Paradigmi di Programmazione e sviluppo

Report finale – ButterflyLFC Simulator

Christ Valtes Medjouwo Diagong

Christ.medjoumo@studio.unibo.it

Idene Prisca Magadjou Tchendjou

Ideneprisca.@studio.unibo.it

Gennaio 2022

Content

Introduzione

Il progetto ButterflyLfC Simulator in ambito della lezione di Paradigmi di Programmazione e sviluppo mira alla realizzazione di un simulatore del ciclo di vita delle farfalle all'interno di mondo usando il paradigma funzionale applicato attraverso il linguaggio Scala.

Il ciclo di vita delle farfalle implementato è composto di stage ben definito partendo dall'uova fino alla nascita di un butterfly adulto. Ogni stage con caratteristiche specifiche e soggetti ad aggiornamenti delle loro proprietà nel tempo.

Durante determinati stage, per sopravvivere le entità del ciclo si devono muovere all'interno del mondo della simulazione in precedenza definito alla ricerca di cibo aumentando ad ogni consumazione la loro vita corrente e raggiunto il tempo di vita necessario passano allo stage successivo del ciclo.

La quantità di diverse creature , e il numero di iterazione della simulazione sono parametrizzabili dall'utente attraverso un'apposita interfaccia di setting dei parametri necessari per far partire la simulazione.

1. **Processo di sviluppo**

Lo sviluppo del sistema verrà effettuato adottando un processo simil-Scrum, visto la ridotta dimensione del Team di sviluppo e la conseguente impossibilità di adottare Scrum in pieno. L'approccio utilizzato prevede la suddivisione in Scrum-Task anche di tutta la parte progettuale del sistema e di bootstrap del progetto, comprese la definizione dei requisiti, la configurazione degli ambienti (IntelliJ, GitHub, GithubAction) e la stesura del report finale. Il ruolo di Product Owner è stato ricoperto da Christ Medjouwo mentre quello di Scrum Master da Prisca Magadjou. Il team effettua meeting settimanale, nei quali si decidono per ogni Sprint, durante il meeting si suddividono i compiti fino allo Sprint successivo.

1.1 Meeting

Tutto il team partecipa ad ogni incontro, svolgendo sempre il ruolo di sviluppatore e quelli di Scrum Master e Product Owner. Si svolgono sprint planning settimanalmente e daily scrum di breve durata per aggiornarsi sull'avanzamento e per possibili questioni da risolvere.

1.1.1 Sprint planning

Con durata variabile tra un'ora e un'ora e trenta, ha obiettivo di pianificare lo Sprint, Infatti durante la seduta si svolgono retrospettive e si discute dell'incremento che si vuole raggiungere alla fine dello Sprint, aggiorniamo il Product Backlog in base allo sprint precedente e quello che si vuole iniziare e fine vengono definite task, assegnamento ai membri del team.

1.1.2 Daily Scrum

Con durata di quindici minuti I Daily Scrum si svolgono molto frequentemente all'interno della settimana sono molto facilitati visto il numero di membro del progetto ed a ogni seduta ognuno mostra il lavoro svolto fino a quel momento evidenziando eventuali problemi riscontrati e se ne valuta il modo di procedere.

1.2 Strumenti utilizzati

I principali strumenti utilizzati durante tutto il processo di sviluppo sono stati:

- **l'IDE IntelliJ** è stato utilizzato l'IDE IntelliJ in grado di fornire un supporto completo per lavorare con il linguaggio Scala
- **Suite GitHub**:
  - ✓ **GitHub Repository** come servizio di hosting del codice sorgente nel quale creare i repository;
  - ✓ **GitHub Actions**, come servizio di continuous integration in modo da controllare i diversi progetti lavorati contemporaneaente;
  - ✓ **GitHub Project**, Come Strumento di project management e di tracciamento per aggiornamento successivo del product backlog.
- **SBT** è il sistema di build automation nativo di Scala ed è necessario per la gestione delle dipendenze e la gestione del codice (test complessivo e/o compilazione) sorgente;
- **Scalastyle** come strumento di controllo di qualità automatizzato del codice;
- **Scalatest** come strumento di scrittura ed esecuzione dei test automatizzati, usando come convenzione l'utilizzo di FunSpec.

2. Requisiti e Analisi del Dominio

In questa sezione verranno trattati in modo dettagliato i requisiti del simulatore da noi realizzato. Questi non sono veramente rimasti invariati in corso d'opera in base ai diversi cicli di analisi e design previsti dalla metodologia di sviluppo agile.

2.1 Requisiti

2.1.1 Requisiti di business

I diversi requisiti di business previsti dal simulatore sono:

- Simulare il ciclo di vita o di crescita delle farfalle in un ambiente soggetto a predatori;
- Le entità del ciclo di vita si differenziano gli uni dagli altri in base alle loro caratteristiche reali che ne influenzano il comportamento (movimento, velocità, nutrizione…)
- Una simulazione composta da diverse giornate in ognuna delle quali certe creature hanno l'obiettivo di ricercare cibo che gli consenta di ottenere le energie necessarie a sopravvivere altre in più hanno obiettivo di trasformarsi passato un certo tempo nel mondo;

2.1.2 Requisiti utente

In questo progetto, Prisca Magadjou ricopre tra i suoi diversi ruoli quelli di Srum master e ha quindi richiesto un simulatore che potesse soddisfare la seguente richieste:

1. Impostare i parametri della simulazione;

    1.1 Impostare un numero iniziale di uova;

    1.2 impostare un numero iniziale di predatori

    1.3 impostare un numero iniziale di cibo nell'ambiente

1.3 Impostare la granularità temporale della simulazione;

2. Osservare l'evoluzione del ciclo di vita delle farfalle graficamente.

2.1. Osservare le creature nell'ambiente di simulazione

2.2 Osservare il movimento degli insetti nell'ambiente 2D;

2.3 Osservare il cambiamento di stage (transizioni) delle creature del ciclo di vita tramite graficamente;

2.4 Osservare l'alimentazione delle creature;

2.5 Osservare l'interazione tra creature e predatori;

2.6 vedere una legenda delle creature presente in ambiente durante la simulazione

Le funzionalità opzionali saranno:

1.Impostare diverse granularità temporali per osservare l'andamento della simulazione;

2. Visualizzazione e controllo della simulazione con possibilità di modificare parametri mediante un'interfaccia grafica.

3. vedere statistiche riassuntive alla fine della simulazione.

2.1.3 Requisiti funzionali

Il simulatore prodotto dovrà essere

1. Essere composto da giornate che si compongono di un numero fisso di iterazioni, le quali sono l'unità temporale di aggiornamento della simulazione;

2. Supportare diverse tipologie di entità
   2.1 entità del ciclo
      2.1.1 **egg**: non si muovono, eseguono transizione verso lo stage larva dopo aver raggiunto il life cycle, in ambiente non possono essere

consumati da predatori, la loro evoluzione nel tempo è graficamente visibile.

2.1.2 **larva**: hanno velocità media, si muovono per cercare cibo, eseguono transizione verso lo stage puppa dopo aver raggiunto il life cycle, periscono se non mangiano mentre stanno in ambiente o se sono mangiati da predatori.

2.1.3 **puppa**: hanno velocità bassa, eseguono transizione verso lo stage butterfly adulto dopo il lyfe cycle, possono anche loro essere mangiati da predatori in movimento

2.1.4 **Butterfly adulto**: velocità alta possono essere mangiati da predatori, consumano il nettare delle piante e si riproducono.

2.2 altre entità

2.2.1 predatori: sono essere viventi, con velocità alta si muovono alla ricerca del cibo e possono morrire nel mondo per mancanza di cibo, e non fanno parte del ciclo di vita

2.2.2 Plant:

2.2.2.1 Flower plant: rappresenta il cibo della larva e delle puppa

2.2.2.2 Nectar plant: sono il cibo del butterfly adulto

2.1.3 Requisiti non funzionali

L'unico requisito funzionale è stato la realizzazione di un simulatore estendibile, cioè usare metodologie e principi che promuovono la scalabilità del software.

2.1.4 Requisiti di implementazione

I requisiti di implementazione che il simulatore dovrà garantire sono:

1. Strutturazione del software in modo da poter aggiungere nuove proprietà ambientali in modo semplice;

2. Sempre fare Testing mediante Scala. Test al fine di poter aggiornare in modo semplice e controllato il simulatore e le sue funzionalità. Come requisito implementativo extra e opzionale ci riserviamo la possibilità dell'impiego di Prolog per l'implementazione di una qualsiasi funzionalità del software.

2.2 Analisi del Dominio

Cui viene riportata una descrizione in contesto reale del ciclo di vita delle farfalle con lo scopo di introdurre alcuni termini e concetti chiave utilizzati nella realizzazione dell'applicazione.

❖ Ciclo di vita delle farfalle

Il ciclo di vita delle farfalle è un ciclo in quattro fasi diverse con caratteristiche specifiche associati ad ogni creatura al suo stage (*https://it.wikipedia.org/wiki/Farfalla*). Per quasi tutte le fasi del ciclo di vita le farfalle sono sottoposte a predatori chi sono altri insetti come lucertole, ragni, pipistrelli, uccelli causando una riduzione nel tempo del numero di farfalle presenti nel mondo.

- ❖ Entità rappresentante di ogni fase del ciclo con le sue caratteristiche:
    1. Egg
    2. Larva
    3. Puppa
    4. Butterfly adulto

- ❖ Entità consumatori di farfalle: Predatori

Le farfalle rappresentano un cibo per i predatori ma devono al loro torno nutrirsi per poter sopravvivere e ci sono variazioni a fasi diversi del ciclo:

1. Flowerplant
2. Nectarplant

L'evoluzione del ciclo di viltà di una creatura del ciclo di vita è strettamente legata al suo tempo di sopravvivenza nel mondo quindi per passare da una fase all'altra dipendente dalle sue caratteristiche reale le creature devono passare un certo tempo nel mondo e non essere mangiati da predatori, in più devono resistere alle temperature considerato con elemento esterno al loro ciclo di vita.

Una volta avviata la simulazione verranno disposte le creature a all'interno del mondo, in seguito del tempo le uova (Egg) cominceranno a trasformassi diventato delle Larva fino a diventare delle Farfalle. Alcune creature inizieranno quindi a

muoversi in cerca di cibo, cercando di non morire in quanto la loro vita si riduce nel tempo. Le creature in base alla loro struttura possono avere diversi valori di dimensione.

3. Design Architetturale

Come pattern di definizione dell'architettura si è sfruttato il modello di design architetturale MVC (Model-View-Controller). Partendo di questo approccio, l'intera logica del sistema è modellata sfruttando il paradigma object oriented mantenendo la modularità del codice.

Il pattern consente sia la separazione dei compiti che dei componenti, favorisce ugualmente la scalabilità del software e la comprensione anche da parte di altri programmatori esterni del codice prodotto. La figura seguente modella ad alto livello l'architettura generale dell'applicativo.

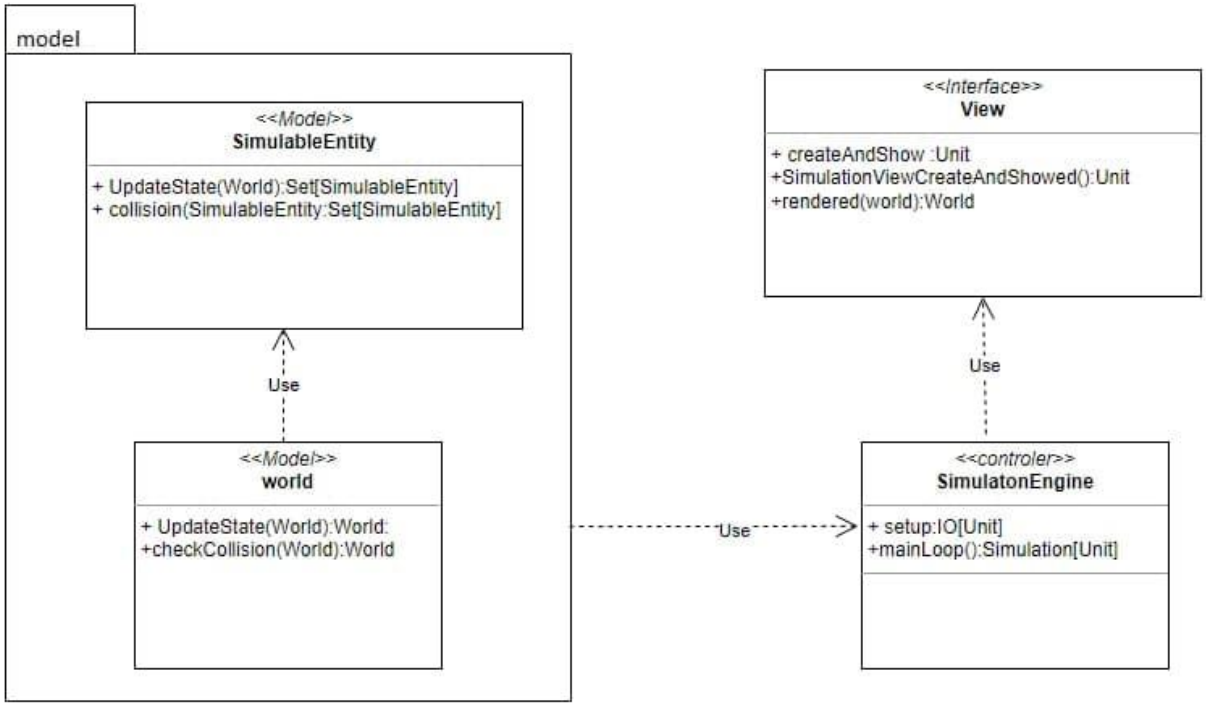

Fig 3.1 diagramma dell'architettura generale della simulazione

3.1 Pattern MVC

Come visibile di sopra i macro-componenti sono interamente indipendenti tra di loro, permettendo una separazione ad alto livello delle funzionalità e degli interessi.
Il nostro Model gestisce la logica e i dati dell'applicazione per questo Interpreta il comportamento specifico dell'applicazione cioè quello delle diverse entità che interagiscono nella simulazione, e questo viene gestito in modo totalmente indipendente dall'interfaccia utente. Il Controller esegue ed applica la logica dell'applicazione e serve punti di interazione tra model e view. La View gestisce l'interazione tra il simulatore e gli utenti reali, implementa la visualizzazione dei dati del modello attraverso diverse interfacce grafiche.

4. Design di Dettaglio

Dopo aver descritto l'architettura del sistema, si procede con il design di dettaglio delle sue componenti principali. volendo seguire in maniera rigida il requirement implementativo, si è deciso di basare il design sui capisaldi del paradigma FP (funzionale) aggiungendo talvolta elementi tipici del paradigma OO-FP Mixed, come dell'interfaccia come essendo delle astrazioni attraverso la quale caratterizzare i component, per scatenare comportamenti diversi su entità soggette ad un comune contratto. La modellazione delle strutture dati infatti, attenendosi al paradigma FP descritto sopra, rispetterà il concetto di immutabilità; pertanto, le operazioni sui data type non avranno side-effect indesiderati, ma solamente la produzione di un nuovo costrutto, con attributi aggiornati. Favorendo così la descrizione lazy della computazione.

Queste caratteristiche si rifletteranno sulla totalità delle classi sviluppate.

4.1 **Model**

4.1.1 **Creature**

La modellazione delle creature si è fatta prima modellando la struttura generale delle creature, poi aggiungendo ad ogni creatura il suo comportamento in base alla sua caratteristica.

4.1.1.1 Struttura delle creature della simulazione

Per quanto riguarda la modellazione della struttura delle creature del Mondo, si sfrutta il meccanismo dei self-type al fine di rendere modulare l'acquisizione di ogni singola caratteristica genetica della popolazione attraverso l'algoritmo di linearizzazione disponibile in scala. Questo è fatto definendo un set d proprietà e i self-type corrispondenti, poi si assegna a ciascuno di essi une dominio ed una singola funzionalità. Ciò porta ad una struttura a più livelli in cui alcuni componenti condividono lo stesso livello.

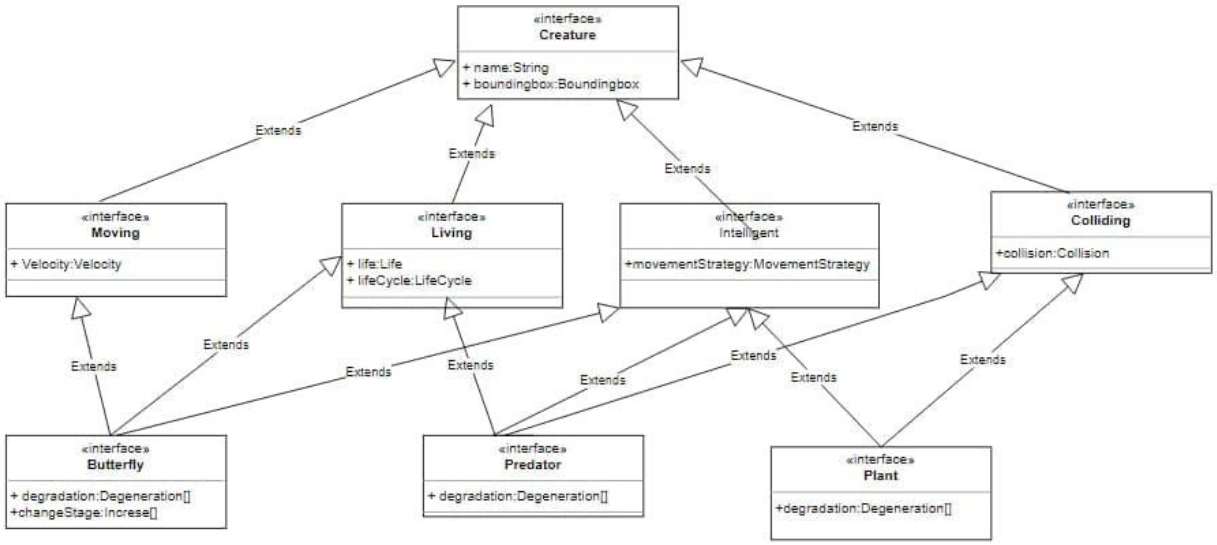

Fig 4.1 Diagramma uml della struttura delle creature

Come proprietà vengono definite ***CollisionEffect***, ***EatingEffect***, ***DegradationEffect***, …, che sono utilizzati come parametri higher order e definiscono il comportamento della creatura per aspetti specifici. Ciò permette di riutilizzare una stessa struttura per modellare una serie di comportamenti diversi, piuttosto che codificare tale diversità di comportamento nel codice, trasferendola dunque a livello di istanza anziché di classificazione (dall'aspetto intensionale a quello estensionale). Invece di esprimere nel codice di una creatura la logica di alcuni aspetti specifici, si incapsula dunque tale gestione in un parametro higher order, permettendo ad esempio di riutilizzare anche per creatura con diverse logiche di degradazione la struttura rimanente.

4.1.2 Comportamento delle creature

Per quanto riguarda L'aspetto comportamentale delle entità, cioè le operazioni che agiscono sul loro stato in occasione dell'evento di aggiornamento del mondo e di collisione con altre creature lo gestiamo tramite due traits con self types: ***UpdatableEntity*** e ***CollidableEntity***.

Questa modellazione prevede di sfruttare la stessa interfaccia per variare il comportamento sotto la stessa astrazione e fornire estendibilità e un approccio funzionale nella gestione dello stato. Vienne definito un altro tipo ***SimulableEntity*** che gioca il ruolo di set di creature e vera utilizzato come signature per i due trait sopra definiti.

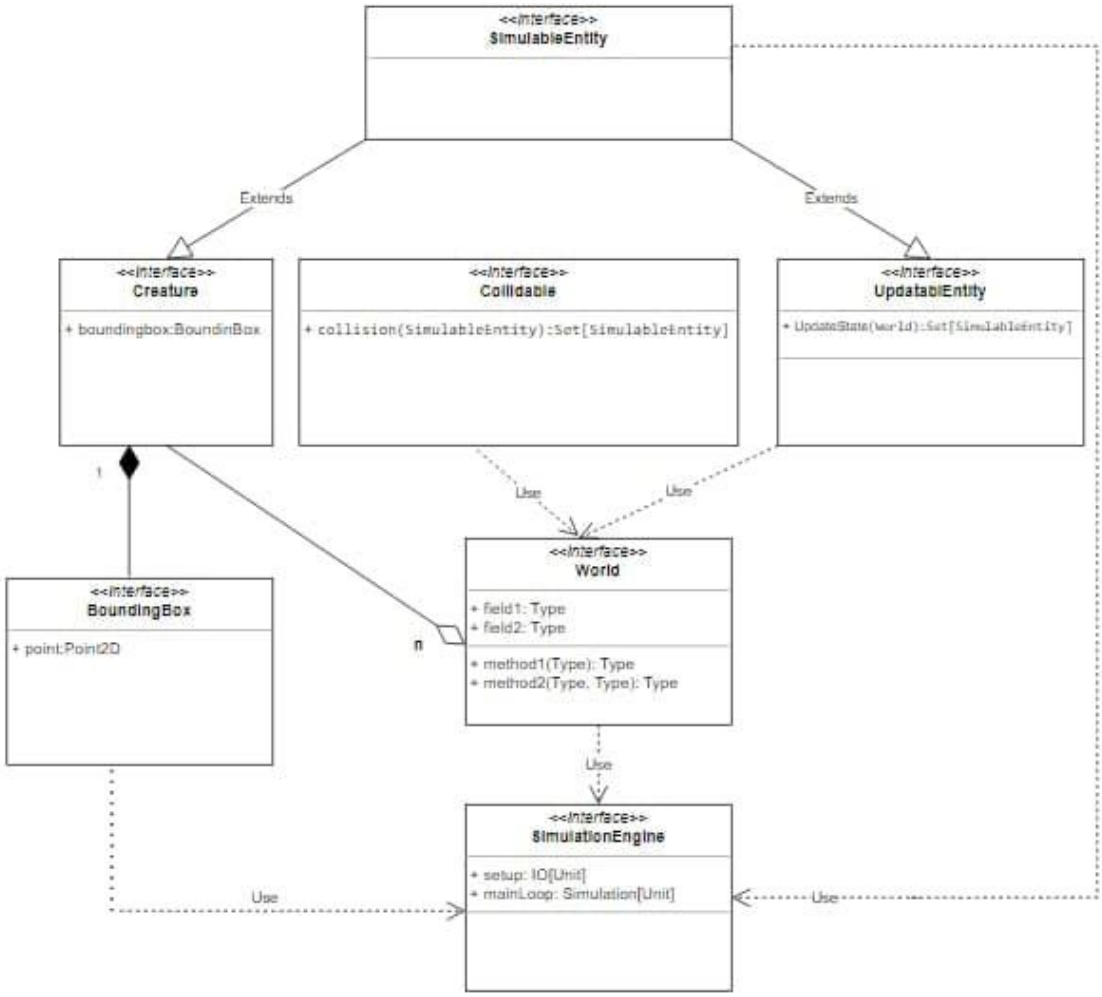

Fig. 4.2 Diagramma uml del comportamento delle creature e l'integrazione

4.2 The World

L'entità mondo come visibile sulla figura 4.1 modella l'ambiente dove le creature si muovono per cercare food. È stato modellato come un contenitore immutabile delle proprietà del simulator e delle creature che si muovono dentro.

Oltre a ciò, viene definita a tal proposito l'entità Width e height che definiscono gli angoli dell'ambiente in modo da dare una limitazione al Mondo stesso e in modo da poter effettuare una generazione controllata di Plant e creature all'interno dei bordi stessi. Possiamo notare l'immutabilità della struttura sviluppata.

4.3 La View

L'interfaccia grafica è composta da due parti:

- La view per il set-up del simulator, nella quale vengono richiesti parametri
- La view della simulazione vera e propria, che appunto mostrerà lo stato Real-time delle entità del sistema e la loro trasformazione.

4.4 SimulationEngine

Il motore della simulazione SimulationEngine consiste in una descrizione monadica delle fasi della simulazione, che prevede l'aggiornamento dei parametri del mondo, il rilevamento e risoluzione delle collisioni fra le creature che lo popolano, la visualizzazione a video del suo stato aggiornato e l'attesa dell'intervallo di tempo prima della successiva iterazione. In questo modo il codice non contiene i side-effect. Questo andamento sequenziale e periodico è stato modellato attraverso una composizione di state monad e di IO monad (per le operazioni di I/O come il display a video delle creature). La IO monad permette di esprimere una computazione in modo lazy, e in più, la state monad consente di rendere implicito il passaggio di stato aggiornato fra le operazioni sequenziali di manipolazione del mondo, ciascuna delle quali determina una sua nuova versione. Al fine di combinare 2 monadi differenti, che n quanto tali non si possono comporre fra di loro è stato utilizzato il monad transformer StateT.

4.5 Diversi pattern di progettazione

Il team ha cercato di utilizzare il più possibile i pattern di progettazione, al fine di trovare una soluzione di design generale a problemi ricorrenti. Questo approccio si è rivelato efficace nel ridurre il debito tecnico.

- **Pattern Strategy**

Abbiamo ampiamente utilizzato il pattern strategy all'interno del progetto in quanto direttamente supportato dal linguaggio come, ad esempio, il passaggio di funzione higher-order.

- **Pattern Singleton**

Nel progetto si è fatto largo uso del pattern singleton, in quanto scala rende particolarmente semplice l'implementazione dello stesso. Nello specifico lo abbiamo utilizzato per.

- Oggetti che definiscono varie strategie,
- Gestione delle collisioni,
- Definizione delle proprietà di oggetti.

- **Pattern Factory**

Abbiamo utilizzato il pattern Factory per la creazione delle caratteristiche. Facendo uso del compagnon object e delle classi private e infatti risultato comodo utilizzare tale pattern per estrarre la creazione delle caratteristiche.

- **Pattern Builder**

Come altri pattern anche il Builder è stato utilizzato nella sua accezione legata a Scala, in quanto ha un supporto nativo per differenti pattern progettuali. Nello specifico per il pattern Builder è stato sfruttato l'utilizzo di immutable case classes con default nei parametri dei costruttori come ad esempio per i Butterfly.

❖ **Self-Type**

I self types come accennato nella sezione più in alto sono stati utilizzati per l'implementazione dei seguenti comportamenti:

- EggBehavior
- LarvaBehavior
- PuppaBehavior
- ButterflyBehavior
- PlantBehavior
- NectarBehavior
- PredatorBehavior

4.4 Organizzazione del codice

Oltre all'impatto dell'uso del pattern MVC sopra descritto sull'organizzazione del codice, è stato usato un approccio organizzativo del codice basato su package, Ciascuno rappresentando una funzionalità ben precisa del sistema.

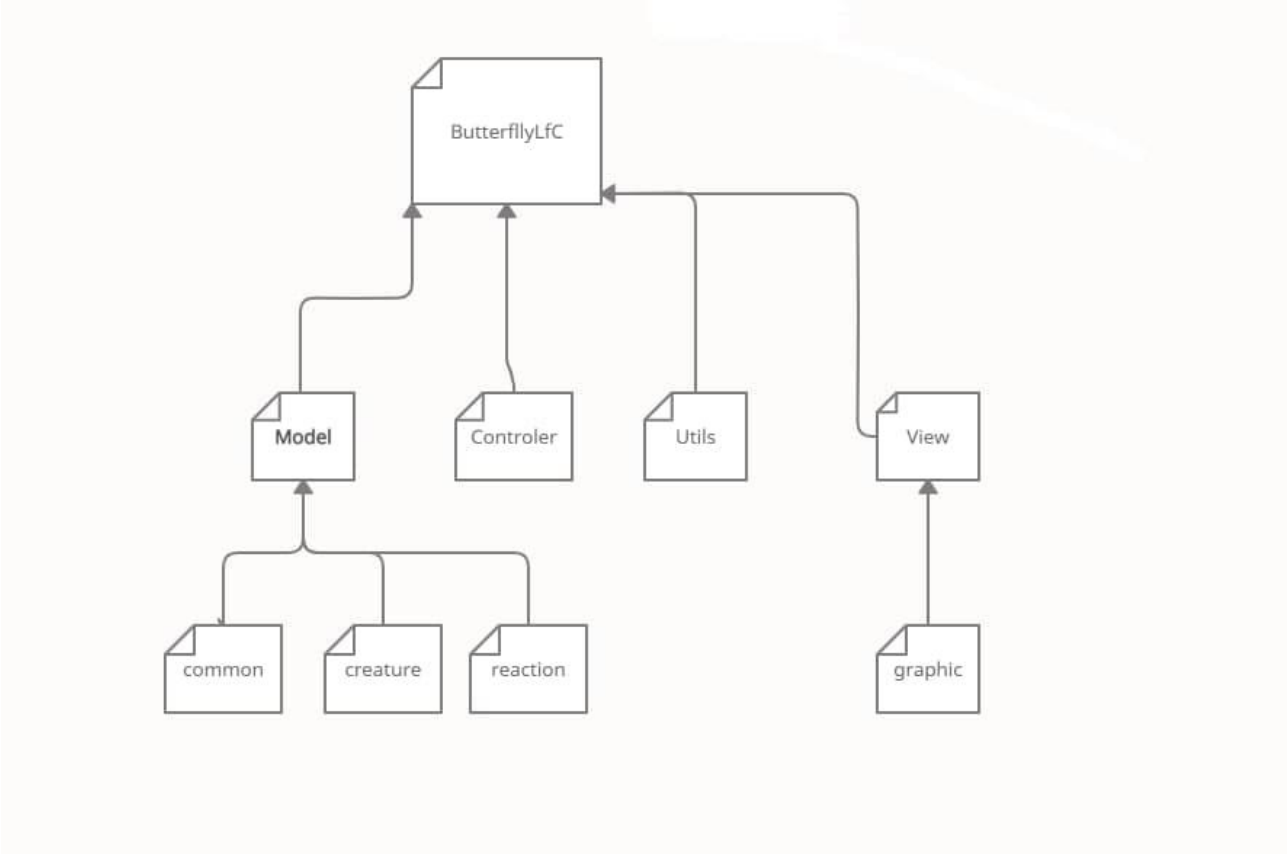

Fig. 4.2 Diagramma della struttura in packaging del codice del software

5. Implementazione

Di seguito sono trattati i punti dell'implementazione da noi considerati rilevanti per una completa comprensione del progetto.

5.1 Applicazione del paradigma funzionale

Avendo trattato nel corso la programmazione funzionale, Il team si è impegnato al fine di sfruttare il più possibile tale paradigma, cercando di esulare al massimo delle sue possibilità dalle classiche abitudini di programmazione object-oriented, ma comunque traendo il meglio dai due paradigmi. Per fare ciò sono state utilizzate diverse metodologie:

- ❖ Utilizzo di funzioni ricorsive:

In molti casi, per rispettare il requisito precedente si è fatto uso della ricorsione. Per gestire la lista delle creature, ad esempio, si è preferita la ricorsione rispetto ai

tradizionali metodi iterativi. Lo stesso loop della simulazione viene realizzato come funzione ricorsiva.

- ❖ Utilizzo di funzioni higher-order:

che permettono una facile ed immediata realizzazione del pattern Strategy, consentono una maggiore riutilizzabilità del codice. In questo modo è possibile passare alle funzioni strategie esterne, non necessitando così di modificare il codice. Rispettando quanto sopra elencato, il Controller e il Model dell'applicazione sono stati realizzati con un approccio puramente funzionale. La View adottare anch'essa il paradigma descritto in questo corso attraverso l'utilizzo della libreria cats.effect.IO

- ❖ Inutilizzo dei side effect:

Nel core della simulazione si è cercato quanto più possibile di non utilizzare side effect. Si è preferito quindi utilizzare oggetti immutabili, preferendo crearne di nuovi piuttosto che modificare lo stato di quelli esistenti. creando ad ogni modifica un nuovo oggetto immutabile.

5.2 Testing

L'utilizzo di test si è rivelato fondamentale per la corretta realizzazione del progetto sempre con l'obiettivo di sfruttare al massimo i principi di programmazione studiato. Seguendo le linee guida della metodologia agile si è cercato di scrivere test che permettano di garantire qualità del sistema realizzato e che servono da documentazione dello stato dell'arte dello stesso.

Per quanto riguarda il core del sistema, si è cercato di seguire un approccio al testing quanto più possibile TDD, cercando di seguire il ciclo Red - Green – Refactor Applicato alle funzioni core del progetto. Nelle fasi successive invece tale approccio è stato sostituito con un metodo di testing più tradizionale, testando una funzionalità dopo l'altra quando pronta. Abbiamo utilizzato principalmente test unitari e uno stile classico FunSuite sfruttando la libreria ScalaTest.

5.3 Suddivisione dei tasks

Inizialmente il team ha effettuato un'analisi preliminare, dettagliando quali caratteristiche e requisiti il software dovesse esibire. In seguito a ciò, abbiamo seguito con una frequente serie di riunioni dove, partendo da piccoli diagrammi UML che descrivevano a grandi linee l'architettura del sistema, siamo arrivati a quelli con un livello di dettaglio più significativo. A seguito di questa fase di analisi ci siamo divisi i lavori però collaborando sempre insieme. Di seguito la descrizione del lavoro fatto da ciascuno di noi 2.

- Prisca Magadjou

Il mio ruolo nel progetto al livello implementativo è stato l'implementazione della struttura e i comportamenti delle diverse creature (*Behaviour*) agli effetti applicabili in caso di collisioni con altra creatura. Ho realizzato le classi e le funzioni relativi alle creature Plant, NectarPLant, FlowerPlant e Predator.

Ho implementato la case class relativa ai parametri dell'environnement e la classe *BoundingBox* relativa alle shape delle entità e le collisioni possibile tra di loro. Per quanto riguarda la View, ho realizzato Il frame *simulationSetting* e l'intero package graphic dov'è è stata implementata la logica di disegno delle nostre creature.

Al livello dei test, ho realizzato una parte *di* ***ButterflyTest***, ***PlantTEst***, ***DegenerationTest***.

- Christ Medjouwo

Il mio ruolo è stato quello di progettare e implementare il core della simulazione SimulationEngine, TimingIO e Launcher e le modalità di interazione con il modello, introducendo le monadi ed impiegandole qui come astrazione per la descrizione delle fasi sequenziali di aggiornamento del world. La verifica e la gestione delle collisioni utilizzando le cats.data.State.

Ho anche implementato la case class World e la sua struttura. Ho implementato la updateState del mondo, la classe temperature e il suo comportamento dentro la classe World. Per la componente view, mi sono occupato del display della simulazione. Ho anche maggiormente lavorato sulla logica di spostamento delle creature all'interno del mondo. Sulla creazione dei trait utilizzati come interfaccia di raggruppamento delle creature immersi dentro il mondo soggetto a alcuni aggiornamenti che sia di stato e di struttura cioè UpdatableEntity e collidibile.

Ho implementato la logica di trasformazione delle diverse fasi delle farfalle cioè da Egg→larva →Puppa →ButteflyAdult (creazione le diverse funzione relative)

Ho contribuito nella creazione della struttura delle diverse creature del sistema. per il testing delle componenti ho realizzato *WorldTests*, *CollitionTest* e una parte de *PlantTEst, DegenerationTest, ButterflyTest.* Oltre delle parti citate, ho redatto anche la scaladoc.

6. Retrospettiva

La fase di avviamento si è incentrata sulla definizione dei requisiti di base del progetto, analisi del problema e definizione dei componenti principali del sistema. Sono stati stabiliti struttura e comportamento delle creature come avvengono le interazioni tra di queste. Queste operazioni hanno portato via molto tempo ma ci ha permesso di avere le idee chiare. Sono stati anche preparati i tool di sviluppo, in particolare SBT e GitHub Action CI. I diagrammi UML delle classi core del progetto sono stati prodotti in questa fase, in modo che si avesse una struttura solida di partenza a cui fare riferimento per lavorare in modo indipendente.

6.1 Sprint 1

Per avere un risultato percettibile, anche di raggiungere gli obiettivi fissati nella parte di definizione dei requisiti, si è deciso di partire da subito con lo sviluppo della View. Si è optato direttamente per l'utilizzo di Swing. Il prodotto finale dello sprint aveva già buona parte della struttura completa necessaria, nonostante nessuna funzione di rilievo fosse ancora presente.

6.2 Sprint 2

A questo punto dello sviluppo il lavoro si è spostata sulla creazione delle creature con il loro comportamento, includendo l'inserimento dei parametri come la temperatura come avendo un effetto su di loro. È anche implementato in questa fase la logica del movimento le classi BoundingBox per anticipare l'idea di collisione futura. Alla fine dello sprint abbiamo qualche problema sullo spostamento delle creature però in modo più generale le creature sono visibile e possono fare qualche azione.

6.3 Sprint 3

Il terzo sprint ha visto in parte la correzione dei problemi incontrati nella fase precedente, e in grand parte la modellazione del simulationEngine che è stato uno dei componenti più importante del lavoro

6.4 Sprint 4

Il focus dell'ultimo sprint si è incentrato sulla rifattorizzazione del codice, e all'introduzione di ottimizzazioni per l'incremento delle performance. sfortunatamente sono stati tantissimi imprevisti dovuto allo stato di salute di uno dei membri e la conclusione dei lavori è stata un po' più difficile che previsto. Infine, questa fase si è conclusa con la documentazione del codice usando la Scaladoc.

7. Sviluppi futuri

In questa sezione sono elencati possibili sviluppi futuri che potrebbe voler sviluppare un programmatore che mette mano al progetto in questione.

- Fare influire variabile stagionale sull'ambiente in cui viene lasciata la simulazione e vedere il comportamento delle creature ad ogni fase del ciclo.

- Sviluppo del senso delle creature per modificare il proprio percorso e dirigersi verso il cibo o scappare da predatori.

- Aggiungere proprietà di ibernazione delle creature durante stagione di basse temperature.

- Produrre statistiche in tempo reale dello stato delle creature in simulazione.

Conclusioni

Sebbene il gruppo non sia riuscito a completare un gruppo da 4 persone, rimanendo in 2, si è comunque riusciti a sviluppare il progetto.

Il lavoro è stato organizzato seguendo gli standard definiti dal framework Scrum e dall'utilizzo della metodologia agile. Nel complesso, all'interno del gruppo non sono state riscontrate grosse problematiche relative alla pianificazione degli Sprint e alla metodologia di sviluppo ma piuttosto una mancanza di parerei diversi davanti a problemi sia progettuale che di sviluppo. Inoltre, è stata una più che valida esplorazione del linguaggio funzionale Scala e delle sue potenzialità in termini di sviluppo e qualità software.

In conclusione, il team ha avuto un responso positivo, trovando un buon bilanciamento nella suddivisione dei lavori, nonostante il caso di malattia chi ha rallentato il lavoro per un bel periodo di tempo.