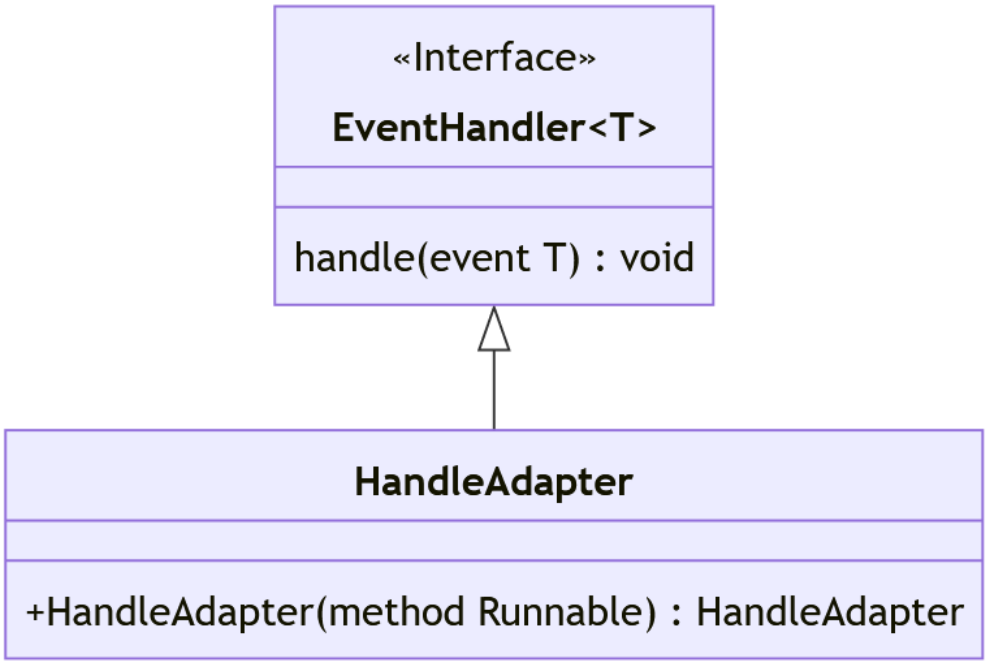

Figura 2.13: Rappresentazione UML delle classi per gestione degli eventi della gui

Problema Disponiamo dei metodi per gestire la simulazione ma per poterli usare effettivamente dobbiamo adattarli a EventHandler in modo da associarli ai click dei bottoni o alle azioni eseguite sulla GUI

Soluzione Utilizziamo il design pattern Adapter per permettere ai metodi di gestione della simulazione di adattarsi all'interfaccia EventHandler richiesta, permettendo ai nostri metodi di lavorare con la libreria JavaFX. Questo permette di semplificare l'integrazione di nuove funzionalità e il riutilizzo di codice esistente in altri contesti, andando creando un adapter per le esigenze dell'applicazione in cui vogliamo importare il nostro codice senza riadattarlo direttamente.

2.2.3 Di Varano Lorenzo

Creazione dei modelli

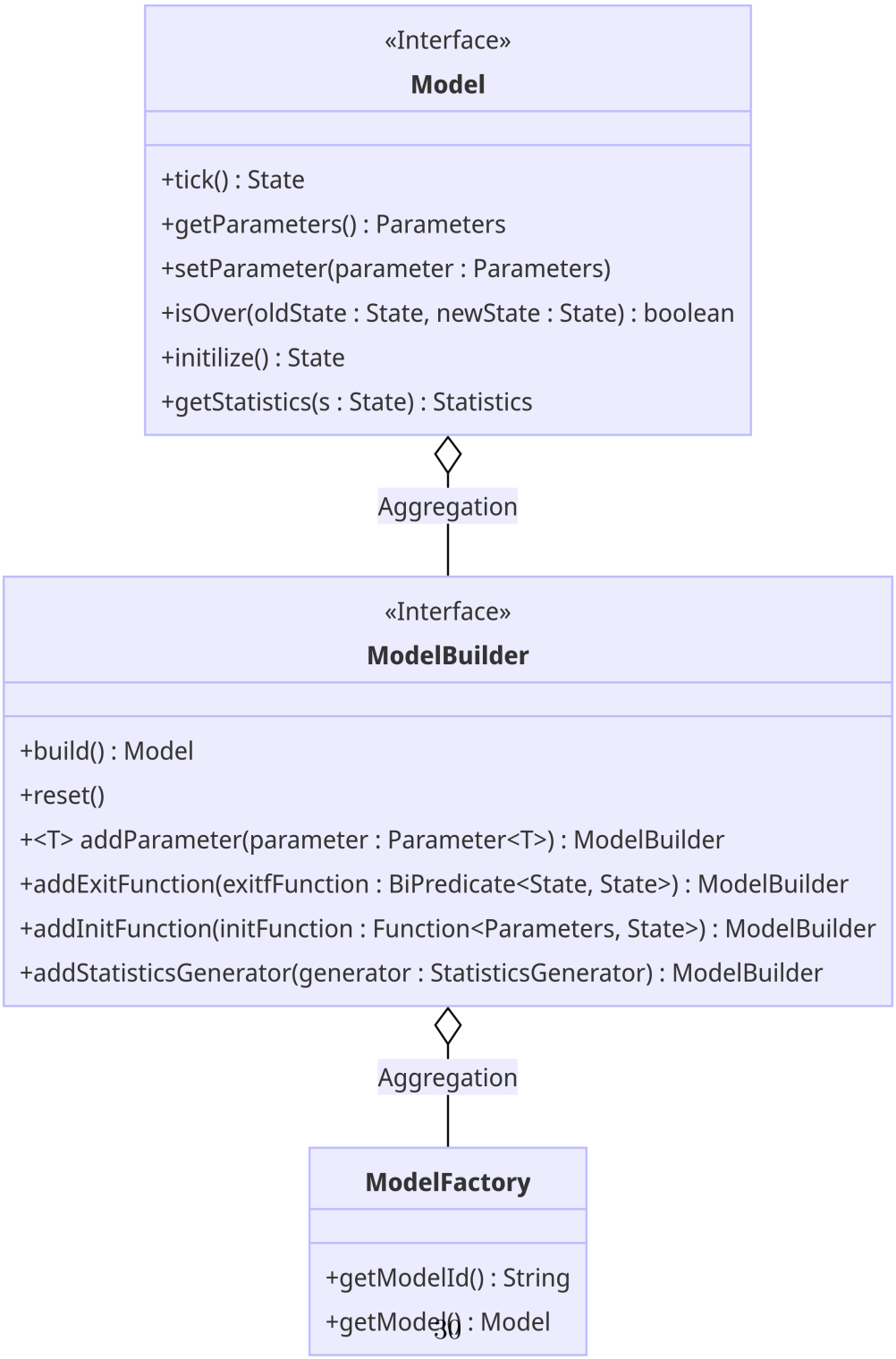

Figura 2.14: Rappresentazione UML delle classi per la creazione dei modelli

Problema Abbiamo la necessità di creare diversi modelli, con diversi parametri iniziali, e di farlo nella maniera più rapida.

Soluzione Sono stati utilizzati i tre pattern creazionali *Builder*, *Factory* e *Strategy*, come da figura. Ogni tipo di modello è generato dalla relativa factory, utilizzando il ModelBuilder per crearli in modo conciso. Oltre ai parametri di inizio della simulazione, sono richieste anche la funzione per istanziare lo stato iniziale del modello, definita all'interno dei singoli modelli, e la condizione di termine di quest'ultimo, un bipredicato che verifica a ogni tick se si è arrivati al termine della simulazione per quel modello specifico.

2.2.4 Abou El Kheir Uhalid

Implementazione del servizio di configurazione

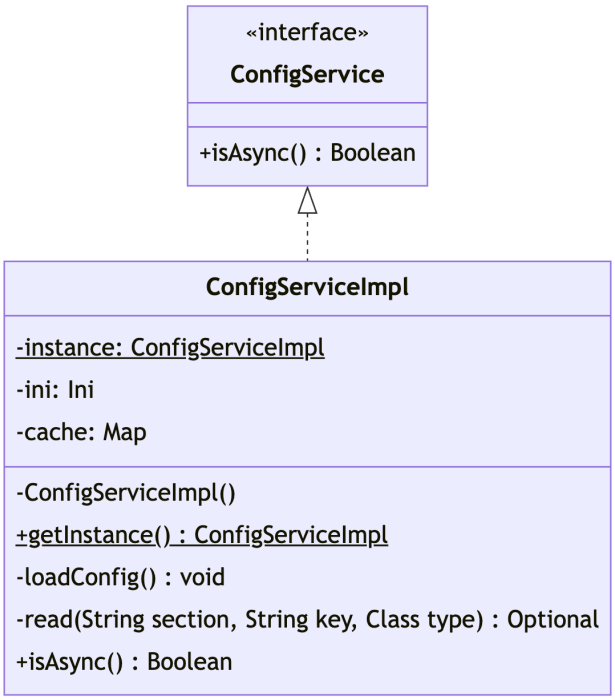

Figura 2.15: Rappresentazione UML delle classi per il servizio di configurazione

Problema È necessario fornire un meccanismo per leggere e accedere alle configurazioni dell'applicazione in modo efficiente e thread-safe, evitando di leggere ripetutamente dal file di configurazione per ogni richiesta.

Soluzione È stata implementata una classe ConfigServiceImpl che implementa l'interfaccia ConfigService, utilizzando il pattern Singleton per garantire una singola istanza del servizio di configurazione. La classe legge le configurazioni da un file .ini e utilizza una cache per memorizzare i valori letti, migliorando le prestazioni.

Caratteristiche principali della soluzione:

- *Pattern Singleton*: Garantisce che esista una sola istanza di ConfigServiceImpl, accessibile attraverso il metodo getInstance().
- *Caricamento lazy*: Il file di configurazione viene caricato solo quando viene creata l'istanza del servizio.
- *Caching*: I valori di configurazione letti vengono memorizzati in una cache per evitare letture ripetute dal file .ini.

Possibili soluzioni alternative Un'alternativa potrebbe essere l'uso di un sistema di configurazione basato su XML o JSON, o l'utilizzo di librerie di configurazione più avanzate come Apache Commons Configuration. Tuttavia, la soluzione attuale offre un buon equilibrio tra semplicità e funzionalità per le esigenze del progetto.

La soluzione implementata permette di accedere facilmente alle configurazioni dell'applicazione in modo efficiente, con la possibilità di estendere facilmente il servizio per supportare nuove configurazioni in futuro.

Utilizzo del pattern singleton

Figura 2.16: Rappresentazione UML della classe AresSupplier

Problema Il motore di calcolo è stato progettato e sviluppato per supportare simulazioni multiple, qualunque utilizzatore di quest'ultimo deve quindi accedere alla stessa istanza del motore e non crearne una nuova ogni volta

Soluzione E' stato utilizzato il pattern singleton in questo modo l'istanza è comune e la logica di gestion di più simulazioni in maniera concorrente è gestita dal motore stesso. In particolare il metodo statico `getInstance` è quello che permette di accedere all'istanza del motore

Utilizzo del pattern Facade

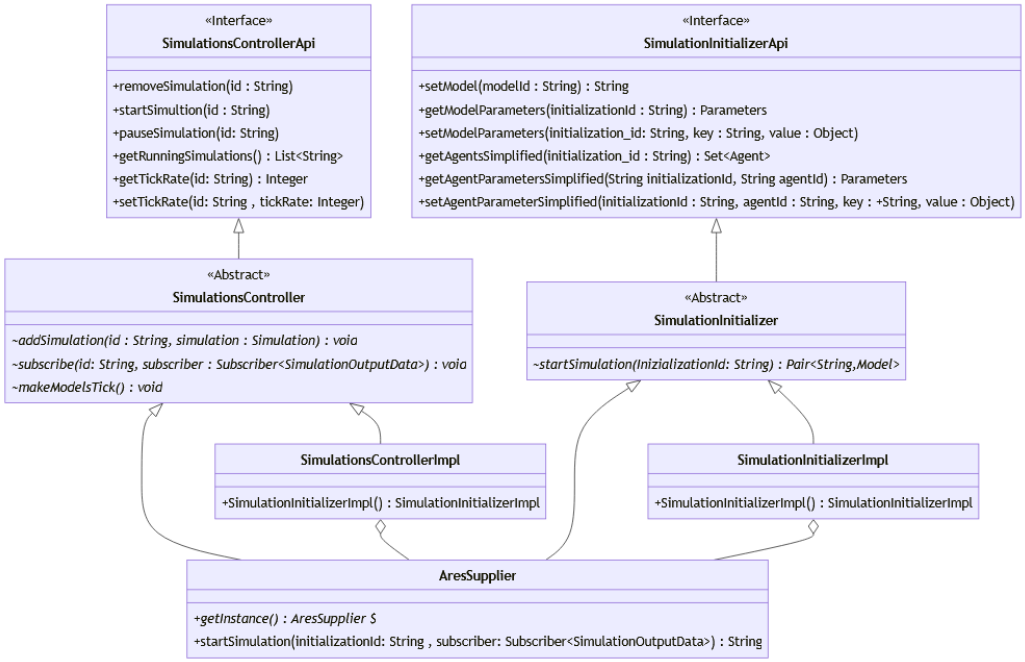

Figura 2.17: Rappresentazione UML delle classi per la gestione delle simulazioni

Problema Evitare un unica classe con tutta la logica dentro per interfacciare il motore con i programmi utilizzatori

Soluzione E' stato utilizzato il pattern facade, dividento la logica in due macroaree ovvero parametrizzazione e gestione della simulazione. Le due aree sono state poi ulteriormente scomposte in due sezioni ovvero parti che debbono essere pubbliche (Api del motore), astratte dalle interfacce e metodi interni al motore (package private) definiti nelle classi astratte. I le logiche dei metodi sono quindi nelle implementazioni concrete mentre la classe AresSupplier effettua solo le chiamate a quest'ultimi.

Crezione di scheduler

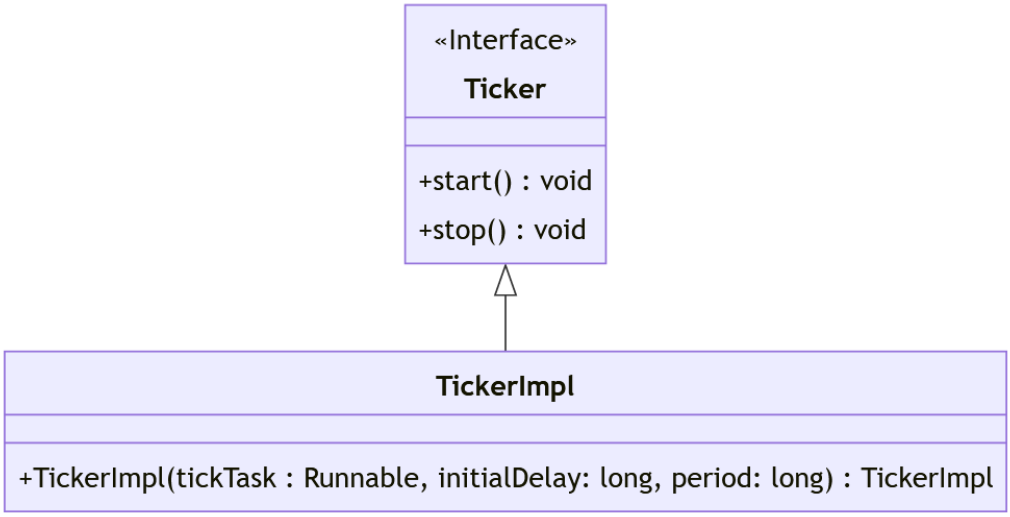

Figura 2.18: Rappresentazione UML delle classi per lo scheduler

Problema Sorge la necessità di eseguire compiti in modo ripetuto e programmato. La gestione di questi task programmabili, con la necessità di avviare e interrompere l'esecuzione in modo sicuro e controllato, rappresenta una sfida significativa.

Soluzione E' stata introdotta l'interfaccia Ticker con la sua implementazione TickerImpl. Utilizzando ScheduledExecutorService, TickerImpl consente di schedulare l'esecuzione di compiti con precisione, offrendo il metodo start() per l'avvio e il metodo stop() per l'interruzione controllata.

Gestione del salvataggio e caricamento delle simulazioni

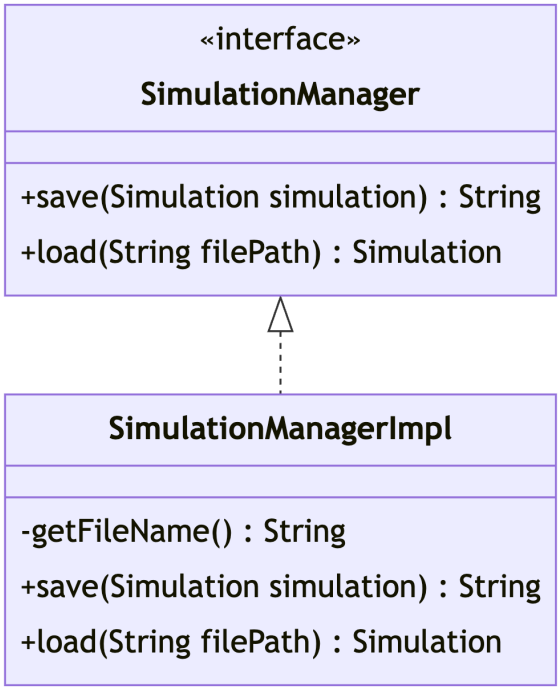

Figura 2.19: Rappresentazione UML delle classi per la gestione del salvataggio e caricamento delle simulazioni

Problema È necessario fornire un meccanismo per salvare e caricare le simulazioni, permettendo agli utenti di interrompere e riprendere le simulazioni in momenti diversi.

Soluzione È stata implementata un'interfaccia SimulationManager con una sua implementazione SimulationManagerImpl. Questa soluzione utilizza la serializzazione Java per salvare e caricare gli oggetti Simulation.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Vista l'elevata complessità intrinseca di una simulazione di tipo ABM, soprattutto quando sono presenti un elevato numero di agenti è stato necessario effettuare una grossa quantità di test a ogni livello implementando quindi test anche per singoli metodi (via reflection in caso di metodi privati) in modo da avere una base solida da cui partire. A fianco dei test sul singolo metodo e singolo algoritmo sono stati implementati test di integration ovvero che mirano a garantire la corretta cooperazione tra le varie componenti. Infine per i modelli, quando non banali, ovvero quando per numero di agenti la complessità, numero di iterazioni oppure numero di possibili risultati dalla singola iterazione risultasse complicato il poter testare direttamente il risultato rispetto ad un rislutato atteso si è scelto di utilizzare solo dei test che verificassero la correttezza del risultato ovvero. Data la serie di regole che determinano il comportamento di agenti e modelli, gli stati prodotti a ogni iterazione sono stati testati per verificare che fossero effettivamente corretti rispetto al modello in uso

3.2 Note di sviluppo

3.2.1 Naddei Leonardo

Scrittura di classi con utilizzo di generici

Creazione di sistema per gestire collezione di classi generiche rappresentanti parametri di configurazione di tipo eterogeneo e garantire il corretto accesso ai dati e l'inserimento di dati consistenti con il tipo stesso degli oggetti precedentementi creati Parameter.java

Utilizzo di progetti multi modulo

L'applicazione è stata utilizzata utilizzando il sistema multi modulo di gradle. L'applicazione è stata suddivisa in modo da isolare le dipendenze nei singoli pacchetti e permettere di utilizzare solo certi pezzi dell'applicativo (si pensi ad esempio ad un uso a libreria dell'engine di simulazione senza però voler utilizzare le interfacce gui e cli)

Utilizzo di `Stream`

Utilizzo di stream per migliorare la scrittura e leggibilita del codice. BoidsAgentFactory.java

Utilizzo `Functional interfaces`

Creazuibe di interfacce interfacce funzionali create appositamente per creare classi comportamento custom in maniera compatta e senza utilizzo di classi anonime (esplicito) o di ulteriori implementazioni. StatisticsGenerator.java

Utilizzo di `Optional`

Utilizzo di optional per astrarre la mancanza di dato invece di utilizzare `null` ParametersImpl.java

Utilizzo di `Java flow API`

Utilizzo di oggetti di Java Flow per creare un interfaccia reattiva per gestire la comunicazione dei dati in uscita a ogni iterazione delle simulazioni, verso i client che utilizzano il motore. SimulationDataProvider.java

Utilizzo di `reflection` per testing

Vista la lentezza della reflection ed i limiti che pone (es. utilizzo compilazione nativa) il suo uso è stato relegato al testing, in particolare è utilizzate per poter testare in maniera atomica a livello di singola funzione il codice dei modelli TestBoids.java

3.2.2 Abou El Kheir Uhalid

Utilizzo di `Stream`

Ho fatto un approfondito uso delle stream, quasi ovunque ne avessi la possibilita', di seguito un paio di esempi: ConsumerAgentFactory.java PredatorAgentFactory.java

Utilizzo di `reflection` per testing

Per il testing del configService ho usato reflection per accedere ai metodi private ConfigServiceImplTest.java

Utilizzo di generici

ConfigServiceImpl.java

Utilizzo di `Optional`

ConfigServiceImpl.java

Utilizzo di `Serializable`

Per consentire la serializzazione delle simulazioni, è stato impiegato il meccanismo di serializzazione fornito nativamente da Java attraverso l'interfaccia `Serializable`. L'implementazione di questa funzionalità ha richiesto alcuni accorgimenti specifici per gestire correttamente la serializzazione di strutture dati complese. SerializableFunction.java

3.2.3 Di Varano Lorenzo

Utilizzo di Stream

FireAgentFactory.java

Utilizzo di Interfaccie funzionali

FireAgentFactory.java

3.2.4 Domeniconi Lorenzo

Utilizzo di Stream

Utilizzati diverse volte. Permalink: FirstGuiController.java

Esecuzione di Modifiche all'Interfaccia Utente da un thread diverso dal JavaFX Application Thread

Permalink: SecondGuiController.java

Utilizzo della reflection per accedere a metodi privati in fase di test

Permalink: TestVirus.java

Uso di lambda expression

Permalink: VirusAgentFactory.java

Definizione metodo generico

Permalink: GuiDinamicWriterImpl.java

Utilizzo della Libreria JavaFX

Permalink: GuiDinamicWriterImpl.java

Utilizzo degli Optional

Permalink: PVirusAgentFactory.java

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

Leonardo Naddei

Sono soddisfatto di quanto prodotto, riguardando quanto prodotto mi ritengo abbastanza soddisfatto, potrei portare avanti questo progetto, magari utilizzandolo per sperimentare e approfondire altri aspetti del linguaggio come utilizzo di CDI, una migliore gestione degli errori e dei logging e la possibile integrazione con servizi esterni per trasformarlo in un prodotto "completo" utilizzando anche tecnologie out of scope rispetto a questo corso come interfacciarsi con database per gestire configurazioni e trasformare quello che è un utilizzo a libreria in una modalità di fruizione più orientata ai serivizi (es. SaaS).

Domeniconi Lorenzo

Ritengo di aver raggiunto gli obiettivi prefissati per il mio lavoro, impiegando circa 72 ore per realizzare la mia parte di progetto. Il mio lavoro permette di comunicare in maniera efficace con il lavoro dei miei compagni e allo stesso tempo di mostrare la simulazione del modello scelto in maniera thread-safe senza causare eccezioni, aggiungendo un punto di forza al sistema. Il sistema è stato implementato in modo tale da coprire tutti gli eventuali tipi di errori che si possono presentare in fase di creazione della simulazione che si desidera avviare, permettendo di avviare una simulazione che non presenti errori a run-time. Questo è un ulteriore punto di forza del sistema. Un punto di debolezza è il non riuscire a gestire il corretto dimensionamento di tabelle di dimensioni elevate (sopra 1300 celle circa, 36x36), questo sarà sicuramente uno sviluppo futuro che sarà implementato nel sistema, coniugando anche

la visualizzazione di statistiche sulla simulazione corrente e la possibilità di modificare i parametri della simulazione a run-time. Per quanto riguarda la collaborazione, ho lavorato bene con i miei colleghi fornendo feedback quando necessario e ricevendo supporto per il corretto uso delle loro implementazioni all'interno del mio lavoro. Ritengo di aver svolto un ruolo abbastanza importante nel gruppo fornendo al sistema un'interfaccia che permetta la reale implementazione dell'intero sistema.

Abou El Kheir Uhalid

Sono concento del lavoro svolto, in quanto ho raggiunto gli obiettivi prefissati. Ho lavorato principalmente sulla gestione delle simulazioni, compressa la seriallizazione, e sulla gestione delle configurazioni.

Per quanto riguarda i lavori futuri, si potrebbe aggiungere un logger per gli errori, migliorando così la tracciabilità e la risoluzione dei problemi. Inoltre, si potrebbe implementare un meccanismo di caching più adeguato per le configurazioni, nel caso in cui aumentassero di numero e creassero qualche forma di bottleneck.

Di Varano Lorenzo

Sono soddisfatto del lavoro svolto per la mia parte, dato che le classi relative al modello permettono la facile implementazione delle diverse tipologie ed evitano di dover creare ulteriori classi per ognuna di esse; perciò penso di essere rientrato negli obiettivi che mi ero prefissato all'inizio, uno sviluppo futuro potrebbe essere implementare altre caratterstiche legate alla creazione e gestione dei modelli per rendere ancora più semplice e completa questa funzionalità, ma ritengo sia abbastanza solida per com'è ora.

4.2 Difficoltà incontrate e commenti per i docenti

Naddei leonardo

Mi sarebbe piaciuto vedere componenti del linguaggio più avanzate, evitando quindi tutta la parte introduttiva e passare subito a pattern/paradigmi più particolari e una visione più approfondita su quella che potrebbe essere la gestione di un progetto complesso, a partire dalla gestione delle dipendenze, applicativi multi package, CI e magari qualche aspetto riguardo la concorrenza. Per quanto riguarda il progetto, ho trovato difficile, lavorando oltre che studiando, riuscire a trovare abbastanza tempo da dedicare al progetto.

Domeniconi Lorenzo

Di Varano Lorenzo

Non ho incontrato particolari difficoltà nella parte da me svolta, l'utilizzo di pattern e stream mi ha preso un po' di tempo per poterlo comprendere appieno per poter partire, ma tutto sommato pensavo potesse andare molto peggio. Il corso, in particolare le slide relative ai laboratori, sono stati abbastanza utili per svolgere il progetto e comprendere i concetti base che servivano per partire.

Abou El Kheir Uhalid

La principale difficolta' che ho incontrato è stata gestire il tempo tra il lavoro e il progetto. Inoltre, alcune funzionalita' sono state implementate verso la fine del progetto, quando sarebbe stato meglio integrarli fin dal inizio. Ad esempio, la serializzazione delle simulazioni, questo ha richiesto tempo e sforzi aggiuntivi che sarebbero potuti essere evitati con una pianificazione piu' anticipata. Per i commenti avrei preferito affrontare da quasi subito le stream e i pattern.

Appendice A

Guida utente

Il software ares ha varie modalità di funzionamento, in particolare può essere utilizzato in modalità grafica (GUI), testuale (CLI) in modalità libreria oppure può essere compilato come utility a libreria (mantenendo solo la parte core) e utilizzato da applicazioni di terze parti. La seguente guida non tratterà l'ultimo caso, coperto dalla documentazione fornita da javadoc. La guida tratterà il caso in cui ci si trovi dinanzi a un applicazion frutto di una compilazione contenente tutti i moduli (in caso di compilazione solo con parte gui o cli non sarà presente la parte iniziale di scelta della modalità (Menu avvio), mostrata a breve)

A.1 Menu di avvio

Una volta avviata l'applicazione ci si troverà dinanzi un menù in cui è possibile scegliere la modalità di funzionamento, basterà inserire il numero della modalità scelta e premere invio. La modalità cli utilizzerà solo il terminale, quella grafica aprirà una gui esterna mentre in modalità 2 (ibrida) verranno utilizzate in contemporanea la gui e la cli (con due simulazioni diverse in contemporanea)

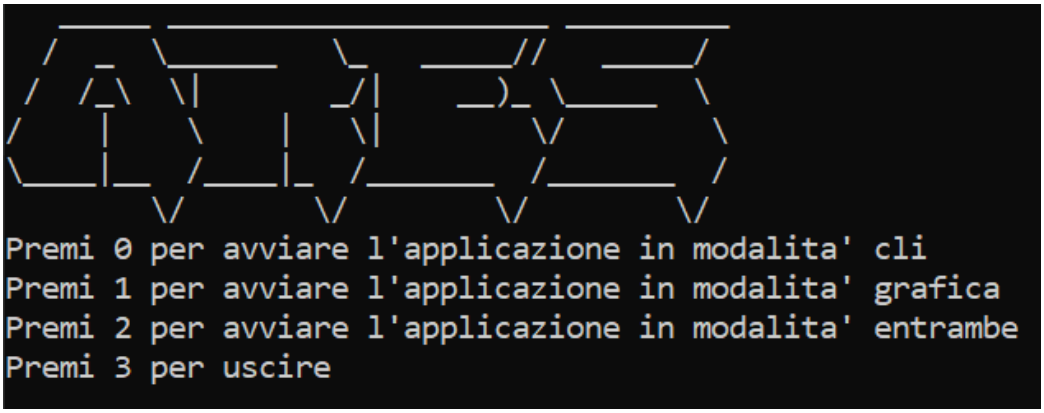

Figura A.1: Schermata di avvio del programma

A.2 Modalità grafica

A.2.1 Parametrizzazione

Una volta avviata la modalità grafica ci si troverà di fronte alla prima schermata in cui è possibile scegliere i vari modelli ed impostarne i parametri. In particolare bisognerà scegliere il modello che si vuole utilizzare attraverso l'apposito menù a tendina (1). Una volta scelto apparità una serie di caselle di testo (2) in cui inserire i valori dei campi parametrizzabili. N.B. è importante rispettare l'eventuale dominio dei campi, altimenti l'applicazione mostrerà un errore e non sarà possibile andare avanti fino al corretto inserimento. Una volta inseriti correttamente i parametri e cliccato il bottone (3) è possibile andare a impostare i parametri dei singoli agenti (per motivi di comodità di utilizzo la parametrizzazione del singolo agente è permessa solo con un utilizzo a libreria, si portà pertanto parametrizzare solo per famiglie di agenti). Attraverso il menu a tendina (5) selezionare man mano tutte le famiglie di agenti presenti ed impostarne i parametri in maniera analoga a quanto fatto per il modello prescelto, avendo cura di premere il bottone (6) per impostarli ogni qualvolta si abbia finito di impostarli (per ogni famiglia di agenti). Una volta completata la parametrizzazione è possibile premere il bottone (7) per iniziare la simulazione.

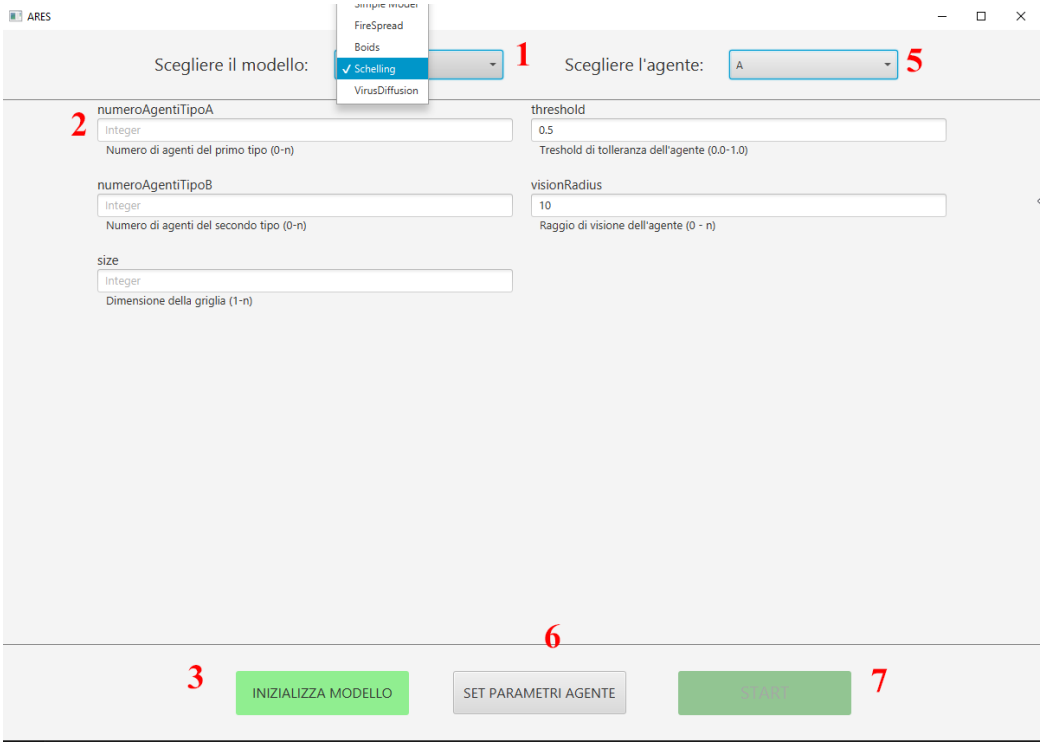

Figura A.2: Schermata con configurazione dei parametri

A.2.2 Simulazione

Durante la simulazione, tramite lo slider (1), è possibile impostare, il tempo tra un tick e l'altro del modello, portandolo in basso il tempo diminuirà (più iterazioni per secondo), portandolo verso l'alto aumenterà (meno iterazioni al secondo). E' inoltre possibile fermare (3), far riprendere (4) e uscire (2) dalla simulazione (in quest'ultimo caso si verrà riportati all'interfaccia di parametrizzazione per poter lanciare un altra simulazione). Nel caso un modello arrivi a termine (alcuni modelli possono finire mentre altri non prevedono condizioni di uscita) verrà mostrato un messaggio e la simulazione terminarà.

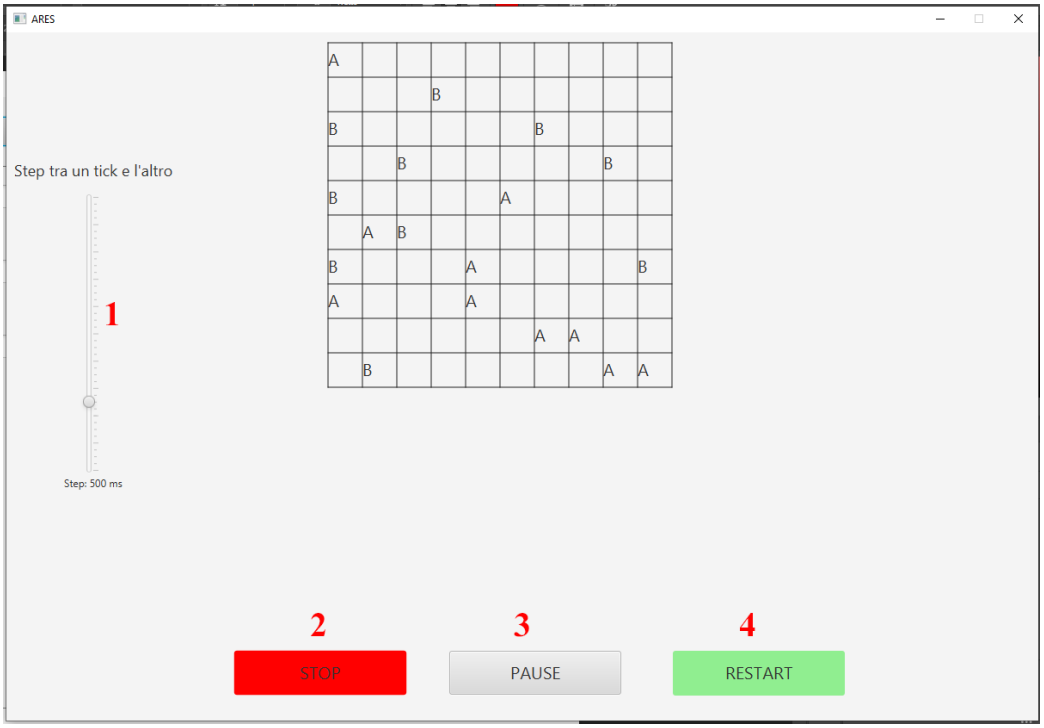

Figura A.3: Schermata durante simulazione

A.3 Modalita cli

A.3.1 Parametrizzazione

Una volta avviata la modalità cli l'applicativo chiederà se si vuole caricare una simulazione precedentemente salvata o crearne una nuova, nel secondo caso bisognera scegliere tra i vari modelli, basterà inserire il numero associato e premere invio. Una volta scelto l'applicativo inizierà a richiedere l'inserimento dei valori dei parametri per il modello stesso. N.B. è importante rispettare l'eventuale dominio dei campi, altimenti l'applicazione mostrerà un errore e non sarà possibile andare avanti fino al corretto inserimento. Una volta inseriti correttamente i parametri il programma, automaticamente, passerà all'inserimento dei parametri per le famiglie di agenti (per motivi di comodità di utilizzo la parametrizzazione del singolo agente è permessa solo con un utilizzo a libreria, si portà pertanto parametrizzare solo per famiglie di agenti). Sarà richiesto man mano di inserire i valori dei parametri delle famigli di agenti presenti. Una volta terminata la paratrizzazione verra' chiesto se si vogliono esportare i parametri appena inseriti, se si scegli di esportarli verrano' messi dentro una cartella ParametersData. In seguito

il programma, chiederà di impostare, il tempo tra un tick e l'altro (in ms) del modello, piu' basso il tempo sarà più iterazioni per secondo ci saranno, nel caso si inserisca 0 il programma andrà il più velocemente possibile. Una volta inserito anche questo parametro, basterà premere invio e la simulazione inizierà.

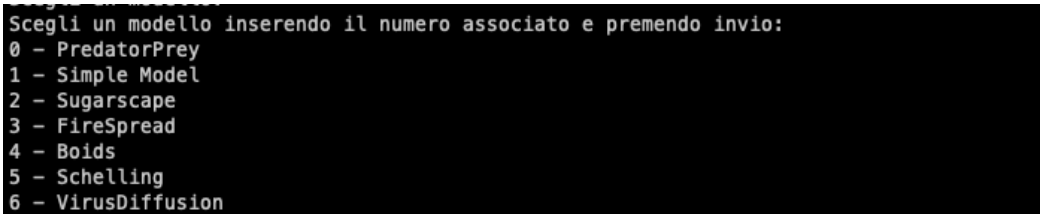

Figura A.4: Scelta modello

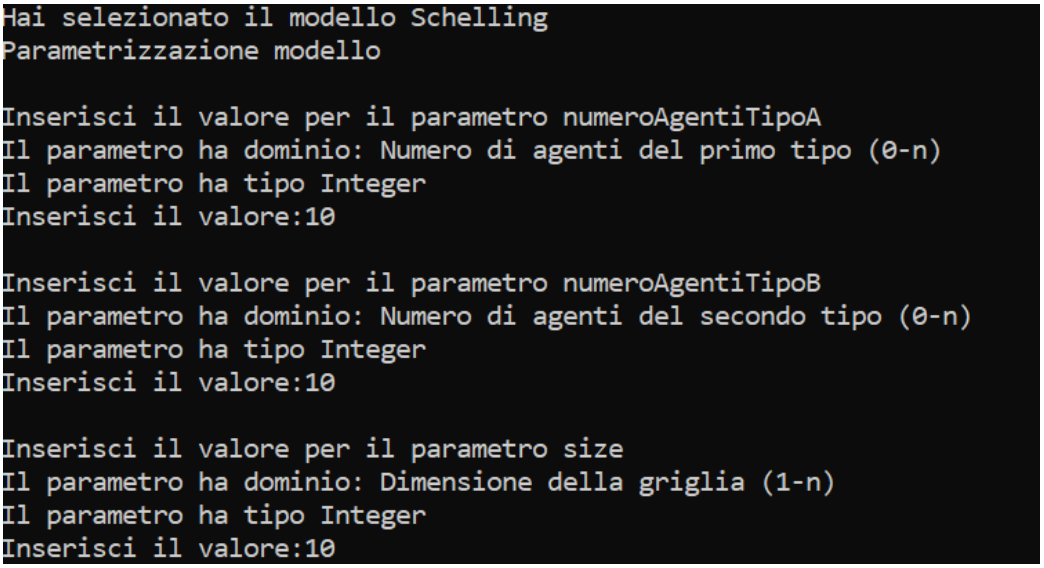

Figura A.5: Parametrizzazione modello

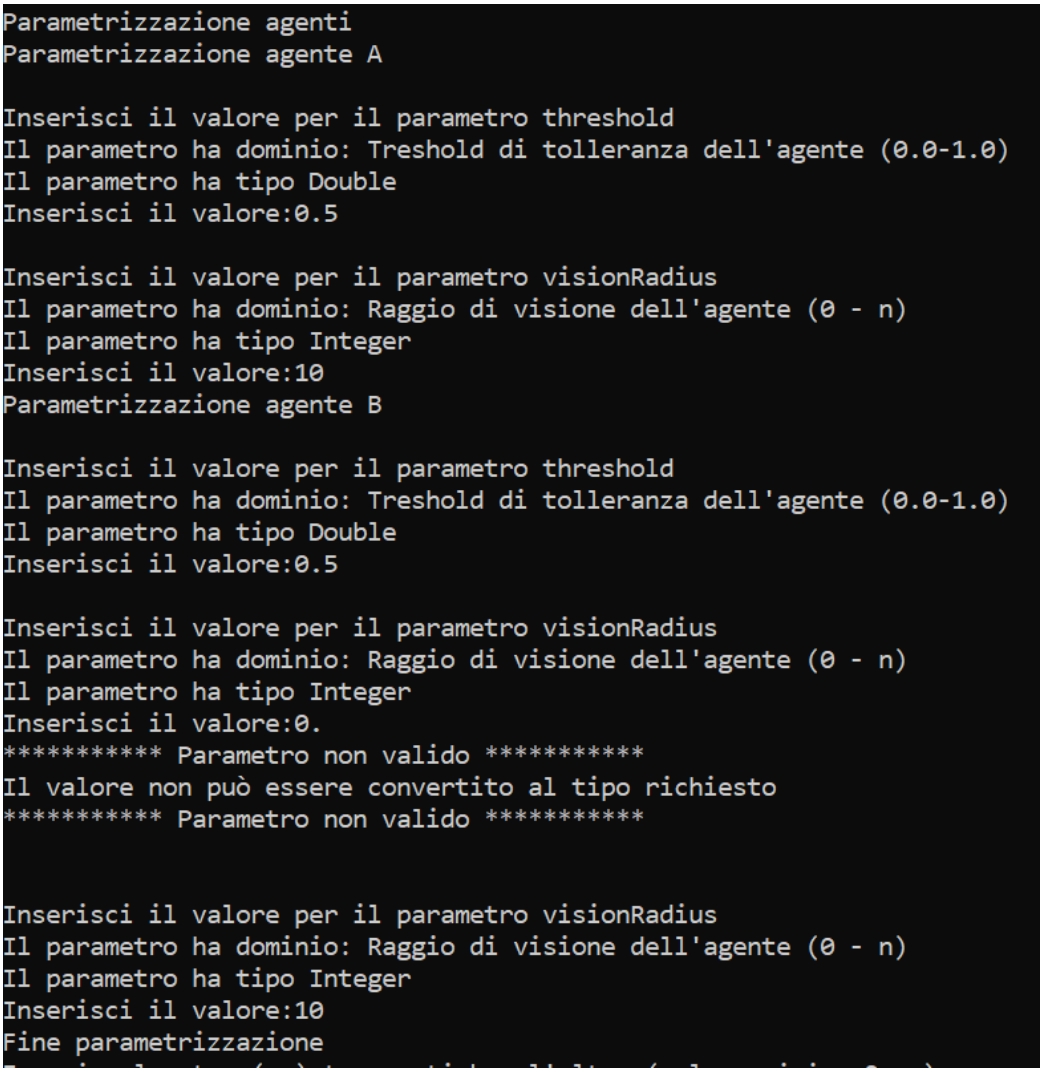

Figura A.6: Parametrizzazione agente

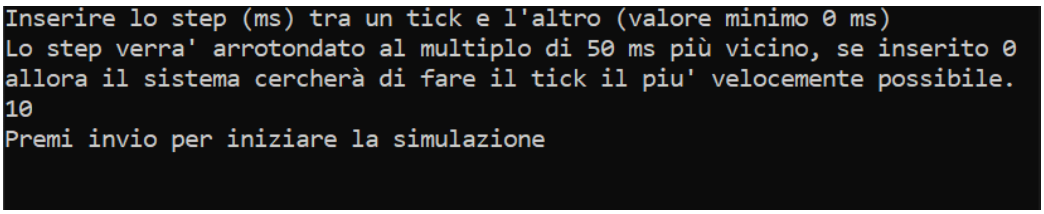

Figura A.7: Impostazione tickRate

A.3.2 Simulazione

Durante la fase di simulazione verrà stampato, a console, lo stato della simulazione e le eventuali statistiche. Sarà possibile mettere in pausa, far ricomiciare, uscire dalla simulazione o salvare la simulazione. per farlo premere rispettivamente p, s, e ed o (case sensetive). Una volta premuti i caratteri inseriti non saranno visibili e il comando sarà eseguito solo in seguito alla pressione del tasto invio. Nel caso di comando di uscita (e) la modalità cli terminerà, se si è in modalità solo CLI il progromma terminerà mentre se si è in modalita ibrida (e la gui è ancora aperta), andrà avanti fino alla chiusura di quest'ultima. Se si sceglie di salvare la simulazione, il programma mostrera il percorso di dove e' stata salvata la simulazione e di seguito il progromma terminera.

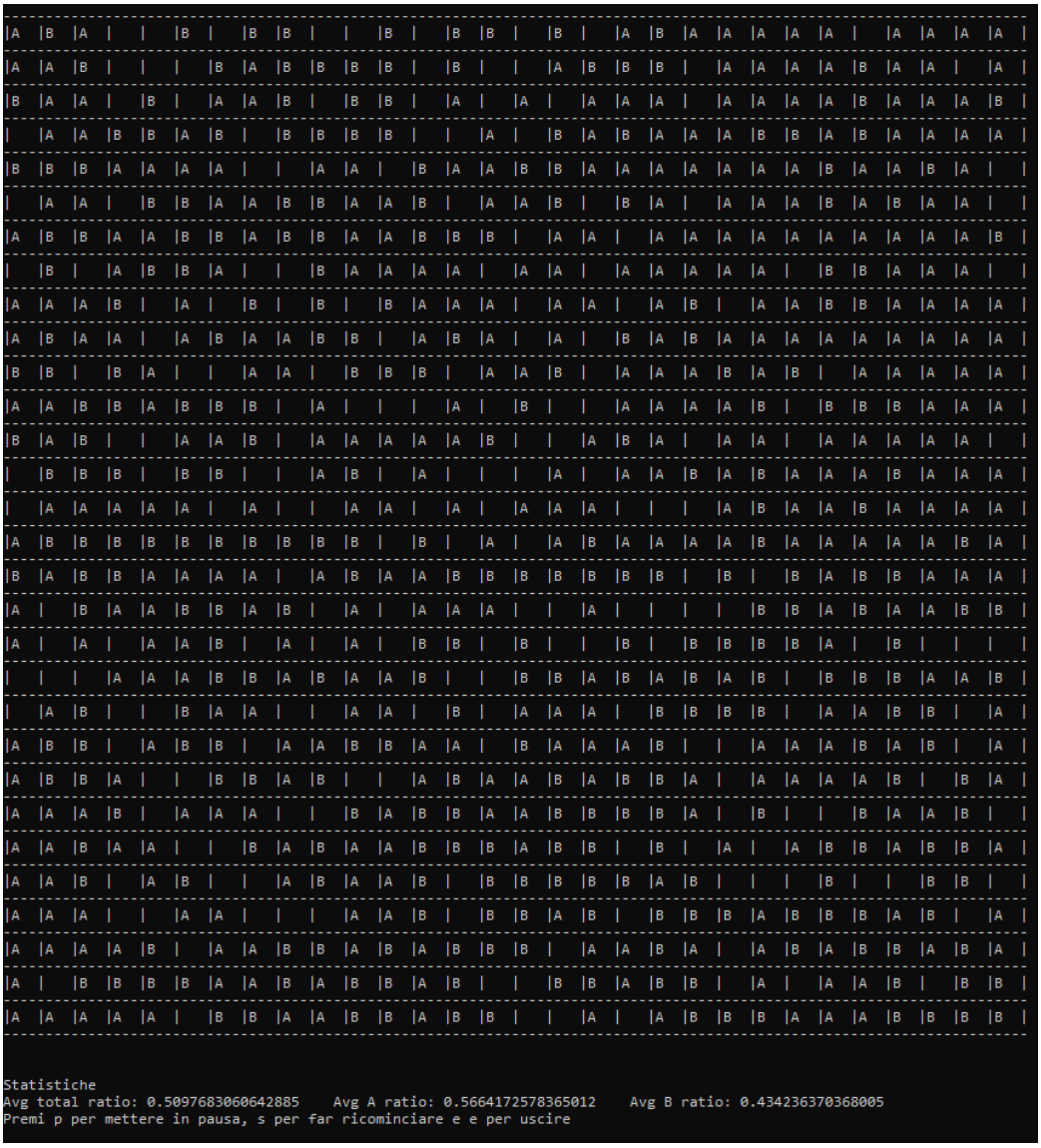

Figura A.8: Esempio di schermata di simulazione

A.3.3 Configurazioni di esempio

A seguito una serie di configurazioni per poter provare i vari modelli (è presente inoltre un modello Simple Model che permette di lanciare una simulazione senza nessuna parametrizzazione allo scopo di demo semplificata)

- Boids
  - Parametri Modello:

        * numeroUccelli : 200
        * size : 30
    - Parametri agente B:
        * alignmentWeight : 0.7
        * angle: 30
        * cohesionWeight : 0.1
        * collisionAvoindanceWeight : 0.2
        * distance : 6
        * stepSize : 2

- Schelling
    - Parametri Modello:
        * numeroAgentiTipoA : 400
        * numeroAgentiTipoB : 300
        * size: 30
    - Parametri agente A:
        * threshold : 0.4
        * visionRadius : 8
    - Parametri agente B:
        * threshold : 0.4
        * visionRadius : 8

- Predator Prey
    - Parametri Modello:
        * numeroAgentiCacciatori: 15
        * numeroAgentiPreda : 20
    - Parametri Preda:
        * visionRadiusPrey: 3
    - Parametri Cacciatori:
        * visionRadiusPredator: 4

- Virus Diffusion
    - Parametri:
        * size : 30

        * sani : 50
        * infetti : 30
        * infection : 70
        * recovery : 5
        * stepsize : 1

- Sugar Scape
    - Parametri Modello:
        * numeroAgentiConsumer: 20
        * numeroAgentiSugar : 5
    - Parametri Consumer:
        * visionRadius: 5
        * metabolismRate: 1
        * sugar: 5
        * maxSugar: 6
    - Parametri Sugar:
        * maxSugar: 6
        * sugarAmount: 2
        * growthRate: 3